

***TMS320C6000***  
***DSP/BIOS***  
***Application Programming Interface***  
***(API) Reference Guide***

Literature Number: SPRU403A  
May 2000



## IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

## Read This First

---

---

---

### ***About This Manual***

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

You should read and become familiar with the TMS320C6000 DSP/BIOS User's Guide (literature number SPRU303a), the companion volume to this reference guide.

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

### ***Notational Conventions***

This document uses the following conventions:

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

Here is a sample program listing:

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

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

## **Related Documentation From Texas Instruments**

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

**TMS320C6000 DSP/BIOS User's Guide** (literature number SPRU303b) gives developers of applications for DSP chips the information necessary to develop and analyze embedded real-time software using the DSP/BIOS software. The **User's Guide** is the companion to this **API Reference Guide**.

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

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

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

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

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

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

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

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

**TMS320C6000 Code Composer Studio Tutorial** (literature number SPRU301b) introduces the Code Composer Studio integrated development environment and software tools.

## **Related Documentation**

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

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

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

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

## **Trademarks**

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, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, BIOSuite, and SPOX.

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

# Contents

---

---

---

<b>1</b>	<b>API Functions</b> .....	<b>1-1</b>
	<i>This chapter describes the DSP/BIOS API functions, which are alphabetized by name. In addition, there are reference sections that describe the overall capabilities of each module.</i>	
1.1	DSP/BIOS Modules .....	1-2
1.2	Naming Conventions .....	1-3
1.3	List of Operations .....	1-3
1.4	Assembly Language Interface .....	1-11
<b>2</b>	<b>Utility Programs</b> .....	<b>2-1</b>
	<i>This chapter provides documentation for utilities that can be used to examine various files from the MS-DOS command line. These programs are provided with DSP/BIOS in the bin subdirectory.</i>	
2.1	cdbprint .....	2-2
2.2	gconfgen .....	2-3
2.3	nmti .....	2-5
2.4	sectti .....	2-6
2.5	size62 .....	2-7
2.6	vers .....	2-8
<b>A</b>	<b>Function Callability and Error Tables</b> .....	<b>A-1</b>
	<i>This appendix provides tables describing errors and function callability.</i>	
A.1	Functions Callable by Tasks, SWI Handlers, or Hardware ISRs .....	A-2
A.2	DSP/BIOS Error Codes .....	A-8

# API Functions

---

---

---

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

Topic	Page
<b>1.1 DSP/BIOS Modules</b> .....	<b>1-2</b>
<b>1.2 Naming Conventions</b> .....	<b>1-3</b>
<b>1.3 List of Operations</b> .....	<b>1-3</b>
<b>1.4 Assembly Language Interface</b> .....	<b>1-11</b>

## 1.1 DSP/BIOS Modules

The DSP/BIOS modules are:

<b>Module</b>	<b>Description</b>
ATM	Atomic functions written in assembly language
C62	Target-specific functions
CLK	System clock manager
DEV	Device driver interface
GBL	Global setting manager
HST	Host channel manager
HWI	Hardware interrupt manager
IDL	Idle function and processing loop manager
LCK	Resource lock manager
LOG	Event Log manager
MBX	Mailboxes manager
MEM	Memory manager
PIP	Buffered pipe manager
PRD	Periodic function manager
QUE	Queue manager
RTDX	Real-time data exchange manager
SEM	Semaphores manager
SIO	Stream I/O manager
STS	Statistics object manager
SWI	Software interrupt manager
SYS	System services manager
TRC	Trace manager
TSK	Multitasking manager
C library stdlib.h	Standard C library I/O functions

## 1.2 Naming Conventions

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

In the Assembly Interface section for each macro, Preconditions lists registers that must be set before using the macro. Postconditions lists the registers set by the macro that you may want to use. Modifies lists all individual registers modified by the macro, including registers in the Postconditions list.

## 1.3 List of Operations

The DSP/BIOS operations are:

Function	Operation
ATM_andi, ATM_andu	Atomically AND two memory locations and return previous value of the second
ATM_cleari, ATM_clearu	Atomically clear memory location and return previous value
ATM_deci, ATM_decu	Atomically decrement memory and return new value
ATM_inci, ATM_incu	Atomically increment memory and return new value
ATM_ori, ATM_oru	Atomically OR memory location and return previous value
ATM_seti, ATM_setu	Atomically set memory and return previous value
C62_disableIER	Disable certain maskable interrupts
C62_dispatchPlug	Plug the HWI dispatcher
C62_enableIER	Enable certain maskable interrupts
C62_plug	C function to plug an interrupt vector
CLK_countspms	Number of hardware timer counts per millisecond
CLK_gethtime	Get high-resolution time
CLK_getltime	Get low-resolution time
CLK_getprd	Get period register value

<b>Function</b>	<b>Operation</b>
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
HST_getpipe	Get corresponding pipe object
HWI_disable	Globally disable hardware interrupts
HWI_enable	Globally enable hardware interrupts
HWI_enter	Hardware interrupt service routine prolog
HWI_exit	Hardware interrupt service routine epilog
HWI_restore	Restore global interrupt enable state
IDL_run	Make one pass through idle functions
LCK_create	Create a resource lock
LCK_delete	Delete a resource lock
LCK_pend	Acquire ownership of a resource lock
LCK_post	Relinquish ownership of a resource lock

---

---

<b>Function</b>	<b>Operation</b>
LOG_disable	Disable a log
LOG_enable	Enable a log
LOG_error/LOG_message	Write a message to the system log
LOG_event	Append an unformatted message to a log
LOG_printf	Append a formatted message to a message log
LOG_reset	Reset a log
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_alloc, MEM_valloc, MEM_calloc	Allocate from a memory section
MEM_define	Define a new memory section
MEM_free	Free a block of memory
MEM_redefine	Redefine an existing memory section
MEM_stat	Return the status of a memory section
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

---

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

---

---

<b>Function</b>	<b>Operation</b>
RTDX_channelBusy	Return status indicating whether a channel is busy
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_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_count	Get current semaphore count
SEM_create	Create a semaphore
SEM_delete	Delete a semaphore
SEM_ipost	Signal a semaphore (interrupt only)
SEM_new	Initialize a semaphore
SEM_pend	Wait for a semaphore
SEM_post	Signal a semaphore
SEM_reset	Reset semaphore
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

---

<b>Function</b>	<b>Operation</b>
SIO_issue	Send a buffer to a stream
SIO_put	Put buffer to a stream
SIO_reclaim	Request a buffer back from a stream
SIO_segid	Memory section used by a stream
SIO_select	Select a ready device
SIO_staticbuf	Acquire static buffer from stream
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_andn	Clear bits from SWI's mailbox and post if becomes 0
SWI_create	Create a software interrupt
SWI_dec	Decrement SWI's mailbox and post if becomes 0
SWI_delete	Delete a software interrupt
SWI_disable	Disable software interrupts
SWI_enable	Enable software interrupts
SWI_getattr	Get attributes of a software interrupt
SWI_getmbx	Return SWI's mailbox value
SWI_getpri	Return an SWI's priority mask
SWI_inc	Increment SWI's mailbox and post
SWI_or	Set or mask in an SWI's mailbox and post
SWI_post	Post a software interrupt
SWI_raisepri	Raise an SWI's priority
SWI_restorepri	Restore an SWI's priority

---

---

<b>Function</b>	<b>Operation</b>
SWI_self	Return address of currently executing SWI object
SWI_setattr	Set attributes of a software interrupt
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_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_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

---

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

---

## 1.4 Assembly Language Interface

When calling DSP/BIOS APIs from assembly source code, you should include the module.h62 header file for any API modules used. This modular approach reduces the assembly time of programs that do not use all the modules.

Where possible, you should use the DSP/BIOS API macros instead of using assembly instructions directly. The DSP/BIOS API macros provide a portable, optimized way to accomplish the same task. For example, use `HWI_disable` instead of the equivalent instruction to temporarily disable interrupts. On some chips, disabling interrupts in a threaded interface is more complex than it appears.

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

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

The Postconditions section lists registers set by the macro.

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

### Example:

#### Assembly Interface

**Syntax** `HWI_enter ABMASK CMASK IEMASK CCMASK`

**Preconditions** interrupts are globally disabled (i.e., `GIE == 0`)

**Postconditions** `amr = 0`

**Modifies** `a0, a1, amr, b0, b1, b2, b3, b14, b15, csr, ier`

**Reentrant** `yes`

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 C Compiler User’s Guide*” for more details.

By default, the Configuration Tool creates two names for each object: one beginning with an underscore, and one without. This allows you to use the name without the underscore in both C and assembly language functions. You can turn off this feature by clicking off the box called Generate C Names for All Objects in the Properties box of the Project Manager in the Configuration Tool.

**ATM Module***Atomic functions written in assembly language***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 atomic functions which are used to manipulate variables with interrupts disabled. These functions may therefore be used on data shared between tasks, and on data shared between tasks and interrupt routines.

**ATM\_andi**

*Atomically AND two Int memory locations and return previous value of the second*

**C Interface**

**Syntax**                   ival = ATM\_andi(idst, isrc);

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

**Return Value**        Int        ival;   /\* previous value of \*idst \*/

**Assembly Interface**   none

**Description**

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

```
`interrupt disable`  
  
val = *dst;  
  
*dst = val & src;  
  
`interrupt enable`  
  
return(val);
```

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 two Uns memory locations and return previous value of the second*

**C Interface**

**Syntax**                    uval = ATM\_andu(udst, usrc);

**Parameters**                volatile Uns \*udst; /\* pointer to unsigned \*/  
                               Uns            usrc; /\* unsigned mask \*/

**Return Value**            Uns        uval; /\* previous value of \*udst \*/

**Assembly Interface**      none

**Description**

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

```
`interrupt disable`
val = *dst;
*dst = val & src;
`interrupt enable`
return(val);
```

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****Syntax**`ival = ATM_cleari(idst);`**Parameters**`volatile Int *idst; /* pointer to integer */`**Return Value**`Int ival; /* previous value of *idst */`**Assembly Interface**

none

**Description**

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

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

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

**See Also**

ATM\_clearu  
ATM\_seti

**ATM\_clearu***Atomically clear Uns memory location and return previous value***C Interface****Syntax**`uval = ATM_clearu(udst);`**Parameters**`volatile Uns *udst; /* pointer to unsigned */`**Return Value**`Uns uval; /* previous value of *udst */`**Assembly Interface**

none

**Description**

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

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

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

**See Also**

ATM\_clearu  
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 \*/

**Assembly Interface**

none

**Description**

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

```
`interrupt disable`  
val = *dst - 1;  
*dst = val;  
`interrupt enable`  
return (val);
```

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 \*/

**Assembly Interface**

none

**Description**

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

```
`interrupt disable`  
  
val = *dst - 1;  
  
*dst = val;  
  
`interrupt enable`  
  
return (val);
```

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 \*dst; /\* pointer to integer \*/

**Return Value**

Int ival; /\* new value after increment \*/

**Assembly Interface**

none

**Description**

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

```
`interrupt disable`  
  
val = *dst + 1;  
  
*dst = val;  
  
`interrupt enable`  
  
return (val);
```

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 \*/

**Assembly Interface**      none

**Description**

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

```
`interrupt disable`  
  
val = *dst + 1;  
  
*dst = val;  
  
`interrupt enable`  
  
return (val);
```

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

**Return Value**        Int        ival;     /\* previous value of \*idst \*/

**Assembly Interface**   none

**Description**

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

```
`interrupt disable`  
  
val = *dst;  
  
*dst = val | src;  
  
`interrupt enable`  
  
return(val);
```

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 mask \*/  
                                  Uns                usrc;            /\* unsigned mask \*/

**Return Value**              Uns            uval;    /\* previous value of \*udst \*/

**Assembly Interface**    none**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`

val = *dst;

*dst = val | src;

`interrupt enable`

return(val);

```

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**

**Syntax**                    iold = ATM\_seti(idst, inew);

**Parameters**                volatile Int    \*idst;    /\* pointer to integer \*/  
                              Int                inew;    /\* new integer value \*/

**Return Value**             Int            iold;    /\* previous value of \*idst \*/

**Assembly Interface**      none

**Description**

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

```
`interrupt disable`  
  
val = *dst;  
  
*dst = new;  
  
`interrupt enable`  
  
return (val);
```

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

**See Also**

ATM\_setu  
ATM\_cleari

**ATM\_setu***Atomically set Uns memory and return previous value***C Interface**

**Syntax**                    uold = ATM\_setu(udst, unew);

**Parameters**                volatile Uns \*udst;    /\* pointer to unsigned \*/  
                               Uns            unew;            /\* new unsigned value \*/

**Return Value**            Uns        uold;    /\* previous value of \*udst \*/

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

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

```

`interrupt disable`

val = *dst;

*dst = new;

`interrupt enable`

return (val);

```

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

**See Also**

ATM\_clearu  
 ATM\_setu

## **C62 Module**

### *Target-specific functions for the TMS320C6000 family*

#### **Functions**

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

#### **Description**

The C62 module provides certain target-specific functions and definitions for the TMS320C6000 family of processors. See `c62.h` for the complete list of definitions for hardware flags for C. `c62.h` contains C language macros, `#defines` for various TMS320C6000 registers, and structure definitions.

`c62.h62` contains assembly language macros for saving and restoring registers in interrupt service routines.

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

**Syntax**                   oldmask = C62\_disableIER(mask); /

**Parameters**               Uns   mask;   /\* disable mask \*/

**Return Value**           Uns   oldmask; /\* actual bits cleared by disable mask \*/

**Assembly Interface**

**Syntax**                   C62\_disableIER IEMASK, REG0, REG1

**Parameters**               IEMASK           ; interrupt disable mask  
REG0               ; temporary register that can be modified  
REG1               ; temporary register that can be modified

**Return Value**           none

**Description**

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

The C version of 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

## **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

### **Assembly Interface**

<b>Syntax</b>	C62_enableIER IEMASK, REG0, REG1
<b>Parameters</b>	IEMASK ; interrupt enable mask REG0 ; temporary register that can be modified REG1 ; temporary register that can be modified
<b>Return Value</b>	none

### **Description**

C62\_disableIER and C62\_enableIER are used to 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:

```

; ASM example

.include c62.h62
...

; disable interrupts specified by IEMASK
C62_disableIER IEMASK, b0, b1

'do some critical operation'

; enable interrupts specified by IEMASK
C62_enableIER IEMASK, b0, b1

/* 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 may cause rescheduling of tasks (e.g., SEM\_post, TSK\_sleep) should be avoided within a C62\_disableIER / C62\_enableIER block since the interrupts may be disabled for an indeterminate amount of time if a task switch occurs.

Alternatively, you may disable DSP/BIOS task scheduling for this block by enclosing it with TSK\_disable / TSK\_enable. You may also use C62\_disableIER / C62\_enableIER to disable selected interrupts, allowing other interrupts to occur. However, if another ISR 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;`
    `OK to be interrupted or call TSK_sleep(), SEM_post(), etc.`
C62_enableIER(oldmask);
TSK_enable();
```

**Note:**

If you use C62\_disableIER / C62\_enableIER to disable only some interrupts, you must surround this region with TSK\_disable / TSK\_enable, to prevent an intervening ISR from causing a task 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

**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 ISR \*/  
 Int dmachan; /\* DMA channel to use for performing plug \*/

**Return Value** Void

**Assembly Interface** none

**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
```

If the IST is stored in external RAM, a DMA 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.

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 2-15.
- dmachan must be 0, 1, 2, or 3 if the IST is in internal program memory.

**See Also**

C62\_enableIER

**CLK Module***System clock manager***Functions**

- CLK\_countspms. Timer counts per millisecond
- CLK\_gethetime. Get high resolution time
- CLK\_getltime. Get low resolution time
- CLK\_getprd. Get period register value

**Description**

The CLK module provides a method for invoking functions periodically.

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

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

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

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

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

**Note: Specifying On-Chip Timer**

The Configuration Tool allows you to specify which on-chip 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, or interrupt 15 must be configured for timer 1.

Therefore, the low-resolution clock ticks at the timer interrupt rate and the clock's value is equal to the number of timer interrupts that have occurred.

You can use the `CLK_gettime` function to get the low-resolution time and the `CLK_getprd` function to get the value of the period register property.

The high-resolution time is the number of times the timer counter register has been incremented (number of instruction cycles divided by 4). Given the high CPU clock rate, the 16-bit timer counter register wraps around quite fast. The 32-bit high-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding the current value of the timer counter register. You can use the `CLK_gettime` function to get the high-resolution time and the `CLK_countspms` function to get the number of hardware timer counter register ticks per millisecond.

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

If you do not want the on-chip timer to drive the system clock, delete the CLK object named `CLK_system`.

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 1–86, for a description of the HWI dispatcher and these HWI properties. *Do 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:

- Object Memory.** The memory section that contains the CLK objects created with the Configuration Tool.
- Timer Selection.** The on-chip 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.
- Enable CLK Manager.** If checked, the on-chip timer hardware is used to drive the high- and low-resolution times and to trigger execution of CLK functions.
- Use high resolution time for internal timings.** If checked, the high-resolution timer is used to monitor internal periods; otherwise the less intrusive, low-resolution timer is used.

- ❑ **Microseconds/Int.** The number of microseconds between timer interrupts. The period register is set to a value that achieves the desired period as closely as possible.
- ❑ **Directly configure on-chip timer registers.** If checked, the period register can be directly set to the desired value. In this case, the Microseconds/Int field is computed based on the value in period register and the CPU clock speed.
- ❑ **PRD Register.** If Directly configure on-chip timer registers is checked, this value is written to the period register.

The following informational fields are also displayed for the CLK module:

- ❑ **CPU Interrupt.** Shows which HWI interrupt is used to drive the timer services. The value is changed automatically when you change the Timer Selection.
- ❑ **Instructions/Int.** The number of instruction cycles represented by the period specified above.

## CLK Object Properties

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

The following properties can be set for a clock function object:

- ❑ **comment.** Type a comment to identify this CLK object.
- ❑ **function.** The function to be executed when the timer hardware interrupt occurs. This function must be written like an HWI function; it must be written in assembly and must save and restore any registers this function modifies. However, this function may 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 functions are performed using the same periodic rate, functions that need to occur at a multiple of that rate should count the number of interrupts and perform their activities when the counter reaches the appropriate value.

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

## **CLK - Code Composer Studio Interface**

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

**CLK\_countspms** *Number of hardware timer counts per millisecond*

### C Interface

**Syntax** ncounts = CLK\_countspms();

**Parameters** Void

**Return Value** Uns ncounts;

### Assembly Interface

**Syntax** CLK\_countspms

**Preconditions** amr = 0

**Postconditions** a4

**Modifies** a4

**Reentrant** yes

### Description

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

CLK\_countspms may be used to compute an absolute length of time from the number of hardware timer counts. For example, the following returns the number of milliseconds since the 32-bit high-resolution time last wrapped back to 0:

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

### See Also

CLK\_geththime  
CLK\_getprd  
STS\_delta

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

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

**Assembly Interface**

<b>Syntax</b>	<code>CLK_gethime</code>
<b>Preconditions</b>	interrupts are disabled b14 = pointer to the start of .bss amr = 0
<b>Postconditions</b>	a4 = high-resolution time value
<b>Modifies</b>	a2, a3, a4, a5, b1, b2, b3, b4, b5
<b>Reentrant</b>	no

**Description**

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

High-resolution time is the number of times the timer counter register has been incremented. When the CLK manager is enabled in the Configuration Tool, the timer counter register is incremented every four CPU cycles.

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

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

High-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding the current value of the timer counter. Although the CLK\_gethime uses the period register value to calculate the high-resolution time, the value of the high-resolution time is independent of the actual value in the period register. This is because the timer counter register is divided by the period register value when

incrementing the low-resolution time, and the result is multiplied by the same period register value to calculate the low-resolution time.

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

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

### Example

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

### See Also

CLK\_gettime  
PRD\_getticks  
STS\_delta

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

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

**Assembly Interface**

<b>Syntax</b>	<code>CLK_gettime</code>
<b>Preconditions</b>	<code>b14 = pointer to the start of .bss</code> <code>amr = 0</code>
<b>Postconditions</b>	<code>a4 = low-resolution time value</code>
<b>Modifies</b>	<code>a4</code>
<b>Reentrant</b>	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.

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

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

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

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

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

Therefore, CLK\_gettime provides a value with greater accuracy than CLK\_gettime, but which wraps back to 0 more frequently. For example, if the chip's clock rate is 200 MHz, and you use the default period register value of 50000, the CLK\_gettime 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.

### Example

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

### See Also

CLK\_gettime  
PRD\_getticks  
STS\_delta

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

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

**Assembly Interface**

<b>Syntax</b>	CLK_getprd
<b>Preconditions</b>	amr = 0
<b>Postconditions</b>	a4
<b>Modifies</b>	a4
<b>Reentrant</b>	yes

**Description**

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

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

**See Also**

CLK\_countspms  
CLK\_gettime  
STS\_delta

**DEV Module***Device driver interface***Functions**

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

**Constants, Types, and Structures**

```

#define DEV_INPUT      0
#define DEV_OUTPUT    1

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

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

typedef struct DEV_Fxns { /* driver function table */
    Int         (*close)( DEV_Handle );
    Int         (*ctrl)( DEV_Handle, Uns, Arg );
    Int         (*idle)( DEV_Handle, Bool );
    Int         (*issue)( DEV_Handle );
    Int         (*open)( DEV_Handle, String );
    Bool        (*ready)( DEV_Handle, SEM_Handle );
    Int         (*reclaim)( DEV_Handle );

```

```
} DEV_Fxns;  
  
typedef struct DEV_Device { /* device specifier */  
    String      name;      /* device name */  
    DEV_Fxns    *fxns;     /* device function table */  
    Int         devid;     /* device ID */  
    Ptr         params;    /* device parameters */  
} DEV_Device;
```

## Description

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

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

## UDEV Manager Properties

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

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

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

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

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

## UDEV Object Properties

The following properties can be set for a user-defined device:

- comment.** Type a comment to identify this object.
- DEV\_Fxns table.** Specify the name of the device functions table contained in `dxx.c`. This table should have a name with the format `DXX_FXNS` where `XX` is the two-letter code for the driver used by this device.  
Use a leading underscore before the table name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- Parameters.** If this device uses additional parameters, provide the name of the parameter structure. This structure should have a name with the format `DXX_Params` where `XX` is the two-letter code for the driver used by this device.  
Use a leading underscore before the structure name.
- Device ID.** Specify the device ID. If the value you provide is non-zero, the value takes the place of a value that would be appended to the device name in a call to `SIO_create`. The purpose of such a value is driver-specific.
- Init Fxn.** Specify the function to run to initialize this device.  
Use a leading underscore before the function name if the function is written in C.
- Stacking Device.** Put a check mark in this box if device uses a stacking driver.

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

substr = DEV\_match(name, device);

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

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

**Assembly Interface**

none

**Description**

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

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

**See Also**

SIO\_create

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

**Syntax**                    status = Dxx\_close(device);

**Parameters**                DEV\_Handle device;     /\* device handle \*/

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

**Assembly Interface**     none

**Description**

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

SIO\_delete calls Dxx\_idle to idle the device before calling Dxx\_close.

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

**Constraints and Calling Context**

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

**See Also**

Dxx\_idle  
Dxx\_open  
SIO\_delete

## **Dxx\_ctrl**

*Device control operation*

### **C Interface**

**Syntax**                    status = Dxx\_ctrl(device, cmd, arg);

**Parameters**                DEV\_Handle device; /\* device handle \*/  
                              Uns        cmd;        /\* driver control code \*/  
                              Arg        arg;        /\* control operation argument \*/

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

**Assembly Interface**      none

### **Description**

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.

Dxx\_ctrl is called by SIO\_ctrl to send control commands to a device.

### **Constraints and Calling Context**

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

### **See Also**

SIO\_ctrl

**Dxx\_idle***Idle device***C Interface**

**Syntax**                    status = Dxx\_idle(device, flush);

**Parameters**            DEV\_Handle device;        /\* device handle \*/  
                               Bool            flush;            /\* flush output flag \*/

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

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

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 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 while returning the device to its initial state. If flush is TRUE, all pending data is discarded and Dxx\_idle does not block waiting for data to be processed. If flush is FALSE, the Dxx\_idle function does not return until all pending output data has been rendered. All pending data in an input stream is always discarded, without waiting.

**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*

**C Interface**

**Syntax**                    Dxx\_init();

**Parameters**                Void

**Return Value**                Void

**Assembly Interface**        none

**Description**

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

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

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

**Syntax**                    status = Dxx\_issue(device);

**Parameters**                DEV\_Handle device;   /\* device handle \*/

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

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

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

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

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

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

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

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

## **Constraints and Calling Context**

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

## **See Also**

`Dxx_reclaim`  
`SIO_issue`

**Dxx\_open***Open device***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 \*/

**Assembly Interface**      none

**Description**

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

The device parameter points to a DEV\_Obj whose fields have been initialized by the calling function (i.e., SIO\_create). These fields may 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 may be referenced in subsequent Dxx driver calls.

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

**See Also**

Dxx\_close  
SIO\_create

## **Dxx\_ready**

*Check if device is ready for I/O*

### **C Interface**

**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 \*/

**Assembly Interface**      none

### **Description**

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

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

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

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

### **See Also**

SIO\_select

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

**Syntax**                    status = Dxx\_reclaim(device);

**Parameters**                DEV\_Handle device;    /\* device handle \*/

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

**Assembly Interface**        none

**Description**

Dxx\_reclaim is used to request a buffer back from a device. Dxx\_reclaim does not return until a buffer is available for the client in the device->fromdevice queue. If the device was opened in DEV\_INPUT mode then Dxx\_reclaim blocks until an input frame has been filled with the number of bytes 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

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

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

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

**Configuring a DAX Device**

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

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

**DAX\_Params Elements**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Data Streaming

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

Tasks block when calling `SIO_get` if a full buffer is not available from DAX. Tasks block when calling `SIO_put` if an empty buffer is not available from DAX.

## Example

The following example declares `DAX_PRMS` as a `DAX_Params` structure:

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

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

**DGN Driver***Software generator driver***Comments**

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 add a DGN device, right-click on the DGN - Software Generator Driver icon and select Insert DGN. From the Object menu, choose Rename and type a new name for the DGN device. Open the Properties dialog for the device you created and modify its properties.

**Data Streaming**

DGN generator devices can be opened for input data streaming only; generators cannot be used as output devices.

The DGN driver places no inherent restrictions on the size or memory section 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.

**Generating Sine Values**

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

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

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.

## 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:

- comment.** Type a comment to identify this object.
- Device category.** The device category (user, sine, random, constant, printHex, or printInt) determines the type of data stream produced by the device. A sine, random, or constant device can be opened for input data streaming only. A printHex or printInt device can be opened for output data streaming only.
  - user.** Uses a custom function to produce or 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
- Use default parameters.** Check this box if you want to use the default parameters shown in this dialog for the Device category you selected.
- Device ID.** This field is set automatically when you select a Device category.
- Constant value.** The constant value to be generated if the Device category is constant.
- Seed value.** The initial seed value used by an internal pseudo-random number generator if the Device category is random. Used to produce a uniformly distributed sequence of numbers ranging between Lower limit and Upper limit.
- Lower limit.** The lowest value to be generated if the Device category is random.
- Upper limit.** The highest value to be generated if the Device category is random.
- Gain.** The amplitude scaling factor of the generated sine wave if the Device category is sine. The scaling factor is applied to each data point.
- Frequency.** The frequency of the generated sine wave (in cycles per second) if the Device category is sine.

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

A user function must have the following form:

```
fxn(arg, buf, nbytes)
```

where `buf` is a buffer of length `nbytes` that contain the values generated or to be processed. `buf` and `nbytes` correspond to the buffer address and buffer size, respectively, for an `SIO_get` operation.

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

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

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

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

**Configuring a DGS Device**

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

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

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

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

The device parameters are:

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

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

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

- ❑ **arg:** Optional argument, default is 0. If the create function is non-NULL, the arg parameter is passed to the create function and the create function's return value is passed as a parameter to the transform function; otherwise, argument is passed to the transform function.
- ❑ **num** (numerator) and **den** (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  $(\text{bufsize} * \text{numerator}) / \text{denominator}$ . For output devices, the size of these buffers is  $(\text{bufsize} * \text{denominator}) / \text{numerator}$ . Data is transformed into or out of these buffers before or after calling the underlying device's output or input functions respectively.

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

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

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

## Example

The following example declares `DGS_PRMS` as a `DGS_Params` structure:

```
#include <dgs.h>

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

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

## See Also

DTR

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

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

**Configuring a DHL Device**

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

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

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

**Data Streaming**

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

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

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

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

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

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

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

## Constraints

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

## DHL Driver Properties

The following global property can be set for the DHL - Host Link Driver:

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

## DHL Object Properties

The following properties can be set for a DHL device:

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

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

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

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

**Configuring a DNL Device**

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

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

**Data Streaming**

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

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

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

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

The DOV driver manages a class of stackable devices that generate an overlapped stream by retaining the last N MAUs of each buffer input from an underlying device. These N points become the first N points of the next input buffer.

**Configuring a DOV Device**

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

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

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

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

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

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

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

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

This causes SIO to open a stack of two devices. `/overlap` designates the device called `overlap`, which you have configured to use the last 16 MAUs of

the previous frame as the first 16 MAUs of the next frame. As in the previous example, codec specifies the name of the physical device that corresponds to the actual source for the data.

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

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

```
#include <dov.h>

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

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

## Data Streaming

DOV devices may only be opened for input.

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

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

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

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

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

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

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

## See Also

DTR  
DGS

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

The DPI driver is a software device used to stream data between tasks on a single processor. It provides a mechanism similar to that of UNIX named pipes; a reader and a writer task may 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 may be stacked on top of DPI. DPI can have only one reader and one writer task.

It is possible to delete one end of a pipe with SIO\_delete and recreate that end with SIO\_create without deleting the other end.

**Configuring a DPI Device**

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

**Data Streaming**

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

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

And in the function for the other task:

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

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

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

and from the writer task call:

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

The DPI driver places no inherent restrictions on the size or memory sections 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` may 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 the buffers from a stream in the same order as they were previously issued. To preserve this mechanism of exchanging buffers with the stream, the default implementation of the DPI driver for `ISSUERECLAIM` copies the full buffers issued by the writer to the empty buffers issued by the reader.

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

```
#define COPYBUFS
```

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

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

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

---

## Converting a Single-Processor Application to a Multiprocessor Application

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

### Constraints

- Only one reader and one writer may 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:

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

**DST Driver***Stackable “split” driver***Description**

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

**Configuring a DST Device**

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

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

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

**Example 1:**

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

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

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

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

Type 4/codec as the Device Control Parameter. Select split from the Device list.

#### Example 2:

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

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

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

#### Example 3:

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

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

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

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

## Data Streaming

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

## Constraints

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

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

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

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

**Configuring a DTR Device**

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

- DEV\_FXNS table:** Type `_DTR_FXNS`
- Parameters:** Enter the name of a `DTR_Params` structure declared in your C application code. See the information following this list for details.

- Device ID:** Type 0 (zero) or `_DTR_multiply`.

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

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

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

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

The `DTR_Params` structure is defined in `dtr.h` as follows:

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

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

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

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

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

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

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

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

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

## Data Streaming

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

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

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

**Global Settings***Global settings manager***Functions**

None

**Description**

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

**Global Settings Properties**

The following Global Settings can be made:

- Target Board Name.** The type of board on which your target chip is mounted.
- DSP MIPS (CLKOUT).** This number, times 1000000, is the number of instructions the processor can execute in 1 second. This value is used by the CLK manager to calculate register settings for the on-chip timers.
- DSP Type.** The target CPU type. If you are using a custom board, you can type a value in this field. Type the number after the C in the chip model. For example, type 62 for a 'C6000 chip.
- 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 field must match the setting in the DSP's CSR register.
- C Autoinitialization Model.** Select the run-time initialization model.
- Call user init function.** Put a checkmark in this box if you want an initialization function to be called early during program initialization—after .cinit processing and before the main function. This function can perform special hardware setup. The code in this function should not use any DSP/BIOS API calls.
- User init function.** Type the name of the initialization function.
- Enable Real Time Analysis.** Remove the checkmark from this box if you want to remove support for DSP/BIOS implicit instrumentation from the program. This optimizes a program by reducing code size, but removes support for the DSP/BIOS plug-ins and the LOG, STS, and TRC module APIs.
- Program Cache Control.** This field specifies the cache mode for the DSP at program initiation.
- Configure L2 Cache Control (c6x11 support).** You may check this box for DSPs that have a L1/L2 cache (for example, the c6211). The following three fields are available if this box is checked.

- ❑ **L2 Mode - CCFG(L2MODE)**. Sets the L2 cache mode. See the *'C6000 Peripherals Manual* for details.
- ❑ **L2 Requestor Priority - CCFG(P)**. Specifies the CPU/DMA cache priority. See the *'C6000 Peripherals Manual* for details.
- ❑ **L2 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.

**HST Module***Host Channel manager***Functions**

- ❑ `HST_getpipe`. Get corresponding pipe object

**Description**

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

**Note:**

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

Each host channel is internally implemented using a data pipe (PIP) object. To use a particular host channel, the program uses `HST_getpipe` to get the corresponding pipe object and then transfers data by calling the `PIP_get` and `PIP_free` operations (for input) or `PIP_alloc` and `PIP_put` operations (for output).

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

The notify function is called from the context of the code that calls `PIP_free` or `PIP_put`. This function may be written in C or assembly. The code that calls `PIP_free` or `PIP_put` should preserve any necessary registers.

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

**HST Manager Properties**

The following global properties can be set for the HST module:

- ❑ **Object Memory.** The memory section that contains the HST objects
- ❑ **Host Link Type.** The underlying physical link to be used for host-target data transfer

Shared memory properties are also shown for the HST module when the Host Link Type is set to Shared Memory. The Shared Memory option is not available for most cards. These properties are:

- ❑ **Shared Memory Segment.** This segment stores the host transfer buffers.
- ❑ **Shared Memory Frame Size (MAUs).** This specifies the size of the transfer buffer.

## 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 may put a variable amount of data in each frame.

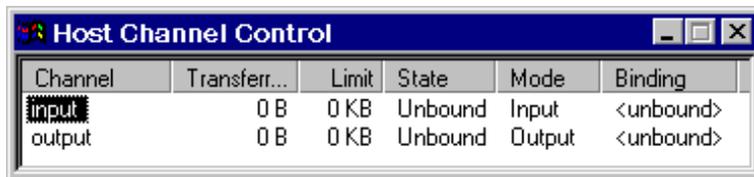
The following properties can be set for a host file object:

- comment** Type a comment to identify this HST object.
- mode** The type of channel: input or output. Input channels are used by the target to read data from the host; output channels are used by the target to transfer data from the target to the host.
- bufseg** The memory section from which the buffer is allocated; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- bufalign** The alignment (in words) of the buffer allocated within the specified memory section.
- framesize** The length of each frame (in words)
- numframes** The number of frames
- statistics** Check this box if you want to monitor this channel with an STS object. You can display the STS object for this channel to see a count of the number of frames transferred with the Statistics View plug-in.
- notify** The function to execute when a frame of data for an input channel (or free space for an output channel) is available. To avoid problems with recursion, this function should not directly call any of the PIP module functions for this HST object.
- arg0, arg1** Two 32-bit arguments passed to the notify function. They can be either unsigned 32-bit constants or symbolic labels.
- Make this channel available for a new DHL device** Check this box if you want to use this HST object with a DHL device. DHL devices allow you to manage data I/O between the host and target using the SIO module, rather than the PIP module. See the DHL driver topic for more details.

## HST - Host Channel Control Interface

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

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



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

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

**Syntax** pipe = HST\_getpipe(hst);

**Parameters** Standee hst /\* host object handle \*/

**Return Value** PIP\_Handle pipe /\* pipe object handle\*/

**Assembly Interface**

**Syntax** HST\_getpipe

**Preconditions** a4 = HST channel object pointer  
amr = 0

**Postconditions** a4 = address of the pipe object

**Modifies** a4

**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  
PIP\_free  
PIP\_get  
PIP\_put

**HWI Module***Hardware interrupt manager***Functions**

- HWI\_disable. Disable hardware interrupts
- HWI\_enable. Enable hardware interrupts
- HWI\_enter. Hardware ISR prolog
- HWI\_exit. Hardware ISR epilog
- HWI\_restore. Restore hardware interrupt state

**Description**

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

You can also dynamically assign routines to interrupts at run-time using C62\_dispatchPlug function.

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

The HWI dispatcher is the preferred method for handling an interrupt. When 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 could post or affect a software interrupt or semaphore, and the HWI\_exit assembly macro must be called at the very end of the function's code. When an HWI object is configured to use the dispatcher, the dispatcher handles the HWI\_enter prolog and the HWI\_exit epilog, and the HWI function can be completely written in C. It would, in fact, cause a system crash were the dispatcher to call a function that contains the HWI\_enter/HWI\_exit macro pair. Using the dispatcher allows you to save code space by including only one instance of the HWI\_enter/HWI\_exit code.

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 application stack—is used for the duration of the ISR. This same stack is also used by all SWI routines.

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

**Note:**

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

**Note:**

You must use HWI\_disable and HWI\_enable to bracket a block of code that atomically makes DSP/BIOS API calls.

**Note:**

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

**Note:**

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

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

- SWI\_andn
- SWI\_dec
- SWI\_inc
- SWI\_or
- SWI\_post
- PIP\_alloc
- PIP\_free
- PIP\_get
- PIP\_put
- PRD\_tick
- SEM\_post

**Note:**

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

**Note:**

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

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

The mask argument to HWI\_enter and HWI\_exit allows you to save and restore registers used within the function.

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

**Note:**

By using HWI\_enter and HWI\_exit as an HWI function's prolog and epilog, an HWI function can be interrupted; i.e., 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; e.g., enabling and disabling the execution of hardware interrupts.

The following global properties can be set for the HWI module:

- Function Stub Memory.** Select the memory section where the dispatch code should be placed for interrupt service routines that are configured to be monitored.
- Interrupt Service Table Memory.** Select the memory section where the Interrupt Service Table (IST) should be placed. IST can be placed

anywhere on the memory map, but a copy of the RESET vector always remains at address 0x00000000.

- 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.

## HWI Object Properties

The following properties can be set for a hardware interrupt service routine object:

- comment** A comment is provided to identify each HWI object.
- interrupt source.** Select the pin, DMA channel, timer, or other source of the interrupt.
- 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 could post or affect a software interrupt. HWI functions can post software interrupts, but they do not run until your HWI function (or the dispatcher) calls the HWI\_exit assembly macro, which must be the last statement in any HWI function that calls HWI\_enter.
- monitor** If set to anything other than Nothing, an STS object is created for this ISR that is passed the specified value on every invocation of the interrupt service routine. The STS update occurs just before entering the ISR.
- addr** If the monitor field above is set to Data Address, this field lets you specify a data memory address to be read; the word-sized value is read and passed to the STS object associated with this HWI object.
- type.** The type of the value to be monitored: unsigned or signed. Signed quantities are sign extended when loaded into the accumulator; unsigned quantities are treated as word-sized positive values.
- operation.** The operation to be performed on the value monitored. You can choose one of several STS operations.
- Use Dispatcher.** A check box that controls whether the HWI dispatcher is used.
- Arg.** This argument is passed to the function as its only parameter. You may use either a literal integer or a symbol defined by the application. This property is available only when using the HWI dispatcher.

- ❑ **Interrupt Mask.** A drop-down menu that specifies which interrupts the dispatcher should disable before calling the function. This property is available only when using the HWI dispatcher.
- ❑ **Interrupt Bit Mask.** An integer field that is writable when the interrupt mask is set as bitmask. This should be a hexadecimal integer bitmask specifying the interrupts to disable.
- ❑ **Don't modify cache control.** 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.
- ❑ **Program Cache Control Mask.** A drop-down menu that becomes writable when the "don't modify cache control" box is not checked. The choices (mapped, cache enable, cache bypass, cache freeze) are the same choices available from the GBL properties.
- ❑ **Data Cache Control Mask.** A drop-down menu that becomes writable when the "don't modify cache control" box is not checked. The choices (mapped, cache enable, cache bypass, cache freeze) are the same choices available from the "program cache control mask" menu.

Although it is not possible to create new HWI objects, most interrupts supported by the chip architecture have a precreated HWI object. Your application may 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.

The following table lists, in priority order (highest to lowest), these precreated objects and their default interrupt sources. The HWI object names are the same as the interrupt names.

HWI interrupts for the TMS320C6000:

Name	Default Interrupt Source
HWI_RESET	Reset
HWI_NMI	NMI
HWI_INT4	INT4

---

<b>Name</b>	<b>Default Interrupt Source</b>
HWI_INT5	INT5
HWI_INT6	INT6
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 - Execution Graph Interface**

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

In addition, if you set the HWI object properties to perform any STS operations on a register, address, or pointer, you can track time spent performing HWI functions in the Statistics View window, which you can open by choosing Tools→DSP/BIOS→Statistics View.

**HWI\_disable** *Disable hardware interrupts***C Interface****Syntax**                   oldCSR = HWI\_disable();**Parameters**               Void**Return Value**             Uns oldCSR;**Assembly Interface****Syntax**                   HWI\_disable**Preconditions**           amr = 0**Postconditions**         GIE = 0  
a4 = CSR when HWI\_disable was invoked**Modifies**                a4, b0, csr**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\_enable to reenables hardware interrupts.

Interrupts that occur while interrupts are disabled are postponed until interrupts are reenables. 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 reenables.

**Example**

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

**See Also**

HWI\_enable  
SWI\_disable  
SWI\_enable

**HWI\_enable***Enable interrupts***C Interface****Syntax** HWI\_enable();**Parameters** Void**Return Value** Void**Assembly Interface****Syntax** HWI\_enable**Preconditions** amr = 0**Postconditions** GIE = 1**Modifies** b0, csr**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.

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.

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

**Example**

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

**See Also**

HWI\_disable  
SWI\_disable  
SWI\_enable

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

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

**Assembly Interface**

<b>Syntax</b>	HWI_enter ABMASK CMASK IEMASK CCMASK
<b>Preconditions</b>	interrupts are globally disabled (i.e., GIE == 0)
<b>Postconditions</b>	amr = 0
<b>Modifies</b>	a0, a1, amr, b0, b1, b2, b3, b14, b15, csr, ier
<b>Reentrant</b>	yes

**Description**

HWI\_enter is an API (assembly macro) used to save the appropriate context for a DSP/BIOS interrupt service routine (ISR).

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

If the HWI dispatcher is not used by an HWI object, HWI\_enter must be used in the ISR before any DSP/BIOS API calls that could trigger a software interrupt; e.g., SWI\_post. HWI\_enter is used in tandem with HWI\_exit to ensure that the DSP/BIOS SWI manager is called at the appropriate time. Normally, HWI\_enter and HWI\_exit must surround all statements in any DSP/BIOS assembly language ISRs.

Two common masks, C62\_ABTEMPS and C62\_CTEMPS, are defined in c62.h62. These masks specify the C temporary registers and should be used when saving the context for an ISR that is written in C.

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 ISR. Some typical values for this mask are defined in c62.h62. The PCC code and DCC code can be ORed together (e.g., C62\_PCC\_ENABLE | C62\_PCC\_DISABLE) to generate CCMASK. If you use 0 as CCMASK, C62\_CCDEFAULT is used. C62\_CCDEFAULT is defined in c62.h62 as C62\_PCC\_DISABLE | C62\_PCC\_DISABLE. You set this value in the Global Settings Properties in the Configuration Tool.

The following parameters and constants are available for HWI\_enter:

- ❑ **ABMASK.** Register mask specifying A, B registers to save
  - **C62\_ABTEMPS.** Mask to use if calling C function from within ISR; defined in c62.h62
  - **C62\_A0-C62\_A15, C62\_B0-C62\_B15.** Individual register constants; can be ORed together for more precise control than using C62\_ABTEMPS
- ❑ **CMASK.** Register mask specifying control registers to save
  - **C62\_CTEMPS.** Mask to use if calling C function from within ISR; defined in c62.h62
  - **C62\_AMR, C62\_CSR, C62\_IER, C62\_IST, C62\_IRP, C62\_NRP.** Individual register constants; can be ORed together for more precise control than using C62\_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.
  - **C62\_NMIE, C62\_IE4, C62\_IE5-C62\_IE15.** These convenience macros can be ORed together to specify the mask of interrupts to disable
- ❑ **CCMASK.** Bit mask specifying cache control bits in CSR
  - **C62\_PCC\_DISABLE, C62\_PCC\_ENABLE, C62\_PCC\_FREEZE, C62\_PCC\_BYPASS.** These macros directly correspond to the possible modes of the program cache specified in the CSR

### Constraints and Calling Context

- ❑ This API should not be used for the NMI HWI function.
- ❑ This API must not be called if the HWI object that runs this function uses the HWI dispatcher.
- ❑ Unless the HWI dispatcher is used, this API must be called within any hardware interrupt function (except NMI's HWI function) before the first operation in an ISR that uses any DSP/BIOS API calls that might post or affect a software interrupt or semaphore. Such functions must be written in assembly language. Alternatively, the HWI dispatcher can be used instead of this API, allowing the function to be written completely in C and allowing you to reduce code size.
- ❑ If an interrupt function calls HWI\_enter, it must end by calling HWI\_exit.

**Example**

CLK\_isr:

```
HWI_enter C62_ABTEMPS, C62_CTEMPS, 0XF0,  
C62_PCC_ENABLE|C62_PCC_DISABLE  
PRD_tick  
HWI_exit C62_ABTEMPS, C62_CTEMPS, 0XF0,  
C62_PCC_ENABLE|C62_PCC_DISABLE
```

**See Also**

HWI\_exit

**HWI\_exit***Hardware ISR epilog***C Interface****Syntax** none**Parameters** none**Return Value** none**Assembly Interface****Syntax** HWI\_exit ABMASK CMASK IERRESTOREMASK CCMASK**Preconditions** b14 = pointer to the start of .bss  
amr = 0**Postconditions** none**Modifies** a0, a1, amr, b0, b1, b2, b3, b14, b15, csr, ier, irp**Reentrant** yes**Description**

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

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

If the HWI dispatcher is not used by an HWI object, HWI\_exit must be the last statement in an ISR that uses DSP/BIOS API calls which could trigger a software interrupt; e.g., SWI\_post.

HWI\_exit restores the registers specified by ABMASK and CMASK. ABMASK and CMASK 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 ISRs that call C functions only for ISRs that are not dispatched by the HWI dispatcher.

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

Of the interrupts in IERRESTOREMASK, HWI\_exit only restores those that were enabled upon entering the ISR. HWI\_exit does not affect the status of interrupt bits that are not in IERRESTOREMASK.

If upon exiting an ISR you do not wish to restore one of the interrupts that were disabled with HWI\_enter, do not set that interrupt bit in the IERRESTOREMASK in HWI\_exit.

If upon exiting an ISR you do wish to enable an interrupt that was disabled upon entering the ISR, set the corresponding bit in IERRESTOREMASK before calling HWI\_exit. (Including the interrupt IER bit in the IERRESTOREMASK of HWI\_exit does not have the effect of enabling the interrupt if it was disabled when the ISR was entered.)

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 CSR to its value at the interrupted context.

The following parameters and constants are available for HWI\_exit:

- ❑ **ABMASK.** Register mask specifying A, B registers to restore
  - **C62\_ABTEMPS.** Mask to use if calling C function from within ISR; defined in c62.h62
  - **C62\_A0-C62\_A15, C62\_B0-C62\_B15.** Individual register constants; can be ORed together for more precise control than using C62\_ABTEMPS
- ❑ **CMASK.** Register mask specifying control registers to restore
  - **C62\_CTEMPS.** Mask to use if calling C function from within ISR; defined in c62.h62
  - **C62\_AMR, C62\_CSR, C62\_IER, C62\_IST, C62\_IRP, C62\_NRP.** Individual register constants; can be ORed together for more precise control than using C62\_CTEMPS
- ❑ **IEMASK.** Bit mask specifying IER bits to restore. Any bit mask can be specified, with bits having a one-to-one correspondence with the assigned values in the IER.
  - **C62\_NMIE, C62\_IE4, C62\_IE5-C62\_IE15.** These convenience macros can be ORed together to specify the mask of interrupts to restore
- ❑ **CCMASK.** Bit mask specifying cache control bits in CSR
  - **C62\_PCC\_DISABLE, C62\_PCC\_ENABLE, C62\_PCC\_FREEZE, C62\_PCC\_BYPASS.** These macros directly correspond to the possible modes of the program cache specified in the CSR

## Constraints and Calling Context

- ❑ This API should not be used for the NMI HWI function.
- ❑ This API must not be called if the HWI object that runs this function uses the HWI dispatcher.
- ❑ Unless the HWI dispatcher is used, this API must be the last operation in an ISR that uses any DSP/BIOS API calls that might post or affect a software interrupt or semaphore. 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.
- ❑ The ABMASK and CMASK parameters must match the corresponding parameters used for HWI\_enter.

## Example

CLK\_isr:

```
HWI_enter C62_ABTEMPS, C62_CTEMPS, 0XF0,  
C62_PCC_ENABLE|C62_PCC_DISABLE  
PRD_tick  
HWI_exit C62_ABTEMPS, C62_CTEMPS, 0XF0,  
C62_PCC_ENABLE|C62_PCC_DISABLE
```

## See Also

HWI\_enter

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

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

**Assembly Interface**

<b>Syntax</b>	HWI_restore
<b>Preconditions</b>	a4 = mask (GIE is set to the value of bit 0) GIE = 0 amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, b0, csr
<b>Reentrant</b>	no

**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.

**Constraints and Calling Context**

- ❑ HWI\_restore cannot be called from an ISR context.

**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

**IDL Module***Event Log manager***Functions**

- IDL\_run. Make one pass through idle functions

**Description**

The IDL module manages the lowest-level threads in the application. In addition to user-configured 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 software interrupts can run. After the return, DSP/BIOS runs the idle loop. Once an application is in this loop, hardware ISRs, SWI software interrupts, PRD periodic functions, TSK task functions, and IDL background threads are all enabled.

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

An application always has an IDL\_cpuLoad object, which runs a function that provides data about the CPU utilization of the application. In addition, the LNK\_dataPump function handles host I/O in the background, and the RTA\_dispatch function handles run-time analysis communication.

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

**IDL Manager Properties**

The following global properties can be set for the IDL module:

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

removing the checkmark from the Include in CPU load calibration box in the Properties dialog for an individual IDL object.

If this box is checked, it is important that no IDL functions are included in the calculation block on this first pass, otherwise your program will never get to main. Also, remember that functions included in the calibration are run before the main function returns. These functions should not access data structures that are not initialized before the main function runs. In particular, do not include functions that perform any of the following actions in functions included in the idle loop calibration:

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

## IDL Object Properties

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

The following properties can be set for an IDL object:

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

## IDL- Execution Graph Interface

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

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

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

**LCK Module***Resource lock manager***Functions**

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

**Constants, Types, and Structures**

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

/* lock object */
typedef struct LCK_Attrs LCK_Attrs;

struct LCK_Attrs {
    Int dummy;
};

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

**Description**

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

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

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

**LCK Manager Properties**

The following global property can be set for the LCK module:

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

**LCK Object Properties**

The following property can be set for a LCK object:

- ❑ **comment.** Type a comment to identify this 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 \*/

**Assembly Interface**    none

**Description**

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

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

**Note:**

At present, no attributes are supported for lock objects.

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

**Constraints and Calling Context**

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

**See Also**

LCK\_delete  
LCK\_pend  
LCK\_post

**LCK\_delete***Delete a resource lock***C Interface**

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

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

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

**Constraints and Calling Context**

- LCK\_delete cannot be called by ISRs.
- No task should be awaiting ownership of the lock.
- No check is performed to prevent LCK\_delete from being used on a statically-created object. If a program attempts to delete a lock object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

LCK\_create  
LCK\_post  
LCK\_pend

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

**Assembly Interface**

none

**Description**

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

The task owning lock may 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 returns TRUE if it successfully acquires ownership of lock, returns FALSE if timeout.

**Constraints and Calling Context**

- lock must be a handle for a resource lock object created through a prior call to LCK\_create.
- LCK\_pend cannot be called by an ISR.

**See Also**LCK\_create  
LCK\_delete  
LCK\_post

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

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

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

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

**Constraints and Calling Context**

- lock must be a handle for a resource lock object created through a prior call to LCK\_create.
- LCK\_post cannot be called by an ISR.

**See Also**

LCK\_create  
LCK\_delete  
LCK\_pend

**LOG Module***Capture events in real time***Functions**

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

**Description**

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

The system log stores messages about system events for the types of log tracing you have enabled. See the TRC Module, page 1–290, 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. Calls that access LOG objects return in less than 2 microseconds.

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 *TMS320C6000 Code Composer Studio Tutorial* for examples of how to use the LOG Manager.

## LOG Manager Properties

The following global property can be set for the LOG module:

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

## LOG Object Properties

The following properties can be set for a log object:

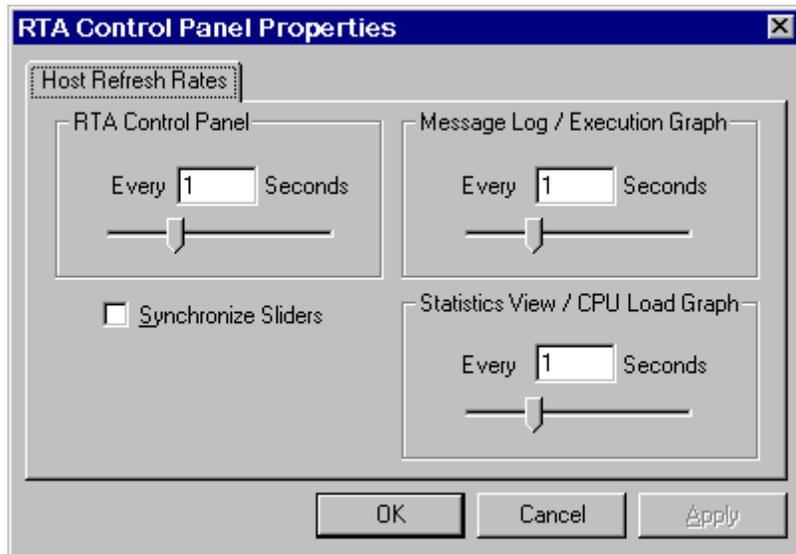
- comment.** Type a comment to identify this LOG object.
- bufseg.** The name of a memory section to contain the log buffer.
- buflen.** The length of the log buffer (in words).
- logtype.** The type of the log: circular or fixed. Events added to a full circular log overwrite the oldest event in the buffer, whereas events added to a full fixed log are dropped.
  - Fixed.** The log stores the first messages it receives and stops accepting messages when its message buffer is full.
  - Circular.** The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.
- datatype.** Choose `printf` if you use `LOG_printf` to write to this log and provide a format string.  
Choose raw data if you want to use `LOG_event` to write to this log and have the Event Log apply a `printf`-style format string to all records in the log.
- format.** If you choose raw data as the datatype, type a `printf`-style format string in this field. Provide up to three (3) conversion characters (such as `%d`) to format words two, three, and four in all records in the log. Do not put quotes around the format string. The format string can use `%d`, `%x`, `%o`, `%s`, and `%r` conversion characters; it cannot use other types of conversion characters.  
See `LOG_printf`, page 1–121, and `LOG_event`, page 1–119, for information about the structure of a log record.

## LOG - Code Composer Studio Interface

You can view log messages in real time while your program is running with the Event Log. To see the system log as a graph, choose `Tools`→`DSP/BIOS`→`Execution Graph`. To see a user log, choose `Tools`→`DSP/BIOS`→`Event Log` and select the log or logs you want to see.

You can also control how frequently the host polls the target for log information. Right-click on the RTA Control Panel and choose the Property Page to set the refresh rate. If you set the refresh rate to 0, the host does not

poll the target unless you right-click on the log window and choose Refresh Window from the pop-up menu.



**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

**Assembly Interface**

<b>Syntax</b>	LOG_disable
<b>Preconditions</b>	a4 = address of the LOG object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0
<b>Reentrant</b>	no

**Description**

LOG\_disable disables the logging mechanism and prevents the log buffer from being modified.

**Example**

```
LOG_disable(&trace);
```

**See Also**

LOG\_enable  
LOG\_reset

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

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

**Assembly Interface**

<b>Syntax</b>	LOG_enable
<b>Preconditions</b>	a4 = address of the LOG object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0
<b>Reentrant</b>	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**

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

**Assembly Interface**

<b>Syntax</b>	LOG_error format [section]
<b>Preconditions</b>	a4 = format b4 = arg0 b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	a0, a1, a2, a3, a4, a6, a7, b0, b2, b3, b5, b6, b7
<b>Reentrant</b>	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 passed to LOG\_error may contain any of the conversion characters supported for LOG\_printf. See LOG\_printf, page 1–121, for details.

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

### Example

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

### See Also

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

**LOG\_message***Write a program-supplied message to the system log***C Interface**

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

**Assembly Interface**

<b>Syntax</b>	LOG_message format [section]
<b>Preconditions</b>	a4 = format b4 = arg0 b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	none (see the description of the section argument below)
<b>Modifies</b>	a0, a1, a2, a3, a4, a6, a7, b0, b2, b3, b5, b6, b7
<b>Reentrant</b>	yes

**Description**

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

The format argument passed to LOG\_message may contain any of the conversion characters supported for LOG\_printf. See LOG\_printf, page 1–121, for details.

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

### Example

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

### See Also

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

**LOG\_event***Append an unformatted message to a message log***C Interface**

**Syntax** LOG\_event(log, arg0, arg1, arg2);

**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 \*/

**Return Value** Void

**Assembly Interface**

**Syntax** LOG\_event

**Preconditions** a4 = address of the LOG object  
 b4 = val1  
 a6 = val2  
 b6 = val3  
 amr = 0

**Postconditions** none

**Modifies** a0, a1, a2, a3, a7, b0, b2, b3, b5, b7

**Reentrant** yes

**Description**

LOG\_event copies a sequence number and three arguments to the specified log buffer. Each log message uses four words. The contents of these four words written by LOG\_event are shown here:

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

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

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

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

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

**Example**

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

**See Also**

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

**LOG\_printf***Append a formatted message to a message log***C Interface**

<b>Syntax</b>	LOG_printf(log, format); or LOG_printf(log, format, arg0); or LOG_printf(log, format, arg0, arg1);
<b>Parameters</b>	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 */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	LOG_printf format [section]
<b>Preconditions</b>	a4 = address of the LOG object b4 = arg0 a6 = arg1 amr = 0
<b>Postconditions</b>	none (see the description of the section parameter below)
<b>Modifies</b>	a0, a1, a2, a3, a7, b0, b2, b3, b5, b6, b7
<b>Reentrant</b>	yes

**Description**

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

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

The arguments passed to LOG\_printf must be integers, strings, or a pointer if the special %r conversion character is used. The format string can use the following conversion characters:

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

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

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

Each log message uses 4 words. The contents of these four words written by LOG\_printf are shown here:

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

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

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

### Constraints and Calling Context

- ❑ LOG\_printf (even the C version) supports 0, 1, or 2 arguments after the format string.
- ❑ The format string address is put in b6 as the third value for LOG\_event.

### Example

```
LOG_printf(&trace, "hello world");
LOG_printf(&trace, "Current time: %d", (Arg)CLK_gettime());
```

### See Also

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

**LOG\_reset***Reset a message log***C Interface**

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

**Assembly Interface**

<b>Syntax</b>	LOG_reset
<b>Preconditions</b>	a4 = address of the LOG object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1
<b>Reentrant</b>	no

**Description**

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

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

**Example**

```
LOG_reset(&trace);
```

**See Also**

LOG\_disable  
LOG\_enable

**MBX Module***Mailbox manager***Functions**

- ❑ `MBX_create`. Create a mailbox
- ❑ `MBX_delete`. Delete a mailbox
- ❑ `MBX_pend`. Wait for a message from mailbox
- ❑ `MBX_post`. Post a message to mailbox

**Constants, Types, and Structures**

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

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

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

**Description**

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

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

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

**MBX Manager Properties**

The following global property can be set for the MBX module:

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

## MBX Object Properties

The following properties can be set for an MBX object:

- comment.** Type a comment to identify this MBX object.
- Message Size.** The size (in 8-bit bytes) of the messages this mailbox can contain.
- Mailbox Length.** The number of messages this mailbox can contain.
- Element memory segment.** The memory section to contain the mailbox data buffers.

## MBX Code Composer Studio Interface

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

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

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

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

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

**Assembly Interface**        `none`

**Description**

MBX\_create creates a mailbox object which is initialized to contain up to mbxlength messages of size msgsize. If successful, MBX\_create returns the handle of the new mailbox object. If unsuccessful, MBX\_create returns NULL unless it aborts (e.g., 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 may be assigned to a variable of type MBX\_Attrs prior to calling MBX\_create.

No task switch occurs when calling MBX\_create.

**Constraints and Calling Context**

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

**See Also**

MBX\_delete  
 SYS\_error

**MBX\_delete***Delete a mailbox***C Interface**

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

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

MBX\_delete frees the mailbox object referenced by mbx.

No task switch occurs when calling MBX\_delete.

**Constraints and Calling Context**

- No tasks should be pending on mbx when MBX\_delete is called.
- MBX\_delete cannot be called by ISRs.
- No check is performed to prevent MBX\_delete from being used on a statically-created object. If a program attempts to delete a mailbox object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

MBX\_create

**MBX\_pend***Wait for a message from mailbox***C Interface**

**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 \*/

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

If the mailbox is not empty, MBX\_pend copies the first message into msg and returns TRUE. Otherwise, MBX\_pend suspends the execution of the current task until MBX\_post is called or the timeout expires. The actual time of task suspension can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout is SYS\_FOREVER, the task remains suspended until MBX\_post is called on this mailbox. If timeout is 0, MBX\_pend returns immediately.

If timeout expires (or timeout is 0) before the mailbox is available, MBX\_pend returns FALSE. Otherwise MBX\_pend returns TRUE.

A task switch occurs when calling MBX\_pend if the mailbox is empty and timeout is not 0, or if a higher priority task is blocked on MBX\_post.

**Constraints and Calling Context**

- ❑ MBX\_pend may only be called from an ISR if timeout is 0.
- ❑ MBX\_pend cannot be called within a TSK\_disable / TSK\_enable block.

**See Also**

MBX\_post

**MBX\_post***Post a message to mailbox***C Interface**

<b>Syntax</b>	status = MBX_post(mbx, msg, timeout);		
<b>Parameters</b>	MBX_Handle	mbx;	/* mailbox object handle */
	Ptr	msg;	/* message pointer */
	Uns	timeout;	/* return after this many system clock ticks */
<b>Return Value</b>	Bool	status;	/* TRUE if successful, FALSE if timeout */

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

MBX\_post checks to see if there are any free message slots before copying msg into the mailbox. MBX\_post readies the first task (if any) waiting on mbx.

If the mailbox is full and timeout is SYS\_FOREVER, the task remains suspended until MBX\_pend is called on this mailbox. If timeout is 0, MBX\_post returns immediately. Otherwise, the task is suspended for timeout system clock ticks. The actual time of task suspension can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout expires (or timeout is 0) before the mailbox is available, MBX\_post returns FALSE. Otherwise MBX\_post returns TRUE.

A task switch occurs when calling MBX\_post if a higher priority task is made ready to run, or if there are no free message slots and timeout is not 0.

**Constraints and Calling Context**

- MBX\_post may only be called from an ISR if timeout is 0.
- MBX\_post cannot be called within a TSK\_disable / TSK\_enable block.

**See Also**

MBX\_pend

**MEM Module***Memory section manager***Functions**

- ❑ MEM\_alloc. Allocate from a memory section
- ❑ MEM\_calloc. Allocate and initialize to 0
- ❑ MEM\_define. Define a new memory section
- ❑ MEM\_free. Free a block of memory
- ❑ MEM\_redefine. Redefine an existing memory section
- ❑ MEM\_stat. Return the status of a memory section
- ❑ 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 section */
    Uns    length;        /* size of the section */
    Uns    space;         /* memory space */
} MEM_Segment;

typedef struct MEM_Stat {
    Uns    size;          /* original size of section */
    Uns    used;          /* number of MAUs used in section */
    Uns    length;        /* length of largest contiguous block */
} MEM_Stat;

```

**Description**

The MEM module provides a set of functions used to allocate storage from one or more disjointed sections of memory. These memory sections are specified with the Configuration Tool. (The terms "memory section" and "memory segment" are used interchangeably in the DSP/BIOS properties and documentation.)

MEM always allocates an even number of words and always aligns buffers on an even boundary. This behavior is used in MEM's implementation to insure that free buffers are always at least two words in length. Note that this behavior does not preclude you from allocating two 512 buffers from a 1K

region of on-chip memory, for example. It does, however, mean that odd allocations consume one more word than expected.

If small code size is important to your application, you can reduce code size significantly by removing the capability to dynamically allocate and free memory. To do this, put a checkmark in the No Dynamic Memory Heaps box in the Properties dialog for the MEM manager. If you remove this capability, your program cannot call any of the MEM functions or any object creation functions (such as `TSK_create`). You will need to create all objects that will be used by your program with the Configuration Tool. You can also use the Configuration Tool to create or remove the dynamic memory heap from an individual memory section.

Software modules in DSP/BIOS that allocate storage at run-time use MEM functions; DSP/BIOS does not use the standard C function `malloc`. DSP/BIOS modules use MEM to allocate storage in the section selected for that module with the Configuration Tool.

The MEM Manager property, Segment for `malloc` / `free`, is used to implement the standard C `malloc`, `free`, and `calloc` functions. These functions actually use the MEM functions (with `segid` = Segment for `malloc/free`) to allocate and free memory.

## MEM Manager Properties

The DSP/BIOS Memory Section Manager allows you to specify the memory sections required to locate the various code and data sections of a DSP/BIOS application.

The following global properties can be set for the MEM module:

- Reuse startup code space.** If this box is checked, the startup code section (`.sysinit`) can be reused after startup is complete
- Map Mode.** Select 'C6000 Memory Map 0 or Memory Map 1. Changing this property affects the base address for some pre-defined memory sections.
- Argument Buffer Size.** The size of the `.args` section. The `.args` section contains the `argc`, `argv`, and `envp` arguments to the program's main function. Code Composer loads arguments for the main function into the `.args` section. The `.args` section is parsed by the boot file.
- Argument Buffer Section (`.args`).** The memory section containing the `.args` section.
- Stack Size (MAUs).** The size of the software stack in MAUs. The upper-left corner of the Configuration Tool window shows the estimated minimum stack size required for this application (as a decimal number).

Stack size is shown as a hex value in Minimum Addressable Units (MAUs). An MAU 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.

- Stack Section (.stack).** The memory section containing the software stack. This section should be located in RAM.
- BIOS Code Section (.bios).** The memory section containing the DSP/BIOS code.
- Startup Code Section (.sysinit).** The memory section containing DSP/BIOS startup initialization code; this memory may be reused after main starts executing.
- DSP/BIOS Init Tables (.gblinit, .trcinit).** The memory section containing the DSP/BIOS global and instrumentation initialization tables.
- DSP/BIOS Kernel State (.sysdata).** The memory section containing system data about the DSP/BIOS kernel state.
- DSP/BIOS Conf Sections (.obj).** The memory section containing configuration properties that can be read by the target program.
- No Dynamic Memory Heaps.** Put a checkmark in this box to completely disable the ability to dynamically allocate memory and the ability to dynamically create and delete objects. If this box is checked, your program may not call the MEM\_alloc, MEM\_valloc, and MEM\_calloc functions or the XXX\_create function for any DSP/BIOS module. If this box is checked and the program calls one of these functions, an error occurs when the program is linked.
- Segment for DSP/BIOS objects.** The default memory section that will contain objects created at run-time with an XXX\_create function.
- Segment for malloc / free.** The memory section from which space is allocated when a program calls malloc and from which space is freed when a program calls free.
- User .cmd file for non-DSP/BIOS sections.** Put a checkmark in this box if you want to have full control over the memory used for the sections that follow. You will need to create your own linker command file that begins by including the linker command file created by the Configuration Tool. Your linker command file should then assign memory for the items normally handled by the following properties. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more details.
- Text Section (.text).** The memory section containing the executable code, string literals, and compiler-generated constants. This section may be located in ROM or RAM.
- Switch jump tables (.switch).** The memory section containing the switch statement tables. This section may be located in ROM or RAM.

- C variables (.bss).** The memory section containing global and static C variables. At boot or load time, the data in the .cinit section is copied to this section. This section should be located in RAM.
- Data Initialization Section (.cinit).** The memory section containing tables for explicitly initialized global and static variables and constants. This section may be located in ROM or RAM.
- Constant Section (.const).** The memory section containing string constants and data defined with the const C qualifier. If the C compiler is not used, this parameter is unused. This section may be located in ROM or RAM.
- Data Sections (.data, .cio, .system).** These data sections contain program data, C standard I/O buffers, and the memory heap used by the standard C functions malloc, calloc, and realloc. If the program does not use the standard C functions to allocate memory (for example, because it uses the DSP/BIOS MEM functions), the .system section is not created by the C compiler. The .system section should be located in RAM. The .data section may be located in ROM or RAM.

## MEM Object Properties

A memory section represents a contiguous length of code or data memory in the address space of the processor. The following properties can be set for MEM objects:

- comment.** Type a comment to identify this MEM object.
- base.** The address at which this memory section begins. This value is shown in hex.
- len.** The length of this memory section in words. This value is shown in hex.
- Create a heap in this memory.** If this box is checked, a heap is created in this memory section. Memory can be allocated dynamically from a heap. In order to remove the heap from a memory section, you may need to select another memory section that contains a heap for properties that dynamically allocate memory in this memory section. 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 section.
- Heap size.** The size of the heap to be created in this memory section.
- space.** Type of memory section. This is set to code for memory sections that store programs, and data for memory sections that store program data.

The predefined memory sections in a configuration file, particularly those for external memory, are dependent on the board template you select. In general, the following sections may be defined for the 'C6000:

<b>Name</b>	<b>Description</b>
IPRAM	Internal (on-chip) program memory
IDRAM	Internal (on-chip) data memory
SBSRAM	External SBSRAM on CE0
SDRAM0	External SDRAM on CE2
SDRAM1	External SDRAM on CE3

### **MEM Code Composer Studio Interface**

The MEM tab of the Kernel/Object View shows information about memory sections.

**MEM\_alloc***Allocate from a memory section***C Interface**

**Syntax**                    `addr = MEM_alloc(segid, size, align);`

**Parameters**                `Int segid; /* memory section identifier */`  
                              `Uns size; /* block size in MAUs */`  
                              `Uns align; /* block alignment */`

**Return Value**              `Void *addr; /* address of allocated block of memory */`

**Assembly Interface**        none

**Description**

MEM\_alloc allocates a contiguous block of storage from the memory section identified by segid and returns the address of this block.

The segid parameter identifies the memory section from which memory is to be allocated. This identifier may be an integer or a memory section name defined in the Configuration Tool. The files created by the Configuration Tool define each configured section name as a variable with an integer value. (The terms "memory section" and "memory segment" are used interchangeably in the DSP/BIOS properties and documentation.)

The block contains size MAUs 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.

**Constraints and Calling Context**

- segid must identify a valid memory section.
- MEM\_alloc cannot be called by ISRs.
- align must be 0, or a power of 2 (e.g., 1, 2, 4, 8).

**See Also**

MEM\_calloc  
MEM\_free  
MEM\_valloc  
SYS\_error  
C library stdlib.h

**MEM\_alloc***Allocate from a memory section and set value to 0***C Interface**

**Syntax**                    `addr = MEM_alloc(segid, size, align)`

**Parameters**                `Int segid; /* memory section identifier */`  
                               `Uns size; /* block size in MAUs */`  
                               `Uns align; /* block alignment */`

**Return Value**              `Void *addr; /* address of allocated block of memory */`

**Assembly Interface**        none

**Description**

MEM\_alloc is functionally equivalent to calling MEM\_valloc with value set to 0.

MEM\_alloc allocates a contiguous block of storage from the memory section identified by segid and returns the address of this block.

The segid parameter identifies the memory section from which memory is to be allocated. This identifier may be an integer or a memory section name defined in the Configuration Tool. The files created by the Configuration Tool define each configured section name as a variable with an integer value. (The terms "memory section" and "memory segment" are used interchangeably in the DSP/BIOS properties and documentation.)

The block contains size MAUs 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\_alloc calls SYS\_error with SYS\_EALLOC and returns MEM\_ILLEGAL.

**Constraints and Calling Context**

- segid must identify a valid memory section.
- MEM\_alloc cannot be called by ISRs.
- align must be 0, or a power of 2 (e.g., 1, 2, 4, 8).

**See Also**

MEM\_alloc  
 MEM\_free  
 MEM\_valloc  
 SYS\_error  
 C library stdlib.h

**MEM\_define***Define a new memory section***C Interface****Syntax**

segid = MEM\_define(base, length, attrs);

**Parameters**

Ptr base; /\* base address of new section \*/  
Uns length; /\* length (in MAUs) of new section \*/  
MEM\_Attrs \*attrs; /\* section attributes \*/

**Return Value**

Int segid; /\* ID of new section \*/

**Assembly Interface**

none

**Description**

MEM\_define defines a new memory section for use by the DSP/BIOS memory module, MEM.

The new section contains length MAUs starting at base. A new table entry is allocated to define the section, and the entry's index into this table is returned as the segid.

The new block should be aligned on a MEM\_HEADERSIZE boundary, and the length should be a multiple of MEM\_HEADERSIZE, otherwise the entire block is not available for allocation.

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

**Note:**

At present, no attributes are supported for sections, and the type MEM\_Attrs is defined as a dummy structure.

MEM\_define and MEM\_redefine must not be called when a task switch is possible. To guard against a task switch, these functions should be used only in the main function, and should be preceded with a call to TSK\_disable and followed with a call to TSK\_enable.

**Note:**

This function is not reentrant and must be called during single-threaded operation, i.e., during DSP/BIOS backplane initialization. It must not be called while other tasks exist in the system. This function is intended to be used inside the application's init function specified in the application configuration file.

**Constraints and Calling Context**

- At least one section must exist at the time MEM\_define is called.

**See Also**

MEM\_redefine

**MEM\_free***Free a block of memory***C Interface****Syntax**                    status = MEM\_free(segid, addr, size);**Parameters**                Int    segid;    /\* memory section identifier \*/  
Ptr    addr;    /\* block address pointer \*/  
Uns    size;    /\* block length \*/**Return Value**              Bool   status;   /\* TRUE if successful \*/**Assembly Interface**        none**Description**

MEM\_free places the memory block specified by addr and size back into the free pool of the section specified by segid. This space is then available for further allocation by MEM\_alloc. The segid may be an integer or a memory section name defined in the Configuration Tool.

**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.

**See Also**

MEM\_alloc  
C library stdlib.h

**MEM\_redefine***Redefine an existing memory section***C Interface****Syntax**

MEM\_redefine(segid, base, length);

**Parameters**

Int segid; /\* section to redefine \*/  
 Ptr base; /\* base address of new block \*/  
 Uns length; /\* length (in MAUs) of new block \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

MEM\_redefine redefines an existing memory section managed by the DSP/BIOS memory module, MEM. All pointers in the old section memory block are automatically freed, and the new section block is completely available for allocations.

The new block should be aligned on a MEM\_HEADERSIZE boundary, and the length should be a multiple of MEM\_HEADERSIZE, otherwise the entire block is not available for allocation.

MEM\_define and MEM\_redefine must not be called when a task switch is possible. To guard against a task switch, these functions should be used only in the main function, and should be preceded with a call to TSK\_disable and followed with a call to TSK\_enable.

**Note:**

This function is not reentrant and must be called during single-threaded operation, i.e., during DSP/BIOS backplane initialization. It must not be called while other tasks exist in the system. This function is intended to be used inside the application's init function specified in the application configuration file.

**See Also**

MEM\_define

**MEM\_stat***Return the status of a memory section***C Interface**

**Syntax**                    status = MEM\_stat(segid, statbuf);

**Parameters**                Int    segid;    /\* memory section identifier \*/  
MEM\_Stat \*statbuf; /\* pointer to stat buffer \*/

**Return Value**              Bool    status;   /\* TRUE if successful \*/

**Assembly Interface**       none

**Description**

MEM\_stat returns the status of the memory section specified by segid in the status structure pointed to by statbuf.

```
struct MEM_Stat {  
    Uns    size;            /* original size of section */  
    Uns    used;            /* number of MAUs used in section */  
    Uns    length;          /* length of largest contiguous block */  
}
```

All values are expressed in terms of minimum addressable units (MAUs).

MEM\_stat returns TRUE if segid corresponds to a valid memory section, and FALSE otherwise. If MEM\_stat returns FALSE, the contents of statbuf are undefined.

**MEM\_valloc***Allocate from a memory section and set value***C Interface**

**Syntax**                    `addr = MEM_valloc(segid, size, align, value);`

**Parameters**                `Int segid; /* memory section identifier */`  
                               `Uns size; /* block size in MAUs */`  
                               `Uns align; /* block alignment */`  
                               `Char value; /* character value */`

**Return Value**              `Void *addr; /* address of allocated block of memory */`

**Assembly Interface**        none

**Description**

MEM\_valloc uses MEM\_alloc to allocate the memory before initializing it to value.

The segid parameter identifies the memory section from which memory is to be allocated. This identifier may be an integer or a memory section name defined in the Configuration Tool. The files created by the Configuration Tool define each configured section name as a variable with an integer value. (The terms "memory section" and "memory segment" are used interchangeably in the DSP/BIOS properties and documentation.)

The block contains size MAUs 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.

**Constraints and Calling Context**

- segid must identify a valid memory section.
- MEM\_valloc cannot be called by ISRs.
- align must be 0, or a power of 2 (e.g., 1, 2, 4, 8).

**See Also**

MEM\_alloc  
 MEM\_calloc  
 MEM\_free  
 SYS\_error  
 C library `stdlib.h`

**PIP Module***Buffered pipe manager***Functions**

- PIP\_alloc.** Get an empty frame from the pipe
- PIP\_free.** Recycle a frame back to the pipe
- PIP\_get.** Get a full frame from the pipe
- PIP\_getReaderAddr.** Get the value of the readerAddr pointer of the pipe
- PIP\_getReaderNumFrames.** Get the number of pipe frames available for reading
- PIP\_getReaderSize.** Get the number of words of data in a pipe frame
- PIP\_getWriterAddr.** Get the value of the writerAddr pointer of the pipe
- PIP\_getWriterNumFrames.** Get the number of pipe frames available to write to
- PIP\_getWriterSize.** Get the number of words that can be written to a pipe frame
- PIP\_peek.** Get the pipe frame size and address without actually claiming the pipe frame
- PIP\_put.** Put a full frame into the pipe
- PIP\_reset.** Reset all fields of a pipe object to their original values
- PIP\_setWriterSize.** Set the number of valid words written to a pipe frame

**PIP\_Obj Structure Members**

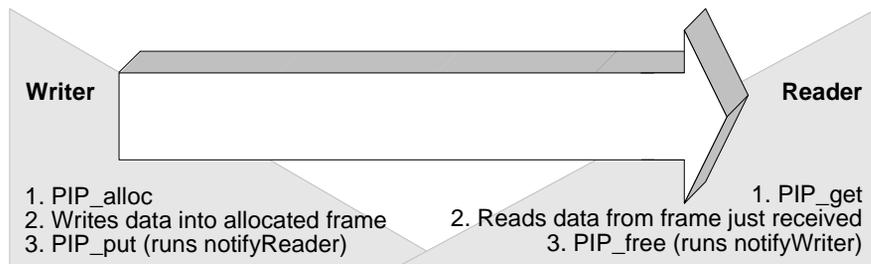
- Ptr readerAddr.** Pointer to the address to begin reading from after calling PIP\_get
- Uns readerSize.** Number of words of data in the frame read with PIP\_get
- Uns readerNumFrames.** Number of frames available to be read
- Ptr writerAddr.** Pointer to the address to begin writing to after calling PIP\_alloc
- Uns writerSize.** Number of words available in the frame allocated with PIP\_alloc
- Uns writerNumFrames.** Number of frames available to be written to

**Description**

The PIP module manages data pipes, which are used to buffer streams of input and output data. These data pipes provide a consistent software data structure you can use to drive I/O between the DSP chip 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 may put a variable amount of data in each frame up to the length of the frame.

A pipe has two ends, as shown in the following figure. 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.



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 may be written in C or assembly. To avoid problems with recursion, the `notifyReader` and `notifyWriter` functions should not directly call any of the PIP module functions for the same pipe. Instead, they should post a software interrupt that uses the PIP module functions.

**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 a hardware ISR and the other end is controlled by a SWI function.

HST objects use PIP objects internally for I/O between the host and the target. Your program only needs to act as the reader or the writer when you use an HST object, because the host controls the other end of the pipe.

Pipes can also be used to transfer data within the program between two application threads.

## PIP Manager Properties

The pipe manager manages objects that allow the efficient transfer of frames of data between a single reader and a single writer. This transfer is often between a hardware ISR and an application software interrupt, but pipes can also be used to transfer data between two application threads.

The following global property can be set for the PIP module:

- Object Memory.** The memory section that contains the PIP objects.

## PIP Object Properties

A pipe object maintains a single contiguous buffer partitioned into a fixed number of fixed length frames. All I/O operations on a pipe deal with one frame at a time; although each frame has a fixed length, the application may put a variable amount of data in each frame (up to the length of the frame).

The following properties can be set for a pipe object:

- comment.** Type a comment to identify this PIP object.
- bufseg.** The memory section that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- bufalign.** The alignment (in words) of the buffer allocated within the specified memory section.
- framesize.** The length of each frame (in words)
- numframes.** The number of frames
- monitor.** The end of the pipe to be monitored by a hidden STS object. Can be set to reader, writer, or nothing. In the Statistics View plug-in, your choice determines whether the STS display for this pipe shows a count of the number of frames handled at the reader or writer end of the pipe.
- notifyWriter.** The function to execute when a frame of free space is available. This function should notify (e.g., by calling SWI\_andn) the object that writes to this pipe that an empty frame is available.  
The notifyWriter function is performed as part of the thread that called PIP\_free or PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any of the PIP module functions for the same pipe.
- nwarg0, nwarg1.** Two 32-bit arguments passed to notifyWriter; these arguments can each be either an unsigned 32-bit constant or a symbolic label.
- notifyReader.** The function to execute when a frame of data is available. This function should notify (e.g., by calling SWI\_andn) the object that

reads from this pipe that a full frame is ready to be processed.

The notifyReader function is performed as part of the thread that called PIP\_put or PIP\_get. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.

- **nrarg0, nrarg1.** Two 32-bit arguments passed to notifyReader; these arguments can each be either an unsigned 32-bit constant or a symbolic label.

### **PIP - Code Composer Studio Interface**

To enable PIP accumulators, choose Tools→DSP/BIOS→RTA Control Panel and put a check in the appropriate box. Then choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a PIP object, you see a count of the number of frames read from or written to the pipe.

**PIP\_alloc***Allocate an empty frame from a pipe***C Interface**

<b>Syntax</b>	PIP_alloc(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PIP_alloc
<b>Preconditions</b>	a4 = address of the pipe object pipe.writerNumFrames > 0 amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9
<b>Reentrant</b>	no

**Description**

PIP\_alloc allocates an empty frame from the pipe object you specify. You can write to this frame and then use PIP\_put to put the frame into the pipe.

If empty frames are available after PIP\_alloc allocates a frame, PIP\_alloc runs the function specified by the notifyWriter property of the PIP object. This function should notify (e.g., by calling SWI\_andn) the object that writes to this pipe that an empty frame is available. The notifyWriter function is performed as part of the thread that calls PIP\_free or PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any PIP module functions for the same pipe.

**Constraints and Calling Context**

- ❑ Before calling PIP\_alloc, a function should check the writerNumFrames member of the PIP\_Obj structure by calling PIP\_getWriterNumFrames to make sure it is greater than 0 (i.e., at least one empty frame is available).
- ❑ PIP\_alloc can only be called one time before calling PIP\_put. You cannot operate on two frames from the same pipe simultaneously.

**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 1–84, also uses a pipe with host channel objects.

**See Also**

[PIP\\_free](#)  
[PIP\\_get](#)  
[PIP\\_put](#)  
[HST\\_getpipe](#)

**PIP\_free***Recycle a frame that has been read to a pipe***C Interface**

<b>Syntax</b>	PIP_free(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PIP_free
<b>Preconditions</b>	a4 = address of the pipe object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, a2, a3, a4, a5, b0, b1, b2, b3, b4
<b>Reentrant</b>	no

**Description**

PIP\_free releases a frame after you have read the frame with PIP\_get. The frame is recycled so that PIP\_alloc can reuse it.

After PIP\_free releases the frame, it runs the function specified by the notifyWriter property of the PIP object. This function should notify (e.g., by calling SWI\_andn) the object that writes to this pipe that an empty frame is available. The notifyWriter function is performed as part of the thread that called PIP\_free or PIP\_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any of the PIP module functions for the same pipe.

**Note:**

Registers used by notifyWriter functions might also be modified.

**Example**

See the example for PIP\_alloc, page 1–148. The example for HST\_getpipe, page 1–84, also uses a pipe with host channel objects.

**See Also**

PIP\_alloc  
PIP\_get  
PIP\_put  
HST\_getpipe

**PIP\_get***Get a full frame from the pipe***C Interface**

<b>Syntax</b>	PIP_get(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PIP_get
<b>Preconditions</b>	a4 = address of the pipe object pipe.readerNumFrames > 0 amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9
<b>Reentrant</b>	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 (e.g., by calling SWI\_andn) the object that reads from this pipe that a full frame is available. The notifyReader function is performed as part of the thread that calls PIP\_get or PIP\_put. To avoid problems with recursion, the notifyReader function should not directly call any PIP module functions for the same pipe.

**Constraints and Calling Context**

- ❑ Before calling PIP\_get, a function should check the readerNumFrames member of the PIP\_Obj structure by calling PIP\_getReaderNumFrames to make sure it is greater than 0 (i.e., 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 1–148. The example for `HST_getpipe`, page 1–84, also uses a pipe with host channel objects.

**See Also**

`PIP_alloc`  
`PIP_free`  
`PIP_put`  
`HST_getpipe`

**PIP\_getReaderAddr** *Get the value of the readerAddr pointer of the pipe*

### C Interface

**Syntax**                    `readerAddr = PIP_getReaderAddr(pipe);`

**Parameters**                `PIP_Handle pipe; /* pipe object handle */`

**Return Value**              `Ptr readerAddr`

### Assembly Interface

**Syntax**                    `mvk pipe, a4;`  
`mvkh pipe, a4;`  
`ldw *+a4(PIP_READPTR), a4;`  
`nop 4`

**Preconditions**            `amr = 0`

**Postconditions**           `none`

**Modifies**                  `none`

**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

```
/*
 * ===== audio =====
 */
Void audio(PIP_Obj *in, PIP_Obj *out)
{
    Uns          *src, *dst;
    Uns          size;

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

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

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

    size = PIP_getReaderSize(in);
    PIP_setWriterSize(out, size);

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

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

**PIP\_getReaderNumFrames***Get the number of pipe frames available for reading***C Interface**

**Syntax**                    num = PIP\_getReaderNumFrames(pipe);

**Parameters**                PIP\_Handle pipe; /\* pip object handle \*/

**Return Value**              Uns num; /\* number of filled frames to be read \*/

**Assembly Interface**

**Syntax**                    mvk pipe, a4;  
                               mvkh pipe, a4;  
                               ldw \*+a4(PIP\_FULLBUFS), a4;  
                               nop 4

**Preconditions**            amr = 0

**Postconditions**           none

**Modifies**                 none

**Reentrant**                yes

**Description**

PIP\_getReaderNumFrames is a C function that returns the value of the readerNumFrames element of a pipe object.

Before a function attempts to read from a pipe it should call PIP\_getReaderNumFrames to ensure at least one full frame is available.

**Example**

See the example for PIP\_getReaderAddr, page 1–153.

**PIP\_getReaderSize** *Get the number of words of data in a pipe frame*

**C Interface**

**Syntax** num = PIP\_getReaderSize(pipe);

**Parameters** PIP\_Handle pipe; /\* pipe object handle\*/

**Return Value** Uns num; /\* number of words to be read from filled frame \*/

**Assembly Interface**

**Syntax** mvk pipe, a4;  
mvkh pipe, a4;  
ldw \*+a4(PIP\_READCNT), a4;  
nop 4

**Preconditions** amr = 0

**Postconditions** none

**Modifies** none

**Reentrant** yes

**Description**

PIP\_getReaderSize is a C function that returns the value of the readerSize element of a pipe object.

As a function reads from a pipe it should use PIP\_getReaderSize to determine the number of valid words of data in the pipe frame.

**Example**

See the example for PIP\_getReaderAddr, page 1–153.

**PIP\_getWriterAddr***Get the value of the writerAddr pointer of the pipe***C Interface**

**Syntax**                   writerAddr = PIP\_getWriterAddr(pipe);

**Parameters**             PIP\_Handle pipe; /\* pipe object handle \*/

**Return Value**           Ptr writerAddr;

**Assembly Interface**

**Syntax**                   mvmk pipe, a4;  
                          mvmkh pipe, a4;  
                          ldw \*+a4(PIP\_WRITEPTR), a4;  
                          nop 4

**Preconditions**          amr = 0

**Postconditions**         none

**Modifies**                none

**Reentrant**               yes

**Description**

PIP\_getWriterAddr is a C function that returns the value of the writerAddr pointer of a pipe object.

The writerAddr pointer is normally used following a call to PIP\_alloc, as the address to begin writing to.

**Example**

See the example for PIP\_getReaderAddr, page 1–153.

**PIP\_getWriterNumFrames***Get number of pipe frames available to be written to***C Interface**

<b>Syntax</b>	num = PIP_getWriterNumFrames(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle*/
<b>Return Value</b>	Uns num; /* number of empty frames to be written */

**Assembly Interface**

<b>Syntax</b>	mvk pipe, a4; mvkh pipe, a4; ldw *+a4(PIP_EMPTYBUFS), a4; nop 4
<b>Preconditions</b>	amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	yes

**Description**

PIP\_getWriterNumFrames is a C function that returns the value of the writerNumFrames element of a pipe object.

Before a function attempts to write to a pipe, it should call PIP\_getWriterNumFrames to ensure at least one empty frame is available.

**Example**

See the example for PIP\_getReaderAddr, page 1–153.

**PIP\_getWriterSize***Get the number of words that can be written to a pipe frame***C Interface**

<b>Syntax</b>	num = PIP_getWriterSize(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle*/
<b>Return Value</b>	Uns num; /* number of words to be written in empty frame */

**Assembly Interface**

<b>Syntax</b>	mvk pipe, a4; mvkh pipe, a4; ldw *+a4(PIP_WRITECNT), a4; nop 4
<b>Preconditions</b>	amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	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 the pipe frame size and address without actually claiming the pipe frame*

**C Interface**

**Syntax** framesize = PIP\_peek(pipe, addr, rw);

**Parameters**

PIP_Handle	pipe;	/* pipe object handle */
Ptr	*addr;	/* the address of the variable that keeps the frame address */
Uns	rw;	/* the flag that indicates the reader or writer side */

**Return Value** Int framesize; /\* the frame size \*/

**Assembly Interface** none

**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***C Interface**

<b>Syntax</b>	PIP_put(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PIP_put
<b>Preconditions</b>	a4 = address of the pipe object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, b0, b1, b2, b3, b4
<b>Reentrant</b>	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 (e.g., by calling SWI\_andn) the object that reads from this pipe that a full frame is ready to be processed. The notifyReader function is performed as part of the thread that called PIP\_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.

**Example**

See the example for PIP\_alloc, page 1–148. The example for HST\_getpipe, page 1–84, also uses a pipe with host channel objects.

**See Also**

PIP\_alloc  
PIP\_free  
PIP\_get  
HST\_getpipe

**PIP\_reset***Reset all fields of a pipe object to their original values***C Interface**

<b>Syntax</b>	PIP_reset(pipe);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */
<b>Return Value</b>	Void

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

PIP\_reset resets all fields of a pipe object to their original values.

The pipe parameter specifies the address of the pipe object that is to be reset.

**Constraints and Calling Context**

- PIP\_reset should not be called between the PIP\_alloc call and the PIP\_put call or between the PIP\_get call and the PIP\_free call.
- PIP\_reset should be called when interrupts are disabled to avoid the race condition.

**See Also**

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

**PIP\_setWriterSize***Set the number of valid words written to a pipe frame***C Interface**

<b>Syntax</b>	PIP_setWriterSize(pipe, size);
<b>Parameters</b>	PIP_Handle pipe; /* pipe object handle */ Uns size; /* size to be set */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	mvk pipe, a4; mvkh pipe, a4; mvk SIZE, b4; mvkh SIZE, b4; stw b4, *+a4(PIP_WRITECNT);
<b>Preconditions</b>	amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	no

**Description**

PIP\_setWriterSize is a C function that sets the value of the writerSize element of a pipe object.

As a function writes to a pipe, it can use PIP\_setWriterSize to indicate the number of valid words being written to a pipe frame.

**Example**

See the example for PIP\_getReaderAddr, page 1–153.

**PRD Module***Periodic function manager***Functions**

- PRD\_getticks. Get the current tick count
- PRD\_start. Arm a periodic function for one-time execution
- PRD\_stop. Stop a periodic function from continuous execution
- PRD\_tick. Advance tick counter, dispatch periodic functions

**Description**

While some applications can schedule functions based on a real-time clock, many applications need to schedule functions based on I/O availability or some other programmatic event.

The PRD module allows you to create PRD objects that schedule periodic execution of program functions. The period may be driven by the CLK module or by calls to PRD\_tick whenever a specific event occurs. There can be several PRD objects, but all are driven by the same period counter. Each PRD object can execute its functions at different intervals based on the period counter.

- To schedule functions based on a real-time clock.** Set the clock interrupt rate you want to use in the Clock Manager property sheet. Put a check mark in the Use On-chip Clock (CLK) box for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object.
- To schedule functions based on I/O availability or some other event.** Remove the check mark from the Use On-chip Clock (CLK) property field for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object. Your program should call PRD\_tick to increment the tick counter.

The function executed by a PRD object is statically defined in the Configuration Tool. PRD functions are called from the context of the function run by the PRD\_swi SWI object. PRD functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

The PRD module uses an SWI object (called PRD\_swi by default) which itself is triggered on a periodic basis to manage execution of period objects. Normally, this SWI object should have the highest software interrupt priority to allow this software interrupt to be performed once per tick. This software interrupt is automatically created (or deleted) by the Configuration Tool if one or more (or no) PRD objects exist.

See the *TMS320C6000 Code Composer Studio Tutorial* for an example that demonstrates the interaction between the PRD module and the SWI module.

When the PRD\_swi object runs its function, the following actions occur:

```
for ("Loop through period objects") {
    if ("time for a periodic function")
        "run that periodic function";
}
```

## PRD Manager Properties

The DSP/BIOS Periodic Function Manager allows the creation of an arbitrary number of objects that encapsulate a function, two arguments, and a period specifying the time between successive invocations of the function. The period is expressed in ticks, where a tick is defined as a single invocation of the PRD\_tick operation. The time between successive invocations of PRD\_tick defines the period represented by a tick.

The following global properties can be set for the PRD module:

- Object Memory.** The memory section that contains the PRD objects.
- Use CLK Manager to drive PRD.** If this field is checked, the on-chip timer hardware (managed by CLK) is used to advance the tick count; otherwise, the application must invoke PRD\_tick on a periodic basis.
- Microseconds/Tick.** The number of microseconds between ticks. If the Use CLK Manager to drive PRD field above is checked, this field is automatically set by the CLK module; otherwise, you must explicitly set this field.

## PRD Object Properties

The following properties can be set for each PRD object:

- comment.** Type a comment to identify this PRD object.
- period (ticks).** The function executes after period ticks have elapsed.
- mode.** If continuous is selected the function executes every period ticks; otherwise it executes just once after each call to PRD\_tick.
- function.** The function to be executed
- arg0, arg1.** Two 32-bit arguments passed to function; these arguments can be either an unsigned 32-bit constant or a symbolic label.

The following informational property is also displayed for each PRD object:

- period (ms).** The number of milliseconds represented by the period specified above

## **PRD - Code Composer Studio Interface**

To enable PRD logging, choose Tools→DSP/BIOS→RTA Control Panel and put a check in the appropriate box. You see indicators for PRD ticks in the PRD ticks row of the Execution Graph, which you can open by choosing Tools→DSP/BIOS→Execution Graph. In addition, you see a graph of activity, including PRD function execution.

You can also enable PRD accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a PRD object, you see statistics about the number of ticks that elapsed during execution of the PRD function.

**PRD\_getticks***Get the current tick count***C Interface**

<b>Syntax</b>	num = PRD_getticks();
<b>Parameters</b>	Void
<b>Return Value</b>	LgUns num /* current tick counter */

**Assembly Interface**

<b>Syntax</b>	PRD_getticks
<b>Preconditions</b>	b14 = pointer to the start of .bss amr = 0
<b>Postconditions</b>	a4 = PRD_D_tick
<b>Modifies</b>	a4
<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-chip 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 1–31, 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\_gethetime  
CLK\_gettime  
STS\_delta

**PRD\_start**

*Arm a periodic function for one-time (or continuous) execution*

**C Interface**

<b>Syntax</b>	PRD_start(prd);
<b>Parameters</b>	PRD_Handle prd; /* prd object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PRD_start
<b>Preconditions</b>	a4 = address of the PRD object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, b1
<b>Reentrant</b>	no

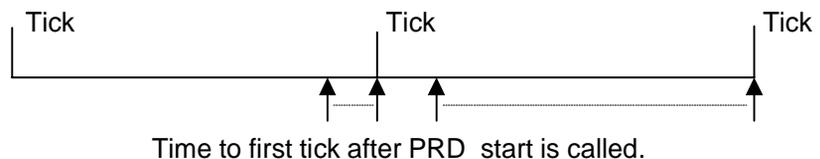
**Description**

PRD\_start starts a period object that has its mode property set to one-shot in the Configuration Tool.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified number of ticks have occurred after a call to PRD\_start.

For example, you might have a function that should be executed a certain number of periodic ticks after some condition is met.

When you use PRD\_start to start a period object, the exact time the function runs can vary by nearly one tick cycle. As this figure shows, PRD ticks occur at a fixed rate and the call to PRD\_start may occur at any point between ticks:



Due to implementation details, if a PRD function calls PRD\_start for a PRD object that is lower in the list of PRD objects, the function sometimes runs a full tick cycle early.

**Example**

```
/* ===== startClock ===== */  
Void startPrd(Int periodID)  
  {  
    if ("condition met") {  
      PRD_start(&periodID);  
    }  
  }
```

**See Also**

PRD\_tick  
PRD\_getticks

**PRD\_stop***Stop a period object to prevent its function execution***C Interface**

<b>Syntax</b>	PRD_stop(prd);
<b>Parameters</b>	PRD_Handle prd; /* prd object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	PRD_stop
<b>Preconditions</b>	a4 = address of the PRD object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, b1
<b>Reentrant</b>	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 Tool.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified 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**Assembly Interface****Syntax** PRD\_tick**Preconditions** GIE = 0 (interrupts are disabled)  
amr = 0**Postconditions** none**Modifies** a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, b0, b1, b2, b3, b4, b5, b6, b7, b8, csr**Reentrant** no**Description**

PRD\_tick advances the period counter by one tick. Unless you are driving PRD functions using the on-chip clock, PRD objects execute their functions at intervals based on this counter.

For example, a hardware ISR could perform PRD\_tick to notify a periodic function when data is available for processing.

**Constraints and Calling Context**

- ❑ This API should be invoked from interrupt service routines. All the registers that are modified by this API should be saved and restored, before and after the API is invoked, respectively.

**See Also**

PRD\_start  
PRD\_getticks

**QUE Module***Atomic queue manager***Functions**

- ❑ QUE\_create. Create an empty queue
- ❑ QUE\_delete. Delete an empty queue
- ❑ QUE\_dequeue. Remove from front of queue (non-atomically)
- ❑ QUE\_empty. Test for an empty queue
- ❑ QUE\_enqueue. Insert at end of queue (non-atomically)
- ❑ QUE\_get. Remove element from front of queue (atomically)
- ❑ QUE\_head. Return element at front of queue
- ❑ QUE\_insert. Insert in middle of queue (non-atomically)
- ❑ QUE\_new. Set a queue to be empty
- ❑ QUE\_next. Return next element in queue (non-atomically)
- ❑ QUE\_prev. Return previous element in queue (non-atomically)
- ❑ QUE\_put. Put element at end of queue (atomically)
- ❑ QUE\_remove. Remove from middle of queue (non-atomically)

**Constants, Types, and Structures**

```

typedef struct QUE_Obj *QUE_Handle;
                                /* handle for queue object */

struct QUE_Attrs{
    Int    dummy;                /* DUMMY */
                                /* queue attributes */
};

QUE_Attrs QUE_ATTRS = {        /* default attribute values */
    0,
};

typedef QUE_Elem;              /* queue element */

```

**Description**

The QUE module makes available a set of functions that manipulate queue objects accessed through handles of type QUE\_Handle. Each queue contains an ordered sequence of zero or more elements referenced through variables of type QUE\_Elem, which are generally embedded as the first field within some struct.

For example, the DEV\_Frame structure which is used by SIO and DEV to enqueue and dequeue I/O buffers is defined as follows:

```
struct DEV_Frame {
    QUE_Elem  link;    /* must be first field! */
    Ptr      addr;
    Uns      size;
}
```

The functions `QUE_put` and `QUE_get` are atomic in that they manipulate the queue with interrupts disabled. These functions may therefore be used to safely share queues between tasks, or between tasks and ISRs. All other QUE functions should only be called by tasks, or by tasks and ISRs when they are used in conjunction with some mutual exclusion mechanism (e.g., `SEM_pend` / `SEM_post`, `TSK_disable` / `TSK_enable`).

### QUE Manager Properties

The following global property can be set for the QUE module:

- Object Memory.** The memory section that contains the QUE objects.

### QUE Object Properties

The following property can be set for a QUE object:

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

**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 \*/

**Assembly Interface**   none

**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 (e.g., 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 may be assigned to a variable of type QUE\_Attrs prior to calling QUE\_create.

**Constraints and Calling Context**

- QUE\_create cannot be called by ISRs.
- You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MEM\_alloc  
QUE\_empty  
QUE\_delete  
SYS\_error

**QUE\_delete***Delete an empty queue***C Interface**

<b>Syntax</b>	QUE_delete(queue);
<b>Parameters</b>	QUE_Handle    queue;    /* queue handle */
<b>Return Value</b>	Void

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

QUE\_delete uses MEM\_free to free the queue object referenced by queue.

**Constraints and Calling Context**

- queue must be empty.
- QUE\_delete cannot be called by ISRs.
- No check is performed to prevent QUE\_delete from being used on a statically-created object. If a program attempts to delete a queue object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

QUE\_create  
QUE\_empty

## QUE\_dequeue

*Remove from front of queue (non-atomically)*

### C Interface

#### Syntax

```
elem = QUE_dequeue(queue);
```

#### Parameters

```
QUE_Handle queue; /* queue object handle */
```

#### Return Value

```
Ptr elem; /* pointer to former first element */
```

### Assembly Interface

none

### Description

QUE\_dequeue removes the element from the front of queue and returns elem.

#### Note:

QUE\_get must be used instead of QUE\_dequeue if queue is shared by multiple tasks, or tasks and ISRs (unless another mutual exclusion mechanism is used).

### 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 \*/**Assembly Interface**     none**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 elem \*/

**Return Value**           Void

**Assembly Interface**    none

**Description**

QUE\_enqueue inserts elem at the end of queue.

**Note:**

QUE\_put must be used instead of QUE\_enqueue if queue is shared by multiple tasks, or tasks and ISRs (unless another mutual exclusion mechanism is used).

**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 */`

**Assembly Interface**        `none`

**Description**

QUE\_get removes the element from the front of queue and returns elem.

Since QUE\_get manipulates queue with interrupts disabled, queue may be shared by multiple tasks, or by tasks and ISRs.

Calling QUE\_get with an empty queue returns the queue itself. This provides a means for using a single atomic action to check if a queue is empty, and to remove and return the first element if it is not empty:

```
if ((QUE_Handle)(elem = QUE_get(q)) != q)
    ` process elem `
```

**See Also**

QUE\_create  
 QUE\_empty  
 QUE\_put

## **QUE\_head**

*Return element at front of queue*

### **C Interface**

**Syntax**

`elem = QUE_head(queue);`

**Parameters**

`QUE_Handle queue;` /\* queue object handle \*/

**Return Value**

`QUE_Elem *elem;` /\* pointer to first element \*/

### **Assembly Interface**

none

### **Description**

QUE\_head returns a pointer to the element at the front of queue. The element is not removed from the queue.

Calling QUE\_head with an empty queue returns the queue itself.

### **See Also**

QUE\_create

QUE\_empty

QUE\_put

**QUE\_insert***Insert in middle of queue (non-atomically)***C Interface****Syntax**

QUE\_insert(qelem, elem);

**Parameters**Ptr qelem; /\* element already in queue \*/  
Ptr elem; /\* element to be inserted in queue \*/**Return Value**

Void

**Assembly Interface**

none

**Description**

QUE\_insert inserts elem in the queue in front of qelem.

**Note:**

If the queue is shared by multiple tasks, or tasks and ISRs, QUE\_insert should be used in conjunction with some mutual exclusion mechanism (e.g., SEM\_pend/ SEM\_post, TSK\_disable / TSK\_enable).

**See Also**QUE\_head  
QUE\_next  
QUE\_prev  
QUE\_remove

**QUE\_new***Set a queue to be empty***C Interface****Syntax**                   QUE\_new(queue);**Parameters**               QUE\_Handle queue; /\* pointer to queue object \*/**Return Value**            Void**Assembly Interface**    none**Description**

QUE\_new adjusts a queue object to make the queue empty. This operation is not atomic. A typical use of QUE\_new is to initialize a queue object that has been statically declared instead of being created with QUE\_create. Note that if the queue is not empty, the element(s) in the queue are not freed or otherwise handled, but are simply abandoned.

**See Also**

QUE\_create  
QUE\_delete  
QUE\_empty

**QUE\_next***Return next element in queue (non-atomically)***C Interface**

**Syntax**                    elem = QUE\_next(qelem);

**Parameters**                Ptr qelem;     /\* element in queue \*/

**Return Value**            Ptr elem;     /\* next element in queue \*/

**Assembly Interface**     none

**Description**

QUE\_next returns elem which points to the element in the queue after qelem.

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 ISRs, QUE\_next should be used in conjunction with some mutual exclusion mechanism (e.g., 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 \*/

**Assembly Interface** none

**Description**

QUE\_prev returns elem which points to the element in the queue before qelem.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE\_prev to return a pointer to the queue itself. Be careful not to call QUE\_remove(elem) in this case.

**Note:**

If the queue is shared by multiple tasks, or tasks and ISRs, QUE\_prev should be used in conjunction with some mutual exclusion mechanism (e.g., 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

**Assembly Interface**

none

**Description**

QUE\_put puts elem at the end of queue.

Since QUE\_put manipulates queue with interrupts disabled, queue may be shared by multiple tasks, or by tasks and ISRs.

**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

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

QUE\_remove removes qelem from the queue.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, be careful not to remove the header node. This can happen when qelem is the return value of QUE\_next or QUE\_prev. The following code sample shows how qelem should be verified before calling QUE\_remove.

```
QUE_Elem *qelem;

/* get pointer to first element in the queue */
qelem = QUE_head(queue);

/* scan entire queue for desired element */
while (qelem != queue) {
    if(' qelem is the elem we're looking for ') {
        break;
    }
    qelem = QUE_next(queue);
}

/* make sure qelem is not the queue itself */
if (qelem != queue) {
    QUE_remove(qelem);
}
```

**Note:**

If the queue is shared by multiple tasks, or tasks and ISRs, QUE\_remove should be used in conjunction with some mutual exclusion mechanism (e.g., SEM\_pend/ SEM\_post and TSK\_disable / TSK\_enable).

### **Constraints and Calling Context**

- ❑ `QUE_remove` should not be called when `qelem` is equal to the queue itself.

### **See Also**

`QUE_head`  
`QUE_insert`  
`QUE_next`  
`QUE_prev`

**RTDX Module***Real-Time Data Exchange Settings***RTDX Data Declaration Macros**

- RTDX\_CreateInputChannel
- RTDX\_CreateOutputChannel

**Functions**

- RTDX\_channelBusy
- RTDX\_disableInput
- RTDX\_disableOutput
- RTDX\_enableInput
- RTDX\_enableOutput
- RTDX\_read
- RTDX\_readNB
- RTDX\_sizeofInput
- RTDX\_write

**Macros**

- RTDX\_isInputEnabled
- RTDX\_isOutputEnabled

**Description**

The RTDX module provides the data types and functions for:

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

Data channels are represented by globally declared structures. A data channel may be used either for input or output, but not both. The contents of an input or output structure are not known to the user. A channel structure contains two states: enabled and disabled. When a channel is enabled, any data written to the channel is sent to the host. Channels are initialized to be disabled.

**RTDX Manager Properties**

The following settings refer to target configuration parameters:

- Enable Real-Time Data Exchange (RTDX).** This box should be checked if you want to link RTDX support into your application.
- RTDX Data Segment.** The memory section used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this section.
- RTDX Buffer Size (MAUs).** The size of the RTDX target-to-host message buffer, in minimum addressable units (MAUs). The default size is 1032 to accommodate a full 1024 byte block and two control words. HST channels that use RTDX are limited by this parameter.

- ❑ **RTDX Text Segment.** The code sections for the RTDX module are placed in this section.
- ❑ **RTDX Interrupt Mask.** This mask identifies RTDX clients and protect RTDX critical sections. The mask specifies the interrupts to be temporarily disabled inside RTDX critical sections. This also temporarily disables other RTDX clients and prevents another RTDX function call. See the RTDX on-line help for details.

### RTDX Object Properties

The following properties can be set for an RTDX object:

- ❑ **comment.** Type a comment to identify this RTDX object.
- ❑ **Channel Mode.** Select output if the RTDX channel handles output from the DSP to the host. Select input if the RTDX channel handles input to the DSP from the host.

**RTDX\_CreateInputChannel** *Declare input channel structure*

**C Interface**

**Syntax** RTDX\_CreateInputChannel( ichan );

**Parameters** ichan /\* Label for the input channel \*/

**Return Value** none

**Assembly Interface** none

**Description**

This macro declares and initializes the RTDX data channel for input.

Data channels must be declared as global objects. A data channel may 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 OLE interface.

**See Also**

RTDX\_CreateOutputChannel

**RTDX\_CreateOutputChannel** *Declare output channel structure***C Interface**

<b>Syntax</b>	RTDX_CreateOutputChannel( ochan );
<b>Parameters</b>	ochan    /* Label for the output channel */
<b>Return Value</b>	none

**Assembly Interface**    none

**Description**

This macro declares and initializes the RTDX data channels for output.

Data channels must be declared as global objects. A data channel may 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 OLE interface.

**See Also**

RTDX\_CreateInputChannel

**RTDX\_channelBusy**     *Return status indicating whether data channel is busy*

### C Interface

**Syntax**                    int RTDX\_channelBusy( RTDX\_inputChannel \*pichan );

**Parameters**                pichan   /\* Identifier for the input data channel \*/

**Return Value**              int       /\* Status: 0 = Channel is not busy. non-zero = Channel is busy. \*/

### Assembly Interface

**Syntax**                    none

**Preconditions**            C callable

**Postconditions**           none

**Modifies**                 none

**Reentrant**                yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

### Description

RTDX\_channelBusy is designed to be used in conjunction with RTDX\_readNB. The return value indicates whether the specified data channel is currently in use or not.

### See Also

RTDX\_readNB

**RTDX\_disableInput** *Disable an input data channel***C Interface**

**Syntax** void RTDX\_disableInput( RTDX\_inputChannel \*ichan );

**Parameters** ichan /\* Identifier for the input data channel \*/

**Return Value** void

**Assembly Interface**

**Syntax** none

**Preconditions** C callable

**Postconditions** none

**Modifies** none

**Reentrant** yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**Description**

A call to a disable function causes the specified channel to be disabled.

**See Also**

RTDX\_disableOutput  
RTDX\_enableInput  
RTDX\_read

**RTDX\_disableOutput** *Enable or disable a data channel***C Interface**

**Syntax** void RTDX\_disableOutput( RTDX\_outputChannel \*ochan );

**Parameters** ochan /\* Identifier for an output data channel \*/

**Return Value** void

**Assembly Interface**

**Syntax** none

**Preconditions** C callable

**Postconditions** none

**Modifies** none

**Reentrant** yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**Description**

A call to an disable function causes the specified data channel to be disabled.

**See Also**

RTDX\_disableInput  
RTDX\_enableOutput  
RTDX\_write

**RTDX\_enableInput** *Enable or disable a data channel***C Interface**

**Syntax** void RTDX\_enableInput( RTDX\_inputChannel \*ichan );

**Parameters** ochan /\* Identifier for an output data channel \*/  
 ichan /\* Identifier for the input data channel \*/

**Return Value** void

**Assembly Interface**

**Syntax** none

**Preconditions** C callable

**Postconditions** none

**Modifies** none

**Reentrant** yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**Description**

A call to an enable function causes the specified data channel to be enabled.

**See Also**

RTDX\_disableInput  
 RTDX\_enableOutput  
 RTDX\_read

**RTDX\_enableOutput** *Enable or disable a data channel***C Interface**

**Syntax** void RTDX\_enableOutput( RTDX\_outputChannel \*ochan );

**Parameters** ochan /\* Identifier for an output data channel \*/

**Return Value** void

**Assembly Interface**

**Syntax** none

**Preconditions** C callable

**Postconditions** none

**Modifies** none

**Reentrant** yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**Description**

A call to an enable function causes the specified data channel to be enabled.

**See Also**

RTDX\_disableOutput  
RTDX\_enableInput  
RTDX\_write

**RTDX\_isInputEnabled** *Return status of the input data channel***C Interface**

**Syntax** RTDX\_isInputEnabled( c );

**Parameter** c /\* Identifier for an input channel. \*/

**Return Value** 0 /\* Not enabled. \*/  
non-zero /\* Enabled. \*/

**Description**

The RTDX\_isInputEnabled macro returns the enabled status of a data channel.

**See Also**

RTDX\_isOutputEnabled

**RTDX\_isOutputEnabled** *Return status of the output data channel*

**C Interface**

**Syntax** RTDX\_isOutputEnabled( c );

**Parameter** c /\* Identifier for an output channel. \*/

**Return Value** 0 /\* Not enabled. \*/  
non-zero /\* Enabled. \*/

**Description**

The RTDX\_isOutputEnabled macro returns the enabled status of a data channel.

**See Also**

RTDX\_isInputEnabled

**RTDX\_read***Read from an input channel***C Interface**

<b>Syntax</b>	int RTDX_read( 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>	> 0 /* The number of address units of data actually supplied in buffer. */ 0 /* Failure. Cannot post read request because target buffer is full. */ RTDX_READ_ERROR /* Failure. Channel currently busy or not enabled. */

**Assembly Interface**

<b>Syntax</b>	none
<b>Preconditions</b>	C callable
<b>Postconditions</b>	none
<b>Modifies</b>	none
<b>Reentrant</b>	yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**Description**

RTDX\_read causes a read request to be posted to the specified input data channel. If the channel is enabled, RTDX\_read busy 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 into the target buffer. When the data is received, the target application continues execution.

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.

**See Also**

`RTDX_channelBusy`  
`RTDX_readNB`  
`RTDX_sizeofInput`

**RTDX\_readNB***Read from input channel without blocking***C Interface**

**Syntax**                   int RTDX\_readNB( RTDX\_inputChannel \*ichan, void \*buffer, int bsize );

**Parameters**

ichan     /\* Identifier for the input data channel \*/  
buffer     /\* A pointer to the buffer that receives the data \*/  
bsize     /\* The size of the buffer in address units \*/

**Return Value**

RTDX\_OK    Success.  
0 (zero)    Failure. The target buffer is full.  
RTDX\_READ\_ERRORChannel is currently busy reading.

**Assembly Interface**

**Syntax**                   none

**Preconditions**           C callable

**Postconditions**         none

**Modifies**                none

**Reentrant**               yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**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.

**See Also**

RTDX\_channelBusy  
RTDX\_read  
RTDX\_sizeofInput

**RTDX\_sizeofInput** *Return the number of bytes read from a data channel*

### C Interface

**Syntax**                   int RTDX\_sizeofInput( RTDX\_inputChannel \*pichan );

**Parameters**             pichan   /\* Identifier for the input data channel \*/

**Return Value**           int       /\* Number of sizeof units of data actually supplied in buffer \*/

### Assembly Interface

**Syntax**                   none

**Preconditions**          C callable

**Postconditions**         none

**Modifies**                none

**Reentrant**               yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

### Description

RTDX\_sizeofInput is designed to be used in conjunction with RTDX\_readNB after a read operation has completed. The function returns the number of sizeof units actually read from the specified data channel.

### See Also

RTDX\_readNB

**RTDX\_write***Write to an output channel***C Interface**

**Syntax**                   int RTDX\_write( RTDX\_outputChannel \*ochan, void \*buffer, int bsize );

**Parameters**           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. \*/

**Assembly Interface**

**Syntax**                 none

**Preconditions**        C callable

**Postconditions**       none

**Modifies**             none

**Reentrant**            yes

**Note:**

No assembly macro is provided for this API. See the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.

**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.

**See Also**

RTDX\_read

**SEM Module***Semaphore Manager***Functions**

- ❑ SEM\_count. Get current semaphore count
- ❑ SEM\_create. Create a semaphore
- ❑ SEM\_delete. Delete a semaphore
- ❑ SEM\_ipost. Signal a semaphore (interrupt only)
- ❑ SEM\_new. Initialize a semaphore
- ❑ SEM\_pend. Wait for a semaphore
- ❑ SEM\_post. Signal a semaphore
- ❑ SEM\_reset. Reset semaphore

**Constants, Types, and Structures**

```
typedef struct SEM_Obj  *SEM_Handle;
                        /* handle for semaphore object */

struct SEM_Attrs {     /* semaphore attributes */
    Int    dummy;     /* DUMMY */
};

SEM_Attrs SEM_ATTRS = { /* default attribute values */
    0,
};
```

**Description**

The SEM module makes available a set of functions that manipulate semaphore objects accessed through handles of type SEM\_Handle. SEM semaphores are counting semaphores that may be used for both task synchronization and mutual exclusion.

SEM\_pend is used to wait for a semaphore. The timeout parameter to SEM\_pend allows the task to wait until a timeout, wait indefinitely, or not wait at all. SEM\_pend's return value is used to indicate if the semaphore was signaled successfully.

SEM\_post is used to signal a semaphore. If a task is waiting for the semaphore, SEM\_post removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, SEM\_post simply increments the semaphore count and returns.

## SEM Manager Properties

The following global property can be set for the SEM module:

- Object Memory.** The memory section that contains the SEM objects created with the Configuration Tool.

## SEM Object Properties

The following properties can be set for a SEM object:

- comment.** Type a comment to identify this SEM object.
- Initial semaphore count.** Set this property to the desired initial semaphore count.

## SEM - Code Composer Studio Interface

The SEM tab of the Kernel/Object View shows information about semaphore objects.

**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 \*/**Assembly Interface**     none**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 \*/

**Assembly Interface** none

**Description**

SEM\_create creates a new semaphore object which is initialized to count. If successful, SEM\_create returns the handle of the new semaphore. If unsuccessful, SEM\_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort).

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

**Note:**

At present, no attributes are supported for semaphore objects, and the type SEM\_Attrs is defined as a dummy structure.

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

No task switch occurs when calling SEM\_create.

**Constraints and Calling Context**

- count must be greater than or equal to 0.
- SEM\_create cannot be called by ISRs.
- You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MEM\_alloc  
SEM\_delete  
SYS\_error

**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

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

SEM\_delete uses MEM\_free to free the semaphore object referenced by sem.

No task switch occurs when calling SEM\_delete.

**Constraints and Calling Context**

- No tasks should be pending on sem when SEM\_delete is called.
- SEM\_delete cannot be called by ISRs.
- No check is performed to prevent SEM\_delete from being used on a statically-created object. If a program attempts to delete a semaphore object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

SEM\_create

**SEM\_ipost***Signal a semaphore (interrupt use only)***C Interface****Syntax**

SEM\_ipost(sem);

**Parameters**

SEM\_Handle sem; /\* semaphore object handle \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SEM\_ipost readies the first task waiting for the semaphore. If no task is waiting, SEM\_ipost simply increments the semaphore count and returns.

SEM\_ipost is the same as SEM\_post in the DSP/BIOS environment. You may call either in an ISR if you have disabled task switching or prepared the ISR for a task switch to occur. SEM\_ipost is provided for source compatibility reasons only. For portable code, use SEM\_ipost within an ISR and SEM\_post within a task.

**See Also**

SEM\_pend

SEM\_post

**SEM\_new***Initialize semaphore object***C Interface****Syntax**

Void SEM\_new(sem, count);

**Parameters**SEM\_Handle sem; /\* pointer to semaphore object \*/  
Int count; /\* initial semaphore count \*/**Return Value**

Void

**Assembly Interface**

none

**Description**

SEM\_new initializes the semaphore object pointed to by sem with count. No task switch occurs when calling SEM\_new.

**Constraints and Calling Context**

- count must be greater than or equal to 0.

**See Also**

QUE\_new

**SEM\_pend***Wait for a semaphore***C Interface**

<b>Syntax</b>	status = SEM_pend(sem, timeout);
<b>Parameters</b>	SEM_Handle sem; /* semaphore object handle */ Uns timeout; /* return after this many system clock ticks */
<b>Return Value</b>	Bool status; /* TRUE if successful, FALSE if timeout */

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

If the semaphore count is greater than zero, SEM\_pend decrements the count and returns TRUE. Otherwise, SEM\_pend suspends the execution of the current task until SEM\_post is called or the timeout expires. If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout is SYS\_FOREVER, the task remains suspended until SEM\_post is called on this semaphore. If timeout is 0, SEM\_pend returns immediately.

If timeout expires (or timeout is 0) before the semaphore is available, SEM\_pend returns FALSE. Otherwise SEM\_pend returns TRUE.

A task switch occurs when calling SEM\_pend if the semaphore count is 0 and timeout is not zero.

**Constraints and Calling Context**

- SEM\_pend may only be called from an ISR if timeout is 0.
- SEM\_pend cannot be called from within a TSK\_disable / TSK\_enable block.
- SEM\_pend should not be called from within an IDL function. Doing so prevents DSP/BIOS plug-ins from gathering run-time information.

**See Also**

SEM\_post

**SEM\_post***Signal a semaphore***C Interface****Syntax**

SEM\_post(sem);

**Parameters**

SEM\_Handle sem; /\* semaphore object handle \*/

**Return Value**

Void

**Assembly Interface**

none

**Description**

SEM\_post readies the first task waiting for the semaphore. If no task is waiting, SEM\_post simply increments the semaphore count and returns.

A task switch occurs when calling SEM\_post if a higher priority task is made ready to run.

**See Also**SEM\_ipost  
SEM\_pend

**SEM\_reset***Reset semaphore count***C Interface****Syntax**

SEM\_reset(sem, count);

**Parameters**SEM\_Handle sem; /\* semaphore object handle \*/  
Int count; /\* semaphore count \*/**Return Value**

Void

**Assembly Interface**

none

**Description**

SEM\_reset resets the semaphore count to count.

No task switch occurs when calling SEM\_reset.

**Constraints and Calling Context**

- count must be greater than or equal to 0.
- No tasks should be waiting for sem when SEM\_reset is called.

**See Also**

SEM\_create

**SIO Module***Stream input and output manager***Functions**

- ❑ SIO\_bufsize. Size of the buffers used by a stream
- ❑ SIO\_create. Create stream
- ❑ SIO\_ctrl. Perform a device-dependent control operation
- ❑ SIO\_delete. Delete stream
- ❑ SIO\_flush. Idle a stream by flushing buffers
- ❑ SIO\_get. Get buffer from stream
- ❑ SIO\_idle. Idle a stream
- ❑ SIO\_issue. Send a buffer to a stream
- ❑ SIO\_put. Put buffer to a stream
- ❑ SIO\_reclaim. Request a buffer back from a stream
- ❑ SIO\_segid. Memory section used by a stream
- ❑ SIO\_select. Select a ready device
- ❑ SIO\_staticbuf. Acquire static buffer from stream

**Constants, Types, and Structures**

```

#define SIO_STANDARD      0 /* open stream for */
                          /* standard streaming model */
#define SIO_ISSUERECLAIM 1 /* open stream for */
                          /* issue/reclaim streaming model */

#define SIO_INPUT        0 /* open for input */
#define SIO_OUTPUT       1 /* open for output */
typedef SIO_Handle;      /* stream object handle */

struct SIO_Attrs {
    Int     nbufs;        /* stream attributes */
    Int     segid;        /* number of buffers */
    Int     align;        /* buffer section ID */
    Bool    flush;        /* buffer alignment */
    Uns     model;        /* TRUE = don't block in DEV_idle() */
    Uns     timeout;     /* usage model: */
                          /* SIO_STANDARD/SIO_ISSUERECLAIM */
};

SIO_Attrs SIO_ATTRS = {
    2,                /* nbufs */
    0,                /* segid */
    0,                /* align */
    FALSE,            /* flush */
    SIO_STANDARD,    /* model */
    SYS_FOREVER       /* timeout */
};

```

## Description

The stream manager provides efficient real-time device-independent I/O through a set of functions that manipulate stream objects accessed through handles of type `SIO_Handle`. The device independence is afforded by having a common high-level abstraction appropriate for real-time applications—continuous streams of data—that can be associated with a variety of devices. All I/O programming is done in a high-level manner using these stream handles to the devices and the stream manager takes care of dispatching into the underlying device drivers.

For efficiency, streams are treated as sequences of fixed-size buffers of data rather than just sequences of bytes.

Streams can be opened and closed at any point during program execution using the functions `SIO_create` and `SIO_delete`, respectively.

The `SIO_issue` and `SIO_reclaim` function calls are enhancements to the basic DSP/BIOS device model. These functions provide a second usage model for streaming, referred to as the issue/reclaim model. It is a more flexible streaming model that allows clients to supply their own buffers to a stream, and to get them back in the order that they were submitted. The `SIO_issue` and `SIO_reclaim` functions also provide a user argument that can be used for passing information between the stream client and the stream devices.

## SIO Manager Properties

The following global properties can be set for the SIO module:

- Object Memory.** The memory section that contains the SIO objects created with the Configuration Tool

## SIO Object Properties

The following properties can be set for an SIO object:

- comment.** Type a comment to identify this SIO object.
- Device.** Select the device to which you want to bind this SIO object. User-defined devices are listed along with DGN and DPI devices.
- Device Control Parameter.** Type the device suffix to be passed to any devices stacked below the device connected to this stream.
- Mode.** Select input if this stream is to be used for input to the application program and output if this stream is to be used for output.
- Buffer size.** If this stream uses the Standard model, this property controls the size of buffers allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle buffers of any size.

- Number of buffers.** If this stream uses the Standard model, this property controls the number of buffers allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle up to the specified Number of buffers.
- Place buffers in memory segment.** Select the memory section to contain the stream buffers if Model is Standard.
- Buffer alignment.** Specify the memory alignment to use for stream buffers if Model is Standard. For example, if you select 16, the buffer must begin at an address that is a multiple of 16. The default is 1, which means the buffer can begin at any address.
- Flush.** Check this box if you want the stream to discard all pending data and return without blocking if this object is idled at run-time with SIO\_idle.
- Model.** Select Standard if you want all buffers to be allocated when the stream is created. Select Issue/Reclaim if your program is to allocate the buffers and supply them using SIO\_issue.
- Allocate Static Buffer(s).** If this box is checked, the Configuration Tool allocates stream buffers for the user. The SIO\_staticbuf function is used to acquire these buffers from the stream. When the Standard model is used, checking this box causes one buffer more than the Number of buffers property to be allocated. When the Issue/Reclaim model is used, buffers are not normally allocated. Checking this box causes the number of buffers specified by the Number of buffers property to be allocated.
- Timeout for I/O operation.** This parameter specifies the length of time SIO\_reclaim waits for I/O. SIO\_reclaim passes this value to the driver's Dxx\_reclaim function. If the timeout expires before a buffer is available, SIO\_reclaim returns SYS\_ETIMEOUT and no buffer is returned.

## **SIO\_bufsize**

*Return the size of the buffers used by a stream*

### **C Interface**

**Syntax**                   size = SIO\_bufsize(stream);

**Parameters**             SIO\_Handle stream;

**Return Value**           Uns     size;

**Assembly Interface**    none

### **Description**

SIO\_bufsize returns the size of the buffers used by stream.

### **See Also**

SIO\_segid

**SIO\_create***Open a stream***C Interface**

**Syntax** stream = SIO\_create(name, mode, bufsize, attrs);

**Parameters**

String	name;	/* name of device */
Int	mode;	/* SIO_INPUT or SIO_OUTPUT */
Uns	bufsize;	/* stream buffer size */
SIO_Attrs	*attrs;	/* pointer to stream attributes */

**Return Value** SIO\_Handle stream; /\* stream object handle \*/

**Assembly Interface** none**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 (e.g., because it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort).

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 {
    Int    nbufs;
    Int    segid;
    Int    align;
    Bool   flush;
    Uns    model;
    Uns    timeout;
};
```

The nbufs attribute specifies the number of buffers allocated by the stream in the SIO\_STANDARD usage model, or the number of buffers to prepare for in

the SIO\_ISSUERECLAIM usage model. The default value of nbufs is 2. In the SIO\_ISSUERECLAIM usage model, nbufs is the maximum number of buffers that can be outstanding (i.e., issued but not reclaimed) at any point in time.

The segid attribute specifies the memory section for stream buffers. Use the memory section names defined using the Configuration Tool. The default value is 0, meaning that buffers are to be allocated from the Segment for DSP/BIOS objects defined for the MEM manager.

The align attribute specifies the memory alignment for stream buffers. The default value is 0, meaning that no alignment is needed.

The flush attribute indicates the desired behavior for an output stream when it is deleted. If flush is TRUE, a call to SIO\_delete or SIO\_idle causes the stream to discard all pending data and return without blocking. If flush is FALSE, a call to SIO\_delete or SIO\_idle causes the stream to block until all pending data has been processed. The default value is FALSE.

The model attribute indicates the usage model that is to be used with this stream. The two usage models are SIO\_ISSUERECLAIM and SIO\_STANDARD. The default usage model is SIO\_STANDARD.

The timeout attribute specifies the length of time the device driver waits for I/O completion before returning an error (e.g., 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. If timeout is 0, SIO\_reclaim returns immediately. If the timeout expires before a buffer is available to be returned, SIO\_reclaim returns (-1 \* SYS\_ETIMEOUT). Otherwise SIO\_reclaim returns the number of valid bytes in the buffer, or -1 multiplied by an error code.

## Constraints and Calling Context

- ❑ A stream can only be used by one task simultaneously. Catastrophic failure may result if more than one task calls SIO\_get (or SIO\_issue / SIO\_reclaim) on the same input stream, or more than one task calls SIO\_put (or SIO\_issue / SIO\_reclaim) on the same output stream.
- ❑ SIO\_create creates a stream dynamically. Do not call SIO\_create on a stream that was created with the Configuration Tool.
- ❑ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX\_create functions. However, streams that are to be used with stacking drivers must be created dynamically with SIO\_create.

**See Also**

MEM\_alloc  
SEM\_pend  
SIO\_delete  
SIO\_issue  
SIO\_reclaim  
SYS\_error

**SIO\_ctrl***Perform a device-dependent control operation***C Interface****Syntax**                    status = SIO\_ctrl(stream, cmd, arg);**Parameters**                SIO\_Handle stream;    /\* stream handle \*/  
Uns                    cmd;                /\* command to device \*/  
Arg                    arg;                /\* arbitrary argument \*/**Return Value**            Int    status;    /\* device status \*/**Assembly Interface**    none**Description**

SIO\_ctrl causes a control operation to be issued to the device associated with stream. cmd and arg are passed directly to the device.

SIO\_ctrl returns SYS\_OK if successful, and a non-zero device-dependent error value if unsuccessful.

**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 \*/

**Assembly Interface**     none

**Description**

SIO\_delete idles the device before freeing the stream object and buffers.

If the stream being deleted was opened for input, then any pending input data is discarded. If the stream being deleted was opened for output, the method for handling data is determined by the value of the object's Flush property in the Configuration Tool or the flush field in the SIO\_Attrs structure (passed in with SIO\_create). If flush is TRUE, SIO\_delete discards all pending data and return without blocking. If flush is FALSE, SIO\_delete blocks until all pending data has been processed by the stream.

SIO\_delete returns SYS\_OK if and only if the operation is successful.

**Constraints and Calling Context**

- SIO\_delete cannot be called by ISRs.
- No check is performed to prevent SIO\_delete from being used on a statically-created object. If a program attempts to delete a stream object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

SIO\_create  
SIO\_flush  
SIO\_idle

**SIO\_flush***Flush a stream***C Interface**

**Syntax**                    status = SIO\_flush(stream);

**Parameters**                SIO\_Handle stream;     /\* stream handle \*/

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

**Assembly Interface**     none

**Description**

SIO\_flush causes all pending data to be discarded regardless of the mode of the stream. SIO\_flush differs from SIO\_idle in that SIO\_flush never suspends program execution to complete processing of data, even for a stream created in output mode.

The underlying device connected to stream is idled as a result of calling SIO\_flush. In general, the interrupt is disabled for the device.

One of the purposes of this function is to provide synchronization with the external environment.

SIO\_flush returns SYS\_OK if and only if the stream is successfully idled.

**See Also**

SIO\_create  
SIO\_idle

**SIO\_get***Get a buffer from stream***C Interface**

**Syntax**                   nbytes = SIO\_get(stream, bufp);

**Parameters**           SIO\_Handle stream; /\* stream handle \*/  
Ptr                       \*bufp; /\* pointer to a buffer \*/

**Return Value**        Int        nbytes; /\* number of bytes read or error if negative \*/

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

SIO\_get exchanges an empty buffer with a non-empty buffer from stream. The bufp parameter is an input/output parameter which points to an empty buffer when SIO\_get is called. When SIO\_get returns, bufp points to a new (different) buffer, and nbytes indicates success or failure of the SIO\_get call.

To indicate success, SIO\_get returns a positive value for nbytes. As a success indicator, nbytes is the number of bytes received from the stream. To indicate failure, SIO\_get returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO\_get.

A task switch occurs when calling SIO\_get if there are no non-empty data buffers in stream.

**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.

**See Also**

SIO\_create  
SIO\_put

**SIO\_idle***Idle a stream***C Interface**

**Syntax**                    status = SIO\_idle(stream);

**Parameters**                SIO\_Handle stream;     /\* stream handle \*/

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

**Assembly Interface**     none

**Description**

If stream is being used for output, SIO\_idle causes any currently buffered data to be transferred to the output device associated with stream. SIO\_idle suspends program execution for as long as is required for the data to be consumed by the underlying device.

If stream is being used for input, SIO\_idle causes any currently buffered data to be discarded. The underlying device connected to stream is idled as a result of calling SIO\_idle. In general, the interrupt is disabled for this device.

If stream is being used for output, the method for handling data is determined by the value of the object's Flush property in the Configuration Tool or the flush field in the SIO\_Attrs structure (passed in with SIO\_create). If flush is TRUE, SIO\_idle discards all pending data and return without blocking. If flush is FALSE, SIO\_idle blocks until all pending data has been processed by the stream.

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.

**See Also**

SIO\_create  
SIO\_flush

**SIO\_issue***Send a buffer to a stream***C Interface**

**Syntax**                    status = SIO\_issue(stream, pbuf, nbytes, arg);

**Parameters**

SIO\_Handle   stream;   /\* stream handle \*/  
 Ptr            pbuf;     /\* pointer to a buffer \*/  
 Uns            nbytes;   /\* number of bytes in the buffer \*/  
 Arg            arg;      /\* user argument \*/

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

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

SIO\_issue is used to send a buffer and its related information to a stream. The buffer-related information consists of the logical length of the buffer (nbytes), 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.

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 nbufs 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 nbytes, 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 bytes of data it contains. For a stream opened in SIO\_INPUT mode, the logical length of a buffer indicates the number of bytes being requested by the client. In either case, the logical size of the buffer must be less than or equal to the physical size of the buffer.

The argument arg is not interpreted by DSP/BIOS, but is offered as a service to the stream client. DSP/BIOS and all DSP/BIOS-compliant device drivers preserve the value of arg and maintain its association with the data that it was issued with. arg provides a user argument as a method for a client to associate additional information with a particular buffer of data.

SIO\_issue is used in conjunction with SIO\_reclaim to operate a stream opened in SIO\_ISSUERECLAIM mode. The SIO\_issue call sends a buffer to a stream, and SIO\_reclaim retrieves a buffer from a stream. In normal operation each SIO\_issue call is followed by an SIO\_reclaim call. Short

bursts of multiple SIO\_issue calls can be made without an intervening SIO\_reclaim call, but over the life of the stream SIO\_issue and SIO\_reclaim must be called the same number of times.

At any given point in the life of a stream, the number of SIO\_issue calls can exceed the number of SIO\_reclaim calls by a maximum of nbufs. The value of nbufs is determined by the SIO\_create call or by setting the Number of buffers property for the object in the Configuration Tool.

**Note:**

An SIO\_reclaim call should not be made without at least one outstanding SIO\_issue call. Calling SIO\_reclaim with no outstanding SIO\_issue calls has undefined results.

**Constraints and Calling Context**

- ❑ The stream must be created with attrs.model set to SIO\_ISSUERECLAIM.

**See Also**

SIO\_create  
SIO\_reclaim

**SIO\_put***Put a buffer to a stream***C Interface**

**Syntax** `nbytes = SIO_put(stream, bufp, nbytes);`

**Parameters**

SIO_Handle	stream;	/* stream handle */
Ptr	*bufp;	/* pointer to a buffer */
Uns	nbytes;	/* number of bytes in the buffer */

**Return Value** Int nbytes; /\* number of bytes returned or negative if error \*/

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

SIO\_put exchanges a non-empty buffer with an empty buffer from stream. The bufp parameter is an input/output parameter which points to a non-empty buffer when SIO\_put is called. When SIO\_put returns, bufp points to a new (different) buffer, and nbytes indicates success or failure of the SIO\_put call.

To indicate success, SIO\_put returns a positive value for nbytes. As a success indicator, nbytes is the number of valid bytes in the buffer returned by the stream (usually zero). To indicate failure, SIO\_put returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO\_put.

A task switch occurs when calling SIO\_put if there are no empty data buffers in stream.

**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.

**See Also**

SIO\_create  
SIO\_get

**SIO\_reclaim***Request a buffer back from a stream***C Interface**

**Syntax** nbytes = SIO\_reclaim(stream, pbufp, parg);

**Parameters**

SIO_Handle	stream;	/* stream handle */
Ptr	*pbufp;	/* pointer to the buffer */
Arg	*parg;	/* pointer to a user argument */

**Return Value** Int nbytes; /\* number of bytes or error if negative \*/

**Assembly Interface** none

**Description**

SIO\_reclaim is used to request a buffer back from a stream. It returns a pointer to the buffer, the number of valid bytes 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 the stream was created in SIO\_OUTPUT mode, then SIO\_reclaim returns an empty buffer, and nbytes is zero, since the buffer is empty. If the stream was opened in SIO\_INPUT mode, SIO\_reclaim returns a non-empty buffer, and nbytes is the number of valid bytes of data in the buffer. In either mode SIO\_reclaim blocks until a buffer can be returned to the caller, or until the timeout expires, and it returns a positive number or zero (indicating success), or a negative number (indicating an error condition). If timeout is not equal to SYS\_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO\_reclaim returns a positive value for nbytes. As a success indicator, nbytes is the number of valid bytes in the buffer. To indicate failure, SIO\_reclaim returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Failure of SIO\_reclaim indicates that no buffer was returned to the client. Therefore, if SIO\_reclaim fails, the client should not attempt to de-reference pbufp, since it is not guaranteed to contain a valid buffer pointer.

SIO\_reclaim is used in conjunction with SIO\_issue to operate a stream opened in SIO\_ISSUERECLAIM mode. The SIO\_issue call sends a buffer to a stream, and SIO\_reclaim retrieves a buffer from a stream. In normal operation each SIO\_issue call is followed by an SIO\_reclaim call. Short bursts of multiple SIO\_issue calls can be made without an intervening SIO\_reclaim call, but over the life of the stream SIO\_issue and SIO\_reclaim must be called the same number of times. The number of SIO\_issue calls can exceed the number of SIO\_reclaim calls by a maximum of nbufs at any given

time. The value of nbufs is determined by the SIO\_create call or by setting the Number of buffers property for the object in the Configuration Tool.

**Note:**

An SIO\_reclaim call should not be made without at least one outstanding SIO\_issue call. Calling SIO\_reclaim with no outstanding SIO\_issue calls has undefined results.

SIO\_reclaim only returns buffers that were passed in using SIO\_issue. It also returns the buffers in the same order that they were issued.

A task switch occurs when calling SIO\_reclaim if timeout is not set to 0, and there are no data buffers available to be returned.

**Constraints and Calling Context**

- The stream must be created with attrs.model set to SIO\_ISSUERECLAIM.
- There must be at least one outstanding SIO\_issue when an SIO\_reclaim call is made.
- All frames issued to a stream must be reclaimed before closing the stream.

**See Also**

SIO\_issue  
SIO\_create

## **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 \*/

**Assembly Interface**     none

### **Description**

SIO\_segid returns the identifier of the memory section 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 \*/

**Assembly Interface**

none

**Description**

SIO\_select waits until one or more of the streams in the streamtab[] array is ready for I/O (i.e., 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.

**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.

**See Also**

SIO\_get  
 SIO\_put  
 SIO\_reclaim

**SIO\_staticbuf***Acquire static buffer from stream***C Interface**

**Syntax** nbytes = SIO\_staticbuf(stream, bufp);

**Parameters** SIO\_Handle stream; /\* stream handle \*/  
Ptr \*bufp; /\* pointer to a buffer \*/

**Return Value** Int nbytes; /\* number of bytes in buffer \*/

**Assembly Interface** none

**Description**

SIO\_staticbuf returns buffers for static streams that were configured using the Configuration Tool. Buffers are allocated for static streams by checking the Allocate Static Buffer(s) check box for the related SIO object.

SIO\_staticbuf returns the size of the buffer or 0 if no more buffers are available from the stream.

SIO\_staticbuf can be called multiple times for SIO\_ISSUERECLAIM model streams.

SIO\_staticbuf must be called to acquire all static buffers before calling SIO\_get, SIO\_put, SIO\_issue or SIO\_reclaim.

**Constraints and Calling Context**

- SIO\_staticbuf should only be called for streams that are defined statically using the Configuration Tool.
- SIO\_staticbuf should only be called for static streams whose Allocate Static Buffer(s) check box has been checked.
- SIO\_staticbuf cannot be called after SIO\_get, SIO\_put, SIO\_issue or SIO\_reclaim have been called for the given stream.

**See Also**

SIO\_get

**STS Module***Statistics Objects Manager***Functions**

- 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.

**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 plug-in 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.

Your program does not need to include any calls to STS functions in order to gather these statistics. The units for the statistics values are controlled by the Statistics Units property of the manager for the module being traced:

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 start to end of execution
SWI	Instruction cycles elapsed from time posted to completion

## Custom STS Objects

You can create custom STS objects using the Configuration Tool. The STS\_add operation updates the count, total, and maximum using the value you provide. The STS\_set operation sets a previous value. The STS\_delta operation accumulates the difference between the value you pass and the previous value and updates the previous value to the value you pass.

By using custom STS objects and the STS operations, you can do the following:

- Count the number of occurrences of an event.** You can pass a value of 0 to STS\_add. The count statistic tracks how many times your program calls STS\_add for this STS object.
- Track the maximum and average values for a variable in your program.** For example, suppose you pass amplitude values to STS\_add. The count tracks how many times your program calls STS\_add for this STS object. The total is the sum of all the amplitudes. The maximum is the largest value. The Statistics View calculates the average amplitude.
- Track the minimum value for a variable in your program.** Negate the values you are monitoring and pass them to STS\_add. The maximum is the negative of the minimum value.
- Time events or monitor incremental differences in a value.** For example, suppose you want to measure the time between hardware interrupts. You would call STS\_set when the program begins running and STS\_delta each time the interrupt routine runs, passing the result of CLK\_gettime each time. STS\_delta subtracts the previous value from the current value. The count tracks how many times the interrupt routine was performed. The maximum is the largest number of clock counts between interrupt routines. The Statistics View also calculates the average number of clock counts.

- ❑ **Monitor differences between actual values and desired values.** For example, suppose you want to make sure a value stays within a certain range. Subtract the midpoint of the range from the value and pass the absolute value of the result to STS\_add. The count tracks how many times your program calls STS\_add for this STS object. The total is the sum of all deviations from the middle of the range. The maximum is the largest deviation. The Statistics View calculates the average deviation.

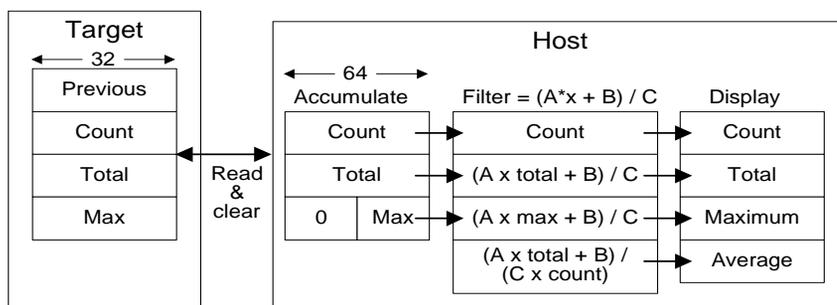
You can further customize the statistics data by setting the STS object properties to apply a printf format to the Total, Max, and Average fields in the Statistics View window and choosing a formula to apply to the data values on the host.

### Statistics Data Gathering by the Statistics View Plug-in

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 232 - 1 samples), and the 32-bit sum of all samples.

These statistics are accumulated on the target in real-time until the host reads and clears these values on the target. The host, however, continues to accumulate the values read from the target in a host buffer which is displayed by the Statistics View real-time analysis tool. Provided that the host reads and clears the target statistics objects faster than the target can overflow the 32-bit wide values being accumulated, no information loss occurs.

Using the Configuration Tool, you can select a Host Operation for an STS object. The statistics are filtered on the host using the operation and variables you specify. This figure shows the effects of the  $(A \times X + B) / C$  operation.



### STS Manager Properties

The following global property can be set for the STS module:

- ❑ **Object Memory.** The memory section that contains the STS objects.

## STS Object Properties

The following properties can be set for a statistics object:

- comment.** Type a comment to identify this STS object
- prev.** The initial 32-bit history value to use in this object
- format.** The printf-style format string used to display the data for this object
- host operation.** The expression evaluated (by the host) on the data for this object before it is displayed by the Statistics View real-time analysis tool. The operation can be:
  - $A \times X$
  - $A \times X + B$
  - $(A \times X + B) / C$
- A, B, C.** The integer parameters used by the expression specified by the Host Operation field above.

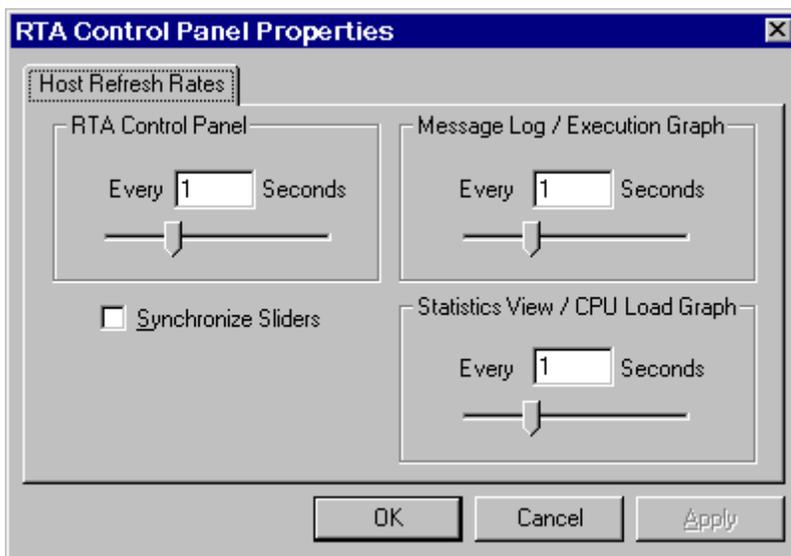
## STS - Statistics View Interface

You can view statistics in real-time with the Statistics View plug-in by choosing the Tools→DSP/BIOS→Statistics View menu item.

Statistics View				
	Count	Total	Max	Average
IDL_busyObj	35376	1110	1	0.03
processing_S\wl	838	4.21629e+007	58704 inst	50313.71 inst

To pause the display, right-click on this window and choose Pause from the pop-up menu. To reset the values to 0, right-click on this window and choose Clear from the pop-up menu.

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



See the *TMS320C6000 Code Composer Studio Tutorial* for more information on how to monitor statistics with the Statistics View plug-in.

**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 */ LgInt value; /* new value to update statistics object */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	STS_add
<b>Preconditions</b>	a4 = STS object handle b4 = 32-bit sample amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, a3, b1, b2, b3
<b>Reentrant</b>	no

**Description**

STS\_add updates a custom STS object's Total, Count, and Max fields using the data value you provide.

For example, suppose your program passes 32-bit amplitude values to STS\_add. The Count field tracks how many times your program calls STS\_add for this STS object. The Total field tracks the total of all the amplitudes. The Max field holds the largest value passed to this point. The Statistics View plug-in 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 plug-in by enabling statistics in the Tools→DSP/BIOS→RTA Control Panel window and choosing your custom STS object in the Tools→DSP/BIOS→Statistics View window.

**See Also**

STS\_delta  
STS\_reset  
STS\_set  
TRC\_disable  
TRC\_enable

**STS\_delta**

*Update statistics using the difference between the provided value and the setpoint*

**C Interface**

<b>Syntax</b>	STS_delta(sts,value);
<b>Parameters</b>	STS_Handle sts; /* statistics object handle */ LgInt value; /* new value to update statistics object */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	STS_delta
<b>Preconditions</b>	a4 = STS object handle b4 = 32-bit sample amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1, a3, b1, b2, b3, b4, b5
<b>Reentrant</b>	no

**Description**

Each STS object contains a previous value that can be initialized with the Configuration Tool or with a call to STS\_set. A call to STS\_delta subtracts the previous value from the value it is passed and then invokes STS\_add with the result to update the statistics. STS\_delta also updates the previous value with the value it is passed.

STS\_delta can be used in conjunction with STS\_set to monitor the difference between a variable and a desired value or to benchmark program performance.

You can benchmark your code by using paired calls to STS\_set and STS\_delta that pass the value provided by CLK\_gethlttime.

```
STS_set(&sts, CLK_gethlttime());
    "processing to be benchmarked"
STS_delta(&sts, CLK_gethlttime());
```

**Constraints and Calling Context**

- ❑ Before the first call to STS\_delta is made, the previous value of the STS object should be initialized either with a call to STS\_set or by setting the prev property of the STS object using the Configuration Tool.

**Example**

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

**See Also**

STS\_add  
STS\_reset  
STS\_set  
CLK\_gethtime  
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

**Assembly Interface**

<b>Syntax</b>	STS_reset
<b>Preconditions</b>	a4 = STS object handle amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a1
<b>Reentrant</b>	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 plug-in 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 */ LgInt value; /* new value to update statistics object */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	STS_set
<b>Preconditions</b>	a4 = STS object handle b4 = new 32-bit value to store as previous amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	none
<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 may need to use STS\_set to reset the previous value before the next call to STS\_delta.

You can also set a previous value for an STS object in the Configuration Tool. STS\_set changes this value.

See STS\_delta for details on how to use the value you set with STS\_set.

**Example**

This example gathers performance information for the processing between STS\_set and STS\_delta.

```
STS_set(&sts, CLK_gettime());  
    "processing to be benchmarked"  
STS_delta(&sts, CLK_gettime());
```

This example gathers information about a value's deviation from the desired value.

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

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

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

### See Also

- STS\_add
- STS\_delta
- STS\_reset
- TRC\_disable
- TRC\_enable

**SWI Module***Software interrupt manager***Functions**

- ❑ SWI\_andn. Clear bits from SWI's mailbox; post if becomes 0
- ❑ SWI\_create. Create a software interrupt
- ❑ SWI\_dec. Decrement SWI's mailbox value; post if becomes 0
- ❑ SWI\_delete. Delete a software interrupt
- ❑ SWI\_disable. Disable software interrupts
- ❑ SWI\_enable. Enable software interrupts
- ❑ SWI\_getattrs. Get attributes of a software interrupt
- ❑ SWI\_getmbox. Return an SWI's mailbox value
- ❑ SWI\_getpri. Return an SWI's priority mask
- ❑ SWI\_inc. Increment SWI's mailbox value
- ❑ SWI\_or. Or mask with value contained in SWI's mailbox field
- ❑ SWI\_post. Post a software interrupt
- ❑ SWI\_raisepri. Raise an SWI's priority
- ❑ SWI\_restorepri. Restore an SWI's priority
- ❑ SWI\_self. Return address of currently executing SWI object
- ❑ SWI\_setattrs. Set attributes of a software interrupt

**Description**

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt service routines.

DSP/BIOS manages four distinct levels of execution threads: hardware interrupt service routines, software interrupts, tasks, and background idle functions. A software interrupt is an object that encapsulates a function to be executed and a priority. Software interrupts are prioritized, preempt tasks, and are preempted by hardware interrupt service routines.

**Note:**

SWI functions are called after the processor register state has been saved. SWI functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

**Note:**

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 C 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 an SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a software interrupt is posted by an API call, the SWI object's function is not executed immediately. Instead, the function is scheduled for execution. DSP/BIOS uses the software interrupt's priority to determine whether to preempt the thread currently running. Note that if a software interrupt is posted several times before it begins running, because HWIs and higher priority interrupts are running, the software interrupt only runs one time.

Software interrupts can be scheduled for execution with a call to SWI\_post or a number of other SWI functions. Each SWI object has a 32-bit mailbox which is used either to determine whether to post the software interrupt or as a value that can be evaluated within the software interrupt's function. SWI\_andn and SWI\_dec post the software interrupt if the mailbox value transitions to 0. SWI\_or and SWI\_inc also modify the mailbox value. (SWI\_or sets bits, and SWI\_andn clears bits.)

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

The SWI\_disable and SWI\_enable operations allow you to post several software interrupts and enable them all for execution at the same time. The software interrupt priorities then determine which software interrupt runs first.

All software interrupts run to completion; you cannot suspend a software interrupt while it waits for something—e.g., a device—to be ready. So, you can use the mailbox to tell the software interrupt when all the devices and other conditions it relies on are ready. Within a software interrupt processing function, a call to SWI\_getmbox returns the value of the mailbox when the

software interrupt started running. The mailbox is automatically reset to its original value when a software interrupt runs.

Software interrupts can have up to 15 priority levels. The highest level is SWI\_MAXPRI (14). The lowest is SWI\_MINPRI (0). The priority level of 0 is reserved for the KNL\_swi object, which runs the task scheduler.

A software interrupt preempts any currently running software interrupt with a lower priority. If two software interrupts with the same priority level have been posted, the software interrupt that was posted first runs first. Hardware interrupts in turn preempt any currently running software interrupt, allowing the target to respond quickly to hardware peripherals. For information about setting software interrupt priorities, you can choose Help→Help Topics in the Configuration Tool, click the Index tab, and type priority.

Interrupt threads—including hardware interrupts and software interrupts—are all executed using the same stack. A context switch is performed when a new thread is added to the top of the stack. The SWI module automatically saves the processor's registers before running a higher-priority software interrupt that preempts a lower-priority software interrupt. After the higher-priority software interrupt finishes running, the registers are restored and the lower-priority software interrupt can run if no other higher-priority software interrupts have been posted. (A separate task stack is used by each task thread.)

See the *TMS320C6000 Code Composer Studio Tutorial* for more information on how to post software interrupts and scheduling issues for the Software Interrupt manager.

## SWI Manager Properties

The following global properties can be set for the SWI module:

- Object Memory.** The memory section that contains the SWI objects.
- Statistics Units.** The units used to display the elapsed instruction cycles or time from when a software interrupt is posted to its completion within the Statistics View plug-in. Raw causes the STS Data to display the number of instruction cycles if the CLK module's Use high resolution time for internal timings parameter is set to True (the default). If this CLK parameter is set to False and the Statistics Units is set to Raw, SWI statistics are displayed in units of timer interrupt periods. You can also choose milliseconds or microseconds.

## SWI Object Properties

The following properties can be set for an SWI object:

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

- ❑ **priority.** This field shows the numeric priority level for this SWI object. Software interrupts can have up to 15 priority levels. The highest level is SWI\_MAXPRI (14). The lowest is SWI\_MINPRI (0). The priority level of 0 is reserved for the KNL\_swi object, which runs the task scheduler. Instead of typing a number in this field, you change the relative priority levels of SWI objects by dragging the objects in the ordered collection view.
- ❑ **function.** The function to execute
- ❑ **mailbox.** The initial value of the 32-bit word used to determine if this software interrupt should be posted.
- ❑ **arg0, arg1.** Two 32-bit arguments passed to function; these arguments can be either an unsigned 32-bit constant or a symbolic label.

### SWI - Code Composer Studio Interface

The SWI tab of the Kernel/Object View shows information about software interrupt objects.

To enable SWI logging, choose Tools→DSP/BIOS→RTA Control Panel and put a check in the appropriate box. To view a graph of activity that includes SWI function execution, choose Tools→DSP/BIOS→Execution Graph.

You can also enable SWI accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose an SWI object, you see statistics about the number of instruction cycles elapsed from the time the SWI was posted to the SWI function's completion.

**SWI\_andn***Clear bits from SWI's mailbox and post if mailbox becomes 0***C Interface**

<b>Syntax</b>	SWI_andn(swi, mask);
<b>Parameters</b>	SWI_Handle swi; /* SWI object handle */ Uns mask /* value to be ANDed */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_andn
<b>Preconditions</b>	a4 = address of the SWI object b4 = mask amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a9, b0, b1, b2, b3, b4, b5, b6, b7, csr
<b>Reentrant</b>	yes

**Description**

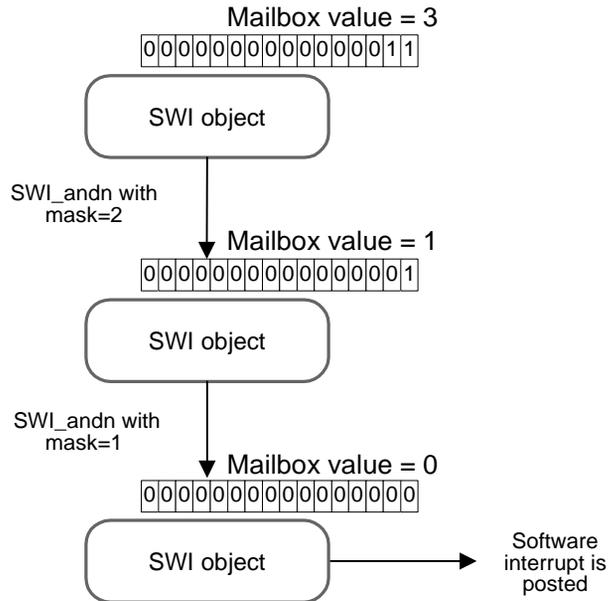
SWI\_andn is used to conditionally post a software interrupt. SWI\_andn clears the bits specified by a mask from SWI's internal mailbox. If SWI's mailbox becomes 0, SWI\_andn posts the software interrupt. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

For example, if there are multiple conditions that must all be met before a software interrupt can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

The following figure shows an example of how a mailbox with an initial value of 3 can be cleared by two calls to SWI\_andn with values of 2 and 1. The entire mailbox could also be cleared with a single call to SWI\_andn with a value of 3.



### Constraints and Calling Context

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

### Example

```
/* ===== ioReady ===== */

Void ioReady(unsigned int mask)
{
    SWI_andn(&copySWI, mask); /* clear bits of "ready mask" */
}
```

### See Also

SWI\_dec  
 SWI\_getmbox  
 SWI\_inc  
 SWI\_or  
 SWI\_post  
 SWI\_self

**SWI\_create***Create a software interrupt***C Interface**

<b>Syntax</b>	<code>swi = SWI_create(attrs);</code>
<b>Parameters</b>	<code>SWI_Attrs *attrs; /* pointer to swi attributes */</code>
<b>Return Value</b>	<code>SWI_Handle swi; /* handle for new swi object */</code>

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

SWI\_create creates a new SWI object. If successful, SWI\_create returns the handle of the new SWI object. If unsuccessful, SWI\_create returns NULL unless it aborts. For example, SWI\_create may abort if it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the object to be created, facilitates setting the SWI object's attributes. If attrs is NULL, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type SWI\_attrs defined as follows:

```
struct SWI_Attrs {
    SWI_Fxn  fxn;
    Arg     arg0;
    Arg     arg1;
    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 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 may be assigned to a variable of type SWI\_Attrs prior to calling SWI\_create.

**Constraints and Calling Context**

- ❑ SWI\_create cannot be called by ISRs.
- ❑ The fxn attribute cannot be NULL.
- ❑ The priority attribute must be less than or equal to 14 and greater than or equal to 0.

**See Also**

SWI\_delete  
SWI\_getattrs  
SWI\_setattrs  
SYS\_error

**SWI\_dec***Decrement SWI's mailbox value and post if mailbox becomes 0***C Interface**

<b>Syntax</b>	SWI_dec(swi);
<b>Parameters</b>	SWI_Handle swi;     /* SWI object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_dec
<b>Preconditions</b>	a4 = address of the SWI object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a9, b0, b1, b2, b3, b4, b5, b6, b7, csr
<b>Reentrant</b>	yes

**Description**

SWI\_dec is used to conditionally post a software interrupt. SWI\_dec decrements the value in SWI's mailbox by 1. If SWI's mailbox value becomes 0, SWI\_dec posts the software interrupt. You can increment a mailbox value by using SWI\_inc, which always posts the software interrupt.

For example, you would use SWI\_dec if you wanted to post a software interrupt after a number of occurrences of an event.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

**Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

**Example**

```
/* ===== strikeOrBall ===== */  
  
Void strikeOrBall(unsigned int call)  
{  
    if (call == 1) {  
        SWI_dec(&strikeoutSwi); /* initial mailbox value is 3 */  
    }  
    if (call == 2) {  
        SWI_dec(&walkSwi);      /* initial mailbox value is 4 */  
    }  
}
```

**See Also**

SWI\_delete  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_post  
SWI\_self

**SWI\_delete***Delete a software interrupt***C Interface**

<b>Syntax</b>	SWI_delete(swi);
<b>Parameters</b>	SWI_Handle swi; /* SWI object handle */
<b>Return Value</b>	Void

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

SWI\_delete uses MEM\_free to free the SWI object referenced by swi.

**Constraints and Calling Context**

- swi cannot be the currently executing SWI object (SWI\_self)
- SWI\_delete cannot be called by ISRs.
- SWI\_delete must not be used to delete a statically-created SWI object. No check is performed to prevent SWI\_delete from being used on a statically-created object. If a program attempts to delete a SWI object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

SWI\_create  
SWI\_getattrs  
SWI\_setattrs  
SYS\_error

## **SWI\_disable** *Disable software interrupts*

### C Interface

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

### Assembly Interface

<b>Syntax</b>	SWI_disable
<b>Preconditions</b>	b14 = address of the start of .bss GIE = 1 (interrupts must be enabled) amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a4
<b>Reentrant</b>	yes

### Description

SWI\_disable and SWI\_enable control SWI software interrupt processing. SWI\_disable disables all other SWI functions from running until SWI\_enable is called. Hardware interrupts can still run.

SWI\_disable and SWI\_enable allow you to ensure that statements that must be performed together during critical processing are not interrupted. In the following example, the critical section is not preempted by any software interrupts.

```
SWI_disable();
    `critical section`
SWI_enable();
```

You can also use SWI\_disable and SWI\_enable to post several software interrupts and allow them to be performed in priority order. See the example that follows.

SWI\_disable calls can be nested—the number of nesting levels is stored internally. Software interrupt handling is not reenabled until SWI\_enable has been called as many times as SWI\_disable.

## Constraints and Calling Context

- The calls to HWI\_enter and HWI\_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenables software interrupt handling. You should not call SWI\_disable or SWI\_enable within a hardware ISR.

## Example

```
/* ===== postEm ===== */  
  
Void postEm()  
{  
    SWI_disable();  
  
    SWI_post(&encoderSwi);  
    SWI_andn(&copySwi, mask);  
    SWI_dec(&strikeoutSwi);  
  
    SWI_enable();  
}
```

## See Also

HWI\_disable  
HWI\_enable  
SWI\_enable

**SWI\_enable** *Enable software interrupts***C Interface****Syntax** SWI\_enable();**Parameters** Void**Return Value** Void**Assembly Interface****Syntax** SWI\_enable**Preconditions** SWI\_D\_lock >= 0 (SWI execution is disabled; i.e., locked)  
GIE = 1 (interrupts must be enabled)  
amr = 0**Postconditions** none**Modifies** a1, a4, b0, b1, b3, b4, csr**Reentrant** yes**Description**

SWI\_disable and SWI\_enable control SWI software interrupt processing. SWI\_disable disables all other software interrupt functions from running until SWI\_enable is called. Hardware interrupts can still run. See the SWI\_disable section for details.

SWI\_disable calls can be nested—the number of nesting levels is stored internally. Software interrupt handling is not be reenabled until SWI\_enable has been called as many times as SWI\_disable.

**Constraints and Calling Context**

- ❑ The calls to HWI\_enter and HWI\_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenables software interrupt handling. You should not call SWI\_disable or SWI\_enable within a hardware ISR.

**See Also**

HWI\_disable  
 HWI\_enable  
 SWI\_disable

**SWI\_getattrs***Get attributes of a software interrupt***C Interface**

**Syntax** SWI\_getattrs(swi, attrs);

**Parameters** SWI\_Handle swi; /\* handle of the swi \*/  
SWI\_Attrs \*attrs; /\* pointer to swi attributes \*/

**Return Value** Void

**Assembly Interface** none

**Description**

SWI\_getattrs retrieves attributes of an existing SWI object.

The swi parameter specifies the address of the SWI object whose attributes are to be retrieved. The attrs parameter, which is the pointer to a structure that contains the retrieved attributes for the SWI object, facilitates retrieval of the attributes of the SWI object.

The SWI object's attributes are specified through a structure of type SWI\_attrs defined as follows:

```
struct SWI_Attrs {
    SWI_Fxn    fxn;
    Arg       arg0;
    Arg       arg1;
    Int       priority;
    Uns       mailbox;
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The priority attribute specifies the SWI object's execution priority and ranges from 0 to 14. The highest level is SWI\_MAXPRI (14). The lowest is SWI\_MINPRI (0). The priority level of 0 is reserved for the KNL\_swi object, which runs the task scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

The following example uses SWI\_getattrs:

```
extern SWI_Handle swi;  
SWI_Attrs attrs;  
  
SWI_getattrs(swi, &attrs);  
attrs.priority = 5;  
SWI_setattrs(swi, &attrs);
```

### Constraints and Calling Context

- ❑ SWI\_getattrs cannot be called by ISRs.
- ❑ The attrs parameter cannot be NULL.

### See Also

SWI\_create  
SWI\_delete  
SWI\_setattrs

**SWI\_getmbox***Return a SWI's mailbox value***C Interface**

<b>Syntax</b>	<code>num = Uns SWI_getmbox();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	Uns num /* mailbox value */

**Assembly Interface**

<b>Syntax</b>	SWI_getmbox
<b>Preconditions</b>	b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	a14 = current software interrupt's mailbox value
<b>Modifies</b>	a4
<b>Reentrant</b>	yes

**Description**

SWI\_getmbox returns the value that SWI's mailbox had when the software interrupt started running. DSP/BIOS saves the mailbox value internally so that SWI\_getmbox can access it at any point within an SWI object's function. DSP/BIOS then automatically resets the mailbox to its initial value (defined with the Configuration Tool) so that other threads can continue to use the software interrupt's mailbox.

SWI\_getmbox should only be called within a function run by a SWI object.

**Example**

This example could be used within a SWI object's function to use the value of the mailbox within the function. For example, if you use SWI\_or or SWI\_inc to post a software interrupt, different mailbox values may require different processing.

```
/* get current SWI mailbox value */  
swicount = SWI_getmbox();
```

**See Also**

SWI\_andn  
SWI\_dec  
SWI\_inc  
SWI\_or  
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 \*/

**Assembly Interface**

**Syntax**                   SWI\_getpri

**Preconditions**          a4 = address of the SWI object  
b14 = address of start of .bss

**Postconditions**         a4 = SWI object's priority mask

**Modifies**               a4

**Reentrant**               yes

**Description**

SWI\_getpri returns the priority mask of the SWI passed in as the argument.

**Example**

```
/* Get the priority key of swi1 */
key = SWI_getpri(&swi1);
/* Get the priorities of swi1 and swi3 */
key = SWI_getpri(&swi1) | SWI_getpri(&swi3);
```

**See Also**

SWI\_raisepri  
SWI\_restorepri

**SWI\_inc***Increment SWI's mailbox value***C Interface**

<b>Syntax</b>	SWI_inc(swi);
<b>Parameters</b>	SWI_Handle swi: /* SWI object handle*/
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_inc
<b>Preconditions</b>	a4 = address of the SWI object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a9, b0, b1, b2, b3, b4, b5, b6, b7, csr
<b>Reentrant</b>	no

**Description**

SWI\_inc increments the value in SWI's mailbox by 1 and posts the software interrupt regardless of the resulting mailbox value. You can decrement a mailbox value by using SWI\_dec, which only posts the software interrupt if the mailbox value is 0.

If a software interrupt is posted several times before it has a chance to begin executing, because HWIs and higher priority software interrupts are running, the software interrupt only runs one time. If this situation occurs, you can use SWI\_inc to post the software interrupt. Within the software interrupt's function, you could then use SWI\_getmbox to find out how many times this software interrupt has been posted since the last time it was executed.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI\_getmbox.

**Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

**Example**

```
/* ===== AddAndProcess ===== */  
  
Void AddAndProcess(int count)  
{  
    int i;  
  
    for (i = 1; I <= count; ++i)  
        SWI_inc(&MySwi);  
    SWI_post(&MySwi);  
}
```

**See Also**

SWI\_andn  
SWI\_dec  
SWI\_getmbox  
SWI\_or  
SWI\_post  
SWI\_self

**SWI\_or***OR mask with the value contained in SWI's mailbox field***C Interface**

<b>Syntax</b>	SWI_or(swi, mask);
<b>Parameters</b>	SWI_Handle swi; /* SWI object handle */ Uns mask; /* value to be ORed */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_or
<b>Preconditions</b>	a4 = address of the SWI object b4 = mask amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a9, b0, b1, b2, b3, b4, b5, b6, b7, csr
<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 software interrupt regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI\_getmbox.

For example, you might use SWI\_or to post a software interrupt if any of three events should cause a software interrupt to be executed, but you want the software interrupt's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

**Constraints and Calling Context**

- ❑ If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

**See Also**

SWI\_andn  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_post  
SWI\_self

**SWI\_post***Post a software interrupt***C Interface**

<b>Syntax</b>	SWI_post(swi);
<b>Parameters</b>	SWI_Handle swi; /* SWI object handle */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_post
<b>Preconditions</b>	a4 = address of the SWI object amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a0, a1, a2, a3, a4, a5, a6, a7, a9, b0, b1, b2, b3, b4, b5, b6, b7, csr
<b>Reentrant</b>	no

**Description**

SWI\_post is used to post a software interrupt regardless of the mailbox value. No change is made to SWI's mailbox value.

To have a PRD object post an SWI object's function, you can set \_SWI\_post as the function property of a PRD object and the name of the software interrupt object you want to post its function as the arg0 property.

**Constraints and Calling Context**

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

**See Also**

SWI\_andn  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_self

**SWI\_raisepri***Raise a SWI's priority***C Interface**

<b>Syntax</b>	key = SWI_raisepri(mask);
<b>Parameters</b>	Uns mask; /* mask of desired priority level */
<b>Return value</b>	Uns key; /* key for use with SWI_restorepri */

**Assembly Interface**

<b>Syntax</b>	SWI_raisepri
<b>Preconditions</b>	b14 = address of start of .bss a4 = priority mask of desired priority level
<b>Postconditions</b>	a4 = key for use with SWI_restorepri
<b>Modifies</b>	a1, a2, a4
<b>Reentrant</b>	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 software interrupts.

SWI\_raisepri should be called before the shared resource is accessed, and SWI\_restorepri should be called after the access to the shared resource.

A call to SWI\_raisepri not followed by a SWI\_restorepri keeps the SWI's priority for the rest of the processing at the raised level. A SWI\_post of the SWI posts the SWI at its original priority level.

A SWI object's execution priority must range from 0 to 14. The highest level is SWI\_MAXPRI (14). The lowest is SWI\_MINPRI (0). The priority level of 0 is reserved for the KNL\_swi object, which runs the task scheduler.

SWI\_raisepri never lowers the current SWI priority.

**Example**

```
/* raise priority to the priority of swi_1 */  
key = SWI_raisepri(SWI_getpri(&swi_1));  
--- access shared resource ---  
SWI_restore(key);
```

**See Also**

SWI\_getpri  
SWI\_restorepri

**SWI\_restorepri***Restore a SWI's priority***C Interface**

<b>Syntax</b>	SWI_restorepri(key);
<b>Parameters</b>	Uns key; /* key to restore original priority level */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	SWI_restorepri
<b>Preconditions</b>	b14 = address of start of .bss a4 = return value from the SWI_raisepri call
<b>Postconditions</b>	none
<b>Modifies</b>	a1, a2, a4, b0,csr
<b>Reentrant</b>	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 software interrupts.

SWI\_raisepri should be called right before the shared resource is referenced, and SWI\_restorepri should be called after the reference to the shared resource.

**Example**

```
/* raise priority to the priority of swi_1 */
key = SWI_raisepri(SWI_getpri(&swi_1));
--- access shared resource ---
SWI_restore(key);
```

**See Also**

SWI\_getpri  
SWI\_raisepri

**SWI\_self***Return address of currently executing SWI object***C Interface**

<b>Syntax</b>	<code>curswi = SWI_self();</code>
<b>Parameters</b>	Void
<b>Return Value</b>	SWI_Handle swi; /* handle for current swi object */

**Assembly Interface**

<b>Syntax</b>	SWI_self
<b>Preconditions</b>	b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	a4 = address of the current SWI object
<b>Modifies</b>	a4, b4
<b>Reentrant</b>	yes

**Description**

SWI\_self returns the address of the currently executing software interrupt.

Within a hardware ISR, SWI\_self returns the address of the software interrupt highest in the processing stack—i.e., the software interrupt that yielded to the hardware interrupt. If no software interrupt is running or yielding, SWI\_self returns NULL.

**Example**

You can use SWI\_self if you want a software interrupt to repost itself:

```
SWI_post(SWI_self());
```

**See Also**

SWI\_andn  
SWI\_dec  
SWI\_getmbox  
SWI\_inc  
SWI\_or  
SWI\_post

**SWI\_setattrs***Set attributes of a software interrupt***C Interface**

<b>Syntax</b>	SWI_setattrs(swi, attrs);
<b>Parameters</b>	SWI_Handle swi; /* handle of the swi */ SWI_Attrs *attrs; /* pointer to swi attributes */
<b>Return Value</b>	Void

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

SWI\_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\_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 must range from 2 to 15. Priority 15 is the highest priority. You cannot use a priority of 1; 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 may 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 cannot be called by ISRs.
- ❑ SWI\_setattrs should 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 15 and greater than or equal to 2.

### See Also

SWI\_create  
SWI\_delete  
SWI\_getattrs

**SYS Module***System Settings***Functions**

- ❑ `SYS_abort`. Abort program execution
- ❑ `SYS_atexit`. Stack an exit handler
- ❑ `SYS_error`. Flag error condition
- ❑ `SYS_exit`. Terminate program execution
- ❑ `SYS_printf`. Formatted output
- ❑ `SYS_putchar`. Output a single character
- ❑ `SYS_sprintf`. Formatted output to string buffer
- ❑ `SYS_vprintf`. Formatted output, variable argument list
- ❑ `SYS_vsprintf`. Output formatted data

**Constants, Types, and Structures**

```

#define SYS_FOREVER (Uns)-1 /* wait forever */
#define SYS_POLL (Uns)0 /* don't wait */

#define SYS_OK 0 /* no error */
#define SYS_EALLOC 1 /* memory allocation error */
#define SYS_EFREE 2 /* memory free error */
#define SYS_ENODEV 3 /* device driver not found */
#define SYS_EBUSY 4 /* device driver busy */
#define SYS_EINVAL 5 /* invalid parameter for device */
#define SYS_EBADIO 6 /* I/O failure */
#define SYS_EMODE 7 /* bad mode for device driver */
#define SYS_EDOMAIN 8 /* domain error */
#define SYS_ETIMEOUT 9 /* call timed out */
#define SYS_EEOF 10 /* end-of-file */
#define SYS_EDEAD 11 /* previously deleted object */
#define SYS_EBADOBJ 12 /* invalid object */
#define SYS_EUSER 256 /* user errors start here */

#define SYS_NUMHANDLERS 8 /* number of atexit handlers */

extern String SYS_errors[]; /* array of error strings */

```

**Description**

The SYS module makes available a set of general-purpose functions that provide basic system services, such as halting program execution and printing formatted text. In general, each SYS function is patterned after a similar function normally found in the standard C library.

SYS does not directly use the services of any other DSP/BIOS module and therefore resides at the bottom of the system. Other DSP/BIOS modules use the services provided by SYS in lieu of similar C library functions. The SYS

module provides hooks for binding system-specific code. This allows programs to gain control wherever other DSP/BIOS modules call one of the SYS functions.

## SYS Manager Properties

The following global properties can be set for the SYS module:

- ❑ **Trace Buffer Size.** The size of the buffer that contains system trace information. For example, by default the Putc function writes to the trace buffer.
- ❑ **Trace Buffer Memory.** The memory section that contains system trace information.
- ❑ **Abort function.** The function to run if the application aborts by calling SYS\_abort. The default function is \_UTL\_doAbort, which logs an error message and calls \_halt.  
If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- ❑ **Error function.** The function to run if an error flagged by SYS\_error occurs. The default function is \_UTL\_doError, which logs an error message.  
If this function is written in C, use a leading underscore before the C function name.
- ❑ **Exit function.** The function to run when the application exits by calling SYS\_exit. The default function is UTL\_halt, which loops forever with interrupts disabled—preventing other processing.  
If this function is written in C, use a leading underscore before the C function name.
- ❑ **Putc function.** The function to run if the application calls SYS\_putchar, SYS\_printf, or SYS\_vprintf. The default function is \_UTL\_doPutc, which writes a character to the trace buffer.  
If this function is written in C, use a leading underscore before the C function name.

## SYS Object Properties

The SYS module does not support the creation of individual SYS objects.

**SYS\_abort***Abort program execution***C Interface**

<b>Syntax</b>	<code>SYS_abort(format, [arg,] ...);</code>
<b>Parameters</b>	String <code>format</code> ; /* format specification string */ Arg <code>arg</code> ; /* optional argument */
<b>Return Value</b>	Void

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

SYS\_abort aborts program execution by calling the function bound to the configuration parameter Abort function, where `vargs` is of type `va_list` and represents the sequence of `arg` parameters originally passed to SYS\_abort.

```
(*(Abort_function))(format, vargs)
```

The function bound to Abort function may 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**

**Syntax** success = SYS\_atexit(handler);

**Parameters** Fxn handler /\* exit handler function \*/

**Return Value** Bool success /\* handler successfully stacked \*/

**Assembly Interface** none

**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 DSP/BIOS plug-ins.)

**Constraints and Calling Context**

- handler cannot be NULL.
- SYS\_atexit cannot be called from ISRs.

**SYS\_error***Flag error condition***C Interface**

**Syntax**                   SYS\_error(s, errno, [arg], ...);

**Parameters**               String s;       /\* error string \*/  
                               Int    errno;   /\* error code \*/  
                               Arg   arg;     /\* optional argument \*/

**Return Value**            Void

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

SYS\_error is used to flag DSP/BIOS error conditions. Application programs as well as internal functions use SYS\_error to handle program errors.

SYS\_error calls the function bound to Error function to handle errors.

The default Error function for the SYS manager is \_UTL\_doError, which logs an error message, disables interrupts, and then runs in an infinite loop.

**Constraints and Calling Context**

- ❑ The only valid error numbers are the error constants defined in sys.h (SYS\_E\*) or numbers greater than or equal to SYS\_EUSER. Passing any other error values to SYS\_error may cause DSP/BIOS to crash.
- ❑ The string passed to SYS\_error must be non-NULL.

**SYS\_exit***Terminate program execution***C Interface**

<b>Syntax</b>	SYS_exit(status);
<b>Parameters</b>	Int status; /* termination status code */
<b>Return Value</b>	Void

**Assembly Interface** none**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 \*/  
String buffer; /\* output buffer \*/  
Arg arg; /\* optional argument \*/  
va\_list vargs; /\* variable argument list reference \*/

**Return Value**       Void

**Assembly Interface**   none

**Description**

SYS\_printf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_printf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_printf are limited to the characters in the following table.

<b>Character</b>	<b>Corresponding Output Format</b>
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

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may 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_vprintf`  
`SYS_sprintf`  
`SYS_vsprintf`

**SYS\_sprintf** *Output formatted data***C Interface**

**Syntax**                   SYS\_sprintf (buffer, format, [arg,] ...);

**Parameters**           String format; /\* format specification string \*/  
String buffer; /\* output buffer \*/  
Arg arg; /\* optional argument \*/  
va\_list vargs; /\* variable argument list reference \*/

**Return Value**       Void

**Assembly Interface**   none

**Description**

SYS\_sprintf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_sprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS\_printf are limited to the characters in the following table.

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

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may 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**

<b>Syntax</b>	SYS_vprintf(format, vars);
<b>Parameters</b>	String format; /* format specification string */ String buffer; /* output buffer */ Arg arg; /* optional argument */ va_list vars; /* variable argument list reference */
<b>Return Value</b>	Void

**Assembly Interface** none

**Description**

SYS\_printf 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\_printf are limited to the characters in the following table.

<b>Character</b>	<b>Corresponding Output Format</b>
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

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may 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\_vsprintf

**SYS\_vsprintf** *Output formatted data***C Interface**

**Syntax**                   SYS\_vsprintf(buffer, format, vargs);

**Parameters**           String format; /\* format specification string \*/  
String buffer; /\* output buffer \*/  
Arg arg; /\* optional argument \*/  
va\_list vargs; /\* variable argument list reference \*/

**Return Value**       Void

**Assembly Interface**   none

**Description**

SYS\_printf provides a subset of the capabilities found in the standard C library function printf.

**Note:**

SYS\_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\_printf are limited to the characters in the following table.

<b>Character</b>	<b>Corresponding Output Format</b>
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

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may 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\_vsprintf 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>	SYS_putchar(c);
<b>Parameters</b>	Char c; /* next output character */
<b>Return Value</b>	Void

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

SYS\_putchar outputs the character c by calling the system-dependent function bound to the configuration parameter Putc function.

```
((Putc function))(c)
```

For systems with limited I/O capabilities, the function bound to Putc function might simply place c into a global buffer that can be examined after program termination.

The default Putc function for the SYS manager is \_UTL\_doPutc, which writes a character to the trace buffer.

SYS\_putchar is also used internally by SYS\_printf and SYS\_vprintf when generating their output.

**Constraints and Calling Context**

- ❑ If the function bound to Putc function is not reentrant, SYS\_putchar must be called atomically.

**See Also**

SYS\_printf

**TRC Module***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.

The following events and statistics can be traced. The constants defined in trc.h and trc.h62 are shown in the left column:

Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Log timer interrupts	off
TRC_LOGPRD	Log periodic ticks and start of periodic functions	off
TRC_LOGSWI	Log events when a software interrupt is posted and completes	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes execution, and terminates	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	off
TRC_USER0 and TRC_USER1	Your program can use these bits to enable or disable sets of explicit instrumentation actions. You can use TRC_query to check the settings of these bits and either perform or omit instrumentation calls based on the result. DSP/BIOS does not use or set these bits.	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 bit is usually set at run time on the host in the RTA Control Panel.	off

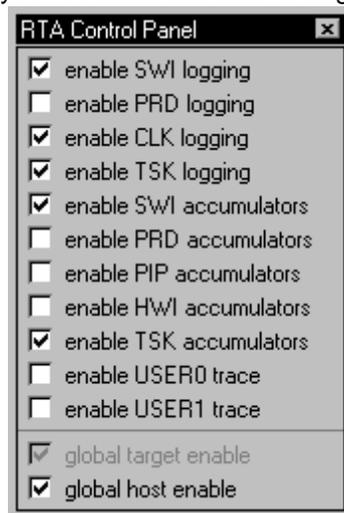
Constant	Tracing Enabled/Disabled	Default
TRC_GBLTARG	This bit must also be set in order for any implicit instrumentation to be performed. This bit can only be set by the target program and is enabled by default.	on
TRC_STSSWI	Gather statistics on length of SWI execution	off

All trace constants except TRC\_GBLTARG are switched off initially. To enable tracing you can use calls to TRC\_enable or the Tools→DSP/BIOS→RTA Control Panel, which uses the TRC module internally. You do not need to enable tracing for messages written with LOG\_printf or LOG\_event and statistics added with STS\_add or STS\_delta.

Your program can call the TRC\_enable and TRC\_disable operations to explicitly start and stop event logging or statistics accumulation in response to conditions encountered during real-time execution. This enables you to preserve the specific log or statistics information you need to see.

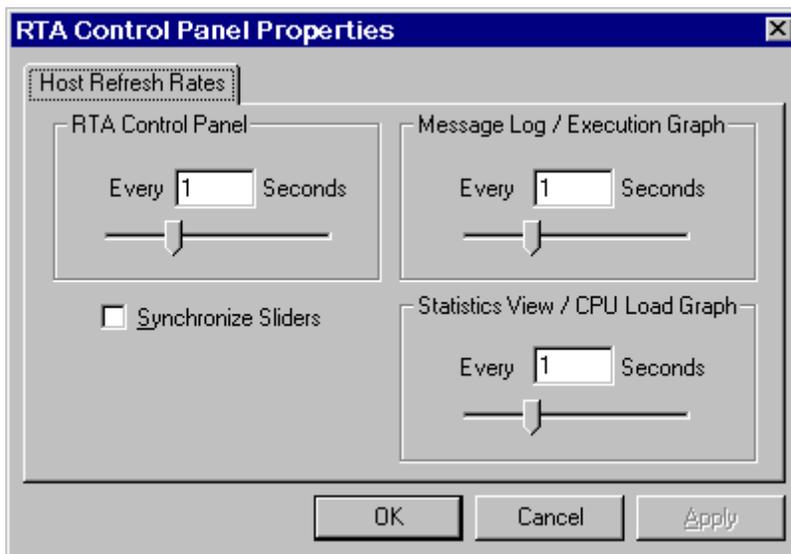
### TRC - Code Composer Studio Interface

You can choose Tools→DSP/BIOS→RTA Control Panel to open a window that allows you to control run-time tracing.



Once you have enabled tracing, you can use Tools→DSP/BIOS→Execution Graph and Tools→DSP/BIOS→Event Log to see log information, and Tools→DSP/BIOS→Statistics View to see statistical information.

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



See the *TMS320C6000 Code Composer Studio Tutorial* for more information on how to enable tracing in the RTA Control Panel.

**TRC\_disable***Disable trace class(es)***C Interface**

<b>Syntax</b>	TRC_disable(mask);
<b>Parameters</b>	Uns mask; /* trace type constant mask */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	TRC_disable mask
<b>Inputs</b>	mask (see the TRC Module for a list of constants to use in the mask)
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...) b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a2, a4
<b>Reentrant</b>	no

**Description**

TRC\_disable disables tracing of one or more trace types. Trace types are specified with a 32-bit 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.

The full list of constants you can use to disable tracing is included in the description of the TRC module.

For example, you might want to use TRC\_disable with a circular log and disable tracing when an unwanted condition occurs. This allows test equipment to retrieve the log events that happened just before this condition started.

**See Also**

TRC\_enable  
TRC\_query  
LOG\_printf  
LOG\_event  
STS\_add  
STS\_delta

**TRC\_enable** *Enable trace type(s)***C Interface**

<b>Syntax</b>	TRC_enable(mask);
<b>Parameters</b>	Uns mask; /* trace type constant mask */
<b>Return Value</b>	Void

**Assembly Interface**

<b>Syntax</b>	TRC_enable mask
<b>Inputs</b>	mask (see the TRC Module for a list of constants to use in the mask)
<b>Preconditions</b>	constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...) b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	none
<b>Modifies</b>	a2, a4
<b>Reentrant</b>	no

**Description**

TRC\_enable enables tracing of one or more trace types. Trace types are specified with a 32-bit mask. The following C code would enable tracing of statistics for software interrupts and periodic functions:

```
TRC_enable(TRC_STSSWI | TRC_STSPRD);
```

Internally, DSP/BIOS uses a bitwise OR operation to enable multiple trace types.

The full list of constants you can use to enable tracing is included in the description of the TRC module.

For example, you might want to use TRC\_enable with a fixed log to enable tracing when a specific condition occurs. This allows test equipment to retrieve the log events that happened just after this condition occurred.

**See Also**

TRC\_disable  
TRC\_query  
LOG\_printf  
LOG\_event  
STS\_add  
STS\_delta

**TRC\_query***Query trace class(es)***C Interface**

<b>Syntax</b>	result = TRC_query(mask);
<b>Parameters</b>	Uns mask; /* trace type constant mask */
<b>Return Value</b>	Int result /* indicates whether all trace types enabled */

**Assembly Interface**

<b>Syntax</b>	TRC_query mask
<b>Inputs</b>	mask (see the TRC Module for a list of constants to use in the mask)
<b>Preconditions</b>	constant - mask for trace types b14 = address of the start of .bss amr = 0
<b>Postconditions</b>	a4 == 0 if all queried trace types enabled a4 != 0 if any of the queried trace types are disabled
<b>Modifies</b>	a2, a4
<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.

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, the following C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)    /* returns 0 */
result = TRC_query(TRC_LOGPRD)    /* returns non-zero */
```

However, if only the TRC\_GBLHOST and TRC\_LOGSWI bits are set, the same C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)    /* returns non-zero */
result = TRC_query(TRC_LOGPRD)    /* returns non-zero */
```

**See Also**

TRC\_enable  
TRC\_disable

**TSK Module***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\_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

**Constants, Types, and Structures**

```

typedef struct TSK_OBJ *TSK_Handle;
                        /* handle for task object */

struct TSK_Attrs {      /* task attributes */
    Int    priority;    /* execution priority */
    Ptr    stack;       /* pre-allocated stack */
    Uns    stacksize;   /* stack size in bytes */
    Int    stackseg;    /* memory seg for stack allocation */
}

```

```

    Ptr    environ; /* global environment data structure */
    String name;    /* printable name */
    Bool   exitflag; /* program termination requires this */
                    /* task to terminate */
    TSK_DBG_Mode debug /* indicates enum type TSK_DBG_YES, */
                    /* TSK_DBG_NO or TSK_DBG_MAYBE */
};

Int TSK_pid; /* MP processor ID */

Int TSK_MAXARGS = 8; /* maximum number of task arguments */
Int TSK_IDLEPRI = 0; /* used for idle task */
Int TSK_MINPRI = 1; /* minimum execution priority */
Int TSK_MAXPRI = 15; /* maximum execution priority */
Int TRG_STACKSTAMP = 0xBEBEBEBE
TSK_Attrs TSK_ATTRS = { /* default attribute values */
    TSK->PRIORITY, /* priority */
    NULL, /* stack */
    TSK->STACKSIZE, /* stacksize */
    TSK->STACKSEG, /* stackseg */
    NULL, /* environ */
    "", /* name */
    TRUE, /* exitflag */
};

enum TSK_Mode { /* task execution modes */
    TSK_RUNNING, /* task is currently executing */
    TSK_READY, /* task is scheduled for execution */
    TSK_BLOCKED, /* task is suspended from execution */
    TSK_TERMINATED, /* task is terminated from execution */
};

struct TSK_Stat { /* task status structure */
    TSK_Attrs attrs; /* task attributes */
    TSK_Mode mode; /* task execution mode */
    Ptr sp; /* task stack pointer */
    Uns used; /* task stack used */
};

```

## Description

The TSK module makes available a set of functions that manipulate task objects accessed through handles of type `TSK_Handle`. Tasks represent independent threads of control that conceptually execute functions in parallel within a single C program; in reality, concurrency is achieved by switching the processor from one task to the next.

When you create each task, it is provided with its own run-time stack, used for storing local variables as well as for further nesting of function calls. The `TRG_STACKSTAMP` value is used to initialize the run-time stack. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher-priority task. All tasks executing within a single program

share a common set of global variables, accessed according to the standard rules of scope defined for C functions.

Each task is in one of four modes of execution at any point in time: running, ready, blocked, or terminated. By design, there is always one (and only one) task currently running, even if it is a dummy idle task managed internally by TSK. The current task can be suspended from execution by calling certain TSK functions, as well as functions provided by other modules like SEM and SIO; the current task can also terminate its execution. In either case, the processor is switched to the next task that is ready to run.

You can assign numeric priorities to tasks through TSK. Tasks are readied for execution in strict priority order; tasks of the same priority are scheduled on a first-come, first-served basis. As a rule, the priority of the currently running task is never lower than the priority of any ready task. Conversely, the running task is preempted and re-scheduled for execution whenever there exists some ready task of higher priority.

The user-definable function pointer configuration parameters Create function, Delete function, and Exit function are described on the TSK\_create, TSK\_delete, and TSK\_exit manual pages, respectively.

Switch function, if not NULL, is invoked during a task switch giving the application access to both the current and next task handles at task switch time:

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

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

The functions attached to the Switch function and Ready function are called from within the kernel and may make only those function calls allowed from within a software interrupt handler. See Appendix A, Function Callability and Error Tables, for a list of functions that can be called within the Kernel. There are no real constraints on what functions are called via Create function, Delete function, or Exit function since these are invoked outside the kernel.

## TSK Manager Properties

The following global properties can be set for the TSK module:

- Enable TSK Manager.** If no tasks are used by the program other than TSK\_idle, you can optimize the program by disabling the task manager. The program must then not use TSK objects created with either the Configuration Tool or the TSK\_create function. If the task manager is disabled, the idle loop still runs and uses the application stack instead of a task stack.
- Object Memory.** The memory section that contains the TSK objects created with the Configuration Tool.
- Default stack size.** The default size of the stack (in MAUs) used by tasks. You can override this value for an individual task you create with the Configuration Tool or TSK\_create. The estimated minimum task size is shown in the status bar of the Configuration Tool. This property applies to TSK objects created both with the Configuration Tool and with TSK\_create.
- Default stack segment for dynamic tasks.** The memory section where the task's stack is placed. You can override this value for an individual task. This property applies to TSK objects created both with the Configuration Tool and with TSK\_create.
- Default task priority.** The default priority level for tasks that are created dynamically with TSK\_create. This property applies to TSK objects created both with the Configuration Tool and with TSK\_create.
- Create function.** The name of a function to call when any task is created at run-time with TSK\_create. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- Delete function.** The name of a function to call when any task is deleted at run-time with TSK\_delete. If this function is written in C, use a leading underscore before the C function name.
- Exit function.** The name of a function to call when any task exits. If this function is written in C, use a leading underscore before the C function name.
- Call switch function.** Check this box if you want a function to be called when any task switch occurs.
- Switch function.** The name of a function to call when any task switch occurs. This function can give the application access to both the current

and next task handles. If this function is written in C, use a leading underscore before the C function name.

- Call ready function.** Check this box if you want a function to be called when any task becomes ready to run.
- Ready function.** The name of a function to call when any task becomes ready to run. If this function is written in C, use a leading underscore before the C function name.
- Statistics Units.** The units used within DSP/BIOS plug-ins to display the elapsed instruction cycles or time from when a task is made ready to run until a call like the following is made:

```
TSK_deltatime(TSK_self())
```

Choose Raw causes the STS Data window to display the number of instruction cycles if the CLK module's Use high resolution time for internal timings parameter is set to True (the default). If this CLK parameter is set to False and the Statistics Units is set to Raw, TSK statistics are displayed in units of timer interrupt periods. You can also choose milliseconds or microseconds.

## TSK Object Properties

The following properties can be set for a TSK object:

- comment.** A comment to identify this TSK object.
- Task function.** The function to be executed when the task runs. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- Task function argument 0-7.** The arguments to pass to the task function. Arguments may be integers or labels. For labels defined in a C program, add a leading underscore before the label name.
- Automatically allocate stack.** Check this box if you want the task's private stack space to be allocated automatically when this task is created. The task's context is saved in this stack before any higher-priority task is allowed to block this task and run.
- Manually allocated stack.** If you did not check the box to Automatically allocate stack, type the name of the manually allocated stack to use for this task. If the stack is defined in a C program, add a leading underscore before the stack name.
- Stack size.** If you checked the box to Automatically allocate stack, type the size (in MAUs) of the stack space to allocate for this task. Each stack must be large enough to handle normal subroutine calls as well as a

single task preemption context. 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.

- Stack Memory Segment.** If you checked the box to Automatically allocate stack, select the memory section to contain the stack space for this task.
- Priority.** The priority level for this task.
- Environment pointer.** A pointer to a globally defined data structure that this task may access. The task can get and set the task environment pointer with the TSK\_getenv and TSK\_setenv functions. If this structure is defined in C, use a leading underscore before the structure name.
- Don't shut down system while this task is still running.** Check this box if you do not want the application to be able to end if this task is still running. The application can still abort. For example, you might clear this box for a monitor task that collects data whenever all other tasks are blocked. The application does not need to explicitly shut down this task.
- Allocate Task Name on Target.** Check this box if you want the name of this TSK object to be retrievable by the TSK\_getname function. Clearing this box saves a small amount of memory. The task name is available in DSP/BIOS plug-ins in either case.

## TSK - DSP/BIOS Plug-ins Interface

The TSK tab of the Kernel/Object View shows information about task objects.

To enable TSK logging, choosing Tools→DSP/BIOS→RTA Control Panel and check the appropriate box. Then you can open the system log by choosing View→System Log. You see a graph of activity that includes TSK function execution states.

Only TSK objects created with the Configuration Tool are traced. The System Log graph includes time spent performing dynamically created TSK functions in the Other Threads row.

You can also enable TSK accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a TSK object, you see statistics about the time elapsed from the time the TSK was posted until TSK\_deltatime(TSK\_self) is called.

**TSK\_checkstacks***Check for stack overflow***C Interface**

<b>Syntax</b>	TSK_checkstacks(oldtask, newtask);
<b>Parameters</b>	TSK_Handle oldtask; /* handle of task switched from */ TSK_Handle newtask; /* handle of task switched to */
<b>Return Value</b>	Void

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

TSK\_checkstacks calls SYS\_abort with an error message if either oldtask or newtask has a stack in which the last location no longer contains the initial value TRG\_STACKSTAMP. The presumption in one case is that oldtask's stack overflowed, and in the other that an invalid store has corrupted newtask's stack.

You may 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 Switch function, you may 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);
    ...
}
```

**TSK\_create***Create a task ready for execution***C Interface****Syntax**`task = TSK_create(fxn, attrs, [arg,] ...);`**Parameters**

```
Fxn      fxn;      /* entry point of the task */
TSK_Attrs *attrs;  /* pointer to task attributes */
Arg      arg;      /* task arguments */
```

**Return Value**`TSK_Handle task; /* task object handle */`**Assembly Interface**

none

**Description**

TSK\_create creates a new task object. If successful, TSK\_create returns the handle of the new task object. If unsuccessful, TSK\_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS\_error, and SYS\_error is configured to abort).

Create\_function, the system specific task create function, has a default value of SYS\_nop. This function is called after the task handle has been initialized but before the task has been placed on its ready queue. It is called with the task handle as its only parameter. Any DSP/BIOS function may be called from Create\_function.

```
(*(Create_function))(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 */
```

TSK\_exit is automatically called if and when the task returns from fxn.

If attrs is NULL, the new task is assigned a default set of attributes. Otherwise, the task's attributes are specified through a structure of type TSK\_Attrs defined as follows:

```
struct TSK_Attrs {
    Int      priority;
    Ptr      stack;
    Uns      stacksize;
    Uns      stackseg;
    Ptr      environ;
    String   name;
    Bool     exitflag;
};
```

The priority attribute specifies the task's execution priority and must be less than or equal to TSK\_MAXPRI (15); this attribute defaults to the value of the

configuration parameter Default task priority (preset to TSK\_MINPRI). If priority is less than 0, task is barred from execution until its priority is raised at a later time by another task. A priority value of 0 is reserved for the TSK\_idle task defined in the default configuration. You should not use a priority of 0 for any other tasks.

The stack attribute specifies a pre-allocated block of stacksize bytes 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 section given by the stackseg attribute.

The stacksize attribute specifies the number of bytes to be allocated for the task's private stack; this attribute defaults to the value of the configuration parameter Default stack size (preset to 1024). Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. 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 section 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 or not the task must terminate before the program as a whole can terminate; this attribute defaults to TRUE.

All default attribute values are contained in the constant TSK\_ATTRS, which may 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.

## Constraints and Calling Context

- ❑ TSK\_create cannot be called by ISRs.
- ❑ 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 may 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 section.
- ❑ Task arguments passed to TSK\_create cannot be greater than 32 bits in length; i.e., 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 the Configuration Tool rather than using the XXX\_create functions.

**See Also**

MEM\_alloc  
SYS\_error  
TSK\_delete  
TSK\_exit

**TSK\_delete***Delete a task***C Interface**

<b>Syntax</b>	TSK_delete(task);
<b>Parameters</b>	TSK_Handle task; /* task object handle */
<b>Return Value</b>	Void

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

TSK\_delete removes the task from all internal queues and calls MEM\_free to free the task object and stack. task should be in a state that does not violate any of the listed constraints.

If all remaining tasks have their exitflag attribute set to FALSE, DSP/BIOS terminates the program as a whole by calling SYS\_exit with a status code of 0.

Delete\_function, the system specific task delete function, has a default value of SYS\_nop. This 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 may be called from \*(Delete\_function).

```
(*(Delete_function))(task)
```

No task switch occurs when calling TSK\_delete.

**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 by ISRs.
- No check is performed to prevent TSK\_delete from being used on a statically-created object. If a program attempts to delete a task object that was created using the Configuration Tool, SYS\_error is called.

**See Also**

MEM\_free  
TSK\_create

**TSK\_deltatime**

*Update task statistics with difference between current time and time task was made ready*

**C Interface**

<b>Syntax</b>	TSK_deltatime(task);
<b>Parameters</b>	TSK_Handle task; /* task object handle */
<b>Return Value</b>	Void

**Assembly Interface** none

**Description**

This function accumulates the time difference from when a task is made ready to the time TSK\_deltatime is called. These time differences are accumulated in the task's internal STS object and can be used to determine whether or not a task misses real-time deadlines.

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 may look something like the following:

```
/* 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 the task's STS object */
    TSK_deltatime(TSK_self());
}
```

Assuming this task blocks on SIO\_get, the device driver posts a semaphore that readies the task. DSP/BIOS sets the time value of the STS object every time the task is made ready to run, so TSK\_settime is only called once when the task is starting up.

**See Also**

TSK\_getsts  
TSK\_settime

**TSK\_disable***Disable DSP/BIOS task scheduler***C Interface****Syntax** TSK\_disable();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_disable disables the DSP/BIOS task scheduler. The current task continues to execute (even if a higher priority task may become ready to run) until TSK\_enable is called.

TSK\_disable does not disable interrupts, but is instead used before disabling interrupts to make sure a context switch to another task does not occur when interrupts are disabled.

TSK\_disable handlers are disabled. TSK\_disable maintains a count which allows nested calls to TSK\_disable. Task switching is not reenabled until TSK\_enable has been called as many times as TSK\_disable. Calls to TSK\_disable may be nested.

Since TSK\_disable may prohibit ready tasks of higher priority from running it should not be used as a general means of mutual exclusion—SEM semaphores should be used for mutual exclusion when possible.

**Constraints and Calling Context**

- ❑ No kernel operations that may cause the current task to block (e.g., SEM\_pend, TSK\_sleep, TSK\_yield) can be made from within a TSK\_disable / TSK\_enable block.
- ❑ TSK\_yield cannot be called within a TSK\_disable / TSK\_enable block.

**See Also**

SEM Module  
TSK\_enable

**TSK\_enable** *Enable DSP/BIOS scheduler***C Interface****Syntax** TSK\_enable();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_enable is used to reenable the DSP/BIOS task scheduler after TSK\_disable has been called. Since TSK\_disable calls may be nested, the task scheduler is not enabled until TSK\_enable is called the same number of times as TSK\_disable.

A task switch occurs when calling TSK\_enable only if there exists a TSK\_READY task whose priority is greater than the currently executing task.

**Constraints and Calling Context**

- ❑ No kernel operations that may cause the current task to block (e.g., SEM\_pend, TSK\_sleep, TSK\_yield) can be made from within a TSK\_disable / TSK\_enable block.

**See Also**

SEM Module  
TSK\_disable  
C62\_enableIER

**TSK\_exit***Terminate execution of the current task***C Interface****Syntax** TSK\_exit();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_exit terminates execution of the current task, changing its mode from TSK\_RUNNING to TSK\_TERMINATED. If all tasks have been terminated, or if all remaining tasks have their exitflag attribute set to FALSE, then DSP/BIOS terminates the program as a whole by calling the function SYS\_exit with a status code of 0.

TSK\_exit is automatically called whenever a task returns from its top-level function.

Exit\_function, the system specific task exit function, has a default value of SYS\_nop. This function is called before the task has been blocked and marked TSK\_TERMINATED. Any DSP/BIOS function may be called from \*(Exit\_function).

```
(*(Exit_function))()
```

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 by ISRs.

**See Also**

MEM\_free  
 TSK\_create  
 TSK\_delete

## **TSK\_getenv**

*Get task environment pointer*

### **C Interface**

**Syntax**                    `environ = TSK_getenv(task);`

**Parameters**                `TSK_Handle task;    /* task object handle */`

**Return Value**              `Ptr        environ; /* task environment pointer */`

**Assembly Interface**        `none`

### **Description**

TSK\_getenv returns the environment pointer of task. The environment pointer, environ, references an arbitrary application-defined data structure.

### **See Also**

TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_geterr***Get task error number***C Interface****Syntax**                    `errno = TSK_geterr(task);`**Parameters**                `TSK_Handle task; /* task object handle */`**Return Value**              `Int      errno; /* error number */`**Assembly Interface**        `none`**Description**

Each task carries a task-specific error number. This number is initially `SYS_OK`, but it can be changed by `TSK_seterr`. `TSK_geterr` returns the current value of this number.

**See Also**

`SYS_error`  
`TSK_setenv`  
`TSK_seterr`  
`TSK_setpri`

**TSK\_getname***Get task name***C Interface****Syntax**                   name = TSK\_getname(task);**Parameters**               TSK\_Handle task; /\* task object handle \*/**Return Value**             String    name; /\* task name \*/**Assembly Interface**      none**Description**

TSK\_getname returns the task's name.

For tasks created with the Configuration Tool, the name is available to this function only if the Allocate Task Name on Target box is checked in the properties for this task. For tasks created with TSK\_create, TSK\_getname returns the attrs.name field value, or an empty string if this attribute was not specified.

**See Also**

TSK\_setenv

TSK\_seterr

TSK\_setpri

**TSK\_getpri***Get task priority***C Interface****Syntax**                    priority = TSK\_getpri(task);**Parameters**                TSK\_Handle task;    /\* task object handle \*/**Return Value**            Int        priority; /\* task priority \*/**Assembly Interface**      none**Description**

TSK\_getpri returns the priority of task.

**See Also**TSK\_setenv  
TSK\_seterr  
TSK\_setpri

**TSK\_getsts***Get the handle of the task's STS object***C Interface****Syntax**`sts = TSK_getsts(task);`**Parameters**`TSK_Handle task; /* task object handle */`**Return Value**`STS_Handle sts; /* statistics object handle */`**Assembly Interface**

none

**Description**

This function provides access to the task's internal STS object. For example, you may want the program to check the maximum value to see if it has exceeded some value.

**See Also**`TSK_deltatime`  
`TSK_settime`

**TSK\_itick***Advance the system alarm clock (interrupt use only)***C Interface****Syntax** TSK\_itick();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_itick increments the system alarm clock, and readies any tasks blocked on TSK\_sleep or SEM\_pend whose timeout intervals have expired.

**Constraints and Calling Context**

- ❑ TSK\_itick may be invoked only inside a hardware ISR.

**See Also**

SEM\_pend  
TSK\_sleep  
TSK\_tick

**TSK\_self***Returns handle to the currently executing task***C Interface****Syntax**

curtask = TSK\_self();

**Parameters**

Void

**Return Value**

TSK\_Handle curtask; /\* handle for current task object \*/

**Assembly Interface**

none

**Description**

TSK\_self returns the object handle for the currently executing task. This function is useful when inspecting the object or when the current task changes its own priority through TSK\_setpri.

No task switch occurs when calling TSK\_self.

**See Also**

TSK\_setpri

**TSK\_setenv** *Set task environment***C Interface**

**Syntax** TSK\_setenv(task, environ);

**Parameters** TSK\_Handle task; /\* task object handle \*/  
Ptr environ; /\* task environment pointer \*/

**Return Value** Void

**Assembly Interface** none

**Description**

TSK\_setenv sets the task environment pointer to environ. The environment pointer, environ, references an arbitrary application-defined data structure.

**See Also**

TSK\_getenv  
TSK\_geterr

**TSK\_seterr***Set task error number***C Interface****Syntax** TSK\_seterr(task, errno);**Parameters** TSK\_Handle task; /\* task object handle \*/  
Int errno; /\* error number \*/**Return Value** Void**Assembly Interface** none**Description**

Each task carries a task-specific error number. This number is initially SYS\_OK, but can be changed to errno by calling TSK\_seterr. TSK\_geterr returns the current value of this number.

**See Also**TSK\_getenv  
TSK\_geterr

**TSK\_setpri***Set a task's execution priority***C Interface**

**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 */`

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

TSK\_setpri sets the execution priority of task to newpri, and returns that task's old priority value. Raising or lowering a task's priority does not necessarily force preemption and re-scheduling of the caller: tasks in the TSK\_BLOCKED mode remain suspended despite a change in priority; and tasks in the TSK\_READY mode gain control only if their (new) priority is greater than that of the currently executing task.

The maximum value of newpri is TSK\_MAXPRI(15). If the minimum value of newpri is TSK\_MINPRI(0). If newpri is less than 0, task is barred from further execution until its priority is raised at a later time by another task; if newpri equals TSK\_MAXPRI, execution of task effectively locks out all other program activity, except for the handling of interrupts.

The current task can change its own priority (and possibly preempt its execution) by passing the output of TSK\_self as the value of the task parameter.

A task switch occurs when calling TSK\_setpri only if there exists some TSK\_READY task whose priority is greater than the currently executing task. This is important and allows TSK\_setpri to 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.
- TSK\_setpri cannot be called from a DSP/BIOS SWI.

**See Also**

TSK\_self  
 TSK\_sleep

**TSK\_settime***Reset task statistics previous value to current time***C Interface**

<b>Syntax</b>	TSK_settime(task);
<b>Parameters</b>	TSK_Handle task; /* task object handle */
<b>Return Value</b>	Void

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

This function initializes the previous time in the task's STS object. It can be used after a task starts running, but before it enters a processing loop. TSK\_settime effectively does an STS\_set operation to the task's internal STS object using the current time.

DSP/BIOS sets the time value of the STS object every time the task is made ready to run, so TSK\_settime should only called once when the task is starting up.

For example, a loop within the task may 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());
    }
}
```

Assuming this task blocks on SIO\_get, the device driver posts a semaphore that readies the task.

**See Also**

TSK\_deltatime  
TSK\_getsts

**TSK\_sleep***Delay execution of the current task***C Interface**

<b>Syntax</b>	TSK_sleep(nticks);
<b>Parameters</b>	Uns nticks; /* number of system clock ticks to sleep */
<b>Return Value</b>	Void

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

TSK\_sleep changes the current task's mode from TSK\_RUNNING to TSK\_BLOCKED, and delays its execution for nticks increments of the system clock. The actual time delayed can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

After the specified period of time has elapsed, the task reverts to the TSK\_READY mode and is scheduled for execution.

A task switch always occurs when calling TSK\_sleep if nticks > 0.

**Constraints and Calling Context**

- TSK\_sleep cannot be called by ISRs, or within a TSK\_disable / TSK\_enable block.
- nticks cannot be SYS\_FOREVER.
- TSK\_sleep should not be called from within an IDL function. Doing so prevents DSP/BIOS plug-ins from gathering run-time information.

**TSK\_stat***Retrieve the status of a task***C Interface**

**Syntax** TSK\_stat(task, statbuf);

**Parameters** TSK\_Handle task; /\* task object handle \*/  
TSK\_Stat \*statbuf; /\* pointer to task status structure \*/

**Return Value** Void

**Assembly Interface** none

**Description**

TSK\_stat retrieves attribute values and status information about task; the current task can inquire about itself by passing the output of TSK\_self as the first argument to TSK\_stat.

Status information is returned through statbuf, which references a structure of type TSK\_Stat defined as follows:

```
struct TSK_Stat { /* task status structure */
    TSK_Attrs attrs; /* task attributes */
    TSK_Mode mode; /* task execution mode */
    Ptr sp; /* task's current stack pointer */
    Uns used; /* max number of words ever used on the */
                /* task stack */
};
```

When a task is preempted by a software or hardware interrupt, the task execution mode returned for that task by TSK\_stat is still TSK\_RUNNING because the task will run when the preemption ends.

**Constraints and Calling Context**

- ❑ statbuf cannot be NULL.

**See Also**

TSK\_create

**TSK\_tick***Advance the system alarm clock***C Interface****Syntax** TSK\_tick();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_tick increments the system clock, and readies any tasks blocked on TSK\_sleep or SEM\_pend whose timeout intervals have expired. TSK\_tick may be invoked by an ISR or by the currently executing task. The latter is particularly useful for testing timeouts in a controlled environment.

A task switch occurs when calling TSK\_tick if the priority of any of the readied tasks is greater than the priority of the currently executing task.

TSK\_tick may also be called from an ISR but TSK\_itick is more efficient. TSK\_tick behaves differently dependent upon whether it is called from an ISR or from a task: TSK\_itick has reduced overhead since it assumes it is called from an ISR and does not run a check.

**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 curtime; /* current time */

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

TSK\_time returns the current value of the system alarm clock.

Note that since the system clock is usually updated asynchronously by an interrupt service routine (via TSK\_itick or TSK\_tick), curtime may lag behind the actual system time. This lag may be even greater if a higher priority task preempts the current task between the call to TSK\_time and when its return value is used. Nevertheless, TSK\_time is useful for getting a rough idea of the current system time.

**TSK\_yield***Yield processor to equal priority task***C Interface****Syntax** TSK\_yield();**Parameters** Void**Return Value** Void**Assembly Interface** none**Description**

TSK\_yield yields the processor to another task of equal priority.

A task switch occurs when you call TSK\_yield if there is an equal priority task ready to run.

**Constraints and Calling Context**

- ❑ TSK\_yield cannot be called by ISRs.

**See Also**

TSK\_sleep

## C library `stdlib.h`

### *DSP/BIOS standard C library functions*

#### Syntax

```
#include <stdlib.h>    /* supplied with the C compiler */
```

#### Functions

```
int  atexit(void (*fcn)(void));
void *calloc(size_t nobj, size_t size);
void  exit(int status);
void  free(void *p);
char *getenv(char *name);
void *malloc(size_t size);
void *realloc(void *p, size_t size);
```

#### Description

The DSP/BIOS library contains some C standard library functions which supersede the library functions bundled with the C compiler. These functions follow the ANSI C specification for parameters and return values. Consult Kernighan and Ritchie for a complete description of these functions.

The functions `calloc`, `free`, `malloc`, and `realloc` use `MEM_alloc` and `MEM_free` (with `segid = Segment` for `malloc/free`) to allocate and free memory.

`getenv` uses the `_environ` variable defined and initialized in the boot file to search for a matching environment string.

`exit` calls the exit functions registered by `atexit` before calling `SYS_exit`.

# Utility Programs

---

---

---

This chapter provides documentation for utilities that can be used to examine various files from the MS-DOS command line. These programs are provided with DSP/BIOS in the bin subdirectory.

Topic	Page
<b>2.1 cdbprint</b> .....	<b>2-2</b>
<b>2.2 gconfgn</b> .....	<b>2-3</b>
<b>2.3 nmti</b> .....	<b>2-5</b>
<b>2.4 sectti</b> .....	<b>2-6</b>
<b>2.5 size62</b> .....	<b>2-7</b>
<b>2.6 vers</b> .....	<b>2-8</b>

**2.1 cdbprint**

*Prints a listing of all parameters defined in a configuration file*

**Syntax**

`cdbprint [-a] [-l] [-w] cdb-file`

**Description**

This utility reads a .cdb file created with the Configuration Tool and creates a list of all the objects and parameters. This tool can be used to compare two configuration files or to simply review the values of a single configuration file.

The -a flag causes cdbprint to list all objects and fields including those that are normally not visible (i.e., unconfigured objects and hidden fields). Without this flag, cdbprint ignores unconfigured objects or modules as well as any fields that are hidden.

The -l flag causes cdbprint to list the internal parameter names instead of the labels used by the Configuration Tool. Without this flag, cdbprint lists the labels used by the Configuration Tool.

The -w flag causes cdbprint to list only those parameters that can also be modified in the Configuration Tool. Without this flag, cdbprint lists both read-only and read-write parameters.

**Example**

The following sequence of commands can be used to compare a configuration file called test62.cdb to the default configuration provided with DSP/BIOS:

```
cdbprint ../../include/bios62.cdb > original.txt
cdbprint test62.cdb > test62.txt
diff original.txt test62.txt
```

## 2.2 gconfgen *Reads a reads a .cdb file created with the Configuration Tool*

**Syntax** gconfgen cdb-file

### Description

This command line utility reads a .cdb file (e.g. program.cdb) created with the Configuration Tool and generates the target configuration files (programcfg.cmd, programcfg.h<54><62>, programcfg.s<54><62>) that are linked with the rest of the application code.

This utility is useful when the build process is controlled by a scripted mechanism, such as a make file, to generate the configuration source files from the configuration database file (.cdb file). Caution should be used, however, following product upgrades, since gconfgen does not detect revision changes. After a product update, use the graphical Configuration Tool to update your .cdb files to the new version. Once updated, gconfgen can be used again to generate the target configuration files.

### Example

You can use gconfgen from a make file, as shown in the following example, from the makefiles provided with the DSP/BIOS examples in the product distribution. To use gconfgen from the command line or makefiles, use its full path (TI\_DIR\plugins\bios\gconfgen) or add its folder (TI\_DIR\plugins\bios) to your PATH environment variable. (Note that TI\_DIR is the root directory of the product distribution).

```
\*
* Makefile for creation of program named by the PROG variable
* The following naming conventions are used by this makefile:

* <prog>.asm      - C62 assembly language source file
* <prog>.obj      - C62 object file (compiled/assembled source)
* <prog>.out      - C62 executable (fully linked program)
* <prog>cfg.s62   - configuration assembly source file generated
*                 by Configuration Tool
* <prog>cfg.h62   - configuration assembly header file generated
*                 by Configuration Tool
* <prog>cfg.cmd   - configuration linker command file generated
*                 by Configuration Tool
*

TI_DIR := $(subst \,/,$(TI_DIR))
include $(TI_DIR)/c6000/bios/include/c62rules.mak

*
* Compiler, assembler, and linker options.
*
* -g enable symbolic debugging
```

```
CC62OPTS = -g
AS62OPTS =

* -q quiet run

LD62OPTS = -q      * -q quiet run
* Every BIOS program must be linked with:
* $(PROG)cfg.o62 - result of assembling $(PROG)cfg.s62
* $(PROG)cfg.cmd - linker command file generated by Config
* Tool. If additional linker command files exist,
* $(PROG)cfg.cmd must appear first.
*
PROG      = tsktest
OBJS      = $(PROG)cfg.obj
LIBS      =
CMDS      = $(PROG)cfg.cmd
*
* Targets:
*
all:: $(PROG).out
$(PROG).out: $(OBJS) $(CMDS)
$(PROG)cfg.obj: $(PROG)cfg.h62
$(PROG).obj:
$(PROG)cfg.s62 $(PROG)cfg.h62 $(PROG)cfg.cmd :: $(PROG).cdb
$(TI_DIR)/plugins/bios/gconfgen $(PROG).cdb
.clean clean::
    @ echo removing generated configuration files ...
    @$ (REMOVE) -f $(PROG)cfg.s62 $(PROG)cfg.h62 $(PROG)cfg.cmd
    @ echo removing object files and binaries ...
    @$ (REMOVE) -f *.obj *.out *.lst *.map
```

## 2.3 nmti

*Display symbols and values in a TI COFF file*

### Syntax

nmti [file1 file2 ...]

### Description

nmti prints the symbol table (name list) for each TI executable file listed on the command line. Executable files must be stored as COFF (Common Object File Format) files.

If no files are listed, the file a.out is searched. The output is sent to stdout. Note that both linked (executable) and unlinked (object) files can be examined with nmti.

Each symbol name is preceded by its value (blanks if undefined) and one of the following letters:

A	absolute symbol
B	bss segment symbol
D	data segment symbol
E	external symbol
S	section name symbol
T	text segment symbol
U	undefined symbol

The type letter is upper case if the symbol is external, and lower case if it is local.

## 2.4 sectti

*Display information about sections in TI COFF files*

### Syntax

sectti [-a] [file1 file2 ...]

### Description

sectti displays location and size information for all the sections in a TI executable file. Executable files must be stored as COFF (Common Object File Format) files.

All values are in hexadecimal. If no file names are given, a.out is assumed. Note that both linked (executable) and unlinked (object) files can be examined with sectti.

Using the -a flag causes sectti to display all program sections, including sections used only on the target by the DSP/BIOS plug-ins. If you omit the -a flag, sectti displays only the program sections that are loaded on the target.

**2.5 size62***Display the section sizes of an object file***Syntax**

size62 [file1 file2 ...]

**Description**

This utility prints the decimal number of MAUs required by all code sections, all data sections, and the .bss and .stack sections for each COFF file argument. If no file is specified, a.out is used. Note that both linked (executable) and unlinked (object) files can be examined with size62.

**2.6 vers**

*Display version information for a DSP/BIOS source or library file*

**Syntax**

vers [file1 file2 ...]

**Description**

The vers utility displays the version number of DSP/BIOS files installed in your system. For example, the following command checks the version number of the bios.a62 file in the lib sub-directory.

```
..\bin\vers bios.a62
bios.a62:
  *** library
  *** "date and time"
  *** bios-c05
  *** "version number"
```

The actual output from vers may contain additional lines of information. To identify your software version number to Technical Support, use the version number shown.

Note that both libraries and source files can be examined with vers.

# Function Callability and Error Tables

---

---

---

This appendix provides tables describing errors and function callability.

<b>Topic</b>	<b>Page</b>
<b>A.1 Functions Callable by Tasks, SWI Handlers, or Hardware ISRs . . .</b>	<b>A-2</b>
<b>A.2 DSP/BIOS Error Codes . . . . .</b>	<b>A-8</b>

## A.1 Functions Callable by Tasks, SWI Handlers, or Hardware ISRs

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
ATM_andi()	Yes	Yes	Yes	No
ATM_andu()	Yes	Yes	Yes	No
ATM_cleari()	Yes	Yes	Yes	No
ATM_clearu()	Yes	Yes	Yes	No
ATM_deci()	Yes	Yes	Yes	No
ATM_decu()	Yes	Yes	Yes	No
ATM_inci()	Yes	Yes	Yes	No
ATM_incu()	Yes	Yes	Yes	No
ATM_ori()	Yes	Yes	Yes	No
ATM_oru()	Yes	Yes	Yes	No
ATM_seti()	Yes	Yes	Yes	No
ATM_setu()	Yes	Yes	Yes	No
C62_disableIER	Yes	Yes	Yes	No
C62_dispatchPlug	Yes	Yes	Yes	No
C62_enableIER	Yes	Yes	Yes	No
C62_plug	Yes	Yes	Yes	No
CLK_countspms()	Yes	Yes	Yes	No
CLK_gethtime()	Yes	Yes	Yes	No
CLK_getttime()	Yes	Yes	Yes	No
CLK_getprd()	Yes	Yes	Yes	No
DEV_match	Yes	Yes	Yes	No
Dxx_close()	Yes	No	No	Yes (see Note 1)
Dxx_ctrl()	Yes	No	No	Yes (see Note 2)
Dxx_idle()	Yes	No	No	Yes (see Note 2)
Dxx_issue	Yes	No	No	No
Dxx_open()	Yes	No	No	Yes (see Note 2)
Dxx_ready()	Yes	No	No	No
Dxx_reclaim	Yes	No	No	No
HST_getpipe	Yes	Yes	Yes	No
HWI_disable()	Yes	Yes	Yes	No
HWI_enable()	Yes	Yes	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
HWI_enter	No	No	Yes	No
HWI_exit	No	No	Yes	Yes
HWI_restore()	Yes	Yes	Yes	No
IDL_run	Yes	No	No	No
LCK_create()	Yes	No	No	Yes (see Note 2)
LCK_delete()	Yes	No	No	Yes (see Note 2)
LCK_pend()	Yes	No	No	Yes (see Note 3)
LCK_post()	Yes	No	No	Yes (see Note 2)
LOG_disable	Yes	Yes	Yes	No
LOG_enable	Yes	Yes	Yes	No
LOG_error()	Yes	Yes	Yes	No
LOG_message	Yes	Yes	Yes	No
LOG_event()	Yes	Yes	Yes	No
LOG_printf()	Yes	Yes	Yes	No
LOG_reset	Yes	Yes	Yes	No
MBX_create()	Yes	No	No	Yes (see Note 2)
MBX_delete()	Yes	No	No	Yes (see Note 2)
MBX_pend()	Yes	Yes (see Note 4)	Yes (see Note 5)	Yes (see Note 5)
MBX_post()	Yes	Yes (see Note 5)	Yes (see Note 5)	Yes (see Note 6)
MEM_alloc()	Yes	No	No	Yes (see Note 2)
MEM_calloc()	Yes	No	No	Yes (see Note 2)
MEM_define	Yes	No	No	No
MEM_free()	Yes	No	No	Yes (see Note 2)
MEM_redefine	Yes	No	No	No
MEM_stat()	Yes	No	No	Yes (see Note 2)
MEM_valloc()	Yes	No	No	Yes (see Note 2)
PIP_alloc	Yes	Yes	Yes	Yes
PIP_free	Yes	Yes	Yes	Yes
PIP_get	Yes	Yes	Yes	Yes
PIP_getReaderAddr	Yes	Yes	Yes	No
PIP_getReaderNumFrames	Yes	Yes	Yes	No
PIP_getReaderSize	Yes	Yes	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
PIP_getWriterAddr	Yes	Yes	Yes	No
PIP_getWriterNumFrames	Yes	Yes	Yes	No
PIP_getWriterSize	Yes	Yes	Yes	No
PIP_put	Yes	Yes	Yes	Yes
PIP_setWriterSize	Yes	Yes	Yes	No
PRD_getticks	Yes	Yes	Yes	No
PRD_start	Yes	Yes	Yes	No
PRD_stop	Yes	Yes	Yes	No
PRD_tick	Yes	Yes	Yes	Yes
QUE_create	Yes	No	No	Yes (see Note 2)
QUE_delete	Yes	No	No	Yes (see Note 2)
QUE_dequeue	Yes	Yes	Yes	No
QUE_empty	Yes	Yes	Yes	No
QUE_enqueue	Yes	Yes	Yes	No
QUE_get	Yes	Yes	Yes	No
QUE_head	Yes	Yes	Yes	No
QUE_insert	Yes	Yes	Yes	No
QUE_new	Yes	Yes	Yes	No
QUE_next	Yes	Yes	Yes	No
QUE_prev	Yes	Yes	Yes	No
QUE_put	Yes	Yes	Yes	No
QUE_remove	Yes	Yes	Yes	No
RTDX_channelBusy	Yes	Yes	No	No
RTDX_disableInput	Yes	Yes	No	No
RTDX_disableOutput	Yes	Yes	No	No
RTDX_enableInput	Yes	Yes	No	No
RTDX_enableOutput	Yes	Yes	No	No
RTDX_read	Yes	Yes	No	No
RTDX_readNB	Yes	Yes	No	No
RTDX_sizeofInput	Yes	Yes	No	No
RTDX_write	Yes	Yes	No	No
SEM_count	Yes	Yes	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
SEM_create	Yes	No	No	Yes (see Note 2)
SEM_delete	Yes	No	No	Yes (see Note 2)
SEM_ipost	No	Yes	Yes	Yes (see Note 7)
SEM_new	Yes	Yes	Yes	No
SEM_pend	Yes	Yes (see Note 2)	No (see Note 2)	Yes (see Note 8)
SEM_post	Yes	Yes	Yes	Yes (see Note 8)
SEM_reset	Yes	No	No	No
SIO_bufsize	Yes	Yes	Yes	No
SIO_create	Yes	No	No	Yes (see Note 2)
SIO_ctrl	Yes	No	No	Yes (see Note 8)
SIO_delete	Yes	No	No	Yes (see Note 8)
SIO_flush	Yes	No	No	No
SIO_get	Yes	No	No	Yes (see Note 2)
SIO_idle)	Yes	No	No	Yes (see Note 2)
SIO_issue	Yes	No	No	No
SIO_put	Yes	No	No	Yes
SIO_reclaim	Yes	No	No	Yes
SIO_segid	Yes	Yes	Yes	No
SIO_select	Yes	No	No	Yes (see Note 9)
SIO_staticbuf	Yes	No	No	No
STS_add	Yes	Yes	Yes	No
STS_delta	Yes	Yes	Yes	No
STS_reset	Yes	Yes	Yes	No
STS_set	Yes	Yes	Yes	No
SWI_andn	Yes	Yes	Yes	No
SWI_create	Yes	No	No	Yes (see Note 2)
SWI_dec	Yes	Yes	Yes	No
SWI_delete	Yes	No	No	Yes (see Note 2)
SWI_disable	Yes	Yes	Yes	No
SWI_enable	Yes	Yes	Yes	Yes (see Note 10)
SWI_getattrs	Yes	Yes	Yes	No
SWI_getmbox	No	Yes	No	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
SWI_getpri	Yes	Yes	No	No
SWI_inc	Yes	Yes	Yes	Yes
SWI_or	Yes	Yes	Yes	Yes
SWI_post()	Yes	Yes	Yes	Yes (see Note 11)
SWI_raisepri	No	Yes	No	Yes
SWI_restorepri	No	Yes	No	Yes
SWI_self	No	Yes	No	No
SWI_setattrs	Yes	Yes	Yes	No
SYS_abort	Yes	Yes	Yes	No
SYS_atexit	Yes	Yes	Yes	No
SYS_error	Yes	Yes	Yes	No
SYS_exit	Yes	Yes	Yes	No
SYS_printf	Yes	Yes	Yes	No
SYS_putchar	Yes	Yes	Yes	No
SYS_sprintf	Yes	Yes	Yes	No
SYS_vprintf	Yes	Yes	Yes	No
SYS_vsprintf	Yes	Yes	Yes	No
TRC_disable	Yes	Yes	Yes	No
TRC_enable	Yes	Yes	Yes	No
TRC_query	Yes	Yes	Yes	No
TSK_checkstacks	Yes	No	No	No
TSK_creat)	Yes	No	No	Yes (see Note 2)
TSK_delete	Yes	No	No	Yes (see Note 2)
TSK_deltatime	Yes	Yes	Yes	No
TSK_disable	Yes	No	No	No
TSK_enable	Yes	No	No	Yes (see Note 11)
TSK_exit	Yes	No	No	Yes (see Note 12)
TSK_getenv	Yes	Yes	Yes	No
TSK_geterr	Yes	Yes	Yes	No
TSK_getname	Yes	Yes	Yes	No
TSK_getpri	Yes	Yes	Yes	No
TSK_getsts	Yes	Yes	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
TSK_itick	No	Yes	Yes	Yes (see Note 8)
TSK_self	Yes	Yes	Yes	No
TSK_setenv	Yes	Yes	Yes	No
TSK_seterr	Yes	Yes	Yes	No
TSK_setpri	Yes	No	No	Yes (see Note 8)
TSK_settime	Yes	Yes	Yes	No
TSK_sleep	Yes	No	No	Yes (see Note 13)
TSK_stat	Yes	Yes	Yes	No
TSK_tick	Yes	Yes	Yes	Yes
TSK_time	Yes	Yes	Yes	No
TSK_yield	Yes	Yes	No	Yes (see Note 13)

- Notes:
- 1) Task switch if memory manager blocks.
  - 2) Task switch if higher priority task is waiting for lock.
  - 3) Task switch if resource already locked by another task.
  - 4) May only be called by SWI handlers or hardware ISRs if timeout parameter equals 0.
  - 5) Task switch if mailbox empty and timeout parameter not 0, or if higher priority task blocked in MBX\_post().
  - 6) Task switch if higher priority task made ready, or mailbox is full and timeout parameter is not equal to 0.
  - 7) Task switch if higher priority task made ready.
  - 8) Task switch if semaphore count equals 0 and timeout parameter is not equal to 0.
  - 9) Task switch if no streams are ready for I/O.
  - 10) SWI\_enable() will allow pending SWI handlers to run. A task switch will occur if one of these routines makes a higher priority task ready.
  - 11) Task switch if higher priority task became ready while within a TSK\_disable()/TSK\_enable() block.
  - 12) Task switch will always occur.
  - 13) Task switch if there is an equal priority task ready.

## A.2 DSP/BIOS Error Codes

Name	Value	SYS_Errors[Value]
SYS_OK	0	"(SYS_OK)"
SYS_EALLOC	1	"(SYS_EALLOC): segid = %d, size = %u, align = %u" Memory allocation error.
SYS_EFREE	2	"(SYS_EFREE): segid = %d, ptr = 0x%x, size = %u" The memory free function associated with the indicated memory segment was unable to free the indicated size of memory at the address indicated by ptr.
SYS_ENODEV	3	"(SYS_ENODEV): device not found" The device being opened is not configured into the system.
SYS_EBUSY	4	"(SYS_EBUSY): device in use" The device is already opened by the maximum number of users.
SYS_EINVAL	5	"(SYS_EINVAL): invalid parameter" An invalid parameter was passed to the device.
SYS_EBADIO	6	"(SYS_EBADIO): device failure" The device was unable to support the I/O operation.
SYS_EMODE	7	"(SYS_EMODE): invalid mode" An attempt was made to open a device in an improper mode; e.g., an attempt to open an input device for output.
SYS_EDOMAIN	8	"(SYS_EDOMAIN): domain error" Used by SPOX-MATH when type of operation does not match vector or filter type.
SYS_ETIMEOUT	9	"(SYS_ETIMEOUT): timeout error" Used by device drivers to indicate that reclaim timed out.
SYS_EEOF	10	"(SYS_EEOF): end-of-file error" Used by device drivers to indicate the end of a file.
SYS_EDEAD	11	"(SYS_EDEAD): previously deleted object" An attempt was made to use an object that has been deleted.
SYS_EBADOBJ	12	"(SYS_EBADOBJ): invalid object" An attempt was made to use an object that does not exist.
SYS_EUSER	>=256	"(SYS_EUSER): <user-defined string>" User-defined error.

## A

- assembly language
  - calling C functions from 1-11
- atexit 1-328
- ATM module 1-13
- ATM\_andi 1-14, 1-15
- ATM\_andu 1-14, 1-15
- ATM\_cleari 1-16, 1-17
- ATM\_clearu 1-16, 1-17
- ATM\_deci 1-18, 1-19
- ATM\_decu 1-18, 1-19
- ATM\_inci 1-20, 1-21
- ATM\_incu 1-20, 1-21
- ATM\_ori 1-22, 1-23
- ATM\_oru 1-22, 1-23
- ATM\_seti 1-24, 1-25
- ATM\_setu 1-24, 1-25
- atomic operations 1-173
- atomic queue 1-173
- average 1-235

## B

- background loop 1-101
- boards
  - setting 1-78
- buffered pipe manager 1-144

## C

- C functions
  - calling from assembly language 1-11
- C standard library 1-328
- C62 module
  - main description 1-26
- C62\_disable
  - main description 1-27
- C62\_disable()
  - main description 1-27

- C62\_enable
  - main description 1-28
- C62\_enable()
  - main description 1-28
- C62\_plug()
  - main description 1-30
- calloc 1-328
- cdbprint utility 2-2, 2-3
- channels 1-80
- CLK module 1-31
  - trace types 1-290
- CLK\_countspms() 1-35
- CLK\_gethtime 1-36
- CLK\_getltime 1-38
- CLK\_getprd 1-40
- clocks
  - real time vs. data-driven 1-164
- configuration files
  - printing 2-2, 2-3
- conversion specifications 1-281, 1-283, 1-285, 1-287
- count 1-235
- counts per millisecond 1-35

## D

- data channels 1-80
- data transfer 1-144
- DAX driver 1-55
- DEV module 1-41
- DEV\_match 1-44
- DGN driver 1-58
- DGS driver 1-61
- DHL driver 1-65
- disable
  - HWI 1-92, 1-100
  - LOG 1-113
  - SWI 1-257
  - TRC 1-293
- disabling
  - hardware interrupts 1-92, 1-100
- DNL driver 1-68
- DOV driver 1-69

DPI driver 1-71  
drivers  
  DAX 1-55  
  DGN 1-58  
  DGS 1-61  
  DHL 1-65  
  DNL 1-68  
  DOV 1-69  
  DPI 1-71  
  DST 1-74  
  DTR 1-76  
DST driver 1-74  
DTR driver 1-76  
Dxx\_close 1-45  
Dxx\_ctrl 1-46  
  error handling 1-46  
Dxx\_idle 1-47  
  error handling 1-47  
Dxx\_init 1-48  
Dxx\_issue 1-49  
Dxx\_open 1-51  
Dxx\_ready 1-52  
Dxx\_reclaim 1-53  
  error handling 1-53

## E

enable  
  HWI 1-93  
  LOG 1-114  
  SWI 1-259  
  TRC 1-294  
enabling  
  hardware interrupts 1-93  
endian mode 1-78  
environ 1-328  
Error Codes A-8  
error handling  
  by Dxx\_close 1-45  
  by Dxx\_ctrl 1-46  
  by Dxx\_idle 1-47  
  by Dxx\_reclaim 1-53  
exit 1-328

## F

free 1-328  
functions  
  list of 1-3

## G

getenv 1-328  
global settings 1-78

## H

hardware interrupts 1-86  
  disabling 1-92, 1-100  
  enabling 1-93  
high-resolution time 1-36  
host data interface 1-80  
HST module 1-80  
HST\_getpipe 1-84  
HWI module 1-86  
  statistics units 1-236  
  trace types 1-290  
HWI\_disable 1-92, 1-100  
  vs. instruction 1-11  
HWI\_enable 1-93  
HWI\_enter 1-94  
HWI\_exit 1-97

## I

IDL module 1-101  
IDL\_run 1-104  
IDRAM0 memory segment 1-135  
IDRAM1 memory segment 1-135  
interrupt service routines 1-86  
IPRAM memory segment 1-135  
ISRs 1-86

## L

LCK module 1-105  
LCK\_create 1-106  
LCK\_delete 1-107  
LCK\_release 1-109  
LCK\_seize 1-108  
LgInt type 1-282, 1-284, 1-286, 1-288  
LgUns type 1-282, 1-284, 1-286, 1-288  
LOG module 1-110  
LOG\_disable 1-113  
LOG\_enable 1-114  
LOG\_error 1-115, 1-117  
LOG\_event 1-119  
LOG\_printf 1-121  
LOG\_reset 1-124  
logged events 1-290  
low-resolution time 1-38

**M**

mailbox  
 clear bits 1-250  
 decrement 1-252, 1-254, 1-256, 1-260, 1-273  
 get value 1-262  
 increment 1-264  
 set bits 1-266  
 malloc 1-328  
 MAU 1-133  
 maximum 1-235  
 MBX module 1-125  
 MBX\_create 1-127  
 MBX\_delete 1-128  
 MBX\_pend 1-129  
 MBX\_post 1-130  
 MEM module 1-131  
 MEM\_alloc 1-136, 1-137, 1-143  
 MEM\_calloc 1-136, 1-137, 1-143  
 MEM\_define 1-138  
 MEM\_free 1-140  
 MEM\_redefine 1-141  
 MEM\_stat 1-142  
 MEM\_valloc 1-136, 1-137, 1-143  
 Minimum Addressable Unit 1-133  
 modifies registers 1-3  
 modules  
 list of 1-2

**N**

naming conventions 1-3  
 nmti utility 2-5  
 notifyReader function  
 use of HWI\_enter 1-88

**O**

on-chip timer 1-31  
 operations  
 list of 1-3

**P**

parameters  
 listing 2-2, 2-3  
 vs. registers 1-11  
 period register 1-40  
 PIP module 1-144  
 statistics units 1-236  
 PIP\_alloc 1-148

PIP\_free 1-150  
 PIP\_get 1-151  
 PIP\_getReaderAddr 1-153  
 PIP\_getReaderNumFrames 1-155  
 PIP\_getReaderSize 1-156  
 PIP\_getWriterAddr 1-157  
 PIP\_getWriterNumFrames 1-158  
 PIP\_getWriterSize 1-159  
 PIP\_put 1-160, 1-161, 1-162  
 PIP\_setWriterSize 1-163  
 pipe object 1-84  
 pipes 1-144  
 postconditions 1-3, 1-11  
 posting software interrupts 1-247, 1-268  
 PRD module 1-164  
 statistics units 1-236  
 trace types 1-290  
 PRD register 1-33  
 PRD\_getticks 1-167  
 PRD\_start 1-168  
 PRD\_stop 1-170  
 PRD\_tick 1-171  
 preconditions 1-3, 1-11  
 printing configuration file 2-2, 2-3  
 priorities 1-248

**Q**

QUE module 1-172  
 QUE\_create 1-174  
 QUE\_delete 1-175  
 QUE\_dequeue 1-176  
 QUE\_empty 1-177  
 QUE\_enqueue 1-178  
 QUE\_get 1-179  
 QUE\_head 1-180  
 QUE\_insert 1-181  
 QUE\_new 1-182  
 QUE\_next 1-183  
 QUE\_prev 1-184  
 QUE\_put 1-185  
 QUE\_remove 1-186

**R**

read data 1-145  
 realloc 1-328  
 registers  
 modified 1-11  
 vs. parameters 1-11  
 RTDX\_bytesRead 1-203  
 RTDX\_channelBusy 1-192  
 RTDX\_CreateInputChannel 1-190, 1-191

RTDX\_CreateOutputChannel 1-190, 1-191  
RTDX\_disableInput 1-193, 1-194, 1-195, 1-196  
RTDX\_disableOutput 1-193, 1-194, 1-195, 1-196  
RTDX\_enableInput 1-193, 1-194, 1-195, 1-196  
RTDX\_enableOutput 1-193, 1-194, 1-195, 1-196  
RTDX\_isInputEnabled 1-197, 1-198  
RTDX\_isOutputEnabled 1-197, 1-198  
RTDX\_read 1-199  
RTDX\_readNB 1-201  
RTDX\_write 1-204

## S

SBSRAM memory segment 1-135  
SDRAM0 memory segment 1-135  
SDRAM1 memory segment 1-135  
sections  
  in executable file 2-6  
sectti utility 2-6  
SEM module 1-205  
SEM\_count 1-207  
SEM\_create 1-208  
SEM\_delete 1-209  
SEM\_ipost 1-210  
SEM\_new 1-211  
SEM\_pend 1-212  
SEM\_post 1-213  
SEM\_reset 1-214  
SIO module 1-215  
SIO\_bufsize 1-218  
SIO\_create 1-219  
SIO\_ctrl 1-222  
SIO\_delete 1-223  
SIO\_flush 1-224  
SIO\_get 1-225  
SIO\_idle 1-226  
SIO\_issue 1-227  
SIO\_put 1-229  
SIO\_reclaim 1-230  
SIO\_segid 1-232  
SIO\_select 1-233, 1-234  
size utility 2-7  
software interrupts 1-246  
SPOX standard C library functions 1-328  
stack overflow check 1-303  
stack, execution 1-248  
standard C library 1-328  
statistics  
  units 1-236, 1-290  
stdlib.h 1-328  
STS manager 1-188, 1-235  
STS\_add 1-240  
STS\_delta 1-241  
STS\_reset 1-243

STS\_set 1-244  
SWI module 1-246  
  statistics units 1-236  
  trace types 1-290  
SWI\_andn 1-250  
SWI\_dec 1-252, 1-254, 1-256, 1-260, 1-273  
SWI\_disable 1-257  
SWI\_enable 1-259  
SWI\_getmbox 1-262  
SWI\_getpri 1-263  
SWI\_inc 1-264  
SWI\_or 1-266  
SWI\_post 1-268  
SWI\_raisepri 1-269  
SWI\_restorepri 1-271  
SWI\_self 1-272  
symbol table 2-5  
SYS module 1-275  
SYS\_abort 1-277  
SYS\_atexit 1-278  
SYS\_EALLOC A-8  
SYS\_EBADIO A-8  
SYS\_EBADOBJ A-8  
SYS\_EBUSY A-8  
SYS\_EDEAD A-8  
SYS\_EDOMAIN A-8  
SYS\_EEOF A-8  
SYS\_EFREE A-8  
SYS\_EINVAL A-8  
SYS\_EMODE A-8  
SYS\_ENODEV A-8  
SYS\_error 1-136, 1-137, 1-143, 1-174, 1-208, 1-219, 1-279  
Sys\_error 1-256  
SYS\_ETIMEOUT A-8  
SYS\_EUSER 1-279, A-8  
SYS\_exit 1-280  
SYS\_OK A-8  
SYS\_printf 1-281, 1-283, 1-285, 1-287  
SYS\_putchar 1-289

## T

target board 1-78  
TDDR register 1-33  
timer 1-31  
total 1-235  
trace types 1-290  
TRC module 1-290  
TRC\_disable 1-293  
TRC\_enable 1-294  
TRC\_query 1-295  
TSK module 1-297  
TSK\_checkstacks 1-303

TSK\_create 1-304  
TSK\_delete 1-307  
TSK\_deltatime 1-308  
TSK\_disable 1-309  
TSK\_enable 1-310  
TSK\_exit 1-311  
TSK\_getenv 1-312  
TSK\_geterr 1-313  
TSK\_getname 1-314  
TSK\_getpri 1-315  
TSK\_getsts 1-316  
TSK\_itick 1-317  
TSK\_self 1-318  
TSK\_setenv 1-319  
TSK\_seterr 1-320  
TSK\_setpri 1-321  
TSK\_settime 1-322  
TSK\_sleep 1-323  
TSK\_stat 1-324  
TSK\_tick 1-325  
TSK\_time 1-326  
TSK\_yield 1-327

## U

underscores in function names 1-11  
units for statistics 1-236  
USER traces 1-290  
utilities  
    cdbprint 2-2, 2-3  
    nmti 2-5  
    sectti 2-6  
    size 2-7  
    vers 2-8

## V

vers utility 2-8  
version information 2-8

## W

write data 1-145