TMS320C28x DSP/BIOS 5.x
Application Programming Interface (API)

Reference Guide

TEXAS INSTRUMENTS

Literature Number: SPRU625L
August 2012

Read This First

About This Manual

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320C28x^{TM} DSP devices the ability to develop embedded real-time software. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

You should read and become familiar with the *TMS320 DSP/BIOS User’s Guide*, a companion volume to this API reference guide.

Before you read this manual, you may use the *Code Composer Studio* online tutorial and the DSP/BIOS section of the online help to get an overview of DSP/BIOS. This manual discusses various aspects of DSP/BIOS in depth and assumes that you have at least a basic understanding of DSP/BIOS.

Notational Conventions

This document uses the following conventions:

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

  Here is a sample program listing:

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

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

- Throughout this manual, 28 represents the two-digit numeric appropriate to your specific DSP platform. For example, DSP/BIOS assembly language API header files for the C28x platform are described as having a suffix of .h28. For the C64x or C67x DSP platform, substitute either 64 or 67 for each occurrence of 62.

Related Documentation From Texas Instruments

The following books describe TMS320 devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

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

TMS320C2000 Optimizing C/C++ Compiler User's Guide (literature number SPRU514) describes the C2000 C/C++ compiler and the assembly optimizer. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the C2000 generation of devices.

TMS320C28x Code Composer Studio Online Help introduces the Code Composer Studio integrated development environment and software tools. Of special interest to new DSP/BIOS users are the *Using DSP/BIOS* tutorial lessons.

Related Documentation

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

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

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

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

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

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

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

Trademarks

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

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, XDS, Code Composer, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, BIOSuite, SPOX, TMS320, TMS320C28x, TMS320C54x, TMS320C55x, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C5000, and TMS320C6000.

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

August 29, 2012

Contents

Figures

Tables

Chapter 1

SPRU625L—August 2012

API Functional Overview

This chapter provides an overview to the TMS320C28x DSP/BIOS API functions.

Topic **Page**

1.1 DSP/BIOS Modules

Table 1–1. DSP/BIOS Modules

Module	Description
ATM Module	Atomic functions written in assembly language
BUF Module	Maintains buffer pools of fixed size buffers
C28 Module	Target-specific functions
CLK Module	System clock manager
DEV Module	Device driver interface
GBL Module	Global setting manager
GIO Module	I/O module used with IOM mini-drivers
HOOK Module	Hook function manager
HST Module	Host channel manager
HWI Module	Hardware interrupt manager
IDL Module	Idle function and processing loop manager
LCK Module	Resource lock manager
LOG Module	Event Log manager
MBX Module	Mailboxes manager
MEM Module	Memory manager
MSGQ Module	Variable-length message manager

Module	Description
PIP Module	Buffered pipe manager
POOL Module	Allocator interface module
PRD Module	Periodic function manager
QUE Module	Queue manager
RTDX Module	Real-time data exchange manager
SEM Module	Semaphores manager
SIO Module	Stream I/O manager
STS Module	Statistics object manager
SWI Module	Software interrupt manager
SYS Module	System services manager
TRC Module	Trace manager
TSK Module	Multitasking manager
std.h and stdlib.h functions	Standard C library I/O functions

1.2 Naming Conventions

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

1.3 Assembly Language Interface Overview

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

Assembly functions can call C functions. Remember that the C compiler adds an underscore prefix to function names, so when calling a C function from assembly, add an underscore to the beginning of the C function name. For example, call _myfunction instead of myfunction. See the TMS320C2x/C2xx/C5x Optimizing C Compiler User’s Guide for more details.

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

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

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

1.4 DSP/BIOS Tconf Overview

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

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

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

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

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

- **Arg.** Arg properties hold arguments to pass to program functions. They may be strings, integers, labels, or other types as needed by the program function.
- **Bool.** You may assign a value of either true or 1 to set a Boolean property to true. You may assign a value of either false or 0 (zero) to set a Boolean property to false. Do not set a Boolean property to the quoted string "true" or "false".
- **EnumInt.** Enumerated integer properties accept a set of valid integer values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- **EnumString.** Enumerated string properties accept certain string values. These values are displayed in a drop-down list in the DSP/BIOS Configuration Tool.
- **Extern.** Properties that hold function names use the Extern type. In order to specify a function Extern, use the prog.extern() method as shown in the examples to refer to objects defined as asm, C, or C++ language symbols. The default language is C.
- **Int16.** Integer properties hold 16-bit unsigned integer values. The value range accepted for a property may have additional limits.
- **Int32.** Long integer properties hold 32-bit unsigned integer values. The value range accepted for a property may have additional limits.
- **Numeric.** Numeric properties hold either 32-bit signed or unsigned values or decimal values, as appropriate for the property.
- **Reference.** Properties that reference other configures objects contain an object reference. Use the prog.get() method to specify a reference to another object.
- **String.** String properties hold text strings.

1.5 List of Operations

Table 1-2: DSP/BIOS Operations

ATM module operations

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

BUF module operations

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

C28 operations

Function	Operation
	Disable certain maskable interrupts
	Enable certain maskable interrupts
	C function to plug an interrupt vector

CLK module operations

Function	Operation
CLK_countspms	Number of hardware timer counts per millisecond
CLK_cpuCyclesPerHtime	Return multiplier for converting high-res time to CPU cycles
CLK_cpuCyclesPerLtime	Return multiplier for converting low-res time to CPU cycles
CLK_gethtime	Get high-resolution time
CLK_getltime	Get low-resolution time

Function	Operation
CLK_getprd	Get period register value
CLK_reconfig	Reset timer period and registers
CLK_start	Restart the low-resolution timer
CLK_stop	Halt the low-resolution timer

DEV module operations

Function	Operation
DEV_createDevice	Dynamically creates device with user-defined parameters
DEV_deleteDevice	Deletes the dynamically created device
DEV_match	Match a device name with a driver
Dxx_close	Close device
Dxx_ctrl	Device control operation
Dxx_idle	Idle device
Dxx_init	Initialize device
Dxx_issue	Send a buffer to the device
Dxx_open	Open device
Dxx_ready	Check if device is ready for I/O
Dxx_reclaim	Retrieve a buffer from a device
DGN Driver	Software generator driver
DGS Driver	Stackable gather/scatter driver
DHL Driver	Host link driver
DIO Driver	Class driver
DNL Driver	Null driver
DOV Driver	Stackable overlap driver
DPI Driver	Pipe driver
DST Driver	Stackable split driver
DTR Driver	Stackable streaming transformer driver

GBL module operations

Function	Operation
GBL_getClkin	Get configured value of board input clock in KHz
GBL_getFrequency	Get current frequency of the CPU in KHz

Function	Operation
GBL_getProcId	Get configured processor ID used by MSGQ
GBL_getVersion	Get DSP/BIOS version information
GBL_setFrequency	Set frequency of CPU in KHz for DSP/BIOS
GBL_setProcId	Set configured value of processor ID used by MSGQ

GIO module operations

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

HOOK module operations

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

HST module operations

Function	Operation
HST_getpipe	Get corresponding pipe object

HWI module operations

Function	Operation
HWI_disable	Globally disable hardware interrupts
HWI_dispatchPlug	Plug the HWI dispatcher
HWI_enable	Globally enable hardware interrupts
HWI_enter	Hardware interrupt service routine prolog
HWI_exit	Hardware interrupt service routine epilog

Function	Operation
HWI_isHWI	Check to see if called in the context of an HWI
HWI_restore	Restore global interrupt enable state

IDL module operations

Function	Operation
IDL_run	Make one pass through idle functions

LCK module operations

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

LOG module operations

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

MBX module operations

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

MEM module operations

Function	Operation
MEM_alloc, MEM_valloc, MEM_calloc	Allocate from a memory heap
MEM_define	Define a new memory heap
MEM_free	Free a block of memory
MEM_getBaseAddress	Get base address of a memory heap
MEM_increaseTableSize	Increase the internal MEM table size
MEM_redefine	Redefine an existing memory heap
MEM_stat	Return the status of a memory heap
MEM_undefine	Undefine an existing memory segment

MSGQ module operations

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

Function	Operation
MSGQ_setMsgId	Sets the message ID in a message.
MSGQ_setSrcQueue	Sets the reply destination in a message.

PIP module operations

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

PRD module operations

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

QUE module operations

Function	Operation
QUE_create	Create an empty queue
QUE_delete	Delete an empty queue
QUE_dequeue	Remove from front of queue (non-atomically)
QUE_empty	Test for an empty queue
QUE_enqueue	Insert at end of queue (non-atomically)
QUE_get	Get element from front of queue (atomically)

Function	Operation
QUE_head	Return element at front of queue
QUE_insert	Insert in middle of queue (non-atomically)
QUE_new	Set a queue to be empty
QUE_next	Return next element in queue (non-atomically)
QUE_prev	Return previous element in queue (non-atomically)
QUE_put	Put element at end of queue (atomically)
QUE_remove	Remove from middle of queue (non-atomically)

RTDX module operations

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

SEM module operations

Function	Operation
SEM_count	Get current semaphore count
SEM_create	Create a semaphore
SEM_delete	Delete a semaphore
SEM_new	Initialize a semaphore
SEM_pend	Wait for a counting semaphore
SEM_pendBinary	Wait for a binary semaphore

Function	Operation
SEM_post	Signal a counting semaphore
SEM_postBinary	Signal a binary semaphore
SEM_reset	Reset semaphore

SIO module operations

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

STS module operations

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

SWI module operations

Function	Operation
SWI_andn	Clear bits from SWI's mailbox and post if becomes 0
SWI_andnHook	Specialized version of SWI_andn
SWI_create	Create a software interrupt
SWI_dec	Decrement SWI's mailbox and post if becomes 0
SWI_delete	Delete a software interrupt
SWI_disable	Disable software interrupts
SWI_enable	Enable software interrupts
SWI_getattrs	Get attributes of a software interrupt
SWI_getmbox	Return SWI's mailbox value
SWI_getpri	Return an SWI's priority mask
SWI_inc	Increment SWI's mailbox and post
SWI_isSWI	Check to see if called in the context of a SWI
SWI_or	Set or mask in an SWI's mailbox and post
SWI_orHook	Specialized version of SWI_or
SWI_post	Post a software interrupt
SWI_raisepri	Raise an SWI's priority
SWI_restorepri	Restore an SWI's priority
SWI_self	Return address of currently executing SWI object
SWI_setattrs	Set attributes of a software interrupt

SYS module operations

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

TRC module operations

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

TSK module operations

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

C library stdlib.h

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

DSP/BIOS std.h special utility C macros

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

Chapter 2

SPRU625L—August 2012

Application Program Interface

This chapter describes the DSP/BIOS API modules and functions.

Topic Page

2.1 ATM Module

The ATM module includes assembly language functions.

Functions

- ATM_andi, ATM_andu. AND memory and return previous value
- ATM_cleari, ATM_clearu. Clear memory and return previous value
- ATM_deci, ATM_decu. Decrement memory and return new value
- ATM_inci, ATM_incu. Increment memory and return new value
- ATM_ori, ATM_oru. OR memory and return previous value
- ATM_seti, ATM_setu. Set memory and return previous value

Description

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

ATM_andi

Atomically AND Int memory location and return previous value

C Interface

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 */

Description

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

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

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

See Also

ATM_andu
ATM_ori

ATM_andu

Atomically AND Uns memory location and return previous value

C Interface

Syntax

uval = ATM_andu(udst, usrc);

Parameters

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

Return Value

Uns	uval;	/* previous value of *udst */

Description

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

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

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

See Also

ATM_andi
ATM_oru

ATM_cleari *Atomically clear Int memory location and return previous value*

C Interface

Syntax

ival = ATM_cleari(idst);

Parameters

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

Return Value

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

Description

ATM_cleari atomically clears an Int memory location and returns its previous value as follows:

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

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

See Also

ATM_clearu
ATM_seti

ATM_clearu

Atomically clear Uns memory location and return previous value

C Interface

Syntax

uval = ATM_clearu(udst);

Parameters

volatile Uns *udst; /* pointer to unsigned */

Return Value

Uns uval; /* previous value of *udst */

Description

ATM_clearu atomically clears an Uns memory location and returns its previous value as follows:

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

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

See Also

ATM_cleari
ATM_setu

ATM_deci *Atomically decrement Int memory and return new value*

C Interface

Syntax

ival = ATM_deci(idst);

Parameters

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

Return Value

Int ival; /* new value after decrement */

Description

ATM_deci atomically decrements an Int memory location and returns its new value as follows:

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

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

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

See Also

ATM_decu
ATM_inci

ATM_decu *Atomically decrement Uns memory and return new value*

C Interface

Syntax

uval = ATM_decu(udst);

Parameters

volatile Uns	*udst;	/* pointer to unsigned */

Return Value

Uns	uval;	/* new value after decrement */

Description

ATM_decu atomically decrements a Uns memory location and returns its new value as follows:

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

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

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

See Also

ATM_deci
ATM_incu

ATM_inci

Atomically increment Int memory and return new value

C Interface

Syntax

ival = ATM_inci(idst);

Parameters

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

Return Value

Int	ival;	/* new value after increment */

Description

ATM_inci atomically increments an Int memory location and returns its new value as follows:

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

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

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

See Also

ATM_deci
ATM_incu

ATM_incu *Atomically increment Uns memory and return new value*

C Interface

Syntax

uval = ATM_incu(udst);

Parameters

volatile Uns *udst; /* pointer to unsigned */

Return Value

Uns uval; /* new value after increment */

Description

ATM_incu atomically increments an Uns memory location and returns its new value as follows:

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

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

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

See Also

ATM_decu
ATM_inci

ATM_ori *Atomically OR Int memory location and return previous value*

C Interface

Syntax
ival = ATM_ori(idst, isrc);

Parameters

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

Return Value

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

Description

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

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

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

See Also

ATM_andi
ATM_oru

ATM_oru

Atomically OR Uns memory location and return previous value

C Interface

Syntax

uval = ATM_oru(udst, usrc);

Parameters

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

Return Value

Uns	uva;	/* previous value of *udst */

Description

ATM_oru atomically ORs the mask contained in usrc with a destination memory location and overwrites the destination value *udst with the result as follows:

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

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

See Also

ATM_andu
ATM_ori

ATM_seti

Atomically set Int memory and return previous value

C Interface

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 */

Description

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

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

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

See Also

ATM_setu
ATM_cleari

ATM_setu

Atomically set Uns memory and return previous value

C Interface

Syntax

uold = ATM_setu(udst, unew);

Parameters

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

Return Value

```
Uns      uold;                    /* previous value of *udst */
```

Description

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

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

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

See Also

ATM_clearu
ATM_seti

2.2 BUF Module

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

Functions

- BUF_alloc. Allocate a fixed-size buffer from the buffer pool
- BUF_create. Dynamically create a buffer pool
- BUF_delete. Delete a dynamically-created buffer pool
- BUF_free. Free a fixed-size buffer back to the buffer pool
- BUF_maxbuff. Get the maximum number of buffers used in a pool
- BUF_stat. Get statistics for the specified buffer pool

Constants, Types, and Structures

```
typedef unsigned long MEM_sizep;

#define BUF_ALLOCSTAMP 0xcafe
#define BUF_FREESTAMP 0xbeef

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

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

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

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the BUF Manager Properties and BUF Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
bufSeg	Reference	prog.get("L0SARAM")
bufCount	Int32	1
size	Int32	4
align	Int32	2
len	Int32	4
postalignsize	Int32	4

Description

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

Within a buffer pool, all buffers have the same size and alignment. Although each frame has a fixed length, the application can put a variable amount of data in each frame, up to the length of the frame. You can create multiple buffer pools, each with a different buffer size.

Buffers can be allocated and freed from a pool as needed at run-time using the BUF_alloc and BUF_free functions.

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

- **Deterministic allocation times.** The BUF_alloc and BUF_free functions require a constant amount of time. Allocating and freeing memory through a heap is not deterministic.
- **Callable from all thread types.** Allocating and freeing buffers is atomic and non-blocking. As a result, BUF_alloc and BUF_free can be called from all types of DSP/BIOS threads: HWI, SWI, TSK, and IDL. In contrast, HWI and SWI threads cannot call MEM_alloc.
- **Optimized for fixed-length allocation.** In contrast MEM_alloc is optimized for variable-length allocation.
- **Less fragmentation.** Since the buffers are of fixed-size, the pool does not become fragmented.

BUF Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

BUF Object Properties

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

```
var myBuf = bios.BUF.create("myBUF");
```

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

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

  Tconf Name: comment Type: String

  Example: `myBuf.comment = "my BUF";`

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

  Tconf Name: bufSeg Type: Reference

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

- **Buffer count.** Specify the number of fixed-length buffers to create in this pool.

  Tconf Name: bufCount Type: Int32

  Example: `myBuf.bufCount = 128;`

- **Buffer size.** Specify the size (in MADUs) of each fixed-length buffer inside this buffer pool. The default size shown is the minimum valid value for that platform. This size may be adjusted to accommodate the alignment in the "Buffer size after alignment" property.

  Tconf Name: size Type: Int32

  Example: `myBuf.size = 4;`

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

  Tconf Name: align Type: Int32

  Example: `myBuf.align = 2;`

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

  Tconf Name: len Type: Int32

  Example: `myBuf.len = 4;`

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

  Tconf Name: postalignsize Type: Int32

  Example: `myBuf.postalignsize = 4;`

BUF_alloc *Allocate a fixed-size buffer from a buffer pool*

C Interface

Syntax

bufaddr = BUF_alloc(buf);

Parameters

BUF_Handle	buf;	/* buffer pool object handle */

Return Value

Ptr	bufaddr;	/* pointer to free buffer */

Reentrant

yes

Description

BUF_alloc allocates a fixed-size buffer from the specified buffer pool and returns a pointer to the buffer. BUF_alloc does not initialize the allocated buffer space.

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

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

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

The time required to successfully execute BUF_alloc is deterministic (constant over multiple calls).

Example

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

Ptr buffPtr;

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

See Also

BUF_free
MEM_alloc

BUF_create *Dynamically create a buffer pool*

C Interface

Syntax

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

Parameters

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

Return Value

BUF_Handle	buf;	/* buffer pool object handle */

Reentrant

no

Description

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

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

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

```
typedef unsigned long MEM_sizep;
```

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

BUF_create ensures that the size and alignment are set to at least the minimum values permitted for the platform. The minimum size permitted is 4 MADUs. The minimum alignment permitted is 2.

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

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

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

BUF_create calls MEM_alloc to dynamically create the BUF object's data structure and the buffer pool.

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

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

Constraints and Calling Context

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

Example

```
BUF_Handle myBufpool;
BUF_Attrs myAttrs;

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

See Also

BUF_delete

BUF_delete *Delete a dynamically-created buffer pool*

C Interface

Syntax

status = BUF_delete(buf);

Parameters

BUF_Handle	buf;	/* buffer pool object handle */

Return Value

Uns	status;	/* returned status */

Reentrant

no

Description

BUF_delete frees the buffer pool object and the buffer pool memory referenced by the handle provided.

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

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

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

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

Constraints and Calling Context

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

Example

```
BUF_Handle myBufpool;
Uns delstat;

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

See Also

BUF_create

BUF_free *Free a fixed memory buffer into the buffer pool*

C Interface

Syntax

status = BUF_free(buf, bufaddr);

Parameters

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

Return Value

Bool	status;	/* returned status */

Reentrant

yes

Description

BUF_free frees the specified buffer back to the specified buffer pool. The newly freed buffer is then available for further allocation by BUF_alloc.

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

The bufaddr parameter is the pointer returned by the corresponding call to BUF_alloc.

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

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

The time required to successfully execute BUF_free is deterministic (constant over multiple calls).

Example

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

...

BUF_free(buffPoolHandle, buffPtr);
```

See Also

BUF_alloc
MEM_free

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

C Interface

Syntax

count = BUF_maxbuff(buf);

Parameters

BUF_Handle	buf;	/* buffer pool object Handle */

Return Value

Uns	count;	/*maximum number of buffers used */

Reentrant

no

Description

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

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

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

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

Constraints and Calling Context

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

Example

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

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

See Also

BUF_stat *Determine the status of a buffer pool*

C Interface

Syntax

BUF_stat(buf,statbuf);

Parameters

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

Return Value

none

Reentrant

yes

Description

BUF_stat returns the status of the specified buffer pool.

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

The statbuf parameter must be a structure of type BUF_Stat. The BUF_stat function fills in all the fields of the structure. The BUF_Stat type has the following fields:

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

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

The time required to successfully execute BUF_stat is deterministic (constant over multiple calls).

Example

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

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

See Also

MEM_stat

2.3 C28 Module

The C28 module includes target-specific functions for the TMS320C28x family

Functions

- C28_disableIER. ASM macro to disable selected interrupts in IER
- C28_enableIER. ASM macro to enable selected interrupts in IER
- C28_plug. Plug interrupt vector

Description

The C28 module provides certain target-specific functions and definitions for the TMS320C28x family of processors.

See the c28.h file for a complete list of definitions for hardware flags for C. The c28.h file contains C language macros, #defines for various TMS320C28x registers, and structure definitions. The c28.h28 file also contains assembly language macros for saving and restoring registers in HWIs.

C28_disableIER *Disable certain maskable interrupts*

C Interface

Syntax

```
oldmask = C28_disableIER(mask);
```

Parameters

Uns	mask;	/* disable mask */

Return Value

Uns	oldmask;	/* actual bits cleared by disable mask */

Description

C28_disableIER disables interrupts by clearing the bits specified by mask in the Interrupt Enable Register (IER).

See C28_enableIER for a description and code examples for safely protecting a critical section of code from interrupts.

See Also

C28_enableIER

C28_enableIER *Enable certain maskable interrupts*

C Interface

Syntax
C28_enableIER(oldmask);

Parameters

Uns	oldmask;	/* enable mask */

Return Value
Void

Description

C28_disableIER and C28_enableIER disable and enable specific internal interrupts by modifying the Interrupt Enable Register (IER).

C28_disableIER clears the bits specified by the mask parameter in the Interrupt Enable Register and returns a mask of the bits it cleared. C28_enableIER sets the bits specified by the oldmask parameter in the Interrupt Enable Register.

C28_disableIER and C28_enableIER are usually used in tandem to protect a critical section of code from interrupts. The following code examples show a region protected from all maskable interrupts:

```
Uns  oldmask;

oldmask = C28_disableIER(~0);
  `do some critical operation; `
  `do not call TSK_sleep, SEM_post, etc.`
C28_enableIER(oldmask);
```

Note: DSP/BIOS kernel calls that can cause rescheduling of tasks (for example, SEM_post and TSK_sleep) should be avoided within a C28_disableIER/C28_enableIER block since the interrupts can be disabled for an indeterminate amount of time if a task switch occurs.

You can use C28_disableIER and C28_enableIER to disable selected interrupts, while allowing other interrupts to occur. However, if another hardware interrupt occurs during this region, it could cause a task switch. You can prevent this by enclosing it with TSK_disable / TSK_enable to disable DSP/BIOS task scheduling.

```
Uns    oldmask;

TSK_disable();
oldmask = C28_disableIER(INTMASK0);
  `do some critical operation;`
  `NOT OK to call TSK_sleep, SEM_post, etc.`
C28_enableIER(oldmask);
TSK_enable();
```

Note: If you use C28_disableIER and C28_enableIER to disable only some interrupts, you must surround this region with SWI_disable / SWI_enable, to prevent an intervening HWI from causing a SWI or TSK switch.

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

See Also

C28_disableIER

C28_plug

C function to plug an interrupt vector

C Interface

Syntax

C28_plug(vecid, fxn);

Parameters

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

Return Value

Void

Description

C28_plug hooks up the specified function as the branch target for a hardware interrupt (fielded by the CPU) at the vector address corresponding to vecid. C28_plug does not enable the interrupt. Use C28_enableIER to enable specific interrupts.

This API can plug the full set of vectors supported by the PIE (0-127). If the PIE is not enabled, the vector will be plugged in the default vector area. If the PIE is enabled, the vector will be plugged in the PIE vector area.

Constraints and Calling Context

- vecid must be a valid interrupt ID in the range of 0-127.

See Also

C28_enableIER

2.4 CLK Module

The CLK module is the clock manager.

Functions

- CLK_countspms. Timer counts per millisecond
- CLK_cpuCyclesPerHtime. Return high-res time to CPU cycles factor
- CLK_cpuCyclesPerLtime. Return low-res time to CPU cycles factor
- CLK_gethtime. Get high-resolution time
- CLK_getltime. Get low-resolution time
- CLK_getprd. Get period register value
- CLK_reconfig. Reset timer period and registers using CPU frequency
- CLK_start. Restart low-resolution timer
- CLK_stop. Stop low-resolution timer

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the CLK Manager Properties and CLK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")
FREERUN	Bool	false
ENABLECLK	Bool	true
HIRESTIME	Bool	true
MICROSECONDS	Int16	1000
CONFIGURETIMER	Bool	false
TCRTDDR	EnumInt	2 (0 to 0xffffffff hex)
PRD	Int16	49999

Instance Configuration Parameters

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

Description

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

DSP/BIOS provides the following timing methods:

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

Timer Counter

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

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

Table 2-1: Timer Counter Rates, Targets, and Resets

Platform	Timer Counter Rate	Target Value	Value Reset
’C28x	Decremented at CLKOUT / (TDDR+1), where CLKOUT is the DSP clock speed in MHz (see GBL Module Properties) and TDDR is the value of the timer divide-down register (see CLK Manager Properties).	0	Counter reset to PRD value.

Low-Resolution Time

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

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

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

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

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

High-Resolution Time

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

Table 2-2: High-Resolution Time Determination

Platform	Description
’C28x	Number of times the timer counter has been decremented.

You can use the CLK_gethtime function to get the high-resolution time and the CLK_countspms function to get the number of hardware timer counter register ticks per millisecond.

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

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

CLK Functions

The CLK functions performed when a timer interrupt occurs are performed in the context of the hardware interrupt that caused the system clock to tick. Therefore, the amount of processing performed within CLK functions should be minimized and these functions can only invoke DSP/BIOS calls that are allowable from within an HWI.

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

CLK Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Continue to run on SW breakpoint**. If this property is set to true, the timer continues to run when a software breakpoint occurs. This is part of the ’C28x real-time mode, which allows time-critical interrupts to run while less critical threads are halted and debugged. See Appendix C for more information. In the case of the timer, the FREE bit and SOFT bit in the TCR register of the on-device timer are both set to 1 when this property is set to true.

  Tconf Name: FREERUN Type: Bool

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

- **Enable CLK Manager**. If this property is set to true, the on-device timer hardware is used to drive the high- and low-resolution times and to trigger execution of CLK functions.

  Tconf Name: ENABLECLK Type: Bool

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

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

  Tconf Name: HIRESTIME Type: Bool

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

- **Microseconds/Int**. The number of microseconds between timer interrupts. The period register is set to a value that achieves the desired period as closely as possible.

  Tconf Name: MICROSECONDS Type: Int16

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

- **Directly configure on-device timer registers**. If this property is set to true, the timer's hardware registers, PRD and TDDR, can be directly set to the desired values. In this case, the Microseconds/Int property is computed based on the values in PRD and TDDR and the CPU clock speed in the GBL Module Properties.

  Tconf Name: CONFIGURETIMER Type: Bool

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

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

Platform	Options	Size	Registers
'C28x	0h to 0ffffffffh	32 bits	TDDRH:TDDR

  Tconf Name: TCRTDDR Type: EnumInt

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

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

  Tconf Name: PRD Type: Int16

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

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

  Tconf Name: N/A

CLK Object Properties

The Clock Manager allows you to create an arbitrary number of CLK objects. Clock objects have functions, which are executed by the Clock Manager every time a timer interrupt occurs. These functions can invoke any DSP/BIOS operations allowable from within an HWI except HWI_enter or HWI_exit.

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

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

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

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

  Tconf Name: comment Type: String

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

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

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

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

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

  Tconf Name: fxn Type: Extern

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

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

  Tconf Name: order Type: Int16

  Example: `myClk.order = 2;`

CLK_countspms *Number of hardware timer counts per millisecond*

C Interface

Syntax

ncounts = CLK_countspms();

Parameters

Void

Return Value

LgUns	ncounts;

Reentrant

yes

Description

CLK_countspms returns the number of high-resolution timer counts per millisecond. See Table 2-2 on page 53 for information about how the high-resolution rate is set.

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

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

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

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

See Also

CLK_gethtime
CLK_getprd
CLK_cpuCyclesPerHtime
CLK_cpuCyclesPerLtime
GBL_getClkin
STS_delta

CLK_cpuCyclesPerHtime *Return multiplier for converting high-res time to CPU cycles*

C Interface

Syntax

ncycles = CLK_cpuCyclesPerHtime(Void);

Parameters

Void

Return Value

Float	ncycles;

Reentrant

yes

Description

CLK_cpuCyclesPerHtime returns the multiplier required to convert from high-resolution time to CPU cycles. High-resolution time is returned by CLK_gethtime.

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

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

See Also

CLK_gethtime
CLK_getprd
GBL_getClkin

CLK_cpuCyclesPerLtime *Return multiplier for converting low-res time to CPU cycles*

C Interface

Syntax

ncycles = CLK_cpuCyclesPerLtime(Void);

Parameters

Void

Return Value

Float	ncycles;

Reentrant

yes

Description

CLK_cpuCyclesPerLtime returns the multiplier required to convert from low-resolution time to CPU cycles. Low-resolution time is returned by CLK_getltime.

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

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

See Also

CLK_getltime
CLK_getprd
GBL_getClkin

CLK_gethtime *Get high-resolution time*

C Interface

Syntax
currtime = CLK_gethtime();

Parameters
Void

Return Value

LgUns	currtime	/* high-resolution time */

Reentrant

no

Description

CLK_gethtime returns the number of high-resolution clock cycles that have occurred as a 32-bit value. When the number of cycles reaches the maximum value that can be stored in 32 bits, the value wraps back to 0. See “High-Resolution Time” on page 53 for information about how the high-resolution rate is set.

CLK_gethtime provides a value with greater accuracy than CLK_getltime, but which wraps back to 0 more frequently. For example, if the timer tick rate is 200 MHz, then regardless of the period register value, the CLK_gethtime value wraps back to 0 approximately every 86 seconds.

CLK_gethtime can be used in conjunction with STS_set and STS_delta to benchmark code. CLK_gethtime can also be used to add a time stamp to event logs.

Constraints and Calling Context

- CLK_gethtime cannot be called from the program’s main() function.

Example

```
/* ======== showTime ======== */

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

See Also

CLK_getltime
PRD_getticks
STS_delta

CLK_getltime *Get low-resolution time*

C Interface

Syntax

currtime = CLK_getltime();

Parameters

Void

Return Value

LgUns currtime /* low-resolution time */

Reentrant

yes

Description

CLK_getltime 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. See “Low-Resolution Time” on page 52 for information about how this rate is set.

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

CLK_gethtime provides a value with more accuracy than CLK_getltime, but which wraps back to 0 more frequently. For example, if the timer tick rate is 80 MHz, and you use the default period register value of 40000, the CLK_gethtime value wraps back to 0 approximately every 107 seconds, while the CLK_getltime value wraps back to 0 approximately every 49.7 days.

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

Constraints and Calling Context

- CLK_getltime cannot be called from the program’s main() function.

Example

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

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

See Also

CLK_gethtime
PRD_getticks
STS_delta

CLK_getprd *Get period register value*

C Interface

Syntax

period = CLK_getprd();

Parameters

Void

Return Value

Uns	period	/* period register value */

Reentrant

yes

Description

CLK_getprd returns the number of high-resolution timer counts per low-resolution interrupt. See Table 2-2 on page 53 for information about how the high-resolution rate is set.

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

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

See Also

CLK_countspms
CLK_gethtime
CLK_cpuCyclesPerHtime
CLK_cpuCyclesPerLtime
GBL_getClkin
STS_delta

CLK_reconfig *Reset timer period and registers using current CPU frequency*

C Interface

Syntax
status = CLK_reconfig();

Parameters
Void

Return Value

Bool	status	/* FALSE if failed */

Reentrant

yes

Description

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

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

When calling CLK_reconfig outside of main(), you must also call CLK_stop and CLK_start to stop and restart the timer. Use the following call sequence:

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

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

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

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

Constraints and Calling Context

- When calling CLK_reconfig from anywhere other than main(), you must also use CLK_stop and CLK_start.

- Call HWI_disable/HWI_restore or SWI_disable/SWI_enable around a block that stops, configures, and restarts the timer as needed to prevent re-entrancy or other problems. That is, you must disable interrupts if an interrupt could lead to another call to CLK_reconfig or if interrupt processing relies on having a running timer to ensure that these non-reentrant functions are not interrupted.
- If you do not stop and restart the timer, CLK_reconfig can only be called from the program's main() function.
- If you use CLK_reconfig, you should also use GBL_setFrequency.

See Also

GBL_getFrequency
GBL_setFrequency
CLK_start
CLK_stop

CLK_start *Restart the low-resolution timer*

C Interface

Syntax

CLK_start();

Parameters

Void

Return Value

Void

Reentrant

no

Description

This function starts the low-resolution timer if it has been halted by CLK_stop. The period and prescalar registers are updated to reflect any changes made by a call to CLK_reconfig. This function then resets the timer counters and starts the timer.

CLK_start should only be used in conjunction with CLK_reconfig and CLK_stop. See the section on CLK_reconfig for details and the allowed calling sequence.

Note that all ’C28x platforms use the same timer to drive low-resolution and high-resolution times. On such platforms, both times are affected by this API.

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

See Also

CLK_reconfig
CLK_stop
GBL_setFrequency

CLK_stop *Halt the low-resolution timer*

C Interface

Syntax

CLK_stop();

Parameters

Void

Return Value

Void

Reentrant

no

Description

This function stops the low-resolution timer. It can be used in conjunction with CLK_reconfig and CLK_start to reconfigure the timer at run-time.

Note that all ’C28x platforms use the same timer to drive low-resolution and high-resolution times. On such platforms, both times are affected by this API.

CLK_stop should only be used in conjunction with CLK_reconfig and CLK_start, and only in the required calling sequence. See the section on CLK_reconfig for details.

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

See Also

CLK_reconfig
CLK_start
GBL_setFrequency

2.5 DEV Module

The DEV module provides the device interface.

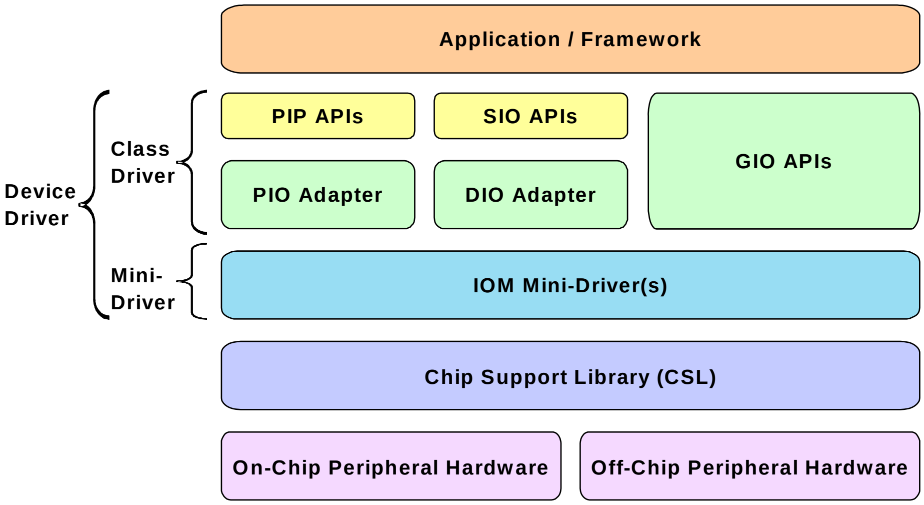
Functions

- DEV_createDevice. Dynamically create device
- DEV_deleteDevice. Delete dynamically-created device
- DEV_match. Match device name with driver
- Dxx_close. Close device
- Dxx_ctrl. Device control
- Dxx_idle. Idle device
- Dxx_init. Initialize device
- Dxx_issue. Send frame to device
- Dxx_open. Open device
- Dxx_ready. Device ready
- Dxx_reclaim. Retrieve frame from device

Description

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

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



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

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

The DEV module provides the following capabilities:

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

Constants, Types, and Structures

```
#define DEV_INPUT       0
#define DEV_OUTPUT      1

typedef struct DEV_Frame {  /* frame object */
   QUE_Elem   link;       /* queue link */
   Ptr        addr;       /* buffer address */
   size_t     size;       /* buffer size */
   Arg        misc;       /* reserved for driver */
   Arg        arg;        /* user argument */
   Uns        cmd;        /* mini-driver command */
   Int        status;     /* status of command */
} DEV_Frame;

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

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

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

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

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DEV Manager Properties and DEV Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Instance Configuration Parameters

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

DEV Manager Properties

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

- **DGN Driver (software generator driver).** pseudo-device that generates one of several data streams, such as a sin/cos series or white noise. This driver can be useful for testing applications that require an input stream of data.
- **DHL Driver (host link driver).** Driver that uses the HST interface to send data to and from the Host Channel Control Analysis Tool.
- **DIO Adapter (class driver).** Driver used with the device driver model.

- **DPI Driver (pipe driver).** Software device used to stream data between DSP/BIOS tasks.

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

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

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

DEV Object Properties

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

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

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

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

  Tconf Name: comment Type: String

  Example: `myDev.comment = "My device";`

- **init function**. Specify the function to run to initialize this device.
  Use a leading underscore before the function name if the function is written in C and you are using the DSP/BIOS Configuration Tool. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

  Tconf Name: initFxn Type: Arg

  Example: `myDev.initFxn = prog.extern("myInit");`

- **function table ptr**. Specify the name of the device functions table for the driver or mini-driver. This table is of type DEV_Fxns or IOM_Fxns depending on the setting for the function table type property.

  Tconf Name: fxnTable Type: Arg

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

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

  Tconf Name: fxnTableType Type: EnumString

  Options: "DEV_Fxns", "IOM_Fxns"

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

- **device id**. Specify the device ID. If the value you provide is non-zero, the value takes the place of a value that would be appended to the device name in a call to SIO_create. The purpose of such a value is driver-specific.

  Tconf Name: deviceId Type: Arg

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

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

  Use a leading underscore before the structure name if the structure is declared in C and you are using the DSP/BIOS Configuration Tool.

Tconf Name:	params	Type: Arg
Example:	`myDev.params = prog.extern("myParams");`	

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

Tconf Name:	deviceGlobalDataPtr	Type: Arg
Example:	`myDev.deviceGlobalDataPtr = 0x00000000;`	

DEV_createDevice *Dynamically create device*

C Interface

Syntax

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

Parameters

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

Return Value

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

Reentrant

no

Description

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

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

The fxns parameter points to the device function table. The function table may be of type DEV_Fxns or IOM_Fxns.

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

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

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

The devid item specifies the device ID. If the value you provide is non-zero, the value takes the place of a value that would be appended to the device name in a call to SIO_create. The purpose of such a value is driver-specific. The default value is NULL.

The params item specifies the name of a parameter structure that may be used to provide additional parameters. This structure should have a name with the format DXX_Params where XX is the two-letter code for the driver used by this device. The default value is NULL.

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

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

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

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

DEV_createDevice returns one of the following status values:

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

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

Constraints and Calling Context

- This function cannot be called from a SWI or HWI.
- This function can only be used if dynamic memory allocation is enabled.
- The device function table must be consistent with the type specified in the attrs structure. DSP/BIOS does not check to ensure that the types are consistent.

Example

```
Int status;

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

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

See Also

SIO_create

DEV_deleteDevice

Delete a dynamically-created device

C Interface

Syntax

status = DEV_deleteDevice(name);

Parameters

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

Return Value

Int	status;	/* result of operation */

Reentrant

no

Description

DEV_deleteDevice deallocates the specified dynamically-created device and deletes it from the list of devices in the application.

The name parameter specifies the device to delete. This name must match a name used with DEV_createDevice.

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

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

DEV_createDevice returns one of the following status values:

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

DEV_deleteDevice calls SYS_error if mdUnBindDev returns a failure condition. The device is deleted even if mdUnBindDev fails, but DEV_deleteDevice returns the IOM error returned by mdUnBindDev.

Constraints and Calling Context

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

Example

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

See Also

SIO_delete

DEV_match *Match a device name with a driver*

C Interface

Syntax

```
substr = DEV_match(name, device);
```

Parameters

```
String          name;       /* device name */
DEV_Device      **device;   /* pointer to device table entry */
```

Return Value

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

Description

DEV_match searches the device table for the first device name that matches a prefix of name. The output parameter, device, points to the appropriate entry in the device table if successful and is set to NULL on error. The DEV_Device structure is defined in dev.h.

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

See Also

SIO_create

Dxx_close

Close device

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

C Interface

Syntax

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

Parameters

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

Return Value

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

Description

Dxx_close closes the device associated with device and returns an error code indicating success (SYS_OK) or failure. device is bound to the device through a prior call to Dxx_open.

SIO_delete first calls Dxx_idle to idle the device. Then it calls Dxx_close.

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

Constraints and Calling Context

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

See Also

Dxx_idle
Dxx_open
SIO_delete

Dxx_ctrl

Device control operation

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

C Interface

Syntax

```
status = Dxx_ctrl(device, cmd, arg);
```

Parameters

```
DEV_Handle        device      /* device handle */
Uns               cmd;        /* driver control code */
Arg               arg;        /* control operation argument */
```

Return Value

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

Description

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

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

C Interface

Syntax

```
status = Dxx_idle(device, flush);
```

Parameters

```
DEV_Handle          device;        /* device handle */
Bool                flush;         /* flush output flag */
```

Return Value

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

Description

Dxx_idle places the device associated with device into its idle state and returns an error code indicating success (SYS_OK) or failure. Devices are initially in this state after they are opened with Dxx_open.

Dxx_idle returns the device to its initial state. Dxx_idle should move any frames from the device->todevice queue to the device->fromdevice queue. In SIO_ISSUERECLAIM mode, any outstanding buffers issued to the stream must be reclaimed in order to return the device to its true initial state.

Dxx_idle is called by SIO_idle, SIO_flush, and SIO_delete to recycle frames to the appropriate queue.

flush is a boolean parameter that indicates what to do with any pending data of an output stream. If flush is TRUE, all pending data is discarded and Dxx_idle does not block waiting for data to be processed. If flush is FALSE, the Dxx_idle function does not return until all pending output data has been rendered. All pending data in an input stream is always discarded, without waiting.

Constraints and Calling Context

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

See Also

SIO_delete
SIO_idle
SIO_flush

Dxx_init

Initialize device

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

C Interface

Syntax

```
Dxx_init();
```

Parameters

Void

Return Value

Void

Description

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

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

C Interface

Syntax

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

Parameters

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

Return Value

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

Description

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

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

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

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

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

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

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

C Interface

Syntax

status = Dxx_open(device, name);

Parameters

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

Return Value

Int	status;	/* result of operation */

Description

Dxx_open is called by SIO_create to open a device. Dxx_open opens a device and returns an error code indicating success (SYS_OK) or failure.

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

name is the string remaining after the device name has been matched by SIO_create using DEV_match.

See Also

Dxx_close
SIO_create

Dxx_ready

Check if device is ready for I/O

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

C Interface

Syntax

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

Parameters

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

Return Value

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

Description

Dxx_ready is called by SIO_select and SIO_ready to determine if the device is ready for an I/O operation. In this context, ready means a call that retrieves a buffer from a device does not block. If a frame exists, Dxx_ready returns TRUE, indicating that the next SIO_get, SIO_put, or SIO_reclaim operation on the device does not cause the calling task to block. If there are no frames available, Dxx_ready returns FALSE. This informs the calling task that a call to SIO_get, SIO_put, or SIO_reclaim for that device would result in blocking.

Dxx_ready registers the device's ready semaphore with the SIO_select semaphore sem. In cases where SIO_select calls Dxx_ready for each of several devices, each device registers its own ready semaphore with the unique SIO_select semaphore. The first device that becomes ready calls SEM_post on the semaphore.

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

SIO_ready calls Dxx_ready with sem = NULL. This is equivalent to the second Dxx_ready call made by SIO_select, and the underlying device driver should just return status without registering a semaphore.

See Also

SIO_select

Dxx_reclaim

Retrieve a buffer from a device

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

C Interface

Syntax

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

Parameters

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

Return Value

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

Description

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

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

If device->timeout is SYS_FOREVER, the task remains suspended until a frame is available on the device's fromdevice queue. If timeout is 0, Dxx_reclaim returns immediately.

If timeout expires before a buffer is available on the device's fromdevice queue, Dxx_reclaim returns SYS_ETIMEOUT. Otherwise Dxx_reclaim returns SYS_OK for success, or an error code.

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

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

SIO_reclaim calls Dxx_reclaim, then it gets the frame from the device->fromdevice queue.

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

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

Constraints and Calling Context

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

See Also

Dxx_issue
SIO_issue
SIO_get
SIO_put

DGN Driver

Software generator driver

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

Description

The DGN driver manages a class of software devices known as generators, which produce an input stream of data through successive application of some arithmetic function. DGN devices are used to generate sequences of constants, sine waves, random noise, or other streams of data defined by a user function.The number of active generator devices in the system is limited only by the availability of memory.

Configuring a DGN Device

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

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

See the DGN Object Properties for the device you created.

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DGN Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
device	EnumString	"user" ("sine", "random", "constant", "printHex", "printInt")
useDefaultParam	Bool	false
deviceId	Arg	prog.extern("DGN_USER", "asm")
constant	Numeric	1
seedValue	Int32	1
lowerLimit	Numeric	-32767
upperLimit	Numeric	32767
gain	Numeric	32767
frequency	Numeric	1
phase	Numeric	0
rate	Int32	256
fxn	Extern	prog.extern("FXN_F_nop")
arg	Arg	0x00000000

Data Streaming

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

Since DGN generates data “on demand,” tasks do not block when calling SIO_get, SIO_put, or SIO_reclaim on a DGN data stream. High-priority tasks must, therefore, be careful when using these streams since lower- or even equal-priority tasks do not get a chance to run until the high-priority task suspends execution for some other reason.

DGN Driver Properties

There are no global properties for the DGN driver manager.

DGN Object Properties

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

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

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

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

  Tconf Name: comment Type: String

  Example: `myDgn.comment = "DGN device";`

- **Device category**. The device category—user, sine, random, constant, printHex, printInt—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 custom function to produce/consume a data stream.

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

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

  — **constant.** Produces a constant stream of data.

  — **printHex.** Writes the stream data buffers to the trace buffer in hexadecimal format.

  — **printInt.** Writes the stream data buffers to the trace buffer in integer format.

  Tconf Name: device Type: EnumString

  Options: "user", "sine", "random", "constant", "printHex", "printInt"

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

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

  Tconf Name: useDefaultParam Type: Bool

  Example: `myDgn.useDefaultParam = false;`

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

  Tconf Name: deviceId Type: Arg

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

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

  Tconf Name: constant Type: Numeric

  Example: `myDgn.constant = 1;`

- **Seed value**. The initial seed value used by an internal pseudo-random number generator if the Device category is random. Used to produce a uniformly distributed sequence of numbers ranging between Lower limit and Upper limit.

  Tconf Name: seedValue Type: Int32

  Example: `myDgn.seedValue = 1;`

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

  Tconf Name: lowerLimit Type: Numeric

  Example: `myDgn.lowerLimit = -32767;`

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

  Tconf Name: upperLimit Type: Numeric

  Example: `myDgn.upperLimit = 32767;`

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

  Tconf Name: gain Type: Numeric

  Example: `myDgn.gain = 32767;`

- **Frequency**. The frequency of the generated sine wave (in cycles per second) if the Device category is sine. DGN uses a static (256 word) sine table to approximate a sine wave. Only frequencies that divide evenly into 256 can be represented exactly with DGN. A “step” value is computed at open time for stepping through this table:

  ```
  step = (256 * Frequency / Rate)
  ```

  Tconf Name: frequency Type: Numeric

  Example: `myDgn.frequency = 1;`

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

  Tconf Name: phase Type: Numeric

  Example: `myDgn.phase = 0;`

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

  Tconf Name: rate Type: Int32

  Example: `myDgn.rate = 256;`

- **User function**. If the Device category is user, specifies the function to be used to compute the successive values of the data sequence in an input device, or to be used to process the data stream, in an output device. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

  Tconf Name: fxn Type: Extern

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

- **User function argument**. An argument to pass to the User function.

A user function must have the following form:

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

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

Tconf Name: arg Type: Arg

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

DGS Driver

Stackable gather/scatter driver

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

Description

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

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

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

Configuring a DGS Device

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

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

Modify the myDgs properties as follows.

- **init function.** Type 0 (zero).
- **function table ptr.** Type _DGS_FXNS
- **function table type**. DEV_Fxns
- **device id.** Type 0 (zero).
- **device params ptr.** Type 0 (zero) to use the default parameters. To use different values, you must declare a DGS_Params structure (as described after this list) containing the values to use for the parameters.

DGS_Params is defined in dgs.h as follows:

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

The device parameters are:

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

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

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

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

Transform Functions

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

- **u32tou8/u8tou32**. These functions provide conversion to/from packed unsigned 8-bit integers to unsigned 32-bit integers. The buffer must contain a multiple of 4 number of 32-bit/8-bit unsigned values.
- **u16tou32/u32tou16**. These functions provide conversion to/from packed unsigned 16-bit integers to unsigned 32-bit integers. The buffer must contain an even number of 16-bit/32-bit unsigned values.
- **i16toi32/i32toi16**. These functions provide conversion to/from packed signed 16-bit integers to signed 32-bit integers. The buffer must contain an even number of 16-bit/32-bit integers.
- **u8toi16/i16tou8**. These functions provide conversion to/from a packed 8-bit format (two 8-bit words in one 16-bit word) to a one word per 16 bit format.
- **i16tof32/f32toi16**. These functions provide conversion to/from packed signed 16-bit integers to 32-bit floating point values. The buffer must contain an even number of 16-bit integers/32-bit floats.
- **localcopy**. This function simply passes the data to the underlying device without packing or compressing it.

Data Streaming

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

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

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

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

Example

The following code example declares DGS_PRMS as a DGS_Params structure:

```
#include <dgs.h>

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

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

See Also

DTR Driver

DHL Driver

Host link driver

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

Description

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

Configuring a DHL Device

To add a DHL device you must first create an HST object and make it available to the DHL driver. To do this, use the following syntax:

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

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

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

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

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

Then, you can set this object's properties to select which HST channel, of those available for DHL, is used by this DHL device. If you plan to use the DHL device for output to the host, be sure to select an HST channel whose mode is output. Otherwise, select an HST channel with input mode.

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DHL Driver Properties and DHL Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

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

Name	Type	Default (Enum Options)
mode	EnumString	"output" ("input")

Data Streaming

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

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

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

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

```
stream = SIO_create("/dhl0", SIO_INPUT, 128, NULL);
```

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

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

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

Constraints

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

DHL Driver Properties

The following global property can be set for the DHL - Host Link Driver on the DHL Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

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

Tconf Name:	OBJMEMSEG	Type: Reference
Example:	`DHL.OBJMEMSEG = prog.get("myMEM");`	

DHL Object Properties

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

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

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

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

Tconf Name:	comment	Type: String
Example:	`myDhl.comment = "DHL device";`	

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

Tconf Name:	hstChannel	Type: Reference
Example:	`myDhl.hstChannel = prog.get("myHST");`	

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

Tconf Name:	mode	Type: EnumString
Options:	"input", "output"	
Example:	`myDhl.mode = "output";`	

DIO Adapter *SIO Mini-driver adapter*

Description

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

Configuring a DIO Device

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

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

Set the DEV Object Properties for the device as follows.

- **init function.** Type 0 (zero).
- **function table ptr.** Type _DIO_FXNS
- **function table type.** IOM_Fxns
- **device id.** Type 0 (zero).
- **device params ptr.** Type 0 (zero).

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

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

DIO Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DIO Driver Properties and DIO Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")
STATICCREATE	Bool	false

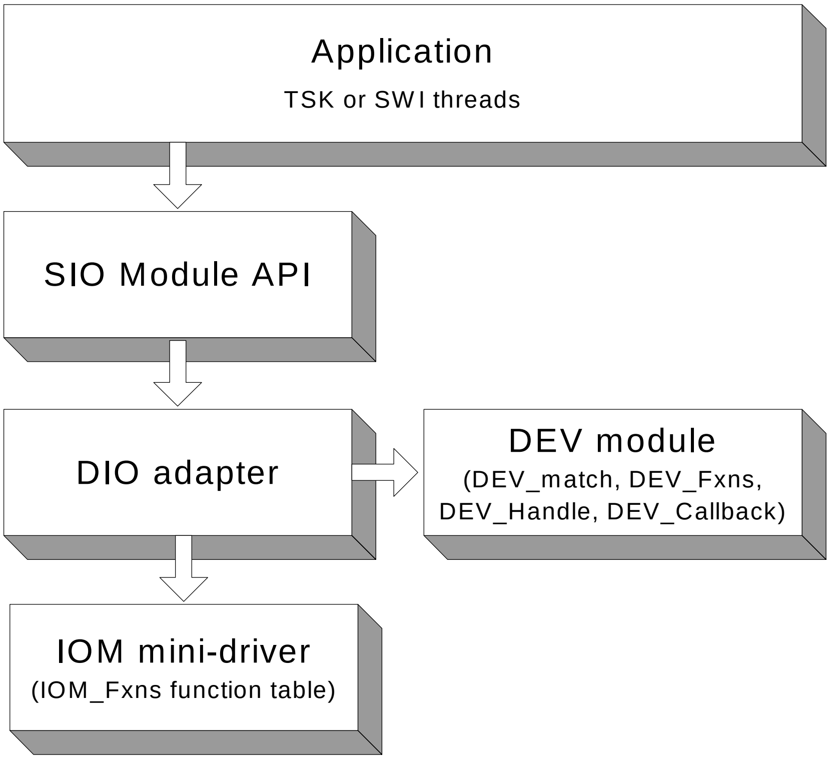
Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
useCallBackFxn	Bool	false
deviceName	Reference	prog.get("UDEV0")
chanParams	Arg	0x00000000

Description

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

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



DIO Driver Properties

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

- **Object memory**. Enter the memory segment from which to allocate DIO objects.

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Create All DIO Objects Statically.** Set this property to true if you want DIO objects to be created completely statically. If this property is false (the default), MEM_calloc is used internally to allocate space for DIO objects. If this property is true, you must create all SIO and DIO objects using the DSP/BIOS Configuration Tool or Tconf. Any calls to SIO_create fail. Setting this property to true reduces the application’s code size (so long as the application does not call MEM_alloc or its related functions elsewhere).

  Tconf Name: STATICCREATE Type: Bool

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

DIO Object Properties

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

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

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

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

  Tconf Name: comment Type: String

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

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

  Tconf Name: useCallBackFxn Type: Bool

  Example: `myDio.useCallBackFxn = false;`

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

  Tconf Name: N/A

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

  Tconf Name: deviceName Type: Reference

  Example: `myDio.deviceName = prog.get("UDEV0");`

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

  Tconf Name: chanParams Type: Arg

  Example: `myDio.chanParams = 0x00000000;`

DNL Driver

Null driver

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

Description

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

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

Configuring a DNL Device

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

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

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

- **init function.** Type 0 (zero).
- **function table ptr.** Type _DNL_FXNS
- **function table type**. DEV_Fxns
- **device id.** Type 0 (zero).
- **device params ptr.** Type 0 (zero).

Data Streaming

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

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

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

DOV Driver

Stackable overlap driver

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

Description

The DOV driver manages a class of stackable devices that generate an overlapped stream by retaining the last N minimum addressable data units (MADUs) of each buffer input from an underlying device. These N points become the first N points of the next input buffer. MADUs are equivalent to a 16-bit word in the data address space of the processor on C28x platforms.

Configuring a DOV Device

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

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

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

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

If you enter 0 for the Device ID, you need to specify the length of the overlap when you create the stream with SIO_create by appending the length of the overlap to the device name. If you statically create the stream (with Tconf) instead, enter the length of the overlap in the Device Control String for the stream.

For example, if you statically create a device called overlap, and use 0 as its Device ID, you can open a stream with:

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

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

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

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

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

If you create the stream statically and enter 16 as the Device ID property, leave the Device Control String blank.

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

```
#include <dov.h>

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

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

Data Streaming

DOV devices can only be opened for input. The overlap size, specified in the string passed to SIO_create, must be greater than 0 and less than the size of the actual input buffers.

DOV does not support any control calls. All SIO_ctrl calls are passed to the underlying device.

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

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

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

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

To create the same streams statically (rather than dynamically with SIO_create), add SIO objects with Tconf. Enter the string that identifies the terminating device preceded by “/” (forward slash) in the SIO object’s Device Control Strings (for example, /codec, /port). Then select the stacking device (overlap, overlapio) from the Device property.

See Also

DTR Driver
DGS Driver

DPI Driver

Pipe driver

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

Description

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

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

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

Configuring a DPI Device

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the DPI Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Instance Configuration Parameters

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

Data Streaming

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

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

And in the function for the other task:

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

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

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

and from the writer task call:

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

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

Tasks block within DPI when using SIO_get, SIO_put, or SIO_reclaim if a buffer is not available. SIO_select can be used to guarantee that a call to one of these functions do not block. SIO_select can be called simultaneously by both the input and the output sides.

DPI and the SIO_ISSUERECLAIM Streaming Model

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

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

```
#define COPYBUFS
```

Rebuild dpi.c. Link your application with this version of dpi.obj instead of the default one. To do this, add this version of dpi.obj to your project explicitly. This buffer exchange alters the way in which the streaming mechanism works. When using this version of the DPI driver, the writer reclaims first the buffers issued by the reader rather than its own issued buffers, and vice versa.

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

Converting a Single Processor Application to a Multiprocessor Application

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

Constraints

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

DPI Driver Properties

There are no global properties for the DPI driver manager.

DPI Object Properties

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

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

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

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

  Tconf Name: comment Type: String

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

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

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

  If this property is set to false, when you use SIO_create, the name of the SIO object must exactly match the name of the DPI object. As a result, only one open stream can use the DPI object. For example, if the DPI object is named "pipe", an attempt to use SIO_create to create pipe0 fails.

  Tconf Name: allowVirtual Type: Bool

  Example: `myDpi.allowVirtual = false;`

DST Driver

Stackable split driver

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

Description

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

Configuring a DST Device

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

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

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

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

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

Example 1:

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

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

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

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

Example 2:

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

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

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

Example 3:

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

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

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

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

Data Streaming

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

Constraints

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

DTR Driver

Stackable streaming transformer driver

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

Description

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

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

Configuring a DTR Device

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

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

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

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

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

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

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

  If you type _DTR_multiplyInt16, a built-in data scaling operation is performed on the data stream to multiply the contents of the buffer by the scale.value of the device parameters. The data stream is assumed to contain values of type Int16.
- **device params ptr**. Enter the name of a DTR_Params structure declared in your C application code. See the information following this list for details.

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

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

In the following code example, DTR_PRMS is declared as a DTR_Params structure:

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

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

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

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

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

If you do not configure one of the built-in scaling functions for the device ID, use user.fxn and user.arg in the DTR_Params structure to define a transformation that is applied to inbound or outbound blocks of data, where buffer is the address of a data block containing size points; if the value of user.fxn is NULL, no transformation is performed at all.

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

Data Streaming

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

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

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

2.6 GBL Module

This module is the global settings manager.

Functions

- GBL_getClkin. Gets configured value of board input clock in KHz.
- GBL_getFrequency. Gets current frequency of the CPU in KHz.
- GBL_getProcId. Gets configured processor ID used by MSGQ.
- GBL_getVersion. Gets DSP/BIOS version information.
- GBL_setFrequency. Set frequency of CPU in KHz for DSP/BIOS.
- GBL_setProcId. Set configured value of processor ID.

Configuration Properties

The following list shows the properties for this module that can be configured in a Tconf script, along with their types and default values. For details, see the GBL Module Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
BOARDNAME	String	"c28xx"
PROCID	Int16	0
CLKIN	Uint32	20000 KHz
CLKOUT	Int16	’C28x: 150
SPECIFYRTSLIB	Bool	false
RTSLIB	String	""
MODIFYPLLCR	Bool	true
PLLCR	Numeric	0xa
USERLIMPMODEABORTFXN	Extern	prog.extern("FXN_F_nop") (C280x only)
PLLWAITCYCLE	Numeric	131072 (C281x only)
MEMORYMODEL	EnumString	"LARGE"
CALLUSERINITFXN	Bool	false
USERINITFXN	Extern	prog.extern("FXN_F_nop")
ENABLEINST	Bool	true
INSTRUMENTED	Bool	true
ENABLEALLTRC	Bool	true

Description

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

GBL Module Properties

The following Global Settings can be made:

- **Target Board Name**. The name of the board or board family.

  Tconf Name: BOARDNAME Type: String

  Example: `bios.GBL.BOARDNAME = "c28xx";`

- **Processor ID (PROCID)**. ID used to communicate with other processors using the MSGQ Module. The procId is also defined in the MSGQ_TransportObj array that is part of the MSGQ_Config structure. This value can be obtained with GBL_getProcId and modified by GBL_setProcId (but only within the User Init Function).

  Tconf Name: PROCID Type: Int16

  Example: `bios.GBL.PROCID = 0;`

- **Board Clock In KHz (Informational Only)**. Frequency of the input clock in KHz. You should set this property to match the actual board clock rate. This property does not change the rate of the board; it is informational only. The configured value can be obtained at run-time using the GBL_getClkin API. The default value is 20000 KHz.

  Tconf Name: CLKIN Type: Uint32

  Example: `bios.GBL.CLKIN = 20000;`

- **DSP Speed In MHz (CLKOUT)**. This number, times 1000000, is the number of instructions the processor can execute in 1 second. You should set this property to match the actual rate. This property does not change the rate of the board. This value is used by the CLK manager to calculate register settings for the on-device timers.

  Tconf Name: CLKOUT Type: Int16

  Example: `bios.GBL.CLKOUT = 150.0000;`

- **Specify RTS Library**. Determines whether a user can specify the run-time support library to which the application is linked. The RTS library contains the printf, malloc, and other standard C library functions. For information about using this library, see “std.h and stdlib.h functions” on page 432. If you do not choose to specify a library, the default library for your platform is used.

  Tconf Name: SPECIFYRTSLIB Type: Bool

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

- **Run-Time Support Library**. The name of the run-time support (RTS) library to which the application is linked. These libraries are located in the appropriate <*ccs_install_dir*>\ccsv5\tools\compiler\<*target*>\lib folder for your target. The library you select is used in the linker command file generated from the Tconf script when you build your application.

  Tconf Name: RTSLIB Type: String

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

- **Modify PLLCR**. Set this property to true if you want to modify the value of the PLL Control Register, which is used to program the PLL (phase-locked loop).

  Tconf Name: MODIFYPLLCR Type: Bool

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

- **PLLCR - PLL Control Register**. The value of the PLL Control Register.

  Tconf Name: PLLCR Type: Numeric

  Example: `bios.GBL.PLLCR = 0xa;`

- **User Limp Mode Abort Function**. (C280x only) This property allows you to plug in an abort function to be called when the CPU is powered by the PLL at the Limp Mode frequency. The property can be set only if MODIFYPLLCR is true.

  Tconf Name: USERLIMPMODEABORTFXNType: Numeric

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

- **Cycles to wait for PLL lock**. (C281x only) The value of this property is the waiting time after the PLLCR register is written for the PLL to be stable. The property can be set only when MODIFYPLLCR is true.

  Tconf Name: PLLWAITCYCLE Type: Numeric

  Example: `bios.CLK.PLLWAITCYCLE = 131072;`

- **Memory Model**. This specifies the address reach within the 'C28x program. The only option is large. In the large model, data addressing uses the full 23-bit range. Program space addressing uses the full 24-bit range.

  Tconf Name: MEMORYMODEL Type: EnumString

  Options: "LARGE"

  Example: `bios.GBL.MEMORYMODEL = "LARGE";`

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

  Tconf Name: CALLUSERINITFXN Type: Bool

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

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

  Tconf Name: USERINITFXN Type: Extern

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

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

  Tconf Name: ENABLEINST Type: Bool

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

- **Use Instrumented BIOS Library**. Specifies whether to link with the instrumented or non-instrumented version of the DSP/BIOS library. The non-instrumented versions are somewhat smaller but do not provide support for LOG, STS, and TRC instrumentation. The libraries are located in appropriate <*ccs_install_dir*>\ccsv5\tools\compiler\<*target*>\lib folder for your target. By default, the instrumented version of the library for your platform is used.

  Tconf Name: INSTRUMENTED Type: Bool

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

- **Enable All TRC Trace Event Classes**. Set this property to false if you want all types of tracing to be initially disabled when the program is loaded. If you disable tracing, you can still use the RTA Control Panel or the TRC_enable function to enable tracing at run-time.

  Tconf Name: ENABLEALLTRC Type: Bool

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

GBL_getClkin *Get configured value of board input clock in KHz*

C Interface

Syntax

```
clkin = GBL_getClkin(Void);
```

Parameters

Void

Return Value

```
Uint32                         clkin;           /* CLKIN frequency */
```

Reentrant

yes

Description

Returns the configured value of the board input clock (CLKIN) frequency in KHz.

See Also

CLK_countspms
CLK_getprd

GBL_getFrequency *Get current frequency of the CPU in KHz*

C Interface

Syntax

frequency = GBL_getFrequency(Void);

Parameters

Void

Return Value

```
Uint32                    frequency;     /* CPU frequency in KHz */
```

Reentrant

yes

Description

Returns the current frequency of the DSP CPU in an integer number of KHz. This is the frequency set by GBL_setFrequency, which must also be an integer. The default value is the value of the CLKOUT property, which is configured as one of the GBL Module Properties.

See Also

GBL_getClkin
GBL_setFrequency

GBL_getProcId *Get configured value of processor ID*

C Interface

Syntax

```
procid = GBL_getProcId(Void);
```

Parameters

Void

Return Value

```
Uint16                  procid;         /* processor ID */
```

Reentrant

yes

Description

Returns the configured value of the processor ID (PROCID) for this processor. This numeric ID value is used by the MSGQ module when determining which processor to communicate with.

The procId is also defined in the MSGQ_TransportObj array that is part of the MSGQ_Config structure. The same processor ID should be defined for this processor in both locations.

During the User Init Function, the application may modify the statically configured processor ID by calling GBL_setProcId. In this case, the User Init Function may need to call GBL_getProcId first to get the statically configured processor ID.

See Also

MSGQ Module: Static Configuration
GBL_setProcId

GBL_getVersion *Get DSP/BIOS version information*

C Interface

Syntax

version = GBL_getVersion(Void);

Parameters

Void

Return Value

Uint16	version;	/* version data */

Reentrant

yes

Description

Returns DSP/BIOS kernel version information as a 4-digit hex number. For example: 0x5100. Note that the kernel version is different from the DSP/BIOS product version.

When comparing versions, compare the highest digits that are different. The digits in the version information are as follows:

Bits	Compatibility with Older DSP/BIOS Versions
12-15 (first hex digit)	Not compatible. Changes to application C, assembly, or configuration (Tconf) code may be required. For example, moving from 0x5100 to 0x6100 may require code changes.
8-11 (second hex digit)	No code changes required but you should recompile. For example, moving from 0x5100 to 0x5200 requires recompilation.
0-7 (third and fourth hex digits)	No code changes or recompile required. You should re-link if either of these digits are different. For example, moving from 0x5100 to 0x5102 requires re-linking.

The version returned by GBL_getVersion matches the version in the DSP/BIOS header files. (For example, tsk.h.) If the header file version is as follows, GBL_getVersion returns 0x5001. If there are three items, the last item uses two digits (for example, 01) in the returned hex number.

```
*  @(#) DSP/BIOS_Kernel 5,0,1 05-30-2004 (cuda-l06)
```

GBL_setFrequency *Set frequency of the CPU in KHz*

C Interface

Syntax

```
GBL_setFrequency( frequency );
```

Parameters

```
Uint32                    frequency;     /* CPU frequency in KHz */
```

Return Value

Void

Reentrant

yes

Description

This function sets the value of the CPU frequency known to DSP/BIOS.

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

If you call GBL_setFrequency to update the CPU frequency known to DSP/BIOS, you should follow the sequence shown in the CLK_reconfig topic to reconfigure the timer.

The frequency must be an integer number of KHz.

Constraints and Calling Context

- If you change the frequency known to DSP/BIOS, you should also reconfigure the timer (with CLK_reconfig) so that the actual frequency is the same as the frequency known to DSP/BIOS.

See Also

CLK_reconfig
GBL_getClkin
GBL_getFrequency

GBL_setProcId *Set configured value of processor ID*

C Interface

Syntax

```
GBL_setProcId( procId );
```

Parameters

```
Uint16          procId;     /* processor ID */
```

Return Value

Void

Reentrant

no

Description

Sets the processor ID (PROCID) for this processor. This numeric ID value is used by the MSGQ module to determine which processor to communicate with.

The procId is also defined in the MSGQ_TransportObj array that is part of the MSGQ_Config structure.

This function can only be called in the User Init Function configured as part of the GBL Module Properties. That is, this function may only be called at the beginning of DSP/BIOS initialization.

The application may determine the true processor ID for the device during the User Init Function and call GBL_setProcId with the correct processor ID. This is useful in applications that run a single binary image on multiple DSP processors.

How the application determines the correct processor ID is application- or board-specific. For example, you might use GPIO. You can call GBL_getProcId from the User Init Function to get the statically configured processor ID.

Constraints and Calling Context

- This function can only be called in the User Init Function configured as part of the GBL Module Properties.

See Also

MSGQ Manager Properties
GBL_getProcId

2.7 GIO Module

The GIO module is the Input/Output Module used with IOM mini-drivers as described in *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

Functions

- GIO_abort. Abort all pending input and output.
- GIO_control. Device specific control call.
- GIO_create. Allocate and initialize a GIO object.
- GIO_delete. Delete underlying mini-drivers and free up the GIO object and any associated IOM packet structures.
- GIO_flush. Drain output buffers and discard any pending input.
- GIO_new. Initialize a GIO object using pre-allocated memory.
- GIO_read. Synchronous read command.
- GIO_submit. Submits a packet to the mini-driver.
- GIO_write. Synchronous write command.

Constants, Types, and Structures

```
/* Modes for GIO_create */
#define IOM_INPUT    0x0001
#define IOM_OUTPUT   0x0002
#define IOM_INOUT    (IOM_INPUT | IOM_OUTPUT)

/* IOM Status and Error Codes */
#define IOM_COMPLETED SYS_OK  /* I/O successful */
#define IOM_PENDING   1 /* I/O queued and pending */
#define IOM_FLUSHED   2 /* I/O request flushed */
#define IOM_ABORTED   3 /* I/O aborted */
#define IOM_EBADIO   -1 /* generic failure */
#define IOM_ETIMEOUT -2 /* timeout occurred */
#define IOM_ENOPACKETS -3 /* no packets available */
#define IOM_EFREE     -4  /* unable to free resources */
#define IOM_EALLOC    -5  /* unable to alloc resource */
#define IOM_EABORT   -6 /* I/O aborted uncompleted*/
#define IOM_EBADMODE -7 /* illegal device mode */
#define IOM_EOF      -8 /* end-of-file encountered */
#define IOM_ENOTIMPL -9 /* operation not supported */
#define IOM_EBADARGS -10 /* illegal arguments used */
#define IOM_ETIMEOUTUNREC -11
                 /* unrecoverable timeout occurred */
#define IOM_EINUSE   -12 /* device already in use */

/* Command codes for IOM_Packet */
#define IOM_READ   0
#define IOM_WRITE  1
#define IOM_ABORT  2
#define IOM_FLUSH  3
#define IOM_USER   128 /* 0-127 reserved for system */
```

```
/* Command codes reserved for control */
#define IOM_CHAN_RESET    0 /* reset channel only */
#define IOM_CHAN_TIMEDOUT 1
                       /* channel timeout occurred */
#define IOM_DEVICE_RESET  2 /* reset entire device */
#define IOM_CNTL_USER   128
                     /* 0-127 reserved for system */

/* Structure passed to GIO_create */
typedef struct GIO_Attrs  {
    Int  nPackets; /* number of asynch I/O packets */
    Uns  timeout;  /* for blocking (SYS_FOREVER) */
} GIO_Attrs;

/* Struct passed to GIO_submit for synchronous use*/
typedef struct GIO_AppCallback {
    GIO_TappCallback      fxn;
    Ptr                   arg;
} GIO_AppCallback;

typedef struct GIO_Obj {
   IOM_Fxns   *fxns;      /* ptr to function table */
   Uns        mode;       /* create mode */
   Uns        timeout;    /* timeout for blocking */
   IOM_Packet syncPacket; /* for synchronous use */
   QUE_Obj    freeList;   /* frames for asynch I/O */
   Ptr        syncObj;    /* ptr to synchro. obj */
   Ptr        mdChan;     /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;

typedef struct IOM_Fxns
{
    IOM_TmdBindDev        mdBindDev;
    IOM_TmdUnBindDev      mdUnBindDev;
    IOM_TmdControlChan    mdControlChan;
    IOM_TmdCreateChan     mdCreateChan;
    IOM_TmdDeleteChan     mdDeleteChan;
    IOM_TmdSubmitChan     mdSubmitChan;
} IOM_Fxns;

typedef struct IOM_Packet {  /* frame object */
   QUE_Elem   link;       /* queue link */
   Ptr        addr;       /* buffer address */
   size_t     size;       /* buffer size */
   Arg        misc;       /* reserved for driver */
   Arg        arg;        /* user argument */
   Uns        cmd;        /* mini-driver command */
   Int        status;     /* status of command */
} IOM_Packet;
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the GIO Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

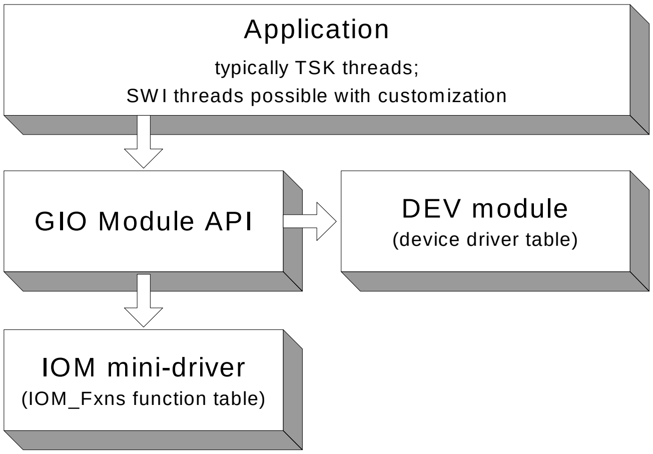
Name	Type	Default
ENABLEGIO	Bool	false
CREATEFXN	Extern	prog.extern("FXN_F_nop")
DELETEFXN	Extern	prog.extern("FXN_F_nop")
PENDFXN	Extern	prog.extern("FXN_F_nop"
POSTFXN	Extern	prog.extern("FXN_F_nop")

Description

The GIO module provides a standard interface to mini-drivers for devices such as UARTs, codecs, and video capture/display devices. The creation of such mini-drivers is not covered in this manual; it is described in *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

The GIO module is independent of the actual mini-driver being used. It allows the application to use a common interface for I/O requests. It also handles response synchronization. It is intended as common "glue" to bind applications to device drivers.

The following figure shows how modules are related in an application that uses the GIO module and an IOM mini-driver:



The GIO module is the basis of communication between applications and mini-drivers. The DEV module is responsible for maintaining the table of device drivers that are present in the system. The GIO module obtains device information by using functions such as DEV_match.

GIO Manager Properties

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

- **Enable General Input/Output Manager.** Set this property to true to enable use of the GIO module. If your application does not use GIO, you should leave it disabled to prevent additional modules (such as SEM) from being linked into your application.

  Tconf Name: ENABLEGIO Type: Bool

  Example: `bios.GIO.ENABLEGIO = false;`

- **Create Function.**The function the GIO module should use to create a synchronization object. This function is typically SEM_create. If you use another function, that function should have a prototype that matches that of SEM_create: Ptr CREATEFXN(Int count, Ptr attrs);

  Tconf Name: CREATEFXN Type: Extern

  Example: `bios.GIO.CREATEFXN = prog.extern("SEM_create");`

- **Delete Function.**The function the GIO module should use to delete a synchronization object. This function is typically SEM_delete. If you use another function, that function should have a prototype that matches that of SEM_delete: Void DELETEFXN(Ptr semHandle);

  Tconf Name: DELETEFXN Type: Extern

  Example: `bios.GIO.DELETEFXN = prog.extern("SEM_delete");`

- **Pend Function.**The function the GIO module should use to pend on a synchronization object. This function is typically SEM_pend. If you use another function, that function should have a prototype that matches that of SEM_pend: Bool PENDFXN(Ptr semHandle, Uns timeout);

  Tconf Name: PENDFXN Type: Extern

  Example: `bios.GIO.PENDFXN = prog.extern("SEM_pend");`

- **Post Function.**The function the GIO module should use to post a synchronization object. This function is typically SEM_post. If you use another function, that function should have a prototype that matches that of SEM_post: Void POSTFXN(Ptr semHandle);

  Tconf Name: POSTFXN Type: Extern

  Example: `bios.GIO.POSTFXN = prog.extern("SEM_post");`

GIO Object Properties

GIO objects cannot be created statically. In order to create a GIO object, the application should call GIO_create or GIO_new.

GIO_abort

Abort all pending input and output

C Interface

Syntax

status = GIO_abort(gioChan);

Parameters

GIO_Handle	gioChan;	/* handle to an instance of the device */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

An application calls GIO_abort to abort all input and output from the device. When this call is made, all pending calls are completed with a status of GIO_ABORTED. An application uses this call to return the device to its initial state. Usually this is done in response to an unrecoverable error at the device level.

GIO_abort returns IOM_COMPLETED upon successfully aborting all input and output requests. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_abort results in a call to the mdSubmit function of the associated mini-driver. The IOM_ABORT command is passed to the mdSubmit function. The mdSubmit call is typically a blocking call, so calling GIO_abort can result in the thread blocking.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- GIO_abort cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

Example

```
/* abort all I/O requests given to the device*/
gioStatus = GIO_abort(gioChan);
```

GIO_control *Device specific control call*

C Interface

Syntax

status = GIO_control(gioChan, cmd, args);

Parameters

GIO_Handle	gioChan;	/* handle to an instance of the device */
Int	cmd;	/* control functionality to perform */
Ptr	args;	/* data structure to pass control information */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

An application calls GIO_control to configure or perform control functionality on the communication channel.

The cmd parameter may be one of the command code constants listed in “Constants, Types, and Structures” on page 117. A mini-driver may add command codes for additional functionality.

The args parameter points to a data structure defined by the device to allow control information to be passed between the device and the application. This structure can be generic across a domain or specific to a mini-driver. In some cases, this argument may point directly to a buffer holding control data. In other cases, there may be a level of indirection if the mini-driver expects a data structure to package many components of data required for the control operation. In the simple case where no data is required, this parameter may just be a predefined command value.

GIO_control returns IOM_COMPLETED upon success. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_control results in a call to the mdControl function of the associated mini-driver. The mdControl call is typically a blocking call, so calling GIO_control can result in blocking.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- GIO_control cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

Example

```
/* Carry out control/configuration on the device*/
gioStatus = GIO_control(gioChan, XXX_RESET, &args);
```

GIO_create *Allocate and initialize a GIO object*

C Interface

Syntax

gioChan = GIO_create(name, mode, *status, chanParams, *attrs)

Parameters

String	name	/* name of the device to open */
Int	mode	/* mode in which the device is to be opened */
Int	*status	/* address to place driver return status */
Ptr	chanParams	/* optional */
GIO_Attrs	*attrs	/* pointer to a GIO_Attrs structure */

Return Value

GIO_Handle	gioChan;	/* handle to an instance of the device */

Description

An application calls GIO_create to create a GIO_Obj object and open a communication channel. This function initializes the I/O channel and opens the lower-level device driver channel. The GIO_create call also creates the synchronization objects it uses and stores them in the GIO_Obj object.

The name argument is the name specified for the device when it was created in the configuration or at runtime.

The mode argument specifies the mode in which the device is to be opened. This may be IOM_INPUT, IOM_OUTPUT, or IOM_INOUT.

If the status returned by the device is non-NULL, a status value is placed at the address specified by the status parameter.

The chanParams parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner.

The attrs parameter is a pointer to a structure of type GIO_Attrs.

```
typedef struct GIO_Attrs  {
  Int  nPackets; /* number of asynch I/O packets */
   Uns  timeout;  /* for blocking calls (SYS_FOREVER) */
} GIO_Attrs;
```

If attrs is NULL, a default set of attributes is used. The default for nPackets is 2. The default for timeout is SYS_FOREVER.

The GIO_create call allocates a list of IOM_Packet items as specified by the nPackets member of the GIO_Attrs structure and stores them in the GIO_Obj object it creates.

GIO_create returns a handle to the GIO_Obj object created upon a successful open. The handle returned by this call should be used by the application in subsequent calls to GIO functions. This function returns a NULL handle if the device could not be opened. For example, if a device is opened in a mode not supported by the device, this call returns a NULL handle.

A call to GIO_create results in a call to the mdCreateChan function of the associated mini-driver.

Constraints and Calling Context

- A GIO stream can only be used by one task simultaneously. Catastrophic failure can result if more than one task calls GIO_read on the same input stream, or more than one task calls GIO_write on the same output stream.
- GIO_create cannot be called from the context of a SWI or HWI thread.
- This function can be called only after the device has been loaded and initialized.

Example

```
/* Create a device instance */
gioAttrs = GIO_ATTRS;
gioChan = GIO_create("\Codec0", IOM_INPUT, NULL, NULL,
                &gioAttrs);
```

GIO_new

GIO_delete

Delete underlying mini-drivers and free GIO object and its structures

C Interface

Syntax

status = GIO_delete(gioChan);

Parameters

GIO_Handle	gioChan;	/* handle to device instance to be closed */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

An application calls GIO_delete to close a communication channel opened prior to this call with GIO_create. This function deallocates all memory allocated for this channel and closes the underlying device. All pending input and output are cancelled and the corresponding interrupts are disabled.

The gioChan parameter is the handle returned by GIO_create or GIO_new.

This function returns IOM_COMPLETED if the channel is successfully closed. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_delete results in a call to the mdDelete function of the associated mini-driver.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.

Example

```
/* close the device instance */
GIO_delete(gioChan);
```

GIO_flush

Drain output buffers and discard any pending input

C Interface

Syntax

status = GIO_flush(gioChan);

Parameters

GIO_Handle	gioChan;	/* handle to an instance of the device */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

An application calls GIO_flush to flush the input and output channels of the device. All input data is discarded; all pending output requests are completed. When this call is made, all pending input calls are completed with a status of IOM_FLUSHED, and all output calls are completed routinely.

The gioChan parameter is the handle returned by GIO_create or GIO_new.

This call returns IOM_COMPLETED upon successfully flushing all input and output. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_flush results in a call to the mdSubmit function of the associated mini-driver. The IOM_FLUSH command is passed to the mdSubmit function. The mdSubmit call is typically a blocking call, so calling GIO_flush can result in the thread blocking while waiting for output calls to be completed.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- GIO_flush cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

Example

```
/* Flush all I/O given to the device*/
GIO_flush(gioChan);
```

GIO_new *Initialize a GIO object with pre-allocated memory*

C Interface

Syntax

gioChan = GIO_new(gioChan, name, mode, *status, optArgs,
packetBuf[], syncObject, *attrs);

Parameters

GIO_Handle	gioChan	/* Handle to GIO Obj */
String	name	/* name of the device to open */
Int	mode	/* mode in which the device is to be opened */
Int	*status	/* address to place driver return status */
Ptr	optArgs	/* optional args to mdCreateChan */
IOM_packet	packetBuf[]	/* to be initialized to zero */
Ptr	syncObject	/* sync Object */
GIO_Attrs	*attrs	/* pointer to a GIO_Attrs structure */

Return Value

GIO_Handle	gioChan;	/* handle to the initialized GIO object */

Description

An application calls GIO_new to initialize a GIO_Obj object and open a communication channel. This function initializes the I/O channel and opens the lower-level device driver channel. The GIO_new call *does not* allocate any memory. It requires pre-allocated memory.

The "gioChan" parameter is a handle to a structure of type GIO_Obj that your program has declared. GIO_new initializes this structure.

```
typedef struct GIO_Obj {
   IOM_Fxns   *fxns;      /* ptr to function table */
   Uns        mode;       /* create mode */
   Uns        timeout;    /* timeout for blocking */
   IOM_Packet syncPacket; /* for synchronous use */
   QUE_Obj    freeList;   /* frames for asynch I/O */
   Ptr        syncObj;    /* ptr to synchro. obj */
   Ptr        mdChan;     /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;
```

The "name" parameter is the name previously specified for the device. It is used to find a matching name in the device table.

The "mode" parameter specifies the mode in which the device is to be opened. This may be IOM_INPUT, IOM_OUTPUT, or IOM_INOUT.

If the status returned by the device is non-NULL, a status value is placed at the address specified by the "status" parameter.

The "optArgs" parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner.

Use the "packetBuf[]" array to pass a list of IOM_Packet items. The number of items should match the nPackets member of the GIO_Attrs structure passed to the "attrs" parameter. GIO_new initializes these IOM_Packet items.

The "syncObject" parameter is usually a SEM handle.

The "attrs" parameter is a pointer to a structure of type GIO_Attrs.

```
typedef struct GIO_Attrs {
  Int nPackets; /* number of asynch I/O packets */
  Uns timeout; /* for blocking calls (SYS_FOREVER) */
} GIO_Attrs;
```

If attrs is NULL, a default set of attributes is used. The default for nPackets is 2. The default for timeout is SYS_FOREVER. GIO_new initializes the packets, but does not allocate them.

GIO_new returns the non-NULL handle to the GIO_Obj when initialization is successful. The handle returned by this call should be used by the application in subsequent calls to GIO functions. Usually, this is the same handle passed to GIO_new. However, GIO_new returns a NULL handle if the device could not be initialized. For example, if a device is opened in a mode not supported by the device, this call returns a NULL handle.

A call to GIO_new results in a call to the mdCreateChan function of the associated mini-driver.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized.

Example

```
/* Initialize a device object */
output = GIO_new(&outObj, "/printf", IOM_OUTPUT,
   &status, NULL, outPacketBuf, outSem, &attrs);
```

GIO_create

GIO_read

Synchronous read command

C Interface

Syntax

status = GIO_read(gioChan, bufp, *pSize);

Parameters

GIO_Handle	gioChan;	/* handle to an instance of the device */
Ptr	bufp	/* pointer to data structure for buffer data */
size_t	*pSize	/* pointer to size of bufp structure */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

An application calls GIO_read to read a specified number of MADUs (minimum addressable data units) from the communication channel.

The gioChan parameter is the handle returned by GIO_create or GIO_new.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the read data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data. For example, for video capture devices this structure may contain pointers to RGB buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of MADUs read from the device. This parameter is relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

GIO_read returns IOM_COMPLETED upon successfully reading the requested number of MADUs from the device. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_read results in a call to the mdSubmit function of the associated mini-driver. The IOM_READ command is passed to the mdSubmit function. The mdSubmit call is typically a blocking call, so calling GIO_read can result in the thread blocking.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- GIO_read cannot be called from a SWI, HWI, or main() unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

Example

```
/* Read from the device */
size = sizeof(readStruct);
status = GIO_read(gioChan, &readStruct, &size);
```

GIO_submit *Submit a GIO packet to the mini-driver*

C Interface

Syntax

```
status = GIO_submit(gioChan, cmd, bufp, *pSize, *appCallback);
```

Parameters

```
GIO_Handle          gioChan;       /* handle to an instance of the device */
Uns                 cmd            /* specified mini-driver command */
Ptr                 bufp           /* pointer to data structure for buffer data */
size_t              *pSize         /* pointer to size of bufp structure */
GIO_AppCallback        *appCallback   /* pointer to callback structure */
```

Return Value

```
Int                 status;        /* returns IOM_COMPLETED if successful */
```

Description

GIO_submit is not typically called by applications. Instead, it is used internally and for user-defined extensions to the GIO module.

GIO_read and GIO_write are macros that call GIO_submit with appCallback set to NULL. This causes GIO to complete the I/O request synchronously using its internal synchronization object (by default, a semaphore). If appCallback is non-NULL, the specified callback is called without blocking. This API is provided to extend GIO functionality for use with SWI threads without changing the GIO implementation.

The gioChan parameter is the handle returned by GIO_create or GIO_new.

The cmd parameter is one of the command code constants listed in “Constants, Types, and Structures” on page 117. A mini-driver may add command codes for additional functionality.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of MADUs transferred to or from the device. This parameter is relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

The appCallback parameter points to either a callback structure that contains the callback function to be called when the request completes, or it points to NULL, which causes the call to be synchronous. When a queued request is completed, the callback routine (if specified) is invoked (i.e. blocking).

GIO_submit returns IOM_COMPLETED upon successfully carrying out the requested functionality. If the request is queued, then a status of IOM_PENDING is returned. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_submit results in a call to the mdSubmit function of the associated mini-driver. The specified command is passed to the mdSubmit function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- This function can be called within the program's main() function only if the GIO channel is asynchronous (non-blocking).

Example

```
/* write asynchronously to the device*/
size = sizeof(userStruct);
status = GIO_submit(gioChan, IOM_WRITE, &userStruct,
            &size, &callbackStruct);

/* write synchronously to the device */
size = sizeof(userStruct);
status = GIO_submit(gioChan, IOM_WRITE, &userStruct,
            &size, NULL);
```

GIO_write *Synchronous write command*

C Interface

Syntax

status = GIO_write(gioChan, bufp, *pSize);

Parameters

GIO_Handle	gioChan;	/* handle to an instance of the device */
Ptr	bufp	/* pointer to data structure for buffer data */
size_t	*pSize	/* pointer to size of bufp structure */

Return Value

Int	status;	/* returns IOM_COMPLETED if successful */

Description

The application uses this function to write a specified number of MADUs to the communication channel.

The gioChan parameter is the handle returned by GIO_create or GIO_new.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the write data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be written to, and other device-dependent data. For example, for video capture devices this structure may contain pointers to RGB buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of MADUs written to the device. This parameter is relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

GIO_write returns IOM_COMPLETED upon successfully writing the requested number of MADUs to the device. If an error occurs, the device returns a negative value. For a list of error values, see “Constants, Types, and Structures” on page 117.

A call to GIO_write results in a call to the mdSubmit function of the associated mini-driver. The IOM_WRITE command is passed to the mdSubmit function. The mdSubmit call is typically a blocking call, so calling GIO_write can result in blocking.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to GIO_create or GIO_new.
- This function can be called within the program’s main() function only if the GIO channel is asynchronous (non-blocking).
- GIO_write cannot be called from a SWI or HWI unless the underlying mini-driver is a non-blocking driver and the GIO Manager properties are set to use non-blocking synchronization methods.

Example

```
/* write synchronously to the device*/
size = sizeof(writeStruct);
status = GIO_write(gioChan, &writeStrct, &size);
```

2.8 HOOK Module

The HOOK module is the Hook Function manager.

Functions

- HOOK_getenv. Get environment pointer for a given HOOK and TSK combination.
- HOOK_setenv. Set environment pointer for a given HOOK and TSK combination.

Constants, Types, and Structures

```
typedef Int HOOK_Id;        /* HOOK instance id */

typedef Void (*HOOK_InitFxn)(HOOK_Id id);
typedef Void (*HOOK_CreateFxn)(TSK_Handle task);
typedef Void (*HOOK_DeleteFxn)(TSK_Handle task);
typedef Void (*HOOK_ExitFxn)(Void);
typedef Void (*HOOK_ReadyFxn)(TSK_Handle task);
typedef Void (*HOOK_SwitchFxn)(TSK_Handle prev,
     TSK_Handle next);
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the HOOK Object Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
initFxn	Extern	prog.extern("FXN_F_nop")
createFxn	Extern	prog.extern("FXN_F_nop")
deleteFxn	Extern	prog.extern("FXN_F_nop")
exitFxn	Extern	prog.extern("FXN_F_nop")
callSwitchFxn	Bool	false
switchFxn	Extern	prog.extern("FXN_F_nop")
callReadyFxn	Bool	false
readyFxn	Extern	prog.extern("FXN_F_nop")
order	Int16	2

Description

The HOOK module is an extension to the TSK function hooks defined in the TSK Manager Properties. It allows multiple sets of hook functions to be performed at key execution points. For example, an application that integrates third-party software may need to perform both its own hook functions and the hook functions required by the third-party software.

In addition, each HOOK object can maintain private data environments for each task for use by its hook functions.

The key execution points at which hook functions can be executed are during program initialization and at several TSK execution points.

The HOOK module manages objects that reference a set of hook functions. Each HOOK object is assigned a numeric identifier during DSP/BIOS initialization. If your program calls HOOK API functions, you must implement an initialization function for the HOOK instance that records the identifier in a variable of type HOOK_Id. DSP/BIOS passes the HOOK object's ID to the initialization function as the lone parameter.

The following function, myInit, could be configured as the Initialization function for a HOOK object using Tconf.

```
#include <hook.h>
HOOK_Id myId;

Void myInit(HOOK_Id id)
{
   myId = id;
}
```

The HOOK_setenv function allows you to associate an environment pointer to any data structure with a particular HOOK object and TSK object combination.

There is no limit to the number of HOOK objects that can be created. However, each object requires a small amount of memory in the .bss section to contain the object.

A HOOK object initially has all of its functions set to FXN_F_nop. You can set some hook functions and use this no-op function for the remaining events. Since the switch and ready events occur frequently during real-time processing, a separate property controls whether any function is called.

When you create a HOOK object, any TSK module hook functions you have specified are automatically placed in a HOOK object called HOOK_KNL. To set any properties of this object other than the Initialization function, use the TSK module. To set the Initialization function property of the HOOK_KNL object, use the HOOK module.

When an event occurs, all HOOK functions for that event are called in the order set by the order property in the configuration. When you select the HOOK manager in the DSP/BIOS Configuration Tool, you can change the execution order by dragging objects within the ordered list.

HOOK Manager Properties

There are no global properties for the HOOK manager. HOOK objects are placed in the C Variables Section (.bss).

HOOK Object Properties

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

```
var myHook = bios.HOOK.create("myHook");
```

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

- **comment**. A comment to identify this HOOK object.

  Tconf Name: comment Type: String

  Example: `myHook.comment = "HOOK funcs";`

- **Initialization function**. The name of a function to call during program initialization. Such functions run during the BIOS_init portion of application startup, which runs before the program's main() function. Initialization functions can call most functions that can be called from the main() function.

However, they should not call TSK module functions, because the TSK module is initialized after initialization functions run. In addition to code specific to the module hook, this function should be used to record the object's ID, if it is needed in a subsequent hook function. This initialization function is intended for use in setting up data structures used by other functions of the same HOOK object. In contrast, the User Init Function property of the GBL Module Properties runs early in the initialization process and is intended to be used to perform hardware setup that needs to run before DSP/BIOS is initialized.

Tconf Name: initFxn Type: Extern

Example: `myHook.initFxn = prog.extern("myInit");`

- **Create function**. The name of a function to call when any task is created. This includes tasks that are created statically and those created dynamically using TSK_create. The TSK_create topic describes the prototype required for the Create function. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally.

  Tconf Name: createFxn Type: Extern

  Example: `myHook.createFxn = prog.extern("myCreate");`

- **Delete function**. The name of a function to call when any task is deleted at run-time with TSK_delete.

  Tconf Name: deleteFxn Type: Extern

  Example: `myHook.deleteFxn = prog.extern("myDelete");`

- **Exit function**. The name of a function to call when any task exits. The TSK_exit topic describes the Exit function.

  Tconf Name: exitFxn Type: Extern

  Example: `myHook.exitFxn = prog.extern("myExit");`

- **Call switch function**. Set this property to true if you want a function to be called when any task switch occurs.

  Tconf Name: callSwitchFxn Type: Bool

  Example: `myHook.callSwitchFxn = false;`

- **Switch function**. The name of a function to call when any task switch occurs. This function can give the application access to both the current and next task handles. The TSK Module topic describes the Switch function.

  Tconf Name: switchFxn Type: Extern

  Example: `myHook.switchFxn = prog.extern("mySwitch");`

- **Call ready function**. Set this property to true if you want a function to be called when any task becomes ready to run.

  Tconf Name: callReadyFxn Type: Bool

  Example: `myHook.callReadyFxn = false;`

- **Ready function**. The name of a function to call when any task becomes ready to run. The TSK Module topic describes the Ready function.

  Tconf Name: readyFxn Type: Extern

  Example: `myHook.readyFxn = prog.extern("myReady");`

- **order**. Set this property for all HOOK function objects match the order in which HOOK functions should be executed.

  Tconf Name: order Type: Int16

  Example: `myHook.order = 2;`

HOOK_getenv *Get environment pointer for a given HOOK and TSK combination*

C Interface

Syntax

```
environ = HOOK_getenv(task, id);
```

Parameters

```
TSK_Handle          task;      /* task object handle */
HOOK_Id             id;        /* HOOK instance id */
```

Return Value

```
Ptr                 environ;   /* environment pointer */
```

Reentrant

yes

Description

HOOK_getenv returns the environment pointer associated with the specified HOOK and TSK objects. The environment pointer, environ, references the data structure specified in a previous call to HOOK_setenv.

See Also

HOOK_setenv
TSK_getenv

HOOK_setenv *Set environment pointer for a given HOOK and TSK combination*

C Interface

Syntax

HOOK_setenv(task, id, environ);

Parameters

TSK_Handle	task;	/* task object handle */
HOOK_Id	id;	/* HOOK instance id */
Ptr	environ;	/* environment pointer */

Return Value

Void

Reentrant

yes

Description

HOOK_setenv sets the environment pointer associated with the specified HOOK and TSK objects to environ. The environment pointer, environ, should reference an data structure to be used by the hook functions for a task or tasks.

Each HOOK object may have a separate environment pointer for each task. A HOOK object may also point to the same data structure for all tasks, depending on its data sharing needs.

The HOOK_getenv function can be used to get the environ pointer for a particular HOOK and TSK object combination.

See Also

HOOK_getenv
TSK_setenv

2.9 HST Module

Important: This module is being deprecated and will no longer be supported in the next major release of DSP/BIOS.

The HST module is the host channel manager.

Functions

- HST_getpipe. Get corresponding pipe object

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the HST Manager Properties and HST Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
OBJMEMSEG	Reference	prog.get("L0SARAM")
HOSTLINKTYPE	EnumString	"RTDX" ("NONE")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
mode	EnumString	"output" ("input")
bufSeg	Reference	prog.get("L0SARAM")
bufAlign	Int16	4
frameSize	Int16	128
numFrames	Int16	2
statistics	Bool	false
availableForDHL	Bool	false
notifyFxn	Extern	prog.extern("FXN_F_nop")
arg0	Arg	3

Description

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

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

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

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

HST Manager Properties

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

- **Object Memory**. The memory segment containing HST objects.

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Host Link Type**. The underlying physical link to be used for host-target data transfer. If None is selected, no instrumentation or host channel data is transferred between the target and host in real time. The Analysis Tool windows are updated only when the target is halted (for example, at a breakpoint). The program code size is smaller when the Host Link Type is set to None because RTDX code is not included in the program.

  Tconf Name: HOSTLINKTYPE Type: EnumString

  Options: "RTDX", "NONE"

  Example: `bios.HST.HOSTLINKTYPE = "RTDX";`

HST Object Properties

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

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

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

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

- **comment**. A comment to identify this HST object.

  Tconf Name: comment Type: String

  Example: `myHst.comment = "my HST";`

- **mode.** The type of channel: input or output. Input channels are used by the target to read data from the host; output channels are used by the target to transfer data from the target to the host.

  Tconf Name: mode Type: EnumString

  Options: "output", "input"

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

- **bufseg.** The memory segment from which the buffer is allocated; all frames are allocated from a single contiguous buffer (of size framesize x numframes).

  Tconf Name: bufSeg Type: Reference

  Example: `myHst.bufSeg = prog.get("myMEM");`

- **bufalign.** The alignment (in words) of the buffer allocated within the specified memory segment.

  Tconf Name: bufAlign Type: Int16

  Options: must be >= 4 and a power of 2

  Example: `myHst.bufAlign = 4;`

- **framesize.** The length of each frame (in words)

  Tconf Name: frameSize Type: Int16

  Example: `myHst.frameSize = 128;`

- **numframes.** The number of frames

  Tconf Name: numFrames Type: Int16

  Example: `myHst.numFrames = 2;`

- **statistics.** Set this property to true if you want to monitor this channel with an STS object. You can display the STS object for this channel to see a count of the number of frames transferred with the Statistics View Analysis Tool.

  Tconf Name: statistics Type: Bool

  Example: `myHst.statistics = false;`

- **Make this channel available for a new DHL device.** Set this property to true if you want to use this HST object with a DHL device. DHL devices allow you to manage data I/O between the host and target using the SIO module, rather than the PIP module. See the DHL Driver topic for more details.

  Tconf Name: availableForDHL Type: Bool

  Example: `myHst.availableForDHL = false;`

- **notify.** The function to execute when a frame of data for an input channel (or free space for an output channel) is available. To avoid problems with recursion, this function should not directly call any of the PIP module functions for this HST object.

  Tconf Name: notifyFxn Type: Extern

  Example: `myHst.notifyFxn = prog.extern("hstNotify");`

  Tconf Name: arg0 Type: Arg

  Tconf Name: arg1 Type: Arg

  Example: `myHst.arg0 = 3;`

HST_getpipe

Get corresponding pipe object

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS.

C Interface

Syntax

pipe = HST_getpipe(hst);

Parameters

HST_Handle	hst	/* host object handle */

Return Value

PIP_Handle	pip	/* pipe object handle*/

Reentrant

yes

Description

HST_getpipe gets the address of the pipe object for the specified host channel object.

Example

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

    in = HST_getpipe(input);
    out = HST_getpipe(output);
    if (PIP_getReaderNumFrames == 0 ||
        PIP_getWriterNumFrames == 0) {
           error;
    }

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

    /* copy input data to output frame */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);
    size = PIP_getReaderSize();
    out->writerSize = size;

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

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

See Also

PIP_alloc

2.10 HWI Module

The HWI module is the hardware interrupt manager.

Functions

- HWI_disable. Disable hardware interrupts
- HWI_dispatchPlug. Plug the HWI dispatcher
- HWI_enable. Enable hardware interrupts
- HWI_enter. Hardware ISR prolog
- HWI_exit. Hardware ISR epilog
- HWI_isHWI. Check current thread calling context.
- HWI_restore. Restore hardware interrupt state

Constants, Types, and Structures

```
typedef struct HWI_Attrs {
   Uns   iermask;      /* IER bitmask, 1 = "self" (default)
   Arg   arg;          /* fxn arg (default = 0)*/
}  HWI_Attrs;

HWI_Attrs HWI_ATTRS = {
    1,           /* interrupt mask (1 => self) */
    0            /* argument to ISR */
};
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the HWI Manager Properties and HWI Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
VMAPMODE	EnumInt	1 (0)
PIESELECT	Bool	true

Instance Configuration Parameters

HWI instances are provided as a default part of the configuration and cannot be created. In the items that follow, HWI_INT* may be any provided instance. Default values for many HWI properties are different for each instance

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
fxn	Extern	prog.extern("HWI_unused", "asm")
monitor	EnumString	"Nothing" ("Data Value", "Stack Pointer", "ah", "al", "xar0", "xar1", "xar2", "xar3", "xar4", "xar5", "xar6", "xar7", "dp", "ifr", "ier", "ph", "pl", "st0", "st1", "t", "tl")
addr	Arg	0x00000000

Name	Type	Default (Enum Options)
dataType	EnumString	"signed" ("unsigned")
operation	EnumString	"STS_add(*addr)" ("STS_delta(*addr)", "STS_add(-*addr)", "STS_delta(-*addr)", "STS_add(|*addr|)", "STS_delta(|*addr|)")
useDispatcher	Bool	false
arg	Arg	3
interruptMask	EnumString	"self" ("all", "none", "bitmask")
interruptBitMask	Numeric	0x0010 *

* Depends on interrupt ID

Description

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

You can also dynamically assign routines to interrupts at run-time using the HWI_dispatchPlug function or the C28_plug function.

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

Note: **RTS Functions Callable from TSK Threads Only.** Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK_pend and LCK_post. As a result, RTS functions that call LCK_pend or LCK_post *must not be called in the context of a SWI or HWI thread*. For a list or RTS functions that should not be called from a SWI or an HWI function, see “LCK_pend” on page 169.

The C++ "new" operator calls malloc, which in turn calls LCK_pend. As a result, the "new" operator cannot be used in the context of a SWI or HWI thread.

HWI Dispatcher vs. HWI_enter/exit

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

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

When an HWI object is configured to use the dispatcher, the dispatcher handles the HWI_enter prolog and HWI_exit epilog, and the HWI function can be completely written in C. It would, in fact, cause a system crash for the dispatcher to call a function that contains the HWI_enter/HWI_exit macro pair. Using the dispatcher allows you to save code space by including only one instance of the HWI_enter/HWI_exit code.

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

Notes

In the following notes, references to the usage of HWI_enter/HWI_exit also apply to usage of the HWI dispatcher since, in effect, the dispatcher calls HWI_enter/HWI_exit.

- Do not call SWI_disable or SWI_enable within an HWI function.
- Do not call HWI_enter, HWI_exit, or any other DSP/BIOS functions from a non-maskable interrupt (NMI) service routine. In addition, the HWI dispatcher cannot be used with the NMI service routine.
- Do not call HWI_enter/HWI_exit from a HWI function that is invoked by the dispatcher.
- The DSP/BIOS API calls that require an HWI function to use HWI_enter and HWI_exit are:
  - SWI_andn
  - SWI_andnHook
  - SWI_dec
  - SWI_inc
  - SWI_or
  - SWI_orHook
  - SWI_post
  - PIP_alloc
  - PIP_free
  - PIP_get
  - PIP_put
  - PRD_tick
  - SEM_post
  - MBX_post
  - TSK_yield
  - TSK_tick

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

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

  If your HWI function and the functions it calls do not call any of these API operations, you do not need to disable SWI scheduling by calling HWI_enter and HWI_exit.

DSP/BIOS and NMI Support

On the C28x, DSP/BIOS does not support returning from an NMI interrupt when the PIE is enabled and other interrupts are using the dispatcher.

When a PIE interrupt occurs, the dispatcher determines which interrupt to service by looking at the PIECTRL register. If a PIE interrupt, such as the ADC interrupt, occurs and then an NMI occurs before the dispatcher has had a chance to read the value of the PIECTRL register, the NMI will overwrite the value and break the dispatcher.

Typically, the NMI is intended for critical failures, and the system is not intended to return from an NMI. If the system must be able to return from an NMI, the workaround is to not use the dispatcher for any interrupts. The dispatcher can be avoided one of two ways:

- If an ISR does not call any DSP/BIOS functions, it does not need to go through the dispatcher.
- Otherwise, the ISR can be written using the HWI_enter and HWI_exit assembly macros. For ISRs written in C, this requires a small assembly function that calls the macros and the ISR.

Registers and Stack

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

The register mask argument to HWI_enter and HWI_exit allows you to save and restore registers used within the function. Other arguments, for example, allow the HWI to control the settings of the IER.

Note: By using HWI_enter and HWI_exit as an HWI function's prolog and epilog, an HWI function can be interrupted; that is, a hardware interrupt can interrupt another interrupt. For the c28x device, you can use the IERDISABLEMASK parameter to prevent this from occurring.

HWI Manager Properties

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

The following global properties can be set for the HWI module in the HWI Manager Properties dialog of Gconf or in a Tconf script:

- **VMAP Mode.** Select the VMAP Mode used for the application: 0 or 1. VMAP Mode determines whether the CPU interrupt vectors (including the reset vector) are mapped to the lowest or highest addresses in program memory. On reset VMAP Mode is 1.
  - 0. CPU interrupt vectors are mapped to the bottom of program memory, addresses: 00 0000h-00 0003Fh.
  - 1. CPU interrupt vectors are mapped to the top of program memory, addresses: 3F FFC0h-3F FFFFh

  Tconf Name: VMAPMODE Type: Enum

  Options: 0, 1

  Example: `bios.HWI.VMAPMODE = 0;`

- **Enable PIE.** If this property is set to true, DSP/BIOS sets the ENPIE bit in the PIE control register (PIECTRL) to 1 during the DSP/BIOS startup process. If this property is set to false, DSP/BIOS does not set the ENPIE bit. The application should enable individual PIE interrupts in the PIEIERx register and the corresponding interrupt in the IER register.

  Tconf Name: PIESELECT Type: Bool

  Example: `bios.HWI.PIESELECT = true;`

If you do not need to use PIE interrupts and want to map the CPU interrupt vectors to the lowest memory address, you should make the following configuration settings:

```
bios.HWI.VMAPMODE = 0;            /* set VMAP mode to zero */
bios.HWI.PIESELECT = false;       /* disable PIE interrupts */
bios.PIEVECT.base = 0x000000;     /* set CPU interrupt vector address */
bios.bios.MSARAM.base = 0x000040; /* move BIOS stack base address away from 0x0 */
bios.MSARAM.len = 0x0700;
```

HWI Object Properties

The following properties can be set for an HWI object in the HWI Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script. The HWI objects for the platform are provided in the default configuration and cannot be created.

- **comment**. A comment is provided to identify each HWI object.

  Tconf Name: comment Type: String

  Example: `bios.HWI_INT2.comment = "myISR";`

- **function**. The function to execute. Interrupt routines that use the dispatcher can be written completely in C or any combination of assembly and C but must not call the HWI_enter/HWI_exit macro pair. Interrupt routines that don't use the dispatcher must be written at least partially in assembly language. Within an HWI function that does not use the dispatcher, the HWI_enter assembly macro must be called prior to any DSP/BIOS API calls that affect other DSP/BIOS objects, such as posting a SWI or a semaphore. HWI functions can post SWIs, but they do not run until your HWI function (or the dispatcher) calls the HWI_exit assembly macro, which must be the last statement in any HWI function that calls HWI_enter.

  Tconf Name: fxn Type: Extern

  Example: `bios.HWI_INT2.fxn = prog.extern("myHWI", "asm");`

- **monitor**. If set to anything other than Nothing, an STS object is created for this HWI that is passed the specified value on every invocation of the HWI function. The STS update occurs just before entering the HWI routine.

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

  Options: "Nothing", "Data Value", "Stack Pointer", "ah", "al", "xar0", "xar1", "xar2", "xar3", "xar4", "xar5", "xar6", "xar7", "dp", "ifr", "ier", "ph", "pl", "st0", "st1", "t", "tl"

  Example: `bios.HWI_INT2.monitor = "Nothing";`

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

  Tconf Name: addr Type: Arg

  Example: `bios.HWI_INT2.addr = 0x00000000;`

- **type**. The type of the value to be monitored: unsigned or signed. Signed quantities are sign extended when loaded into the accumulator; unsigned quantities are treated as word-sized positive values.

  Tconf Name: dataType Type: EnumString

  Options: "signed", "unsigned"

  Example: `bios.HWI_INT2.dataType = "signed";`

- **operation**. The operation to be performed on the value monitored. You can choose one of several STS operations.

  Tconf Name: operation Type: EnumString

  Options: "STS_add(*addr)", "STS_delta(*addr)", "STS_add(-*addr)", "STS_delta(-*addr)", "STS_add(|*addr|)", "STS_delta(|*addr|)"

  Example:
  ```
  bios.HWI_INT2.operation = "STS_add(*addr)";
  ```

- **Use Dispatcher**. A check box that controls whether the HWI dispatcher is used. The HWI dispatcher cannot be used for the non-maskable interrupt (NMI) service routine.

  Tconf Name: useDispatcher Type: Bool

  Example:
  ```
  bios.HWI_INT2.useDispatcher = false;
  ```

- **Arg**. This argument is passed to the function as its only parameter. You can use either a literal integer or a symbol defined by the application. This property is available only when using the HWI dispatcher.

  Tconf Name: arg Type: Arg

  Example:
  ```
  bios.HWI_INT2.arg = 3;
  ```

- **Interrupt Mask**. Specifies which interrupts the dispatcher should disable before calling the function. This property is available only when using the HWI dispatcher.

  — The "self" option causes the dispatcher to disable only the current interrupt.

  — The "all" option disables all interrupts.

  — The "none" option disables no interrupts.

  — The "bitmask" option causes the interruptBitMask property to be used to specify which interrupts to disable.

  Tconf Name: interruptMask Type: EnumString

  Options: "self", "all", "none", "bitmask"

  Example:
  ```
  bios.HWI_INT2.interruptMask = "self";
  ```

- **Interrupt Bit Mask**. An integer property that is writable when the interrupt mask is set to "bitmask". This should be a hexadecimal integer bitmask specifying the interrupts to disable.

  Tconf Name: interruptBitMask Type: Numeric

  Example:
  ```
  bios.HWI_INT2.interruptBitMask = 0x0010;
  ```

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

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

Table 2-3 lists these precreated objects and their default interrupt sources. The HWI object names are the same as the interrupt names.

Table 2-3: HWI Interrupts for the ‘C28x

Name	Interrupt Type
HWI_RESET	Reset interrupt.
HWI_INT1 through HWI_INT13	Maskable (IER, bit0) hardware interrupt through Maskable (IER, bit12) hardware interrupt.

Name	Interrupt Type
HWI_TINT	Timer interrupt. (IER, bit13)
HWI_DLOG	Maskable (IER, bit14) data log interrupt.
HWI_RTOS	Maskable (IER1, bit10) RTOS interrupt.
HWI_RESERVED	Reserved for use by RTDX
HWI_NMI	Non-maskable interrupt.
HWI_ILLEGAL	Illegal interrupt
HWI_USER1 through HWI_USER12	Non-maskable user-defined software interrupts.
PIE_INT1_1 through PIE_INT12_8	Peripheral Interrupt Expansion interrupts 1.1 to 12.8. (That is 1.1 to 1.8, 2.1 to 2.8, ..., and 12.1 to 12.8)

HWI_disable *Disable hardware interrupts*

C Interface

Syntax
oldST1 = HWI_disable();

Parameters
Void

Return Value
Uns oldST1;

Reentrant

yes

Description

HWI_disable disables hardware interrupts by setting the intm bit in the status register. Call HWI_disable before a portion of a function that needs to run without interruption. When critical processing is complete, call HWI_restore or HWI_enable to reenable hardware interrupts.

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

A context switch can occur when calling HWI_enable or HWI_restore if an enabled interrupt occurred while interrupts are disabled.

HWI_disable may be called from main(). However, since HWI interrupts are already disabled in main(), such a call has no effect.

Example

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

See Also

HWI_enable
HWI_restore
SWI_disable
SWI_enable

HWI_dispatchPlug *Plug the HWI dispatcher*

C Interface

Syntax

HWI_dispatchPlug(vecid, fxn, attrs);

Parameters

Int	vecid;	/* interrupt id */
Fxn	fxn;	/* pointer to HWI function */
HWI_Attrs	*attrs	/*pointer to HWI dispatcher attributes */

Return Value

Void

Reentrant

yes

Description

HWI_dispatchPlug fills the HWI dispatcher table with the function specified by the fxn parameter and the attributes specified by the attrs parameter.

HWI_dispatchPlug also writes two instruction words into the Interrupt-Vector Table, at the address corresponding to vecid. The instructions written in the Interrupt-Vector Table create a call to the HWI dispatcher.

This API can plug the full set of vectors supported by the PIE (0-127). If the PIE is not enabled, the vector is plugged in the default vector area. If the PIE is enabled, the vector is plugged in the PIE vector area.

HWI_dispatchPlug does not enable the interrupt. Use C28_enableIER to enable specific interrupts.

If attrs is NULL, the HWI's dispatcher properties are assigned a default set of attributes. Otherwise, the HWI's dispatcher properties are specified by a structure of type HWI_Attrs defined as follows.

```
typedef struct HWI_Attrs {
   Uns   iermask;     /* IER bitmask, 1 = "self" (default)
   Arg   arg;         /* fxn arg (default = 0)*/
}  HWI_Attrs;
```

The iermask bitmask specifies ier interrupts to mask while executing the HWI. The bit positions in iermask correspond to those of IER.

The default values are defined as follows:

```
HWI_Attrs HWI_ATTRS = {
    1,            /* interrupt mask (1 => self) */
    0             /* argument to ISR */
};
```

The arg element is a generic argument that is passed to the plugged function as its only parameter. The default value is 0.

Constraints and Calling Context

- vecid must be a valid interrupt ID in the range of 0-127.

See Also

HWI_enable
HWI_restore
SWI_disable
SWI_enable

HWI_enable *Enable interrupts*

C Interface

Syntax

HWI_enable();

Parameters

Void

Return Value

Void

Reentrant

yes

Description

HWI_enable enables hardware interrupts by clearing the intm bit in the status register.

Hardware interrupts are enabled unless a call to HWI_disable disables them. DSP/BIOS enables hardware interrupts after the program’s main() function runs. Your main() function can enable individual interrupt mask bits, but it should not call HWI_enable to globally enable interrupts.

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

Any call to HWI_enable enables interrupts, even if HWI_disable has been called several times.

Constraints and Calling Context

- HWI_enable cannot be called from the program’s main() function.

Example

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

See Also

HWI_disable
HWI_restore
SWI_disable
SWI_enable

HWI_enter *Hardware ISR prolog*

C Interface

Syntax
none

Parameters
none

Return Value
none

Assembly Interface

Syntax
HWI_enter AR_MASK, ACC_MASK, MISC_MASK, IERDISABLEMASK

Preconditions
intm = 1

Postconditions
arp = 000 [Auxiliary register pointer points to AR0]
pm = 1 [Product shift mode is 1]
ovm = 0 [Overflow Mode is Normal]
sxm = 0 [Sign Extension Mode disabled]
The M0M1MAP, OBJMODE, AMODE are to be set for 28x.
page0 = 0 [Stack addressing mode]
SP is aligned to an even address boundary.

Modifies
ah, ier, sp

Reentrant

yes

Description

HWI_enter is an API (assembly macro) used to save the appropriate context for a DSP/BIOS hardware interrupt (HWI).

The arguments to HWI_enter are bitmasks that define the set of registers to be saved and bitmasks that define which interrupts are to be masked during the execution of the HWI.

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

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

The following are the definitions of the masks in the HWI_enter and HWI_exit API syntax:

- **AR_MASK.** Mask of registers belonging to xar0-7

- **ACC_MASK.** Mask of registers belonging to acc, p, t
- **MISC_MASK.** Mask of registers ier, ifr, dbier, st0, st1,dp
- **IERDISABLEMASK.** Mask of ier bits to turn off

Note: The C28_saveCcontext, C28_restoreCcontext C28_saveBiosContext and C28_restoreBiosContext macros preserve processor register context per C and DSP/BIOS requirements, respectively.

Constraints and Calling Context

- This API should not be used in the NMI HWI function.
- This API must not be called if the HWI object that runs this function uses the HWI dispatcher.
- This API cannot be called from the program's main() function.
- This API cannot be called from a SWI, TSK, or IDL function.
- This API cannot be called from a CLK function.
- Unless the HWI dispatcher is used, this API must be called within any hardware interrupt function (except NMI's HWI function) before the first operation in an HWI that uses any DSP/BIOS API calls that might post or affect a SWI or semaphore. Such functions must be written in assembly language. Alternatively, the HWI dispatcher can be used instead of this API, allowing the function to be written completely in C and allowing you to reduce code size.
- If an interrupt function calls HWI_enter, it must end by calling HWI_exit.
- Do not use the interrupt keyword or the INTERRUPT pragma in C functions that run in the context of an HWI.
-

Examples

Example #1:

```
.include c28.h28

AR_MASK_clk           .set C28_AR_CALLER_MASK
ACC_MASK_clk          .set C28_ALL_ACC_REGS
MISC_MASK_clk         .set C28_MISC_CALLER_MASK
IERDISABLEMASK_clk    .set 0008h

CLK_isr:
HWI_enter AR_MASK_clk, ACC_MASK_clk, MISC_MASK_clk, \
IERDISABLEMASK_clk
PRD_tick
HWI_exit AR_MASK_clk, ACC_MASK_clk, MISC_MASK_clk, \
IERDISABLEMASK_clk
```

Example #2: Calling a C function from within an HWI_enter/HWI_exit block:

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

```
HWI_enter C28_AR_CALLER_MASK, C28_AC_CALLER_MASK, \
C28_MISC_CALLER_MASK, user_ier_mask
```

The HWI_enter macro

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

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

```
lcr _myCfunction;
```

When exiting the hardware interrupt, you need to call HWI_exit with the following macro:

```
HWI_exit C28_AR_CALLER_MASK, C28_AC_CALLER_MASK, \
C28_MISC_CALLER_MASK, user_ier_mask
```

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

See Also

HWI_exit

HWI_exit *Hardware ISR epilog*

C Interface

Syntax
none

Parameters
none

Return Value
none

Assembly Interface

Syntax
HWI_exit AR_MASK, ACC_MASK, MISC_MASK, IERDISABLEMASK

Preconditions
none

Postconditions
intm=0
all status bits set to values expected by C compiler

Modifies

Restores all registers saved with the HWI_enter mask

Reentrant

yes

Description

HWI_exit is an API (assembly macro) which is used to restore the context that existed before a DSP/BIOS hardware interrupt (HWI) was invoked.

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

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

HWI_exit restores the registers specified by AR_MASK, ACC_MASK and MISC_MASK. These masks are used to specify the set of registers that were saved by HWI_enter.

HWI_enter and HWI_exit must surround all statements in any DSP/BIOS assembly language HWIs that call C functions only for HWIs that are not dispatched by the HWI dispatcher.

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

Of the interrupts in IERDISABLEMASK, HWI_exit only restores those that were disabled upon entering the HWI. HWI_exit does not affect the status of interrupt bits that are not in IERDISABLEMASK.

- If upon exiting an HWI you do not wish to restore one of the interrupts that were disabled with HWI_enter, do not set that interrupt bit in the IERDISABLEMASK in HWI_exit.
- If upon exiting an HWI you do wish to enable an interrupt that was disabled upon entering the HWI, set the corresponding bit in IERDISABLEMASK before calling HWI_exit. (Simply setting bits in IERDISABLEMASK that is passed as argument to HWI_exit does not result in enabling the corresponding interrupts if those were not originally disabled by the HWI_enter macro.)

For a list of parameters and constants available for use with HWI_exit, see the description of HWI_enter. In addition, see the c28.h28 file.

Constraints and Calling Context

- This API should not be used for the NMI HWI function.
- This API must not be called if the HWI object that runs the function uses the HWI dispatcher.
- If the HWI dispatcher is not used, this API must be the last operation in an HWI that uses any DSP/BIOS API calls that might post or affect a SWI or semaphore. The HWI dispatcher can be used instead of this API, allowing the function to be written completely in C and allowing you to reduce code size.
- On the ’C28x platform, the AR_MASK, ACC_MASK and MISC_MASK parameters must match the corresponding parameters used for HWI_enter.
- This API cannot be called from the program’s main() function.
- This API cannot be called from a SWI, TSK, or IDL function.
- This API cannot be called from a CLK function.

Examples

Example #1:

```
.include C28.h28

AR_MASK_clk          .set C28_AR_CALLER_MASK
ACC_MASK_clk         .set C28_ALL_ACC_REGS
MISC_MASK_clk        .set C28_MISC_CALLER_MASK
IERDISABLEMASK_clk   .set 0008h
```

CLK_isr:

```
HWI_enter AR_MASK_clk, ACC_MASK_clk, MISC_MASK_clk,
IERDISABLEMASK_clk
PRD_tick
HWI_exit AR_MASK_clk, ACC_MASK_clk, MISC_MASK_clk, \
IERDISABLEMASK_clk
```

Example #2:

Calling a C function from within an HWI_enter/HWI_exit:

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

```
HWI_enter C28_AR_CALLER_MASK, C28_AC_CALLER_MASK, \
C28_MISC_CALLER_MASK, user_ier_mask
```

The HWI_enter macro

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

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

```
lcr _myCfunction;
```

When exiting the hardware interrupt, you need to call HWI_exit with the following macro:

```
HWI_exit C28_AR_CALLER_MASK, C28_AC_CALLER_MASK,  \
C28_MISC_CALLER_MASK, user_ier1_mask
```

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

See Also

HWI_enter

HWI_isHWI *Check to see if called in the context of an HWI*

C Interface

Syntax

```
result = HWI_isHWI(Void);
```

Parameters

Void

Return Value

Bool	result;	/* TRUE if in HWI context, FALSE otherwise */

Reentrant

yes

Description

This macro returns TRUE when it is called within the context of an HWI or CLK function. This macro returns FALSE in all other contexts.

In previous versions of DSP/BIOS, calling HWI_isHWI() from main() resulted in TRUE. This is no longer the case; main() is identified as part of the TSK context.

See Also

SWI_isSWI
TSK_isTSK

HWI_restore

Restore global interrupt enable state

C Interface

Syntax

HWI_restore(oldST1);

Parameters

Uns oldST1;

Returns

Void

Reentrant

yes

Description

HWI_restore sets the intm bit in the st1 register using bit 0 of the oldst1 parameter. If bit 0 is 1, the intm bit is not modified. If bit 0 is 0, the intm bit is set to 0, which enables interrupts.

When you call HWI_disable, the previous contents of the st1 register are returned. You can use this returned value with HWI_restore.

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

HWI_restore may be called from main(). However, since HWI_enable cannot be called from main(), interrupts are always disabled in main(), and a call to HWI_restore has no effect.

Constraints and Calling Context

- HWI_restore must be called with interrupts disabled. The parameter passed to HWI_restore must be the value returned by HWI_disable.

Example

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

See Also

HWI_enable
HWI_disable

2.11 IDL Module

The IDL module is the idle thread manager.

Functions

- IDL_run. Make one pass through idle functions.

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the IDL Manager Properties and IDL Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")
AUTOCALCULATE	Bool	true
LOOPINSTCOUNT	Int32	1000

Instance Configuration Parameters

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

Description

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

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

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

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

When RTA is enabled (see page 2–109), an application contains an IDL_cpuLoad object, which runs a function that provides data about the CPU utilization of the application. In addition, the LNK_dataPump function handles host I/O in the background, and the RTA_dispatch function handles run-time analysis communication.

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

IDL Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Auto calculate idle loop instruction count**. When this property is set to true, the program runs the IDL functions one or more times at system startup to get an approximate value for the idle loop instruction count. This value, saved in the global variable CLK_D_idletime, is read by the host and used in the CPU load calculation. By default, the instruction count includes all IDL functions, not just LNK_dataPump, RTA_dispatcher, and IDL_cpuLoad. You can remove an IDL function from the calculation by setting the "Include in CPU load calibration" property for an IDL object to false.

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

  — enabling hardware interrupts or the SWI or TSK schedulers
  — using CLK APIs to get the time
  — accessing PIP objects
  — blocking tasks
  — creating dynamic objects

  Tconf Name: AUTOCALCULATE Type: Bool

  Example: `bios.IDL.AUTOCALCULATE = true;`

- **Idle Loop Instruction Count**. This is the number of instruction cycles required to perform the IDL loop and the default IDL functions (LNK_dataPump, RTA_dispatcher, and IDL_cpuLoad) that communicate with the host. Since these functions are performed whenever no other processing is needed, background processing is subtracted from the CPU load before it is displayed.

  Tconf Name: LOOPINSTCOUNT Type: Int32

  Example: `bios.IDL.LOOPINSTCOUNT = 1000;`

IDL Object Properties

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

To create an IDL object in a configuration script, use the following syntax. The Tconf examples assume the object is created as shown here.

```
var myIdl = bios.IDL.create("myIdl");
```

The following properties can be set for an IDL object:

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

  Tconf Name: comment Type: String

  Example: `myIdl.comment = "IDL function";`

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

  Tconf Name: fxn Type: Extern

  Example: `myIdl.fxn = prog.extern("myIDL");`

- **Include in CPU load calibration**. You can remove an individual IDL function from the CPU load calculation by setting this property to false. The CPU load calibration is performed only if the "Auto calculate idle loop instruction count" property is true in the IDL Manager Properties. You should remove a function from the calculation if it blocks or depends on variables or structures that are not initialized until the main() function runs.

  Tconf Name: calibration Type: Bool

  Example: `myIdl.calibration = true;`

- **order**. Set this property for all IDL objects so that the numbers match the sequence in which IDL functions should be executed.

  Tconf Name: order Type: Int16

  Example: `myIdl.order = 2;`

IDL_run

Make one pass through idle functions

C Interface

Syntax

IDL_run();

Parameters

Void

Return Value

Void

Description

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

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

Note: BIOS_init and BIOS_start must be called before IDL_run to ensure that DSP/BIOS has been initialized. For example, the DSP/BIOS boot file contains the following system calls around the call to main:

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

Constraints and Calling Context

- IDL_run cannot be called by an HWI or SWI function.

2.12 LCK Module

The LCK module is the resource lock manager.

Functions

- LCK_create. Create a resource lock
- LCK_delete. Delete a resource lock
- LCK_pend. Acquire ownership of a resource lock
- LCK_post. Relinquish ownership of a resource lock

Constants, Types, and Structures

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

/* lock object */
typedef struct LCK_Attrs LCK_Attrs;

struct LCK_Attrs {
    Int dummy;
};

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the LCK Manager Properties and LCK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameter.

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Description

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

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

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

LCK Manager Properties

The following global property can be set for the LCK module on the LCK Manager Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: OBJMEMSEG Type: Reference

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

LCK Object Properties

To create a LCK object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myLck = bios.LCK.create("myLck");
```

The following property can be set for a LCK object in the LCK Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myLck.comment = "LCK object";`

LCK_create *Create a resource lock*

C Interface

Syntax

```
lock = LCK_create(attrs);
```

Parameters

```
LCK_Attrs      attrs;      /* pointer to lock attributes */
```

Return Value

```
LCK_Handle     lock;       /* handle for new lock object */
```

Description

LCK_create creates a new lock object and returns its handle. The lock has no current owner and its corresponding resource is available for acquisition through LCK_pend.

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

Note: At present, no attributes are supported for lock objects.

All default attribute values are contained in the constant LCK_ATTRS, which can be assigned to a variable of type LCK_Attrs prior to calling LCK_create.

LCK_create calls MEM_alloc to dynamically create the object's data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module, page 2–192.

Constraints and Calling Context

- LCK_create cannot be called from a SWI or HWI.
- You can reduce the size of your application program by creating objects with Tconf rather than using the XXX_create functions.

See Also

LCK_delete
LCK_pend
LCK_post

LCK_delete *Delete a resource lock*

C Interface

Syntax

```
LCK_delete(lock);
```

Parameters

```
LCK_Handle        lock;        /* lock handle */
```

Return Value

Void

Description

LCK_delete uses MEM_free to free the lock referenced by lock.

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

Constraints and Calling Context

- LCK_delete cannot be called from a SWI or HWI.
- No task should be awaiting ownership of the lock.
- No check is performed to prevent LCK_delete from being used on a statically-created object. If a program attempts to delete a lock object that was created using Tconf, SYS_error is called.

See Also

LCK_create
LCK_pend
LCK_post

LCK_pend

Acquire ownership of a resource lock

C Interface

Syntax

```
status = LCK_pend(lock, timeout);
```

Parameters

```
LCK_Handle        lock;        /* lock handle */
Uns               timeout;     /* return after this many system clock ticks */
```

Return Value

```
Bool              status;      /* TRUE if successful, FALSE if timeout */
```

Description

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

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

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

LCK_pend returns TRUE if it successfully acquires ownership of lock, returns FALSE if a timeout occurs before it can acquire ownership. LCK_pend returns FALSE if it is called from the context of a SWI or HWI, even if the timeout is zero.

Note: **RTS functions callable from TSK threads only.** Many run-time support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK_pend and LCK_post. As a result, RTS functions that call LCK_pend or LCK_post *must not be called in the context of a SWI or HWI*.

To determine whether a particular RTS function uses LCK_pend or LCK_post, refer to the source code for that function shipped with Code Composer Studio. The following table lists some RTS functions that call LCK_pend and LCK_post in certain versions of Code Composer Studio:

fprintf	printf	vfprintf	sprintf
vprintf	vsprintf	clock	strftime
minit	malloc	realloc	free
calloc	rand	srand	getenv

The C++ new operator calls malloc, which in turn calls LCK_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

Constraints and Calling Context

- The lock must be a handle for a resource lock object created through a prior call to LCK_create.
- LCK_pend should not be called from a SWI or HWI thread.
- LCK_pend should not be called from main().

See Also

LCK_create
LCK_delete
LCK_post

LCK_post *Relinquish ownership of a resource LCK*

C Interface

Syntax

LCK_post(lock);

Parameters

LCK_Handle	lock;	/* lock handle */

Return Value

Void

Description

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

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

Constraints and Calling Context

- lock must be a handle for a resource lock object created through a prior call to LCK_create.
- LCK_post should not be called from a SWI or HWI thread.
- LCK_post should not be called from main().

See Also

LCK_create
LCK_delete
LCK_pend

2.13 LOG Module

The LOG module captures events in real time.

Functions

- LOG_disable. Disable the system log.
- LOG_enable. Enable the system log.
- LOG_error. Write a user error event to the system log.
- LOG_event. Append unformatted message to message log.
- LOG_message. Write a user message event to the system log.
- LOG_printf. Append formatted message to message log.
- LOG_reset. Reset the system log.

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the LOG Manager Properties and LOG Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
bufSeg	Reference	prog.get("L0SARAM")
bufLen	EnumInt	64 (0, 8, 16, 32, 64, ..., 32768)
logType	EnumString	"circular" ("fixed)
dataType	EnumString	"printf" ("raw data")
format	String	"0x%x, 0x%x, 0x%x"

Description

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

The system log stores messages about system events for the types of log tracing you have enabled. See the TRC Module, page 2–392, for a list of events that can be traced in the system log.

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

LOG_error writes a user error event to the system log. This operation is not affected by any TRC trace bits; an error event is always written to the system log. LOG_message writes a user message event to the system log, provided that both TRC_GBLHOST and TRC_GBLTARG (the host and target trace bits, respectively) traces are enabled.

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

Each log event buffer uses eight words in the 'C28x large memory model.

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

LOG Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

LOG Object Properties

To create a LOG object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myLog = bios.LOG.create("myLog");
```

The following properties can be set for a log object on the LOG Object Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myLog.comment = "trace LOG";`

- **bufseg**. The name of a memory segment to contain the log buffer.

  Tconf Name: bufSeg Type: Reference

  Example: `myLog.bufSeg = prog.get("myMEM");`

- **buflen**. The length of the log buffer (in words).

  Tconf Name: bufLen Type: EnumInt

  Options: 0, 8, 16, 32, 64, ..., 32768

  Example: `myLog.bufLen = 64;`

- **logtype**. The type of the log: circular or fixed. Events added to a full circular log overwrite the oldest event in the buffer, whereas events added to a full fixed log are dropped.

  — **Fixed**. The log stores the first messages it receives and stops accepting messages when its message buffer is full.

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

  Tconf Name: logType Type: EnumString

  Options: "circular", "fixed"

  Example: `myLog.logType = "circular";`

- **datatype**. Choose printf if you use LOG_printf to write to this log and provide a format string.

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

Tconf Name:	dataType	Type: EnumString
Options:	"printf", "raw data"	
Example:	`myLog.dataType = "printf";`	

- **format**. If you choose raw data as the datatype, type a printf-style format string for this property. Provide up to three (3) conversion characters (such as %d) to format words two, three, and four in all records in the log. Do not put quotes around the format string. The format string can use %d, %u, %x, %o, %s, %r, and %p conversion characters; it cannot use other types of conversion characters. See LOG_printf, page 2–179, and LOG_event, page 2–177, for information about the structure of a log record.

Tconf Name:	format	Type: String
Example:	`myLog.format = "0x%x, 0x%x, 0x%x";`	

LOG_disable *Disable a message log*

C Interface

Syntax

LOG_disable(log);

Parameters

LOG_Handle	log;	/* log object handle */

Return Value

Void

Reentrant

no

Description

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

Example

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

See Also

LOG_enable
LOG_reset

LOG_enable *Enable a message log*

C Interface

Syntax

LOG_enable(log);

Parameters

LOG_Handle	log;	/* log object handle */

Return Value

Void

Reentrant

no

Description

LOG_enable enables the logging mechanism and allows the log buffer to be modified.

Example

```
LOG_enable(&trace);
```

See Also

LOG_disable
LOG_reset

LOG_error *Write an error message to the system log*

C Interface

Syntax

LOG_error(format, arg0);

Parameters

String	format;	/* printf-style format string */
Arg	arg0;	/* copied to second word of log record */

Return Value

Void

Reentrant

yes

Description

LOG_error writes a program-supplied error message to the system log, which is defined in the default configuration by the LOG_system object. LOG_error is not affected by any TRC bits; an error event is always written to the system log.

The format argument can contain any of the conversion characters supported for LOG_printf. See LOG_printf for details.

Example

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

See Also

LOG_event
LOG_message
LOG_printf
TRC_disable
TRC_enable

LOG_event *Append an unformatted message to a message log*

C Interface

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

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 the four words written by LOG_event are shown here:

LOG_event	Sequence #	arg0	arg1	arg2

You can format the log by using LOG_printf instead of LOG_event.

If you want the Event Log to apply the same printf-style format string to all records in the log, use Tconf to choose raw data for the datatype property and type a format string for the format property (see “LOG Object Properties” on page 172).

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

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

Example

```
LOG_event(&trace, (Arg)value1, (Arg)value2,
          (Arg)CLK gethtime());
```

See Also

LOG_error
LOG_printf
TRC_disable
TRC_enable

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

C Interface

Syntax

LOG_message(format, arg0);

Parameters

String	format;	/* printf-style format string */
Arg	arg0;	/* copied to second word of log record */

Return Value

Void

Reentrant

yes

Description

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

The format argument passed to LOG_message can contain any of the conversion characters supported for LOG_printf. See LOG_printf, page 2–179, for details.

Example

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

See Also

LOG_error
LOG_event
LOG_printf
TRC_disable
TRC_enable

LOG_printf *Append a formatted message to a message log*

C Interface

Syntax

```
LOG_printf(log, format);
   or
LOG_printf(log, format,arg0);
   or
LOG_printf(log, format, arg0, arg1);
```

Parameters

```
LOG_Handle    log;       /* log object handle */
String        format;    /* printf format string */
Arg           arg0;      /* value for first format string token */
Arg           arg1;      /* value for second format string token */
```

Return Value

Void

Reentrant

yes

Description

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 or %p conversion character is used).

The format string can use any conversion character found in Table Table 2-4.

Table 2-4: Conversion Characters for LOG_printf

Conversion Character	Description
%d	Signed integer
%u	Unsigned integer
%x	Unsigned hexadecimal integer
%o	Unsigned octal integer

Conversion Character	Description

%s	Character string
	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:
	`char *msg = "Hello world!";`
	`LOG_printf(&trace, "%s", msg);`
	However, the following example is not supported:
	`char msg[100];`
	`strcpy(msg, "Hello world!");`
	`LOG_printf(&trace, "%s", msg);`
	If the string appears in the COFF file and a pointer to the string is passed to LOG_printf, then the string in the COFF file is used by the Event Log to generate the output.
	If the string can not be found in the COFF file, the format string is replaced with *** ERROR: 0x%x 0x%x ***\n, which displays all arguments in hexadecimal.

%r	Symbol from symbol table
	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:
	`Int testval = 17;`
	`LOG_printf("%r = %d", &testval, testval);`
	displays:
	`testval = 17`
	If no symbol is found for the value passed to %r, the Event Log uses the string <unknown symbol>.

%p	data pointer

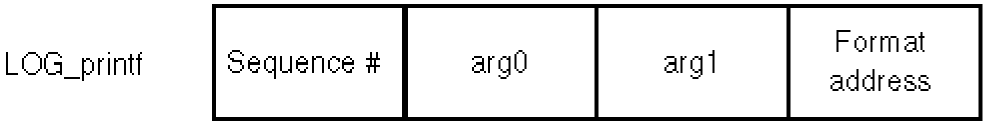
Since LOG_printf does not provide a conversion character for long integers, you may want to use 0x%p instead. Another solution is to use bitwise shifting and ANDing to break a 32-bit number into its 16-bit counterparts. In following example, `(Int)(maincount >> 16)` is the upper 16 bits of maincount shifted into the 16-bits of an Int. And, `(Int)(maincount & 0xffff)` is the lower 16 bits of maincount.

```
LOG_printf(&trace, "total count = 0x%04x%04x",
    (Int)(maincount >> 16),
    (Int)(maincount & 0xffff));
```

The 0x%04x%04x format string used in this example causes a literal string of "0x" to precede the value to indicate that it is a hex value. Then, each %04x tells LOG_printf to display the value as hex, padding to 4 characters with leading zeros.

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

Each log message uses eight words. The contents of the message written by LOG_printf are shown here:



LOG_printf	Sequence #	arg0	arg1	Format address

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

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

Constraints and Calling Context

- LOG_printf supports only 0, 1, or 2 arguments after the format string.

Example

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

See Also

LOG_error
LOG_event
TRC_disable
TRC_enable

LOG_reset *Reset a message log*

C Interface

Syntax

LOG_reset(log);

Parameters

LOG_Handle	log	/* log object handle */

Return Value

Void

Reentrant

no

Description

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

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

Example

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

See Also

LOG_disable
LOG_enable

2.14 MBX Module

The MBX module is the mailbox manager.

Functions

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

Constants, Types, and Structures

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

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

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the MBX Manager Properties and MBX Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
messageSize	Int16	1
length	Int16	1
elementSeg	Reference	prog.get("L0SARAM")

Description

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

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

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

MBX Manager Properties

The following global property can be set for the MBX module on the MBX Manager Properties dialog in the DSP/BIOS Configuration Tool or in a Tconf script:

- **Object Memory**. The memory segment that contains the MBX objects created with Tconf.

  Tconf Name: OBJMEMSEG Type: Reference

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

MBX Object Properties

To create an MBX object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myMbx = bios.MBX.create("myMbx");
```

The following properties can be set for an MBX object in the MBX Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myMbx.comment = "my MBX";`

- **Message Size**. The size (in MADUs) of the messages this mailbox can contain.

  Tconf Name: messageSize Type: Int16

  Example: `myMbx.messageSize = 1;`

- **Mailbox Length**. The number of messages this mailbox can contain.

  Tconf Name: length Type: Int16

  Example: `myMbx.length = 1;`

- **Element memory segment**. The memory segment to contain the mailbox data buffers.

  Tconf Name: elementSeg Type: Reference

  Example: `myMbx.elementSeg = prog.get("myMEM");`

MBX_create *Create a mailbox*

C Interface

Syntax

mbx = MBX_create(msgsize, mbxlength, attrs);

Parameters

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

Return Value

MBX_Handle	mbx;	/* mailbox object handle */

Description

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

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

All default attribute values are contained in the constant MBX_ATTRS, which can be assigned to a variable of type MBX_Attrs prior to calling MBX_create.

MBX_create calls MEM_alloc to dynamically create the object's data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module, page 2–192.

Constraints and Calling Context

- MBX_create cannot be called from a SWI or HWI.
- You can reduce the size of your application program by creating objects with Tconf rather than using the XXX_create functions.

See Also

MBX_delete
SYS_error

MBX_delete *Delete a mailbox*

C Interface

Syntax

```
MBX_delete(mbx);
```

Parameters

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

Return Value

Void

Description

MBX_delete frees the mailbox object referenced by mbx.

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

Constraints and Calling Context

- No tasks should be pending on mbx when MBX_delete is called.
- MBX_delete cannot be called from a SWI or HWI.
- No check is performed to prevent MBX_delete from being used on a statically-created object. If a program attempts to delete a mailbox object that was created using Tconf, SYS_error is called.

See Also

MBX_create

MBX_pend

Wait for a message from mailbox

C Interface

Syntax

```
status = MBX_pend(mbx, msg, timeout);
```

Parameters

MBX_Handle	mbx;	/* mailbox object handle */
Ptr	msg;	/* message pointer */
Uns	timeout;	/* return after this many system clock ticks */

Return Value

Bool	status;	/* TRUE if successful, FALSE if timeout */

Description

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

If timeout is SYS_FOREVER, the task remains suspended until MBX_post is called on this mailbox. If timeout is 0, MBX_pend returns immediately.

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

A task switch occurs when calling MBX_pend if the mailbox is empty and timeout is not 0, or if a higher priority task is blocked on MBX_post.

Constraints and Calling Context

- This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- If you need to call MBX_pend within a TSK_disable/TSK_enable block, you must use a timeout of 0.
- MBX_pend cannot be called from the program's main() function.

See Also

MBX_post

MBX_post *Post a message to mailbox*

C Interface

Syntax

```
status = MBX_post(mbx, msg, timeout);
```

Parameters

MBX_Handle	mbx;	/* mailbox object handle */
Ptr	msg;	/* message pointer */
Uns	timeout;	/* return after this many system clock ticks */

Return Value

Bool	status;	/* TRUE if successful, FALSE if timeout */

Description

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

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

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

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

Constraints and Calling Context

- If you need to call MBX_post within a TSK_disable/TSK_enable block, you must use a timeout of 0.
- This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- MBX_post can be called from the program's main() function. However, the number of calls should not be greater than the number of messages the mailbox can hold. Additional calls have no effect.

See Also

MBX_pend

2.15 MEM Module

The MEM module is the memory segment manager.

Functions

- MEM_alloc. Allocate from a memory segment.
- MEM_calloc. Allocate and initialize to 0.
- MEM_define. Define a new memory segment.
- MEM_free. Free a block of memory.
- MEM_getBaseAddress. Get base address of memory heap.
- MEM_increaseTableSize. Increase the internal MEM table size.
- MEM_redefine. Redefine an existing memory segment.
- MEM_stat. Return the status of a memory segment.
- MEM_undefine. Undefine an existing memory segment.
- MEM_valloc. Allocate and initialize to a value.

Constants, Types, and Structures

```
MEM->MALLOCSEG = 0;    /* segid for malloc, free */

#define MEM_HEADERSIZE /* free block header size */
#define MEM_HEADERMASK /* mask to align on
                          MEM_HEADERSIZE */
#define MEM_ILLEGAL    /* illegal memory address */

MEM_Attrs MEM_ATTRS ={ /* default attribute values */
    0
};
typedef struct MEM_Segment {
    Ptr       base;     /* base of the segment */
    MEM_sizep length;   /* size of the segment */
    Uns       space;    /* memory space */
} MEM_Segment;

typedef struct MEM_Stat {
  MEM_sizep   size;   /* original size of segment */
  MEM_sizep   used;   /* MADUs used in segment */
  size_t      length; /* largest contiguous block */
} MEM_Stat;


typedef unsigned long MEM_sizep;
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the MEM Manager Properties and MEM Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters.

Name	Type	Default
REUSECODESPACE	Bool	false
ARGSSIZE	Numeric	0x0004
STACKSIZE	Numeric	0x0100

Name	Type	Default
NOMEMORYHEAPS	Bool	false
BIOSOBJSEG	Reference	prog.get("MEM_NULL")
MALLOCSEG	Reference	prog.get("MEM_NULL")
ARGSSEG	Reference	prog.get("L0SARAM")
STACKSEG	Reference	prog.get("M1SARAM")
GBLINITSEG	Reference	prog.get("H0SARAM")
TRCDATASEG	Reference	prog.get("H0SARAM")
SYSDATASEG	Reference	prog.get("L0SARAM")
OBJSEG	Reference	prog.get("L0SARAM")
BIOSSEG	Reference	prog.get("H0SARAM")
SYSINITSEG	Reference	prog.get("H0SARAM")
HWISEG	Reference	prog.get("L0SARAM")
HWIVECSEG	Reference	prog.get("VECT")
RTDXTEXTSEG	Reference	prog.get("H0SARAM")
USERCOMMANDFILE	Bool	false
TEXTSEG	Reference	prog.get("H0SARAM")
SWITCHSEG	Reference	prog.get("H0SARAM")
BSSSEG	Reference	prog.get("L0SARAM")
CINITSEG	Reference	prog.get("H0SARAM")
PINITSEG	Reference	prog.get("H0SARAM")
CONSTSEG	Reference	prog.get("L0SARAM")
DATASEG	Reference	prog.get("L0SARAM")
CIOSEG	Reference	prog.get("L0SARAM")
ENABLELOADADDR	Bool	false
LOADBIOSSEG	Reference	prog.get("H0SARAM")
LOADSYSINITSEG	Reference	prog.get("H0SARAM")
LOADGBLINITSEG	Reference	prog.get("H0SARAM")
LOADTRCDATASEG	Reference	prog.get("H0SARAM")
LOADTEXTSEG	Reference	prog.get("H0SARAM")
LOADSWITCHSEG	Reference	prog.get("H0SARAM")
LOADCINITSEG	Reference	prog.get("H0SARAM")
LOADPINITSEG	Reference	prog.get("H0SARAM")
LOADCONSTSEG	Reference	prog.get("H0SARAM")
LOADHWISEG	Reference	prog.get("L0SARAM")
LOADHWIVECSEG	Reference	prog.get("VECT")
LOADRTDXTEXTSEG	Reference	prog.get("H0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
base	Numeric	0x000000
len	Numeric	0x000000
createHeap	Bool	true
heapSize	Numeric	0x000ff
enableHeapLabel	Bool	false
heapLabel	Extern	prog.extern("segment_name","asm")
space	EnumString	"data" ()

Description

The MEM module provides a set of functions used to allocate storage from one or more disjointed segments of memory. These memory segments are specified with Tconf.

MEM always allocates an even number of MADUs and always aligns buffers on an even boundary. This behavior is used to insure that free buffers are always at least two MADUs in length. This behavior does not preclude you from allocating two 512 buffers from a 1K region of on-device memory, for example. It does, however, mean that odd allocations consume one more MADU than expected.

If small code size is important to your application, you can reduce code size significantly by removing the capability to dynamically allocate and free memory. To do this, set the "No Dynamic Memory Heaps" property for the MEM manager to true. If you remove this capability, your program cannot call any of the MEM functions or any object creation functions (such as TSK_create). You need to create all objects to be used by your program statically (with Tconf). You can also create or remove the dynamic memory heap from an individual memory segment in the configuration.

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

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 segments 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 in the MEM Manager Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

General tab

- **Reuse Startup Code Space.** If this property is set to true, the startup code section (.sysinit) can be reused after startup is complete.

  Tconf Name: REUSECODESPACE Type: Bool

  Example: `bios.MEM.REUSECODESPACE = false;`

- **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.

  Tconf Name: ARGSSIZE Type: Numeric

  Example: `bios.MEM.ARGSSIZE = 0x0004;`

- **Stack Size**. The size of the global stack in MADUs. The upper-left corner of the DSP/BIOS Configuration Tool window shows the estimated minimum global stack size required for this application (as a decimal number).

  This size is shown as a hex value in Minimum Addressable Data Units (MADUs). An MADU is the smallest unit of data storage that can be read or written by the CPU. For the c28x this is a 16-bit word.

  Tconf Name: STACKSIZE Type: Numeric

  Example: `bios.MEM.STACKSIZE = 0x0400;`

- **System Stack Size (MADUs)**. The size of the system stack in MADUs, applicable only on the ’C28x device.

- **No Dynamic Memory Heaps**. Put a checkmark in this box to completely disable the ability to dynamically allocate memory and the ability to dynamically create and delete objects. If this property is set to true, the program may not call the MEM_alloc, MEM_valloc, MEM_calloc, and malloc or the XXX_create function for any DSP/BIOS module. If this property is set to true, the Segment For DSP/BIOS Objects, Segment for malloc()/free(), and Stack segment for dynamic tasks properties are set to MEM_NULL.

  When you set this property to true, heaps already specified in MEM segments are removed from the configuration. If you later reset this property to false, recreate heaps by configuring properties for individual MEM objects as needed.

  Tconf Name: NOMEMORYHEAPS Type: Bool

  Example: `bios.MEM.NOMEMORYHEAPS = false;`

- **Segment For DSP/BIOS Objects**. The default memory segment to contain objects created at run-time with an XXX_create function. The XXX_Attrs structure passed to the XXX_create function can override this default. If you select MEM_NULL for this property, creation of DSP/BIOS objects at run-time via the XXX_create functions is disabled.

  Tconf Name: BIOSOBJSEG Type: Reference

  Example: `bios.MEM.BIOSOBJSEG = prog.get("myMEM");`

- **Segment For malloc() / free()**. The memory segment from which space is allocated when a program calls malloc and from which space is freed when a program calls free. If you select MEM_NULL for this property, dynamic memory allocation at run-time is disabled.

  Tconf Name: MALLOCSEG Type: Reference

  Example: `bios.MEM.MALLOCSEG = prog.get("myMEM");`

BIOS Data tab

- **Argument Buffer Section (.args)**. The memory segment containing the .args section.

  Tconf Name: ARGSSEG Type: Reference

  Example: `bios.MEM.ARGSSEG = prog.get("myMEM");`

- **Stack Section (.stack)**. The memory segment containing the data stack. This segment should be located in RAM.

  Tconf Name: STACKSEG Type: Reference

  Example: `bios.MEM.STACKSEG = prog.get("myMEM");`

- **System Stack Section (.sysstack).** The memory segment containing the system stack, applicable only on the ’C28x device.
- **DSP/BIOS Init Tables (.gblinit).** The memory segment containing the DSP/BIOS global initialization tables.

  Tconf Name: GBLINITSEG Type: Reference

  Example: `bios.MEM.GBLINITSEG = prog.get("myMEM");`
- **TRC Initial Value (.trcdata)**. The memory segment containing the TRC mask variable and its initial value. This segment must be placed in RAM.

  Tconf Name: TRCDATASEG Type: Reference

  Example: `bios.MEM.TRCDATASEG = prog.get("myMEM");`
- **DSP/BIOS Kernel State (.sysdata).** The memory segment containing system data about the DSP/BIOS kernel state.

  Tconf Name: SYSDATASEG Type: Reference

  Example: `bios.MEM.SYSDATASEG = prog.get("myMEM");`
- **DSP/BIOS Conf Sections (.obj).** The memory segment containing configuration properties that can be read by the target program.

  Tconf Name: OBJSEG Type: Reference

  Example: `bios.MEM.OBJSEG = prog.get("myMEM");`

BIOS Code tab

- **BIOS Code Section (.bios)**. The memory segment containing the DSP/BIOS code.

  Tconf Name: BIOSSEG Type: Reference

  Example: `bios.MEM.BIOSSEG = prog.get("myMEM");`
- **Startup Code Section (.sysinit).** The memory segment containing DSP/BIOS startup initialization code; this memory can be reused after main starts executing.

  Tconf Name: SYSINITSEG Type: Reference

  Example: `bios.MEM.SYSINITSEG = prog.get("myMEM");`
- **Function Stub Memory (.hwi).** The memory segment containing dispatch code for HWIs that are configured to be monitored in the HWI Object Properties.

  Tconf Name: HWISEG Type: Reference

  Example: `bios.MEM.HWISEG = prog.get("myMEM");`
- **Interrupt Service Table Memory (.hwi_vec).** The memory segment containing the Interrupt Service Table (IST). This property is read-only.
- **RTDX Text Segment (.rtdx_text).** The memory segment containing the code sections for the RTDX module.

  Tconf Name: RTDXTEXTSEG Type: Reference

  Example: `bios.MEM.RTDXTEXTSEG = prog.get("myMEM");`

Compiler Sections tab

- **User .cmd File For Compiler Sections**. Put a checkmark in this box if you want to have full control over the memory used for the sections that follow. You must then create a linker command file that begins by including the linker command file created by the configuration. Your linker command file should then assign memory for the items normally handled by the following properties. See the

TMS320C28x Optimizing Compiler User's Guide, (literature number SPRU514) for more details.

Tconf Name: USERCOMMANDFILE Type: Bool

Example: `bios.MEM.USERCOMMANDFILE = false;`

- **Text Section (.text)**. The memory segment containing the executable code, string literals, and compiler-generated constants. This segment can be located in ROM or RAM.

  Tconf Name: TEXTSEG Type: Reference

  Example: `bios.MEM.TEXTSEG = prog.get("myMEM");`

- **Switch Jump Tables (.switch)**. The memory segment containing the jump tables for switch statements. This segment can be located in ROM or RAM.

  Tconf Name: SWITCHSEG Type: Reference

  Example: `bios.MEM.SWITCHSEG = prog.get("myMEM");`

- **C Variables Section (.bss)**. The memory segment containing global and static C variables. At boot or load time, the data in the .cinit section is copied to this segment. This segment should be located in RAM.

  Tconf Name: BSSSEG Type: Reference

  Example: `bios.MEM.BSSSEG = prog.get("myMEM");`

- **Data Initialization Section (.cinit)**. The memory segment containing tables for explicitly initialized global and static variables and constants. This segment can be located in ROM or RAM.

  Tconf Name: CINITSEG Type: Reference

  Example: `bios.MEM.CINITSEG = prog.get("myMEM");`

- **C Function Initialization Table (.pinit)**. The memory segment containing the table of global object constructors. Global constructors must be called during program initialization. The C/C++ compiler produces a table of constructors to be called at startup. The table is contained in a named section called .pinit. The constructors are invoked in the order that they occur in the table. This segment can be located in ROM or RAM.

  Tconf Name: PINITSEG Type: Reference

  Example: `bios.MEM.PINITSEG = prog.get("myMEM");`

- **Constant Sections (.const, .printf)**. These sections can be located in ROM or RAM. The .const section contains string constants and data defined with the const C qualifier. The DSP/BIOS .printf section contains other constant strings used by the Real-Time Analysis tools. The .printf section is not loaded onto the target. Instead, the (COPY) directive is used for this section in the .cmd file. The .printf section is managed along with the .const section, since it must be grouped with the .const section to make sure that no addresses overlap. If you specify these sections in your own .cmd file, you'll need to do something like the following:

```
GROUP {
   .const: {}
   .printf (COPY): {}
} > IRAM
```

  Tconf Name: CONSTSEG Type: Reference

  Example: `bios.MEM.CONSTSEG = prog.get("myMEM");`

- **Data Section (.data)**. This memory segment contains program data. This segment can be located in ROM or RAM.

  Tconf Name: DATASEG Type: Reference

  Example: `bios.MEM.DATASEG = prog.get("myMEM");`

- **Data Section (.cio)**. This memory segment contains C standard I/O buffers.

  Tconf Name: CIOSEG Type: Reference

  Example: `bios.MEM.CIOSEG = prog.get("myMEM");`

Load Address tab

- **Specify Separate Load Addresses**. If you put a checkmark in this box, you can select separate load addresses for the sections listed on this tab.

  Load addresses are useful when, for example, your code must be loaded into ROM, but would run faster in RAM. The linker allows you to allocate sections twice: once to set a load address and again to set a run address.

  If you do not select a separate load address for a section, the section loads and runs at the same address.

  If you do select a separate load address, the section is allocated as if it were two separate sections of the same size. The load address is where raw data for the section is placed. References to items in the section refer to the run address. The application must copy the section from its load address to its run address. For details, see the topics on Runtime Relocation and the .label Directive in the Code Generation Tools help or manual.

  Tconf Name: ENABLELOADADDR Type: Bool

  Example: `bios.MEM.ENABLELOADADDR = false;`

- **Load Address - BIOS Code Section (.bios)**. The memory segment containing the load allocation of the section that contains DSP/BIOS code.

  Tconf Name: LOADBIOSSEG Type: Reference

  Example: `bios.MEM.LOADBIOSSEG = prog.get("myMEM");`

- **Load Address - Startup Code Section (.sysinit)**. The memory segment containing the load allocation of the section that contains DSP/BIOS startup initialization code.

  Tconf Name: LOADSYSINITSEG Type: Reference

  Example: `bios.MEM.LOADSYSINITSEG = prog.get("myMEM");`

- **Load Address - DSP/BIOS Init Tables (.gblinit)**. The memory segment containing the load allocation of the section that contains the DSP/BIOS global initialization tables.

  Tconf Name: LOADGBLINITSEG Type: Reference

  Example: `bios.MEM.LOADGBLINITSEG = prog.get("myMEM");`

- **Load Address - TRC Initial Value (.trcdata)**. The memory segment containing the load allocation of the section that contains the TRC mask variable and its initial value.

  Tconf Name: LOADTRCDATASEG Type: Reference

  Example: `bios.MEM.LOADTRCDATASEG = prog.get("myMEM");`

- **Load Address - Text Section (.text)**. The memory segment containing the load allocation of the section that contains the executable code, string literals, and compiler-generated constants.

  Tconf Name: LOADTEXTSEG Type: Reference

  Example: `bios.MEM.LOADTEXTSEG = prog.get("myMEM");`

- **Load Address - Switch Jump Tables (.switch)**. The memory segment containing the load allocation of the section that contains the jump tables for switch statements.

  Tconf Name: LOADSWITCHSEG Type: Reference

  Example: `bios.MEM.LOADSWITCHSEG = prog.get("myMEM");`

- **Load Address - Data Initialization Section (.cinit)**. The memory segment containing the load allocation of the section that contains tables for explicitly initialized global and static variables and constants.

  Tconf Name: LOADCINITSEG Type: Reference

  Example: `bios.MEM.LOADCINITSEG = prog.get("myMEM");`

- **Load Address - C Function Initialization Table (.pinit)**. The memory segment containing the load allocation of the section that contains the table of global object constructors.

  Tconf Name: LOADPINITSEG Type: Reference

  Example: `bios.MEM.LOADPINITSEG = prog.get("myMEM");`

- **Load Address - Constant Sections (.const, .printf)**. The memory segment containing the load allocation of the sections that contain string constants, data defined with the const C qualifier, and other constant strings used by the Real-Time Analysis tools. The .printf section is managed along with the .const section to make sure that no addresses overlap.

  Tconf Name: LOADCONSTSEG Type: Reference

  Example: `bios.MEM.LOADCONSTSEG = prog.get("myMEM");`

- **Load Address - Function Stub Memory (.hwi)**. The memory segment containing the load allocation of the section that contains dispatch code for HWIs configured to be monitored.

  Tconf Name: LOADHWISEG Type: Reference

  Example: `bios.MEM.LOADHWISEG = prog.get("myMEM");`

- **Load Address - Interrupt Service Table Memory (.hwi_vec)**. The memory segment containing the load allocation of the section that contains the Interrupt Service Table (IST).

  Tconf Name: LOADHWIVECSEG Type: Reference

  Example: `bios.MEM.LOADHWIVECSEG = prog.get("myMEM");`

- **Load Address - RTDX Text Segment (.rtdx_text)**. The memory segment containing the load allocation of the section that contains the code sections for the RTDX module.

  Tconf Name: LOADRTDXTEXTSEG Type: Reference

  Example: `bios.MEM.LOADRTDXTEXTSEG = prog.get("myMEM");`

MEM Object Properties

A memory segment represents a contiguous length of code or data memory in the address space of the processor.

To create a MEM object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myMem = bios.MEM.create("myMem");
```

The following properties can be set for a MEM object in the MEM Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myMem.comment = "my MEM";`

- **base**. The address at which this memory segment begins. This value is shown in hex.

  Tconf Name: base Type: Numeric

  Example: `myMem.base = 0x000000;`

- **len**. The length of this memory segment in MADUs. This value is shown in hex.

  Tconf Name: len Type: Numeric

  Example: `myMem.len = 0x000000;`

- **create a heap in this memory.** If this property is set to true, a heap is created in this memory segment. Memory can by allocated dynamically from a heap. In order to remove the heap from a memory segment, you can select another memory segment that contains a heap for properties that dynamically allocate memory in this memory segment. The properties you should check are in the Memory Section Manager (the Segment for DSP/BIOS objects and Segment for malloc/free properties) and the Task Manager (the Default stack segment for dynamic tasks property). If you disable dynamic memory allocation in the Memory Section Manager, you cannot create a heap in any memory segment.

  Tconf Name: createHeap Type: Bool

  Example: `myMem.createHeap = true;`

- **heap size**. The size of the heap in MADUs to be created in this memory segment. You cannot control the location of the heap within its memory segment except by making the segment and heap the same sizes. Note that if the base of the heap ends up at address 0x0, the base address of the heap is offset by MEM_HEADERSIZE and the heap size is reduced by MEM_HEADERSIZE.

  Tconf Name: heapSize Type: Numeric

  Example: `myMem.heapSize = 0x000ff;`

- **enter a user defined heap identifier**. If this property is set to true, you can define your own identifier label for this heap.

  Tconf Name: enableHeapLabel Type: Bool

  Example: `myMem.enableHeapLabel = false;`

- **heap identifier label**. If the property above is set to true, type a name for this segment's heap.

  Tconf Name: heapLabel Type: Extern

  Example: `myMem.heapLabel = prog.extern("seg_name", "asm");`

- **space**. Type of memory segment. This is set to code for memory segments that store programs, and data for memory segments that store program data.

  Tconf Name: space Type: EnumString

  Options:

  Example: `myMem.space = "data";`

The predefined memory segments in a configuration file, particularly those for external memory, are dependent on the board template you select. In general, Table 2-5 lists segments that can be defined for the c2800:

Table 2-5: Typical Memory Segments for ’C28x Boards

Name	Memory Segment Type
BOOTROM	Boot code memory
FLASH	Internal flash program memory
VECT	Interrupt vector table when VMAP=0
VECT1	Interrupt vector table when VMAP=1
OTP	One-time programmable memory via flash registers
H0SARAM	Internal program RAM

Name	Memory Segment Type
L0SARAM	Internal data RAM
M1SARAM	Internal user and task stack RAM
PIEVECT	Space for PIE interrupt vectors

MEM_alloc *Allocate from a memory segment*

C Interface

Syntax

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

Parameters

```
Int           segid;      /* memory segment identifier */
size_t        size;       /* block size in MADUs */
size_t        align;      /* block alignment */
```

Return Value

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

Description

MEM_alloc allocates a contiguous block of storage from the memory segment identified by segid and returns the address of this block.

The segid parameter identifies the memory segment to allocate memory from. This identifier can be an integer or a memory segment name defined in the configuration. Files created by the configuration define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

MEM_alloc does not initialize the allocated memory locations.

If the memory request cannot be satisfied, MEM_alloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK_pend and LCK_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_alloc cannot be called from the context of a SWI or HWI. MEM_alloc checks the context from which it is called. It calls SYS_error and returns MEM_ILLEGAL if it is called from the wrong context.

A number of other DSP/BIOS APIs call MEM_alloc internally, and thus also cannot be called from the context of a SWI or HWI. See the “Function Callability Table” on page 434 for a detailed list of calling contexts for each DSP/BIOS API.

Constraints and Calling Context

- segid must identify a valid memory segment.
- MEM_alloc cannot be called from a SWI or HWI.
- MEM_alloc cannot be called if the TSK scheduler is disabled.
- align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

See Also

MEM_calloc
MEM_free
MEM_valloc

MEM_calloc *Allocate from a memory segment and set value to 0*

C Interface

Syntax

```
addr = MEM_calloc(segid, size, align)
```

Parameters

```
Int       segid;    /* memory segment identifier */
size_t    size;     /* block size in MADUs */
size_t    align;    /* block alignment */
```

Return Value

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

Description

MEM_calloc is functionally equivalent to calling MEM_valloc with value set to 0. MEM_calloc allocates a contiguous block of storage from the memory segment identified by segid and returns the address of this block.

The segid parameter identifies the memory segment from which memory is to be allocated. This identifier can be an integer or a memory segment name defined in the configuration. The files created by the configuration define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM_calloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK_pend and LCK_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_calloc cannot be called from the context of a SWI or HWI.

Constraints and Calling Context

- segid must identify a valid memory segment.
- MEM_calloc cannot be called from a SWI or HWI.
- MEM_calloc cannot be called if the TSK scheduler is disabled.
- align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

See Also

MEM_alloc
MEM_free
MEM_valloc
SYS_error
std.h and stdlib.h functions

MEM_define

Define a new memory segment

C Interface

Syntax

segid = MEM_define(base, length, attrs);

Parameters

Ptr	base;	/* base address of new segment */
MEM_sizep	length;	/* length (in MADUs) of new segment */
MEM_Attrs	*attrs;	/* segment attributes */

Return Value

Int	segid;	/* ID of new segment */

Reentrant

yes

Description

MEM_define defines a new memory segment for use by the DSP/BIOS MEM Module.

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

The new block should be aligned on a MEM_HEADERSIZE boundary, and the length should be a multiple of MEM_HEADERSIZE.

If attrs is NULL, the new segment is assigned a default set of attributes. Otherwise, the segment's attributes are specified through a structure of type MEM_Attrs.

Note: No attributes are supported for segments, and the type MEM_Attrs is defined as a dummy structure.

If there are undefined slots available in the internal table of memory segment identifiers, one of those slots is (re)used for the new segment. If there are no undefined slots available in the internal table, the table size is increased via MEM_alloc. See MEM_increaseTableSize to manage performance in this situation.

Constraints and Calling Context

- At least one segment must exist at the time MEM_define is called.
- MEM_define internally locks the memory by calling LCK_pend and LCK_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_define cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.
- The length parameter must be a multiple of MEM_HEADERSIZE and must be at least equal to MEM_HEADERSIZE.
- The base Ptr cannot be NULL.

See Also

MEM_redefine
MEM_undefine

MEM_free *Free a block of memory*

C Interface

Syntax

```
status = MEM_free(segid, addr, size);
```

Parameters

```
Int        segid;    /* memory segment identifier */
Ptr        addr;     /* block address pointer */
size_t     size;     /* block length in MADUs*/
```

Return Value

```
Bool       status;   /* TRUE if successful */
```

Description

MEM_free places the memory block specified by addr and size back into the free pool of the segment specified by segid. The newly freed block is combined with any adjacent free blocks. This space is then available for further allocation by MEM_alloc. The segid can be an integer or a memory segment name defined in the configuration.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK_pend and LCK_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_free cannot be called from the context of a SWI or HWI.

Constraints and Calling Context

- addr must be a valid pointer returned from a call to MEM_alloc.
- segid and size are those values used in a previous call to MEM_alloc.
- MEM_free cannot be called by HWI or SWI functions.
- MEM_free cannot be called if the TSK scheduler is disabled.

See Also

MEM_alloc
std.h and stdlib.h functions

MEM_getBaseAddress *Get base address of a memory heap*

C Interface

Syntax

```
addr = MEM_getBaseAddress(segid);
```

Parameters

```
Int            segid;       /* memory segment identifier */
```

Return Value

```
Ptr            addr;        /* heap base address pointer */
```

Description

MEM_getBaseAddress returns the base address of the memory heap with the segment ID specified by the segid parameter.

Constraints and Calling Context

- The segid can be an integer or a memory segment name defined in the configuration.

See Also

MEM Object Properties

MEM_increaseTableSize *Increase the internal MEM table size*

C Interface

Syntax

status = MEM_increaseTableSize(numEntries);

Parameters

Uns	numEntries;	/* number of segments to increase table by */

Return Value

Int	status;	/* TRUE if successful */

Reentrant

yes

Description

MEM_increaseTableSize allocates numEntries of undefined memory segments. When MEM_define is called, undefined memory segments are re-used. If no undefined memory segments exist, one is allocated. By using MEM_increaseTableSize, the application can avoid the use of MEM_alloc (thus improving performance and determinism) within the MEM_define call.

MEM_increaseTableSize internally locks memory by calling LCK_pend and LCK_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_increaseTableSize cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.

MEM_increaseTableSize returns SYS_OK to indicate success and SYS_EALLOC if an allocation error occurred.

Constraints and Calling Context

- Do not call from the context of a SWI or HWI.

See Also

MEM_define
MEM_undefine

MEM_redefine *Redefine an existing memory segment*

C Interface

Syntax

```
MEM_redefine(segid, base, length);
```

Parameters

```
Int            segid;     /* segment to redefine */
Ptr            base;      /* base address of new block */
MEM_sizep      length;    /* length (in MADUs) of new block */
```

Return Value

Void

Reentrant

yes

Description

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

The new block should be aligned on a MEM_HEADERSIZE boundary, and the length should be a multiple of MEM_HEADERSIZE.

Constraints and Calling Context

- MEM_redefine internally locks the memory by calling LCK_pend and LCK_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_redefine cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.
- The length parameter must be a multiple of MEM_HEADERSIZE and must be at least equal to MEM_HEADERSIZE.
- The base Ptr cannot be NULL.

See Also

MEM_define
MEM_undefine

MEM_stat

Return the status of a memory segment

C Interface

Syntax

status = MEM_stat(segid, statbuf);

Parameters

Int	segid;	/* memory segment identifier */
MEM_Stat	*statbuf;	/* pointer to stat buffer */

Return Value

Bool	status;	/* TRUE if successful */

Description

MEM_stat returns the status of the memory segment specified by segid in the status structure pointed to by statbuf.

```
typedef struct MEM_Stat {
  MEM_sizep   size;   /* original size of segment */
  MEM_sizep   used;   /* MADUs used in segment */
  size_t      length; /* largest contiguous block */
} MEM_Stat;
```

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

MEM_stat returns TRUE if segid corresponds to a valid memory segment, and FALSE otherwise. If MEM_stat returns FALSE, the contents of statbuf are undefined. If the segment has been undefined with MEM_undefine, this function returns FALSE.

MEM functions that access memory internally lock the memory by calling the LCK_pend and LCK_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_stat cannot be called from the context of a SWI or HWI.

Constraints and Calling Context

- MEM_stat cannot be called from a SWI or HWI.
- MEM_stat cannot be called if the TSK scheduler is disabled.

MEM_undefine *Undefine an existing memory segment*

C Interface

Syntax

```
MEM_undefine(segid);
```

Parameters

```
Int                    segid;          /* segment to undefine */
```

Return Value

Void

Reentrant

yes

Description

MEM_undefine removes a memory segment from the internal memory tables. Once a memory segment has been undefined, the segid cannot be used in any of the MEM APIs (except MEM_stat). Note: The undefined segid might later be returned by a subsequent MEM_define call.

MEM_undefine internally locks the memory by calling LCK_pend and LCK_post. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_undefine cannot be called from the context of a SWI or HWI. It can be called from main() or a TSK. The duration that the API holds the memory lock is variable.

Constraints and Calling Context

- Do not call from the context of a SWI or HWI.
- MEM_undefine does not free the actual memory buffer managed by the memory segment.

See Also

MEM_define
MEM_redefine

MEM_valloc

Allocate from a memory segment and set value

C Interface

Syntax

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

Parameters

```
Int        segid;    /* memory segment identifier */
size_t     size;     /* block size in MADUs */
size_t     align;    /* block alignment */
Char       value;    /* character value */
```

Return Value

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

Description

MEM_valloc uses MEM_alloc to allocate the memory before initializing it to value.

The segid parameter identifies the memory segment from which memory is to be allocated. This identifier can be an integer or a memory segment name defined in the configuration. The files created by the configuration define each configured segment name as a variable with an integer value.

The block contains size MADUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM_valloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

MEM functions that allocate and deallocate memory internally lock the memory by calling the LCK_pend and LCK_post functions. If another task already holds a lock to the memory, there is a context switch. For this reason, MEM_valloc cannot be called from the context of a SWI or HWI.

Constraints and Calling Context

- segid must identify a valid memory segment.
- MEM_valloc cannot be called from a SWI or HWI.
- MEM_valloc cannot be called if the TSK scheduler is disabled.
- align must be 0, or a power of 2 (for example, 1, 2, 4, 8).

See Also

MEM_alloc
MEM_calloc
MEM_free
SYS_error
std.h and stdlib.h functions

2.16 MSGQ Module

The MSGQ module allows for the structured sending and receiving of variable length messages. This module can be used for homogeneous or heterogeneous multi-processor messaging.

Functions

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

Constants, Types, and Structures

```
/* Attributes used to open message queue */
typedef struct MSGQ_Attrs {
    Ptr        notifyHandle;
    MSGQ_Pend  pend;
    MSGQ_Post  post;
} MSGQ_Attrs;

MSGQ_Attrs MSGQ_ATTRS = {
   NULL,                /* notifyHandle */
   (MSGQ_Pend)SYS_zero, /* NOP pend */
   FXN_F_nop            /* NOP post */
};

/* Attributes for message queue location */
typedef struct MSGQ_LocateAttrs {
    Uns         timeout;
} MSGQ_LocateAttrs;

MSGQ_LocateAttrs  MSGQ_LOCATEATTRS = {SYS_FOREVER};

/* Attrs for asynchronous message queue location */
typedef struct MSGQ_LocateAsyncAttrs {
    Uint16      poolId;
    Arg         arg;
} MSGQ_LocateAttrs;

MSGQ_LocateAsyncAttrs  MSGQ_LOCATEASYNCATTRS = {0, 0};


/* Configuration structure */
typedef struct MSGQ_Config {
  MSGQ_Obj          *msgqQueues;          /* Array of MSGQ handles */
  MSGQ_TransportObj *transports;          /* Transport array */
  Uint16             numMsgqQueues;       /* Number of MSGQ handles */
  Uint16             numProcessors;       /* Number of processors */
  Uint16             startUninitialized;  /* 1st MSGQ to init */
  MSGQ_Queue         errorQueue;          /* Receives transport err */
  Uint16             errorPoolId;         /* Alloc errors from poolId */
} MSGQ_Config;

/* Asynchronous locate message */
typedef struct MSGQ_AsyncLocateMsg {
    MSGQ_MsgHeader  header;
    MSGQ_Queue      msgqQueue;
    Arg             arg;
} MSGQ_AsyncLocateMsg;

/* Asynchronous error message */
typedef struct MSGQ_AsyncErrorMsg {
    MSGQ_MsgHeader  header;
    MSGQ_MqtError   errorType;
    Uint16          mqtId;
    Uint16          parameter;
} MSGQ_AsyncErrorMsg;

/* Transport object */
```

```
typedef struct MSGQ_TransportObj {
  MSGQ_MqtInit  initFxn;    /* Transport init func */
  MSGQ_TransportFxns *fxns; /* Interface funcs */
  Ptr         params; /* Setup parameters */
  Ptr         object; /* Transport-specific object */
  Uint16      procId; /* Processor Id talked to */
} MSGQ_TransportObj;
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the MSGQ Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
ENABLEMSGQ	Bool	false

Description

The MSGQ module allows for the structured sending and receiving of variable length messages. This module can be used for homogeneous or heterogeneous multi-processor messaging. The MSGQ module with a substantially similar API is implemented in DSP/BIOS Link for certain TI general-purpose processors (GPPs), particularly those used in OMAP devices.

MSGQ provides more sophisticated messaging than other modules. It is typically used for complex situations such as multi-processor messaging. The following are key features of the MSGQ module:

- Writers and readers can be relocated to another processor with no runtime code changes.
- Timeouts are allowed when receiving messages.
- Readers can determine the writer and reply back.
- Receiving a message is deterministic when the timeout is zero.
- Sending a message is non-blocking.
- Messages can reside on any message queue.
- Supports zero-copy transfers.
- Can send and receive from HWIs, SWIs and TSKs.
- Notification mechanism is specified by application.
- Allows QoS (quality of service) on message buffer pools. For example, using specific buffer pools for specific message queues.

Messages are sent and received via a *message queue*. A reader is a thread that gets (reads) messages from a message queue. A writer is a thread that puts (writes) a message to a message queue. Each message queue has one reader and can have many writers. A thread may read from or write to multiple message queues.

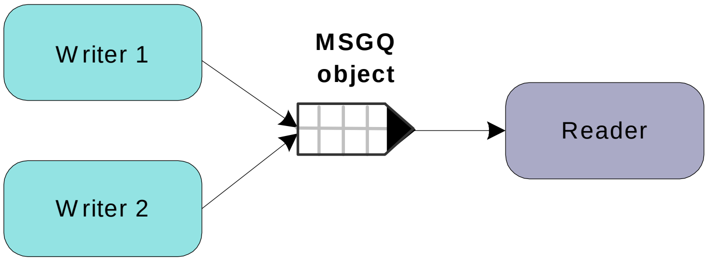


Figure 2-1. Writers and Reader of a Message Queue

Conceptually, the reader thread owns a message queue. The processor where the reader resides opens a message queue. Writer threads locate existing message queues to get access to them.

Messages must be allocated from the MSGQ module. Once a message is allocated, it can be sent on any message queue. Once a message is sent, the writer loses ownership of the message and should not attempt to modify the message. Once the reader receives the message, it owns the message. It may either free the message or re-use the message.

Messages in a message queue can be of variable length. The only requirement is that the first field in the definition of a message must be a MSGQ_MsgHeader element.

```
typedef struct MyMsg {
    MSGQ_MsgHeader header;
    ...
} MyMsg;
```

The MSGQ API uses the MSGQ_MsgHeader internally. Your application should not modify or directly access the fields in the MSGQ_MsgHeader.

The MSGQ module has the following components:

- **MSGQ API.** Applications call the MSGQ functions to open and use a message queue object to send and receive messages. For an overview, see “MSGQ APIs” on page 213. For details, see the sections on the individual APIs.
- **Allocators.** Messages sent via MSGQ must be allocated by an allocator. The allocator determines where and how the memory for the message is allocated. For more about allocators, see the *DSP/BIOS User’s Guide* (SPRU423F).
- **Transports.** Transports are responsible for locating and sending messages with other processors. For more about transports, see the *DSP/BIOS User’s Guide* (SPRU423F).

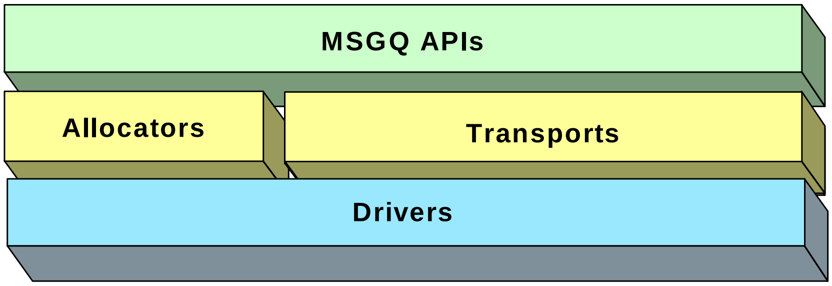


Figure 2-2. Components of the MSGQ Architecture

For more about using the MSGQ module—including information about multi-processor issues and a comparison of data transfer modules—see the *DSP/BIOS User's Guide* (SPRU423F).

MSGQ APIs

The MSGQ APIs are used to open and close message queues and to send and receive messages. The MSGQ APIs shield the application from having to contain any knowledge about transports and allocators.

The following figure shows the call sequence of the main MSGQ functions:

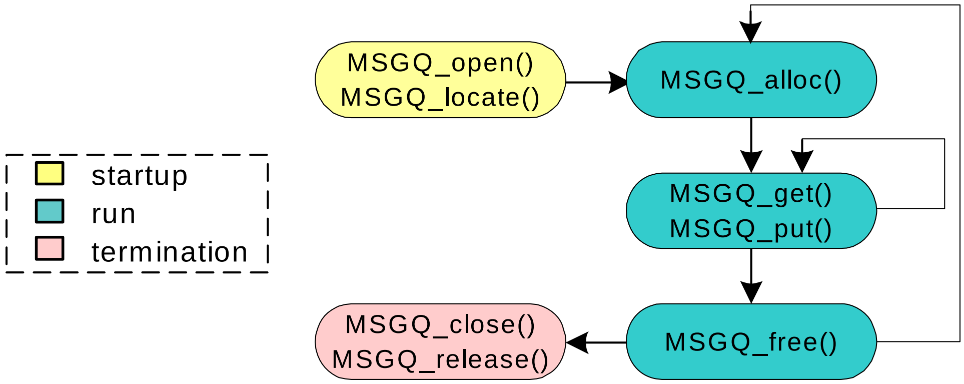


Figure 2-3. MSGQ Function Calling Sequence

The reader calls the following APIs:

- MSGQ_open
- MSGQ_get
- MSGQ_free
- MSGQ_close

A writer calls the following APIs:

- MSGQ_locate or MSGQ_locateAsync
- MSGQ_alloc
- MSGQ_put
- MSGQ_release

Wherever possible, the MSGQ APIs have been written to have a deterministic execution time. This allows application designers to be certain that messaging will not consume an unknown number of cycles.

In addition, the MSGQ functions support use of message queues from all types of DSP/BIOS threads: HWIs, SWIs, and TSKs. That is, calls that may be synchronous (blocking) have an asynchronous (non-blocking) alternative.

Static Configuration

In order to use the MSGQ module and the allocators it depends upon, you must statically configure the following:

- ENABLEMSGQ property of the MSGQ module using Tconf (see “MSGQ Manager Properties” on page 217)
- MSGQ_config variable in application code (see below)

- PROCID property of the GBL module using Tconf (see “GBL Module Properties” on page 108)
- ENABLEPOOL property of the POOL module using Tconf (see “POOL Manager Properties” on page 265)
- POOL_config variable in application code (see “Static Configuration” on page 263)

An application must provide a filled in MSGQ_config variable in order to use the MSGQ module.

```
MSGQ_Config MSGQ_config;
```

The MSGQ_Config type has the following structure:

```
typedef struct MSGQ_Config {
   MSGQ_Obj           *msgqQueues;      /* Array of message queue handles */
   MSGQ_TransportObj *transports;      /* Array of transports */
   Uint16             numMsgqQueues;   /* Number of message queue handles*/
   Uint16             numProcessors;   /* Number of processors */
   Uint16             startUninitialized;  /* First msgq to init */
   MSGQ_Queue         errorQueue;      /* Receives async transport errors*/
   Uint16             errorPoolId;     /* Alloc error msgs from poolId */
} MSGQ_Config;
```

The fields in the MSGQ_Config structure are described in the following table:

Field	Type	Description
msgqQueues	MSGQ_Obj *	Array of message queue objects. The fields of each object do not need to be initialized.
transports	MSGQ_TransportObj *	Array of transport objects. The fields of each object must be initialized.
numMsgqQueues	Uint16	Length of the msgqQueues array.
numProcessors	Uint16	Length of the transports array.
startUninitialized	Uint16	Index of the first message queue to initialize in the msgqQueue array. This should be set to 0.
errorQueue	MSGQ_Queue	Message queue to receive transport errors. Initialize to MSGQ_INVALIDMSGQ.
errorPoolId	Uint16	Allocator to allocate transport errors. Initialize to POOL_INVALIDID.

Internally, MSGQ references its configuration via the MSGQ_config variable. If the MSGQ module is enabled (via Tconf) but the application does not provide the MSGQ_config variable, the application cannot be linked successfully.

In the MSGQ_Config structure, an array of MSGQ_TransportObj items defines transport objects with the following structure:

```
typedef struct MSGQ_TransportObj {
  MSGQ_MqtInit  initFxn;    /* Transport init func */
  MSGQ_TransportFxns *fxns; /* Interface funcs */
  Ptr         params; /* Setup parameters */
  Ptr         object; /* Transport-specific object */
  Uint16      procId; /* Processor Id talked to */
} MSGQ_TransportObj;
```

The following table describes the fields in the MSGQ_TransportObj structure:

Field	Type	Description
initFxn	MSGQ_MqtInit	Initialization function for this transport. This function is called during DSP/BIOS startup. More explicitly it is called before main().
fxns	MSGQ_TransportFxns *	Pointer to the transport's interface functions.
params	Ptr	Pointer to the transport's parameters. This field is transport-specific. Please see documentation provided with your transport for a description of this field.
info	Ptr	State information needed by the transport. This field is initialized and managed by the transport. Refer to the specific transport implementation to determine how to use this field
procId	Uint16	Numeric ID of the processor that this transport communicates with. The current processor must have a procId field that matches the GBL.PROCID property.

If no parameter structure is specified (that is, NULL is used) for the MSGQ_TransportObj, the transport uses its default parameters.

The order of the transports array is by processor. The first entry communicates with processor 0, the next entry with processor 1, and so on. On processor n, the nth entry in the transport array should be MSGQ_NOTRANSPORT, since there is no transport to itself. The following example shows a configuration for a single-processor application (that is, processor 0). Note that the 0th entry is MSGQ_NOTRANSPORT

```
#define NUMMSGQUEUES  4  /* # of local message queues*/
#define NUMPROCESSORS 1 /* Single processor system */

static MSGQ_Obj           msgQueues[NUMMSGQUEUES];
static MSGQ_TransportObj  transports[NUMPROCESSOR] =
                            {MSGQ_NOTRANSPORT};

MSGQ_Config MSGQ_config = {
   msgQueues,
   transports,
   NUMMSGQUEUES,
   NUMPROCESSORS,
   0,
   MSGQ_INVALIDMSGQ,
   POOL_INVALIDID
};
```

Managing Transports at Run-Time

As described in the previous section, MSGQ uses an array of transports of type MSGQ_TransportObj in the MSGQ_config variable. This array is processor ID based. For example, MSGQ_config->transports[0] is the transport to processor 0. Therefore, if a single binary is used on multiple processors, the array must be changed at run-time.

As with the GBL_setProcId API, the transports array can be managed in the User Init Function (see GBL Module Properties). DSP/BIOS only uses MSGQ_config and the transports array after the User Init Function returns.

There are several ways to manage the transports array. Two common ways are as follows:

- **Create a static two-dimensional transports array and select the correct one.** Assume a single image will be used for two processors (procId 0 and 1) in a system with NUMPROCESSORS (3 in this example) processors. The transports array in the single image might look like this:

  ```
  MSGQ_TransportObj transports[2][NUMPROCESSORS] =
  { { MSGQ_NOTRANSPORT,    // proc 0 talk to proc 0
      {...},               // proc 0 talk to proc 1
      {...},               // proc 0 talk to proc 2
    },
    { {...},               // proc 1 talk to proc 0
      MSGQ_NOTRANSPORT,    // proc 1 talk to proc 1
      {...},               // proc 1 talk to proc 2
    }
  }
  ```

  In the User Init Function, the application would call GBL_setProcId with the correct processor ID. Then it would assign the correct transport array to MSGQ_config. For example, for processor 1, it would do the following:

  ```
  MSGQ_config.transports = transports[1];
  ```

Note that this approach does not scale well as the number of processors in the system increases.

- **Fill in the transports array in the User Init Function.** In the User Init Function, you can fill in the contents of the transports array. You would still statically define a 1-dimensional transports array as follows:

  ```
  MSGQ_TransportObj transports[NUMPROCESSORS];
  ```

  This array would not be initialized. The initialization would occur in the User Init Function. For example on processor 1, it would fill in the transports array as follows.

```
transports[0].initFxn = ...
transports[0].fxns = ...
transports[0].object = ...
transports[0].params = ...
transports[0].procId = 0;
transports[1] = MSGQ_NOTRANSPORT;//no self-transport
transports[2].initFxn = ...
transports[2].fxns = ...
...
transports[2].procId = 2;
MSGQ_config.transport = transports;
```

Note that some of the parameters may not be able to be determined easily at run-time, therefore you may need to use a mixture of these two options.

Message Queue Management

When a message queue is closed, the threads that located the closing message queue are not notified. No messages should be sent to a closed message queue. Additionally, there should be no active call to MSGQ_get or MSGQ_getAttrs to a message queue that is being closed. When a message queue is closed, all unread messages in the message queue are freed.

MSGQ Manager Properties

To configure the MSGQ manager, the MSGQ_Config structure must be defined in the C code. See “Static Configuration” on page 213.

The following global property must also be set in order to use the MSGQ module:

- **Enable Message Queue Manager**. If ENABLEMSGQ is TRUE, each transport and message queue specified in the MSGQ_config structure (see “Static Configuration” on page 213) is initialized.

  Tconf Name: ENABLEMSGQ Type: Bool

  Example: `bios.MSGQ.ENABLEMSGQ = true;`

MSGQ_alloc *Allocate a message*

C Interface

Syntax

status = MSGQ_alloc(poolId, msg, size);

Parameters

Uint16	poolId;	/* allocate the message from this allocator */
MSGQ_Msg	*msg;	/* pointer to the returned message */
Uint16	size;	/* size of the requested message */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_alloc returns a message from the specified allocator. The size is in minimum addressable data units (MADUs).

This function is performed by a writer. This call is non-blocking and can be called from a HWI, SWI or TSK.

All messages must be allocated from an allocator. Once a message is allocated it can be sent. Once a message is received, it must either be freed or re-used.

The poolId must correspond to one of the allocators specified by the allocators field of the POOL_Config structure specified by the application. (See “Static Configuration” on page 263.)

If a message is allocated, SYS_OK is returned. Otherwise, SYS_EINVAL is returned if the poolId is invalid, and SYS_EALLOC is returned if no memory is available to meet the request.

Constraints and Calling Context

- All message definitions must have MSGQ_MsgHeader as its first field. For example:

```
struct MyMsg {
   MSGQ_MsgHeader header; /* Required field */
   ...                    /* User fields */
}
```

Example

```
/* Allocate a message */
status = MSGQ_alloc(STATICPOOLID, (MSGQ_Msg *)&msg,
       sizeof(MyMsg));
if (status != SYS_OK) {
    SYS_abort("Failed to allocate a message");
}
```

See Also

MSGQ_free

MSGQ_close *Close a message queue*

C Interface

Syntax

```
status = MSGQ_close(msgqQueue);
```

Parameters

```
MSGQ_Queue          msgqQueue;   /* Message queue to close */
```

Return Value

```
Int                 status;      /* status */
```

Reentrant

yes

Description

MSGQ_close closes a message queue. If any messages are in the message queue, they are deleted.

This function is performed by the reader.

Threads that have located (with MSGQ_locate or MSGQ_locateAsync) the message queue being closed are not notified about the closure.

If successful, this function returns SYS_OK.

Constraints and Calling Context

- The message queue must have been returned from MSGQ_open.

See Also

MSGQ_open

MSGQ_count *Return the number of messages in a message queue*

C Interface

Syntax

status = MSGQ_count(msgqQueue, count);

Parameters

MSGQ_Queue	msgqQueue;	/* Message queue to count */
Uns	*count;	/* Pointer to returned count */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

This API determines the number of messages in a specific message queue. Only the processor that opened the message queue should call this API to determine the number of messages in the reader's message queue. This API is not thread safe with MSGQ_get when accessing the same message queue, so the caller of MSGQ_count must prevent any calls to MSGQ_get.

If successful, this function returns SYS_OK.

Constraints and Calling Context

- The message queue must have been returned from a MSGQ_open call.

Example

```
status = MSGQ_count(readerMsgQueue, &count);
if (status != SYS_OK) {
   return;
}
LOG_printf(&trace, "There are %d messages.", count);
```

See Also

MSGQ_open

MSGQ_free *Free a message*

C Interface

Syntax
status = MSGQ_free(msg);

Parameters

MSGQ_Msg	msg;	/* Message to be freed */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_free frees a message back to the allocator.

If successful, this function returns SYS_OK.

This call is non-blocking and can be called from a HWI, SWI or TSK.

Constraints and Calling Context

- The message must have been allocated via MSGQ_alloc.

Example

```
status = MSGQ_get(readerMsgQueue, (MSGQ_Msg *)msg,
              SYS_FOREVER);
if (status != SYS_OK) {
   SYS_printf("MSGQ_get call failed.");
}
// process message

MSGQ_free(msg);
```

See Also

MSGQ_alloc

MSGQ_get *Receive a message from the message queue*

C Interface

Syntax

status = MSGQ_get(msgqQueue, msg, timeout);

Parameters

MSGQ_Queue	msgqQueue;	/* Message queue */
MSGQ_Msg	*msg;	/* Pointer to the returned message */
Uns	timeout;	/* Duration to block if no message */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_get returns a message sent via MSGQ_put. The order of retrieval is FIFO.

This function is performed by the reader. Once a message has been received, the reader is responsible for freeing or re-sending the message.

If no messages are present, the pend() function specified in the MSGQ_Attrs passed to MSGQ_open for this message queue is called. The pend() function blocks up to the timeout value (SYS_FOREVER = forever). The timeout units are system clock ticks.

This function is deterministic if timeout is zero. MSGQ_get can be called from a TSK with any timeout. It can be called from a HWI or SWI if the timeout is zero.

If successful, this function returns SYS_OK. Otherwise, SYS_ETIMEOUT is returned if the timeout expires before the message is received.

Constraints and Calling Context

- Only one reader of a message queue is allowed concurrently.
- The message queue must have been returned from a MSGQ_open call.

Example

```
status = MSGQ_get(readerMsgQueue, (MSGQ_Msg *)&msg, 0);
if (status != SYS_OK) {
   /* No messages to process */
   return;
}
```

See Also

MSGQ_put
MSGQ_open

MSGQ_getAttrs *Returns the attributes of a message queue*

C Interface

Syntax

status = MSGQ_getAttrs(msgqQueue, attrs);

Parameters

MSGQ_Queue	msgqQueue;	/* Message queue */
MSGQ_Attrs	*attrs;	/* Attributes of message queue */

Return Value

Int	status	/* status */

Reentrant

yes

Description

MSGQ_getAttrs fills in the attrs structure passed to it with the attributes of a local message queue. These attributes are set by MSGQ_open.

The API returns SYS_OK unless the message queue is not local (that is, it was opened on another processor). If the message queue is not local, the API returns SYS_EINVAL and does not change the contents of the passed in attrs structure.

Example

```
status = MSGQ_getAttrs (msgqQueue, &attrs);
if (status != SYS_OK) {
   return;
}
notifyHandle = attrs.notifyHandle;
```

Constraints and Calling Context

- The message queue must have been returned from a MSGQ_open call and must be valid.
- This function can be called from a HWI, SWI or TSK.

See Also

MSGQ_open

MSGQ_getDstQueue *Get destination message queue field in a message*

C Interface

Syntax

```
MSGQ_getDstQueue(msg, msgqQueue);
```

Parameters

```
MSGQ_Msg          msg;          /* Message */
MSGQ_Queue        *msgqQueue;   /* Message queue */
```

Return Value

Void

Reentrant

yes

Description

This API allows the application to determine the destination message queue of a message. This API is generally used by transports to determine the final destination of a message. This API can also be used by the application once the message is received.

This function can be called from a HWI, SWI or TSK.

Constraints and Calling Context

- The message must have been sent via MSGQ_put.

MSGQ_getMsgId *Return the message ID from a message*

C Interface

Syntax

msgId = MSGQ_getMsgId(msg);

Parameters

MSGQ_Msg	msg;	/* Message */

Return Value

Uint16	msgId;	/* Message ID */

Reentrant

yes

Description

MSGQ_getMsgId returns the message ID from a received message. This message ID is specified via the MSGQ_setMsgId function.

This function can be called from a HWI, SWI or TSK.

Example

```
/* Make sure the message is the one expected */
if (MSGQ_getMsgId((MSGQ_Msg)msg) != MESSAGEID) {
   SYS_abort("Unexpected message");
}
```

See Also

MSGQ_setMsgId

MSGQ_getMsgSize *Return the message size from a message*

C Interface

Syntax

```
size = MSGQ_getMsgSize(msg);
```

Parameters

```
MSGQ_Msg          msg;          /* Message */
```

Return Value

```
Uint16            size;         /* Message size */
```

Reentrant

yes

Description

MSGQ_getMsgSize returns the size of the message buffer out of the received message. The size is in minimum addressable data units (MADUs).

This function can be used to determine if a message can be re-used.

This function can be called from a HWI, SWI or TSK.

See Also

MSGQ_alloc

MSGQ_getSrcQueue *Extract the reply destination from a message*

C Interface

Syntax

status = MSGQ_getSrcQueue(msg, msgqQueue);

Parameters

MSGQ_Msg	msg;	/* Received message */
MSGQ_Queue	*msgqQueue;	/* Message queue */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

Many times a receiver of a message wants to reply to the sender of the message (for example, to send an acknowledgement). When a valid msgqQueue is specified in MSGQ_setSrcQueue, the receiver of the message can extract the message queue via MSGQ_getSrcQueue.

This is basically the same as a MSGQ_locate function without knowing the name of the message queue. This function can be used even if the queueName used with MSGQ_open was NULL or non-unique.

Note: The msgqQueue may not be the sender's message queue handle. The sender is free to use any created message queue handle.

This function can be called from a HWI, SWI or TSK.

If successful, this function returns SYS_OK.

Example

```
/* Get the handle and send the message back. */
status = MSGQ_getSrcQueue((MSGQ_Msg)msg, &replyQueue);
if (status != SYS_OK) {
   /* Free the message and abort */
   MSGQ_free((MSGQ_Msg)msg);
   SYS_abort("Failed to get handle from message");
}
status = MSGQ_put(replyQueue, (MSGQ_Msg)msg);
```

See Also

MSGQ_getAttrs
MSGQ_setSrcQueue

MSGQ_isLocalQueue *Return whether message queue is local or on other processor*

C Interface

Syntax

flag = MSGQ_isLocalQueue(msgqQueue);

Parameters

MSGQ_Queue	msgqQueue;	/* Message queue */

Return Value

Bool	flag;	/* status */

Reentrant

yes

Description

This API determines whether the message queue is local (that is, opened on this processor) or remote (that is, opened on a different processor).

If the message queue is local, the flag returned is TRUE. Otherwise, it is FALSE.

Constraints and Calling Context

- This function can be called from a HWI, SWI or TSK.

Example

```
flag = MSGQ_isLocalQueue(readerMsgQueue);
if (flag == TRUE) {
    /* Message queue is local */
   return;
}
```

See Also

MSGQ_open

MSGQ_locate *Synchronously find a message queue*

C Interface

Syntax

status = MSGQ_locate(queueName, msgqQueue, locateAttrs);

Parameters

String	queueName;	/* Name of message queue to locate */
MSGQ_Queue	*msgqQueue;	/* Return located message queue here */
MSGQ_LocateAttrs	*locateAttrs;	/* Locate attributes */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

The MSGQ_locate function is used to locate an opened message queue. This function is synchronous (that is, it can block if timeout is non-zero).

This function is performed by a writer. The reader must have already called MSGQ_open for this queueName.

MSGQ_locate firsts searches the local message queues for a name match. If a match is found, that message queue is returned. If no match is found, the transports are queried one at a time. If a transport locates the queueName, that message queue is returned. If the transport does not locate the message queue, the next transport is queried. If no transport can locate the message queue, an error is returned.

In a multiple-processor environment, transports can block when they are queried if you call MSGQ_locate. The timeout in the MSGQ_LocateAttrs structure specifies the maximum time each transport can block. The default is SYS_FOREVER (that is, each transport can block forever). Remember that if you specify 1000 clock ticks as the timeout, the total blocking time could be 1000 * number of transports.

Note that timeout is not a fixed amount of time to wait. It is the maximum time each transport waits for a positive or negative response. For example, suppose your timeout is 1000, but the response (found or not found) comes back in 600 ticks. The transport returns the response then; it does not wait for another 400 ticks to recheck for a change.

If you do not want to allow blocking, call MSGQ_locateAsync instead of MSGQ_locate.

The locateAttrs parameter is of type MSGQ_LocateAttrs. This type has the following structure:

```
typedef struct MSGQ_LocateAttrs {
    Uns          timeout;
} MSGQ_LocateAttrs;
```

The timeout is the maximum time a transport can block on a synchronous locate in system clock ticks. The default attributes are as follows:

```
MSGQ_LocateAttrs  MSGQ_LOCATEATTRS = {SYS_FOREVER};
```

If successful, this function returns SYS_OK. Otherwise, it returns SYS_ENOTFOUND to indicate that it could not locate the specified message queue.

Constraints and Calling Context

- Cannot be called from main().
- Cannot be called in a SWI or HWI context.

Example

```
status = MSGQ_locate("reader", &readerMsgQueue, NULL);
   if (status != SYS_OK) {
    SYS_abort("Failed to locate reader message queue");
}
```

See Also

MSGQ_locateAsync
MSGQ_open

MSGQ_locateAsync *Asynchronously find a message queue*

C Interface

Syntax

```
status = MSGQ_locateAsync(queueName, replyQueue, locateAsyncAttrs);
```

Parameters

String	queueName;	/* Name of message queue to locate */
MSGQ_Queue	replyQueue;	/* Msgq to send locate message */
MSGQ_LocateAsyncAttrs	*locateAsyncAttrs;	/* Locate attributes */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_locateAsync firsts searches the local message queues for a name match. If one is found, an asynchronous locate message is sent to the specified message queue (in the replyQueue parameter). If it is not, all transports are asked to start an asynchronous locate search. After all transports have been asked to start the search, the API returns.

If a transport locates the message queue, an asynchronous locate message is sent to the specified replyQueue. If no transport can locate the message queue, no message is sent.

This function is performed by a writer. The reader must have already called MSGQ_open for this queueName. An asynchronous locate can be performed from a SWI or TSK. It cannot be performed in main().

The message ID for an asynchronous locate message is:

```
/* Asynchronous locate message ID */
#define MSGQ_ASYNCLOCATEMSGID   0xFF00
```

The MSGQ_LocateAsyncAttrs structure has the following fields:

```
typedef struct MSGQ_LocateAsyncAttrs {
    Uint16      poolId;
    Arg         arg;
} MSGQ_LocateAttrs;
```

The default attributes are as follows:

```
MSGQ_LocateAsyncAttrs  MSGQ_LOCATEASYNCATTRS = {0, 0};
```

The locate message is allocated from the allocator specified by the locateAsyncAttrs->poolId field.

The locateAsyncAttrs->arg value is included in the asynchronous locate message. This field allows you to correlate requests with the responses.

Once the application receives an asynchronous locate message, it is responsible for freeing the message. The asynchronous locate message received by the replyQueue has the following structure:

```
typedef struct MSGQ_AsyncLocateMsg {
    MSGQ_MsgHeader  header;
    MSGQ_Queue      msgqQueue;
    Arg             arg;
} MSGQ_AsyncLocateMsg;
```

Field	Type	Description
header	MSGQ_MsgHeader	Required field for every message.
msgqQueue	MSGQ_Queue	Located message queue handle.
Arg	Arg	Value specified in MSGQ_LocateAttrs for this asynchronous locate.

This function returns SYS_OK to indicated that an asynchronous locate was started. This status does not indicate whether or not the locate will be successful. The SYS_EALLOC status is returned if the message could not be allocated.

Constraints and Calling Context

- The allocator must be able to allocate an asynchronous locate message.
- Cannot be called in the context of main().

Example

The following example shows an asynchronous locate performed in a task. Time spent blocking is dictated by the timeout specified in the MSGQ_get call. (Error handling statements were omitted for brevity.)

```
status = MSGQ_open("myMsgQueue", &myQueue, &msgqAttrs);

locateAsyncAttrs              = MSGQ_LOCATEATTRS;
locateAsyncAttrs.poolId       = STATICPOOLID;

MSGQ_locateAsync("msgQ1", myQueue, &locateAsyncAttrs);
status = MSGQ_get(myQueue, &msg, SYS_FOREVER);
if (MSGQ_getMsgId((MSGQ_Msg)msg) ==
                         MSGQ_ASYNCLOCATEMSGID) {
    readerQueue = msg->msgqQueue;
}
MSGQ_free((MSGQ_Msg)msg);
```

See Also

MSGQ_locate
MSGQ_free
MSGQ_open

MSGQ_open *Open a message queue*

C Interface

Syntax

status = MSGQ_open(queueName, msgqQueue, attrs);

Parameters

String	queueName;	/* Unique name of the message queue */
MSGQ_Queue	*msgqQueue;	/* Pointer to returned message queue */
MSGQ_Attrs	*attrs;	/* Attributes of the message queue */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_open is the function to open a message queue. This function selects and returns a message queue from the array provided in the static configuration (that is, MSGQ_config->msgqQueues).

This function is on the processor where the reader resides. The reader then uses this message queue to receive messages.

If successful, this function returns SYS_OK. Otherwise, it returns SYS_ENOTFOUND to indicate that no empty spot was available in the message queue array.

If the application will use MSGQ_locate or MSGQ_locateAsync to find this message queue, the queueName must be unique. If the application will never need to use the locate APIs, the queueName may be NULL or a non-unique name.

Instead of using a fixed notification mechanism, such as SEM_pend and SEM_post, the MSGQ notification mechanism is supplied in the attrs parameter, which is of type MSGQ_Attrs. If attrs is NULL, the new message queue is assigned a default set of attributes. The structure for MSGQ_Attrs is as follows:

```
typedef struct MSGQ_Attrs {
    Ptr         notifyHandle;
    MSGQ_Pend   pend;
    MSGQ_Post   post;
} MSGQ_Attrs;
```

The MSGQ_Attrs fields are as follows:

Field	Type	Description
notifyHandle	Ptr	Handle to use in the pend() and post() functions.
Pend	MSGQ_Pend	Function pointer to a user-specified pend function.
Post	MSGQ_Post	Function pointer to a user-specified post function.

The default attributes are:

```
MSGQ_Attrs MSGQ_ATTRS = {
   NULL,                /* notifyHandle */
   (MSGQ_Pend)SYS_zero, /* NOP pend */
   FXN_F_nop            /* NOP post */
};
```

The following typedefs are provided by the MSGQ module to allow easier casting of the pend and post functions:

```
                typedef Bool (*MSGQ_Pend)(Ptr notifyHandle, Uns timeout);
                typedef Void (*MSGQ_Post)(Ptr notifyHandle);
```

The post() function you specify is always called within MSGQ_put when a writer sends a message.

A reader calls MSGQ_get to receive a message. If there is a message, it returns that message, and the pend() function is not called. The pend() function is only called if there are no messages to receive.

The pend() and post() functions must act in a binary manner. For instance, SEM_pend and SEM_post treat the semaphore as a counting semaphore instead of binary. So SEM_pend and SEM_post are an invalid pend/post pair. The following example, in which the reader calls MSGQ_get with a timeout of SYS_FOREVER, shows why:

1. A writer sends 10 messages, making the count 10 in the semaphore.
2. The reader then calls MSGQ_get 10 times. Each call returns a message without calling the pend() function.
3. The reader then calls MSGQ_get again. Since there are no messages, the pend() function is called. Since the semaphore count was 10, SEM_pend returns TRUE immediately from the pend(). MSGQ would check for messages and there would still be none, so pend() would be called again. This would repeat 9 more times until the count was zero.

If the pend() function were binary (for example, a binary semaphore), the pend() function would be called at most two times in step 3.

So instead of using SEM_pend and SEM_post for synchronous (blocking) opens, you should use SEM_pendBinary and SEM_postBinary.

The following notification attributes could be used if the reader is a SWI function (which cannot block):

```
MSGQ_Attrs attrs   = MSGQ_ATTRS; // default attributes
// leave attrs.pend as a NOP
attrs.notifyHandle = (Ptr)swiHandle;
attrs.post         = (MSGQ_Pend)SWI_post;
```

The following notification attributes could be used if the reader is a TSK function (which can block):

```
MSGQ_Attrs attrs   = MSGQ_ATTRS; // default attributes
attrs.notifyHandle = (Ptr)semHandle;
attrs.pend         = (MSGQ_Pend)SEM_pendBinary;
attrs.post         = (MSGQ_Post)SEM_postBinary;
```

Constraints and Calling Context

- The message queue returned is to be used by the caller of MSGQ_get. It should not be used by writers to that message queue (that is, callers of MSGQ_put). Writers should use the message queue returned by MSGQ_locate, MSGQ_locateAsync, or MSGQ_getSrcQueue.

- If a post() function is specified, the function must be non-blocking.
- If a pend() function is specified, the function must be non-blocking when timeout is zero.
- Each message queue must have a unique name if the application will use MSGQ_locate or MSGQ_locateAsync.
- The queueName must be persistent. The MSGQ module references this name internally; that is, it does not make a copy of the name.

Example

```
/* Open the reader message queue.
 * Using semaphores as notification mechanism */
msgqAttrs              = MSGQ_ATTRS;
msgqAttrs.notifyHandle = (Ptr)readerSemHandle;
msgqAttrs.pend         = (MSGQ_Pend)SEM_pendBinary;
msgqAttrs.post         = (MSGQ_Post)SEM_postBinary;
status = MSGQ_open("reader", &readerMsgQueue,
                       &msgqAttrs);
if (status != SYS_OK) {
  SYS_abort("Failed to open the reader message queue");
}
```

See Also

MSGQ_close
MSGQ_locate
MSGQ_locateAsync
SEM_pendBinary
SEM_postBinary

MSGQ_put

Place a message on a message queue

C Interface

Syntax

```
status = MSGQ_put(msgqQueue, msg);
```

Parameters

MSGQ_Queue	msgqQueue;	/* Destination message queue */
MSGQ_Msg	msg;	/* Message */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

MSGQ_put places a message into the specified message queue.

This function is performed by a writer. This function is non-blocking, and can be called from a HWI, SWI or TSK.

The post() function for the destination message queue is called as part of the MSGQ_put. The post() function is specified MSGQ_open call in the MSGQ_Attrs parameter.

If successful, this function returns SYS_OK. Otherwise, it may return an error code returned by the transport.

There are several features available when sending a message.

- A msgId passed to MSGQ_setMsgId can be used to indicate the type of message it is. Such a type is completely application-specific, except for IDs defined for MSGQ_setMsgId. The reader of a message can use MSGQ_getMsgId to get the ID from the message.
- The source message queue parameter to MSGQ_setSrcQueue allows the sender of the message to specify a source message queue. The receiver of the message can use MSGQ_getSrcQueue to extract the embedded message queue from the message. A client/server application might use this mechanism because it allows the server to reply to a message without first locating the sender. For example, each client would have its own message queue that it specifies as the source message queue when it sends a message to the server. The server can use MSGQ_getSrcQueue to get the message queue to reply back to.

If MSGQ_put returns an error, the user still owns the message and is responsible for freeing the message (or re-sending it).

Constraints and Calling Context

- The msgqQueue must have been returned from MSGQ_locate, MSGQ_locateAsync or MSGQ_getSrcQueue (or MSGQ_open if the reader of the message queue wants to send themselves a message).
- If MSGQ_put does not return SYS_OK, the message is still owned by the caller and must either be freed or re-used.

Example

```
/* Send the message back. */
status = MSGQ_put(replyMsgQueue, (MSGQ_Msg)msg);
if (status != SYS_OK) {
   /* Need to free the message */
   MSGQ_free((MSGQ_Msg)msg);
   SYS_abort("Failed to send the message");
}
```

See Also

MSGQ_get
MSGQ_open
MSGQ_setMsgId
MSGQ_getMsgId
MSGQ_setSrcQueue
MSGQ_getSrcQueue

MSGQ_release *Release a located message queue*

C Interface

Syntax
status = MSGQ_release(msgqQueue);

Parameters

MSGQ_Queue	msgqQueue;	/* Message queue to release */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

This function releases a located message queue. That is, it releases a message queue returned from MSGQ_locate or MSGQ_locateAsync.

This function is performed by a writer.

If successful, this function returns SYS_OK. Otherwise, it may return an error code returned by the transport.

Constraints and Calling Context

- The handle must have been returned from MSGQ_locate or MSGQ_locateAsync.

See Also

MSGQ_locate
MSGQ_locateAsync

MSGQ_setErrorHandler *Set up handling of internal MSGQ errors*

C Interface

Syntax

status = MSGQ_setErrorHandler(errorQueue, poolId);

Parameters

MSGQ_Queue	errorQueue;	/* Message queue to receive errors */
Uint16	poolId;	/* Allocator to allocate error messages */

Return Value

Int	status;	/* status */

Reentrant

yes

Description

Asynchronous errors that need to be communicated to the application may occur in a transport. If an application calls MSGQ_setErrorHandler, all asynchronous errors are then sent to the message queue specified.

The specified message queue receives asynchronous error messages (if they occur) via MSGQ_get.

poolId specifies the allocator the transport should use to allocate error messages. If the transports cannot allocate a message, no action is performed.

If this function is not called or if errorHandler is set to MSGQ_INVALIDMSGQ, no error messages will be allocated and sent.

This function can be called multiple times with only the last handler being active.

If successful, this function returns SYS_OK.

The message ID for an asynchronous error message is:

```
/* Asynchronous error message ID */
#define MSGQ_ASYNCERRORMSGID   0xFF01
```

The following is the structure for an asynchronous error message:

```
typedef struct MSGQ_AsyncErrorMsg {
    MSGQ_MsgHeader  header;
    MSGQ_MqtError   errorType;
    Uint16          mqtId;
    Uint16          parameter;
} MSGQ_AsyncErrorMsg;
```

The following table describes the fields in the MSGQ_AsyncErrorMsg structure:

Field	Type	Description
header	MSGQ_MsgHeader	Required field for every message
errorType	MSGQ_MqtError	Error ID

Field	Type	Description
mqtId	Uint16	ID of the transport that sent the error message
parameter	Uint16	Error-specific field

The following table lists the valid errorType values and the meanings of their arg fields:

errorType	mqtId	parameter
MSGQ_MQTERROREXIT	ID of the transport that is exiting.	Not used.
MSGQ_MQTFAILEDPUT	ID of the transport that failed to send a message.	ID of destination queue. The parameter is 16 bits, so only the lower 16 bits of the msgqQueue is logged. The top 16 bits of the msgQueue contain the destination processor ID, which is also the mqtId. You can OR the mqtId shifted over by 16 bits with the parameter to get the full destination msgqQueue.
MSGQ_MQTERRORINTERNAL	Generic internal error.	Transport defined.
MSGQ_MQTERRORPHYSICAL	Problem with the physical link.	Transport defined.
MSGQ_MQTERRORALLOC	Transport could not allocate memory.	Size of the requested memory.

MSGQ_open
MSGQ_get

MSGQ_setMsgId *Set the message ID in a message*

C Interface

Syntax

```
MSGQ_setMsgId(msg, msgId);
```

Parameters

```
MSGQ_MSG        msg;      /* Message */
Uint16          msgId;    /* Message id  */
```

Return Value

Void

Reentrant

yes

Description

Inside each message is a message id field. This API sets this field. The value of msgId is application-specific. MSGQ_getMsgId can be used to extract this field from a message.

When a message is allocated, the value of this field is MSGQ_INVALIDMSGID. When MSGQ_setMsgId is called, it updates the field accordingly. This API can be called multiple times on a message.

If a message is sent to another processor, the message Id field is converted by the transports accordingly (for example, endian conversion is performed).

The message IDs used when sending messages are application-specific. They can have any value except values in the following ranges:

- Reserved for the MSGQ module messages: 0xFF00 - 0xFF7F
- Reserved for internal transport usage: 0xFF80 - 0xFFFE
- Used to signify an invalid message ID: 0xFFFF

The following table lists the message IDs currently used by the MSGQ module.

Constant Defined in msgq.h	Value	Description
MSGQ_ASYNCLOCATEMSGID	0xFF00	Used to denote an asynchronous locate message.
MSGQ_ASYNCERRORMSGID	0xFF01	Used to denote an asynchronous transport error.
MSGQ_INVALIDMSGID	0xFFFF	Used as initial value when message is allocated.

Constraints and Calling Context

- Message must have been allocated originally from MSGQ_alloc.

Example

```
/* Fill in the message */
msg->sequenceNumber = 0;
MSGQ_setMsgId((MSGQ_Msg)msg, MESSAGEID);

/* Send the message */
status = MSGQ_put(readerMsgQueue, (MSGQ_Msg)msg);
    if (status != SYS_OK) {
    SYS_abort("Failed to send the message");
}
```

See Also

MSGQ_getMsgId
MSGQ_setErrorHandler

MSGQ_setSrcQueue *Set the reply destination in a message*

C Interface

Syntax

MSGQ_setSrcQueue(msg, msgqQueue);

Parameters

MSGQ_MSG	msg;	/* Message */
MSGQ_Queue	msgqQueue;	/* Message queue */

Return Value

Void

Reentrant

yes

Description

This API allows the sender to specify a message queue that the receiver of the message can reply back to (via MSGQ_getSrcQueue). The msgqQueue must have been returned by MSGQ_open.

Inside each message is a source message queue field. When a message is allocated, the value of this field is MSGQ_INVALIDMSGQ. When this API is called, it updates the field accordingly. This API can be called multiple times on a message.

If a message is sent to another processor, the source message queue field is managed by the transports accordingly.

Constraints and Calling Context

- Message must have been allocated originally from MSGQ_alloc.
- msgqQueue must have been returned from MSGQ_open.

Example

```
/* Fill in the message */
msg->sequenceNumber = 0;
MSGQ_setSrcQueue((MSGQ_Msg)msg, writerMsgQueue);

/* Send the message */
status = MSGQ_put(readerMsgQueue, (MSGQ_Msg)msg);
    if (status != SYS_OK) {
    SYS_abort("Failed to send the message");
}
```

See Also

MSGQ_getSrcQueue

2.17 PIP Module

Important: The PIP module is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

The PIP module is the buffered pipe manager.

Functions

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

PIP_Obj Structure

Members

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PIP Manager Properties and PIP Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
bufSeg	Reference	prog.get("L0SARAM")
bufAlign	Int16	1
frameSize	Int16	8
numFrames	Int16	2
monitor	EnumString	"reader" ("writer", "none")
notifyWriterFxn	Extern	prog.extern("FXN_F_nop")
notifyWriterArg0	Arg	0
notifyWriterArg1	Arg	0
notifyReaderFxn	Extern	prog.extern("FXN_F_nop")
notifyReaderArg0	Arg	0
notifyReaderArg1	Arg	0

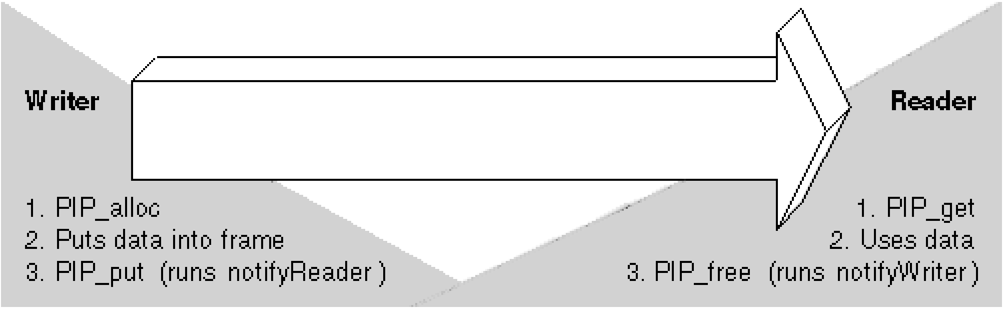
Description

The PIP module manages data pipes, which are used to buffer streams of input and output data. These data pipes provide a consistent software data structure you can use to drive I/O between the DSP device and all kinds of real-time peripheral devices.

Each pipe object maintains a buffer divided into a fixed number of fixed length frames, specified by the numframes and framesize properties. All I/O operations on a pipe deal with one frame at a time; although each frame has a fixed length, the application can put a variable amount of data in each frame up to the length of the frame.

A pipe has two ends, as shown in Figure Figure 2-4. The writer end (also called the producer) is where your program writes frames of data. The reader end (also called the consumer) is where your program reads frames of data

Figure 2-4. Pipe Schematic



Internally, pipes are implemented as a circular list; frames are reused at the writer end of the pipe after PIP_free releases them.

The notifyReader and notifyWriter functions are called from the context of the code that calls PIP_put or PIP_free. These functions can be written in C or assembly. To avoid problems with recursion, the notifyReader and notifyWriter functions normally should not directly call any of the PIP module functions for the same pipe. Instead, they should post a SWI that uses the PIP module functions. However, PIP calls may be made from the notifyReader and notifyWriter functions if the functions have been protected against re-entrancy.

Note: When DSP/BIOS starts up, it calls the notifyWriter function internally for each created pipe object to initiate the pipe's I/O.

The code that calls PIP_free or PIP_put should preserve any necessary registers.

Often one end of a pipe is controlled by an HWI and the other end is controlled by a SWI function, such as SWI_andnHook.

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

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

PIP Manager Properties

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

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

PIP Object Properties

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

To create a PIP object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myPip = bios.PIP.create("myPip");
```

The following properties can be set for a PIP object in the PIP Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myPip.comment = "my PIP";`

- **bufseg**. The memory segment that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes).

  Tconf Name: bufSeg Type: Reference

  Example: `myPip.bufSeg = prog.get("myMEM");`

- **bufalign**. The alignment (in words) of the buffer allocated within the specified memory segment.

  Tconf Name: bufAlign Type: Int16

  Example: `myPip.bufAlign = 1;`
- **framesize.** The length of each frame (in words)

  Tconf Name: frameSize Type: Int16

  Example: `myPip.frameSize = 8;`
- **numframes**. The number of frames

  Tconf Name: numFrames Type: Int16

  Example: `myPip.numFrames = 2;`
- **monitor**. The end of the pipe to be monitored by a hidden STS object. Can be set to reader, writer, or nothing. In the Statistics View analysis tool, your choice determines whether the STS display for this pipe shows a count of the number of frames handled at the reader or writer end of the pipe.

  Tconf Name: monitor Type: EnumString

  Options: "reader", "writer", "none"

  Example: `myPip.monitor = "reader";`
- **notifyWriter**. The function to execute when a frame of free space is available. This function should notify (for example, by calling SWI_andnHook) the object that writes to this pipe that an empty frame is available.

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

  Tconf Name: notifyWriterFxn Type: Extern

  Example: `myPip.notifyWriterFxn = prog.extern("writerFxn");`
- **nwarg0, nwarg1**. Two Arg type arguments for the notifyWriter function.

  Tconf Name: notifyWriterArg0 Type: Arg

  Tconf Name: notifyWriterArg1 Type: Arg

  Example: `myPip.notifyWriterArg0 = 0;`
- **notifyReader**. The function to execute when a frame of data is available. This function should notify (for example, by calling SWI_andnHook) the object that reads from this pipe that a full frame is ready to be processed.

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

  Tconf Name: notifyReaderFxn Type: Extern

  Example: `myPip.notifyReaderFxn = prog.extern("readerFxn");`
- **nrarg0, nrarg1**. Two Arg type arguments for the notifyReader function.

  Tconf Name: notifyReaderArg0 Type: Arg

  Tconf Name: notifyReaderArg1 Type: Arg

  Example: `myPip.notifyReaderArg0 = 0;`

PIP_alloc

Allocate an empty frame from a pipe

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_alloc(pipe);
```

Parameters

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

Return Value

Void

Reentrant

no

Description

PIP_alloc allocates an empty frame from the pipe you specify. You can write to this frame and then use PIP_put to put the frame into the pipe.

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

Constraints and Calling Context

- Before calling PIP_alloc, a function should check the writerNumFrames member of the PIP_Obj structure by calling PIP_getWriterNumFrames to make sure it is greater than 0 (that is, at least one empty frame is available).
- PIP_alloc can only be called one time before calling PIP_put. You cannot operate on two frames from the same pipe simultaneously.

Example

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

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

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

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

    /* copy input data to output frame */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);
    size = PIP_getReaderSize(in);
    PIP_setWriterSize(out, size);
    for (; size > 0; size--) {
        *dst++ = *src++;
    }

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

The example for HST_getpipe, page 2–141, also uses a pipe with host channel objects.

See Also

PIP_free
PIP_get
PIP_put
HST_getpipe

PIP_free

Recycle a frame that has been read to a pipe

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_free(pipe);
```

Parameters

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

Return Value

Void

Reentrant

no

Description

PIP_free releases a frame after you have read the frame with PIP_get. The frame is recycled so that PIP_alloc can reuse it.

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

Constraints and Calling Context

- When called within an HWI, the code sequence calling PIP_free must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

See the example for PIP_alloc, page 2–248. The example for HST_getpipe, page 2–141, 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

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_get(pipe);
```

Parameters

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

Return Value

Void

Reentrant

no

Description

PIP_get gets a frame from the pipe after some other function puts the frame into the pipe with PIP_put.

If full frames are available after PIP_get gets a frame, PIP_get runs the function specified by the notifyReader property of the PIP object. This function should notify (for example, by calling SWI_andnHook) the object that reads from this pipe that a full frame is available. The notifyReader function is performed as part of the thread that calls PIP_get or PIP_put. To avoid problems with recursion, the notifyReader function should not directly call any PIP module functions for the same pipe.

Constraints and Calling Context

- Before calling PIP_get, a function should check the readerNumFrames member of the PIP_Obj structure by calling PIP_getReaderNumFrames to make sure it is greater than 0 (that is, at least one full frame is available).
- PIP_get can only be called one time before calling PIP_free. You cannot operate on two frames from the same pipe simultaneously.

Example

See the example for PIP_alloc, page 2–248. The example for HST_getpipe, page 2–141, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_free
PIP_put
HST_getpipe

PIP_getReaderAddr *Get the value of the readerAddr pointer of the pipe*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

readerAddr = PIP_getReaderAddr(pipe);

Parameters

PIP_Handle	pipe;	/* pipe object handle */

Return Value

Ptr	readerAddr

Reentrant

yes

Description

PIP_getReaderAddr is a C function that returns the value of the readerAddr pointer of a pipe object. The readerAddr pointer is normally used following a call to PIP_get, as the address to begin reading from.

Example

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

    if (PIP_getReaderNumFrames(in) == 0 ||
    PIP_getWriterNumFrames(out) == 0) {
        error;    }
    PIP_get(in);      /* get input data */
    PIP_alloc(out);   /* allocate output buffer */

    /* copy input data to output buffer */
    src = PIP_getReaderAddr(in);
    dst = PIP_getWriterAddr(out);
    size = PIP_getReaderSize(in);
    PIP_setWriterSize(out,size);
    for (; size > 0; size--) {
        *dst++ = *src++;
    }

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

PIP_getReaderNumFrames *Get the number of pipe frames available for reading*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
num = PIP_getReaderNumFrames(pipe);
```

Parameters

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

Return Value

```
Uns                  num;           /* number of filled frames to be read */
```

Reentrant

yes

Description

PIP_getReaderNumFrames is a C function that returns the value of the readerNumFrames element of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 2–252.

PIP_getReaderSize *Get the number of words of data in a pipe frame*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
num = PIP_getReaderSize(pipe);
```

Parameters

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

Return Value

```
Uns                 num;           /* number of words to be read from filled frame */
```

Reentrant

yes

Description

PIP_getReaderSize is a C function that returns the value of the readerSize element of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 2–252.

PIP_getWriterAddr

Get the value of the writerAddr pointer of the pipe

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
writerAddr = PIP_getWriterAddr(pipe);
```

Parameters

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

Return Value

```
Ptr                  writerAddr;
```

Reentrant

yes

Description

PIP_getWriterAddr is a C function that returns the value of the writerAddr pointer of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 2–252.

PIP_getWriterNumFrames *Get number of pipe frames available to be written to*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
num = PIP_getWriterNumFrames(pipe);
```

Parameters

PIP_Handle	pipe;	/* pipe object handle*/

Return Value

Uns	num;	/* number of empty frames to be written */

Reentrant

yes

Description

PIP_getWriterNumFrames is a C function that returns the value of the writerNumFrames element of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 2–252.

PIP_getWriterSize *Get the number of words that can be written to a pipe frame*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

num = PIP_getWriterSize(pipe);

Parameters

PIP_Handle	pipe;	/* pipe object handle*/

Return Value

Uns	num;	/* num of words to be written in empty frame */

Reentrant

yes

Description

PIP_getWriterSize is a C function that returns the value of the writerSize element of a pipe object.

As a function writes to a pipe, it can use PIP_getWriterSize to determine the maximum number words that can be written to a pipe frame.

Example

```
if (PIP_getWriterNumFrames(rxPipe) > 0) {
    PIP_alloc(rxPipe);
    DSS_rxPtr = PIP_getWriterAddr(rxPipe);
    DSS_rxCnt = PIP_getWriterSize(rxPipe);
}
```

PIP_peek

Get pipe frame size and address without actually claiming pipe frame

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
framesize = PIP_peek(pipe, addr, rw);
```

Parameters

```
PIP_Handle     pipe;       /* pipe object handle */
Ptr            *addr;      /* address of variable with frame address */
Uns            rw;         /* flag to indicate the reader or writer side */
```

Return Value

```
Int            framesize;  /* the frame size */
```

Description

PIP_peek can be used before calling PIP_alloc or PIP_get to get the pipe frame size and address without actually claiming the pipe frame.

The pipe parameter is the pipe object handle, the addr parameter is the address of the variable that keeps the retrieved frame address, and the rw parameter is the flag that indicates what side of the pipe PIP_peek is to operate on. If rw is PIP_READER, then PIP_peek operates on the reader side of the pipe. If rw is PIP_WRITER, then PIP_peek operates on the writer side of the pipe.

PIP_getReaderNumFrames or PIP_getWriterNumFrames can be called to ensure that a frame exists before calling PIP_peek, although PIP_peek returns –1 if no pipe frame exists.

PIP_peek returns the frame size, or –1 if no pipe frames are available. If the return value of PIP_peek in frame size is not –1, then *addr is the location of the frame address.

See Also

PIP_alloc
PIP_free
PIP_get
PIP_put
PIP_reset

PIP_put

Put a full frame into the pipe

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_put(pipe);
```

Parameters

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

Return Value

Void

Reentrant

no

Description

PIP_put puts a frame into a pipe after you have allocated the frame with PIP_alloc and written data to the frame. The reader can then use PIP_get to get a frame from the pipe.

After PIP_put puts the frame into the pipe, it runs the function specified by the notifyReader property of the PIP object. This function should notify (for example, by calling SWI_andnHook) the object that reads from this pipe that a full frame is ready to be processed. The notifyReader function is performed as part of the thread that called PIP_get or PIP_put. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.

Constraints and Calling Context

- When called within an HWI, the code sequence calling PIP_put must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

See the example for PIP_alloc, page 2–248. The example for HST_getpipe, page 2–141, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_free
PIP_get
HST_getpipe

PIP_reset

Reset all fields of a pipe object to their original values

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_reset(pipe);
```

Parameters

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

Return Value

Void

Description

PIP_reset resets all fields of a pipe object to their original values.

The pipe parameter specifies the address of the pipe object that is to be reset.

Constraints and Calling Context

- PIP_reset should not be called between the PIP_alloc call and the PIP_put call or between the PIP_get call and the PIP_free call.
- PIP_reset should be called when interrupts are disabled to avoid the race condition.

See Also

PIP_alloc
PIP_free
PIP_get
PIP_peek
PIP_put

PIP_setWriterSize *Set the number of valid words written to a pipe frame*

Important: This API is being deprecated and will no longer be supported in the next major release of DSP/BIOS. We recommend that you use the SIO module instead.

C Interface

Syntax

```
PIP_setWriterSize(pipe, size);
```

Parameters

```
PIP_Handle          pipe;          /* pipe object handle */
Uns                 size;          /* size to be set */
```

Return Value

```
Void
```

Reentrant

no

Description

PIP_setWriterSize is a C function that sets the value of the writerSize element of a pipe object.

As a function writes to a pipe, it can use PIP_setWriterSize to indicate the number of valid words being written to a pipe frame.

Example

See the example for PIP_getReaderAddr, page 2–252.

2.18 POOL Module

The POOL module describes the interface that allocators must provide.

Functions

None; this module describes an interface to be implemented by allocators

Constants, Types, and Structures

```
POOL_Config POOL_config;

typedef struct POOL_Config {
   POOL_Obj *allocators;    /* Array of allocators */
   Uint16    numAllocators; /* Num of allocators */
} POOL_Config;

typedef struct POOL_Obj {
   POOL_Init  initFxn;  /* Allocator init function */
   POOL_Fxns *fxns;     /* Interface functions */
   Ptr         params;  /* Setup parameters */
   Ptr         object;  /* Allocator's object */
} POOL_Obj, *POOL_Handle;
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the POOL Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
ENABLEPOOL	Bool	false

Description

The POOL module describes standard interface functions that allocators must provide. The allocator interface functions are called internally by the MSGQ module and not by user applications. A simple static allocator, called STATICPOOL, is provided with DSP/BIOS. Other allocators can be implemented by following the standard interface.

Note: This document does not discuss how to write an allocator. Information about designing allocators will be provided in a future document.

All messages sent via the MSGQ module must be allocated by an allocator. The allocator determines where and how the memory for the message is allocated.

An allocator is an instance of an implementation of the allocator interface. An application may instantiate one or more instances of an allocator.

An application can use multiple allocators. The purpose of having multiple allocators is to allow an application to regulate its message usage. For example, an application can allocate critical messages from one pool of fast on-chip memory and non-critical messages from another pool of slower external memory.

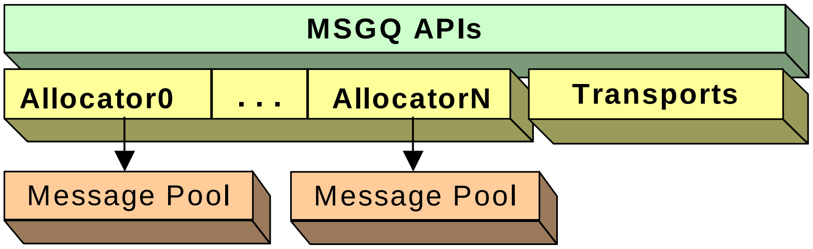


Figure 2-5. Allocators and Message Pools

Static Configuration

In order to use an allocator and the POOL module, you must statically configure the following:

- ENABLEPOOL property of the POOL module using Tconf (see "POOL Manager Properties" on page 265)
- POOL_config variable in application code (see below)

An application must provide a filled in POOL_config variable if it uses one or more allocators.

```
POOL_Config POOL_config;
```

Where the POOL_Config structure has the following structure:

```
typedef struct POOL_Config {
   POOL_Obj *allocators;    /* Array of allocators */
   Uint16    numAllocators; /* Num of allocators */
} POOL_Config;
```

The fields in this structure are as follows:

Field	Type	Description
allocators	POOL_Obj	Array of allocator objects
numAllocators	Uint16	Number of allocators in the allocator array.

If the POOL module is enabled via Tconf and the application does not provide the POOL_config variable, the application cannot be linked successfully.

The following is the POOL_Obj structure:

```
typedef struct POOL_Obj {
   POOL_Init  initFxn;  /* Allocator init function */
   POOL_Fxns *fxns;     /* Interface functions */
   Ptr        params;   /* Setup parameters */
   Ptr        object;   /* Allocator's object */
} POOL_Obj, *POOL_Handle;
```

The fields in the POOL_Obj structure are as follows:

Field	Type	Description
initFxn	POOL_Init	Initialization function for this allocator. This function will be called during DSP/BIOS initialization. More explicitly it is called before main().
fxns	POOL_Fxns *	Pointer to the allocator's interface functions.
params	Ptr	Pointer to the allocator's parameters. This field is allocator-specific. Please see the documentation provided with your allocator for a description of this field.
object	Ptr	State information needed by the allocator. This field is initialized and managed by the allocator. See the allocator documentation to determine how to specify this field.

One allocator implementation (STATICPOOL) is shipped with DSP/BIOS. Additional allocator implementations can be created by application writers.

STATICPOOL Allocator

The STATICPOOL allocator takes a user-specified buffer and allocates fixed-size messages from the buffer. The following are its configuration parameters:

```
typedef struct STATICPOOL_Params {
    Ptr         addr;
    size_t      length;
    size_t      bufferSize;
} STATICPOOL_Params;
```

The following table describes the fields in this structure:

Field	Type	Description
addr	Ptr	User supplied block of memory for allocating messages from. The address will be aligned on an 8 MADU boundary for correct structure alignment on all ISAs. If there is a chance the buffer is not aligned, allow at least 7 extra MADUs of space to allow room for the alignment.
length	size_t	Size of the block of memory pointed to by addr.
bufferSize	size_t	Size of the buffers in the block of memory. The bufferSize must be a multiple of 8 to allow correct structure alignment.

The following figure shows how the fields in STATICPOOL_Params define the layout of the buffer:

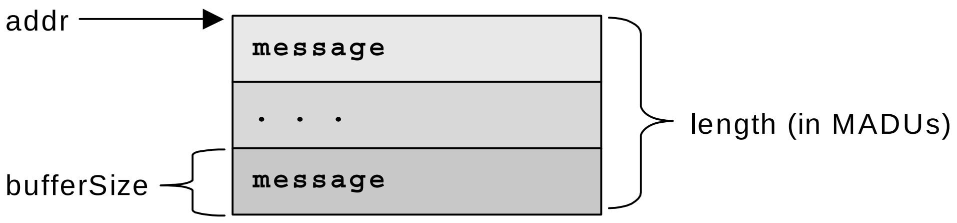


Figure 2-6. Buffer Layout as Defined by STATICPOOL_Params

Since the STATICPOOL buffer is generally used in static systems, the application must provide the memory for the STATICPOOL_Obj. So the object field of the POOL_Obj must be set to STATICPOOL_Obj instead of NULL.

The following is an example of an application that has two allocators (two instances of the STATICPOOL implementation).

```
#define NUMMSGS  8  /* Number of msgs per allocator */

/* Size of messages in the two allocators. Must be a
 * multiple of 8 as required by static allocator. */
#define MSGSIZE0       64
#define MSGSIZE1       128

enum {  /* Allocator ID and number of allocators */
    MQASTATICID0 = 0,
    MQASTATICID1,
    NUMALLOCATORS
};

static Char staticBuf0[MSGSIZE0 * NUMMSGS];
static Char staticBuf1[MSGSIZE1 * NUMMSGS];

static MQASTATIC_Params poolParams0 = {staticBuf0,
         sizeof(staticBuf0), MSGSIZE0};
static MQASTATIC_Params poolParams1 = {staticBuf1,
         sizeof(staticBuf1), MSGSIZE1};

static STATICPOOL_Obj poolObj0, poolObj1;

static POOL_Obj allocators[NUMALLOCATORS] =
    {{STATICPOOL_init, (POOL_Fxns *)&STATICPOOL_FXNS,
        &poolParams0, &poolObj0}
    {{STATICPOOL_init, (POOL_Fxns *)&STATICPOOL_FXNS,
        &poolParams1, &poolObj1}};

POOL_Config  POOL_config =
        {allocators, NUMALLOCATORS};
```

POOL Manager Properties

To configure the POOL manager, the POOL_Config structure must be defined in the application code. See “Static Configuration” on page 263.

The following global property must also be set in order to use the POOL module:

- **Enable POOL Manager**. If ENABLEPOOL is TRUE, each allocator specified in the POOL_config structure (see “Static Configuration” on page 263) is initialized and opened.

  Tconf Name: ENABLEPOOL Type: Bool

  Example: `bios.POOL.ENABLEPOOL = true;`

2.19 PRD Module

The PRD module is the periodic function manager.

Functions

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

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the PRD Manager Properties and PRD Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")
USECLK	Bool	true
MICROSECONDS	Int16	1000.0

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
period	Int16	32767
mode	EnumString	"continuous" ("one-shot")
fxn	Extern	prog.extern("FXN_F_nop")
arg0	Arg	0
arg1	Arg	0
order	Int16	0

Description

While some applications can schedule functions based on a real-time clock, many applications need to schedule functions based on I/O availability or some other programmatic event.

The PRD module allows you to create PRD objects that schedule periodic execution of program functions. The period can be driven by the CLK module or by calls to PRD_tick whenever a specific event occurs. There can be several PRD objects, but all are driven by the same period counter. Each PRD object can execute its functions at different intervals based on the period counter.

- **To schedule functions based on a real-time clock**. Set the clock interrupt rate you want to use in the CLK Object Properties. Set the "Use On-chip Clock (CLK)" property of the PRD Manager Properties to true. Set the frequency of execution (in number of clock interrupt ticks) in the period property for the individual period object.

- **To schedule functions based on I/O availability or some other event**. Set the "Use On-chip Clock (CLK)" property of the PRD Manager Properties to false. Set the frequency of execution (in number of ticks) in the period property for the individual period object. Your program should call PRD_tick to increment the tick counter.

The function executed by a PRD object is statically defined in the configuration. PRD functions are called from the context of the function run by the PRD_swi SWI object. PRD functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

The PRD module uses a SWI object (called PRD_swi by default) which itself is triggered on a periodic basis to manage execution of period objects. Normally, this SWI object should have the highest SWI priority to allow this SWI to be performed once per tick. This SWI is automatically created (or deleted) by the configuration if one or more (or no) PRD objects exist. The total time required to perform all PRD functions must be less than the number of microseconds between ticks. Any more lengthy processing should be scheduled as a separate SWI, TSK, or IDL thread.

See the *Code Composer Studio* online tutorial for an example that demonstrates the interaction between the PRD module and the SWI module.

When the PRD_swi object runs its function, the following actions occur:

```
for ("Loop through period objects") {
   if ("time for a periodic function")
       "run that periodic function";
}
```

PRD Manager Properties

The DSP/BIOS Periodic Function Manager allows the creation of an arbitrary number of objects that encapsulate a function, two arguments, and a period specifying the time between successive invocations of the function. The period is expressed in ticks, and a tick is defined as a single invocation of the PRD_tick operation. The time between successive invocations of PRD_tick defines the period represented by a tick.

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

- **Object Memory**. The memory segment containing the PRD objects.

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Use CLK Manager to drive PRD**. If this property is set to true, the on-device timer hardware (managed by the CLK Module) is used to advance the tick count; otherwise, the application must invoke PRD_tick on a periodic basis. If the CLK module is used to drive PRDs, the ticks are equal to the low-resolution time increment rate.

  Tconf Name: USECLK Type: Bool

  Example: `bios.PRD.USECLK = true;`

- **Microseconds/Tick**. The number of microseconds between ticks. If the "Use CLK Manager to drive PRD field" property above is set to true, this property is automatically set by the CLK module; otherwise, you must explicitly set this property. The total time required to perform all PRD functions must be less than the number of microseconds between ticks.

  Tconf Name: MICROSECONDS Type: Int16

  Example: `bios.PRD.MICROSECONDS = 1000.0;`

PRD Object Properties

To create a PRD object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myPrd = bios.PRD.create("myPrd");
```

If you cannot create a new PRD object (an error occurs or the Insert PRD item is inactive in the DSP/BIOS Configuration Tool), increase the Stack Size property in the MEM Manager Properties before adding a PRD object.

The following properties can be set for a PRD object in the PRD Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myPrd.comment = "my PRD";`

- **period (ticks)**. The function executes after this number of ticks have elapsed.

  Tconf Name: period Type: Int16

  Example: `myPrd.period = 32767;`

- **mode**. If "continuous" is used, the function executes every "period" number of ticks. If "one-shot" is used, the function executes just once after "period" ticks.

  Tconf Name: mode Type: EnumString

  Options: "continuous", "one-shot"

  Example: `myPrd.mode = "continuous";`

- **function**. The function to be executed. The total time required to perform all PRD functions must be less than the number of microseconds between ticks.

  Tconf Name: fxn Type: Extern

  Example: `myPrd.fxn = prog.extern("prdFxn");`

- **arg0, arg1**. Two Arg type arguments for the user-specified function above.

  Tconf Name: arg0 Type: Arg

  Tconf Name: arg1 Type: Arg

  Example: `myPrd.arg0 = 0;`

- **period (ms)**. The number of milliseconds represented by the period specified above. This is an informational property only.

  Tconf Name: N/A

- **order**. Set this property to all PRD objects so that the numbers match the sequence in which PRD functions should be executed.

  Tconf Name: order Type: Int16

  Example: `myPrd.order = 2;`

PRD_getticks *Get the current tick count*

C Interface

Syntax

num = PRD_getticks();

Parameters

Void

Return Value

LgUns	num	/* current tick counter */

Reentrant

yes

Description

PRD_getticks returns the current period tick count as a 32-bit value.

If the periodic functions are being driven by the on-device timer, the tick value is the number of low resolution clock ticks that have occurred since the program started running. When the number of ticks reaches the maximum value that can be stored in 32 bits, the value wraps back to 0. See the CLK Module, page 2–51, for more details.

If the periodic functions are being driven programmatically, the tick value is the number of times PRD_tick has been called.

Example

```
/* ======== showTicks ======== */
Void showTicks
{
      LOG_printf(&trace, "ticks = %d", PRD_getticks());
}
```

See Also

PRD_start
PRD_tick
CLK_gethtime
CLK_getltime
STS_delta

PRD_start

Arm a periodic function for one-shot execution

C Interface

Syntax

PRD_start(prd);

Parameters

PRD_Handle	prd;	/* prd object handle*/

Return Value

Void

Reentrant

no

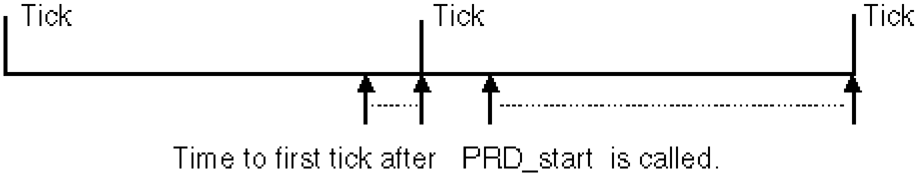
Description

PRD_start starts a period object that has its mode property set to one-shot in the configuration. Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified number of ticks have occurred after a call to PRD_start.

For example, you might have a function that should be executed a certain number of periodic ticks after some condition is met.

When you use PRD_start to start a period object, the exact time the function runs can vary by nearly one tick cycle. As Figure Figure 2-7 shows, PRD ticks occur at a fixed rate and the call to PRD_start can occur at any point between ticks

Figure 2-7. PRD Tick Cycles



If PRD_start is called again before the period for the object has elapsed, the object's tick count is reset. The PRD object does not run until its "period" number of ticks have elapsed.

Example

```
/* ======== startPRD ======== */
Void startPrd(Int periodID)
    {
       if ("condition met") {
          PRD_start(&periodID);
       }
    }
```

See Also

PRD_tick
PRD_getticks

PRD_stop

Stop a period object to prevent its function execution

C Interface

Syntax
PRD_stop(prd);

Parameters

PRD_Handle	prd;	/* prd object handle*/

Return Value
Void

Reentrant

no

Description

PRD_stop stops a period object to prevent its function execution. In most cases, PRD_stop is used to stop a period object that has its mode property set to one-shot in the configuration.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified numbers of ticks have occurred after a call to PRD_start.

PRD_stop is the way to stop those one-shot PRD objects once started and before their period counters have run out.

Example

```
PRD_stop(&prd);
```

See Also

PRD_getticks
PRD_start
PRD_tick

PRD_tick

Advance tick counter, enable periodic functions

C Interface

Syntax

PRD_tick();

Parameters

Void

Return Value

Void

Reentrant

no

Description

PRD_tick advances the period counter by one tick. Unless you are driving PRD functions using the on-device clock, PRD objects execute their functions at intervals based on this counter.

For example, an HWI could perform PRD_tick to notify a periodic function when data is available for processing.

Constraints and Calling Context

- All the registers that are modified by this API should be saved and restored, before and after the API is invoked, respectively.
- When called within an HWI, the code sequence calling PRD_tick must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.
- Interrupts need to be disabled before calling PRD_tick.

See Also

PRD_start
PRD_getticks

2.20 QUE Module

The QUE module is the atomic queue manager.

Functions

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

Constants, Types, and Structures

```
typedef struct QUE_Obj *QUE_Handle;  /* queue obj handle */
struct QUE_Attrs{     /* queue attributes */
   Int  dummy;     /* DUMMY */
};

QUE_Attrs QUE_ATTRS = {       /* default attribute values */
    0,
};

typedef QUE_Elem;        /* queue element */
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the QUE Manager Properties and QUE Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"

Description

The QUE module makes available a set of functions that manipulate queue objects accessed through handles of type QUE_Handle. Each queue contains an ordered sequence of zero or more elements referenced through variables of type QUE_Elem, which are generally embedded as the first field within a structure. The QUE_Elem item is used as an internal pointer.

For example, the DEV_Frame structure, which is used by the SIO Module and DEV Module to enqueue and dequeue I/O buffers, contains a field of type QUE_Elem:

```
struct DEV_Frame {    /* frame object */
   QUE_Elem   link;        /* must be first field! */
   Ptr        addr;        /* buffer address */
   size_t     size;        /* buffer size */
   Arg        misc;        /* reserved for driver */
   Arg        arg;         /* user argument */
   Uns        cmd;         /* mini-driver command */
   Int        status;      /* status of command */
} DEV_Frame;
```

Many QUE module functions either are passed or return a pointer to an element having the structure defined for QUE elements.

The functions QUE_put and QUE_get are atomic in that they manipulate the queue with interrupts disabled. These functions can therefore be used to safely share queues between tasks, or between tasks and SWIs or HWIs. All other QUE functions should only be called by tasks, or by tasks and SWIs or HWIs when they are used in conjunction with some mutual exclusion mechanism (for example, SEM_pend / SEM_post, TSK_disable / TSK_enable).

Once a queue has been created, use MEM_alloc to allocate elements for the queue.

QUE Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

QUE Object Properties

To create a QUE object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myQue = bios.QUE.create("myQue");
```

The following property can be set for a QUE object in the PRD Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myQue.comment = "my QUE";`

QUE_create *Create an empty queue*

C Interface

Syntax

```
queue = QUE_create(attrs);
```

Parameters

```
QUE_Attrs          *attrs;     /* pointer to queue attributes */
```

Return Value

```
QUE_Handle         queue;      /* handle for new queue object */
```

Description

QUE_create creates a new queue which is initially empty. If successful, QUE_create returns the handle of the new queue. If unsuccessful, QUE_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

If attrs is NULL, the new queue is assigned a default set of attributes. Otherwise, the queue's attributes are specified through a structure of type QUE_Attrs.

Note: At present, no attributes are supported for queue objects, and the type QUE_Attrs is defined as a dummy structure.

All default attribute values are contained in the constant QUE_ATTRS, which can be assigned to a variable of type QUE_Attrs prior to calling QUE_create.

You can also create a queue by declaring a variable of type QUE_Obj and initializing the queue with QUE_new.

QUE_create calls MEM_alloc to dynamically create the object's data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module, page 2–192.

Constraints and Calling Context

- QUE_create cannot be called from a SWI or HWI.
- You can reduce the size of your application program by creating objects with the Tconf rather than using the XXX_create functions.

See Also

MEM_alloc
QUE_empty
QUE_delete
SYS_error

QUE_delete *Delete an empty queue*

C Interface

Syntax

```
QUE_delete(queue);
```

Parameters

```
QUE_Handle              queue;           /* queue handle */
```

Return Value

Void

Description

QUE_delete uses MEM_free to free the queue object referenced by queue.

QUE_delete calls MEM_free to delete the QUE object. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Constraints and Calling Context

- queue must be empty.
- QUE_delete cannot be called from a SWI or HWI.
- No check is performed to prevent QUE_delete from being used on a statically-created object. If a program attempts to delete a queue object that was created using Tconf, SYS_error is called.

See Also

QUE_create
QUE_empty

QUE_dequeue *Remove from front of queue (non-atomically)*

C Interface

Syntax

```
elem = QUE_dequeue(queue);
```

Parameters

```
QUE_Handle          queue;      /* queue object handle */
```

Return Value

```
Ptr                 elem;       /* pointer to former first element */
```

Description

QUE_dequeue removes the element from the front of queue and returns elem.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Calling QUE_dequeue with an empty queue returns the queue itself. However, QUE_dequeue is non-atomic. Therefore, the method described for QUE_get of checking to see if a queue is empty and returning the first element otherwise is non-atomic.

Note: You should use QUE_get instead of QUE_dequeue if multiple threads share a queue. QUE_get runs atomically and is never interrupted; QUE_dequeue performs the same action but runs non-atomically. You can use QUE_dequeue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue. An HWI or task that preempts QUE_dequeue and operates on the same queue can corrupt the data structure.

QUE_dequeue is somewhat faster than QUE_get, but you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.

See Also

QUE_get

QUE_empty *Test for an empty queue*

C Interface

Syntax

```
empty = QUE_empty(queue);
```

Parameters

```
QUE_Handle          queue;      /* queue object handle */
```

Return Value

```
Bool                empty;      /* TRUE if queue is empty */
```

Description

QUE_empty returns TRUE if there are no elements in queue, and FALSE otherwise.

See Also

QUE_get

QUE_enqueue *Insert at end of queue (non-atomically)*

C Interface

Syntax

QUE_enqueue(queue, elem);

Parameters

QUE_Handle	queue;	/* queue object handle */
Ptr	elem;	/* pointer to queue element */

Return Value

Void

Description

QUE_enqueue inserts elem at the end of queue.

The elem parameter must be a pointer to an element to be placed in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Note: Use QUE_put instead of QUE_enqueue if multiple threads share a queue. QUE_put is never interrupted; QUE_enqueue performs the same action but runs non-atomically. You can use QUE_enqueue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE_enqueue is somewhat faster than QUE_put, but you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.

See Also

QUE_put

QUE_get *Get element from front of queue (atomically)*

C Interface

Syntax
elem = QUE_get(queue);

Parameters

QUE_Handle	queue;	/* queue object handle */

Return Value

Void	*elem;	/* pointer to former first element */

Description

QUE_get removes the element from the front of queue and returns elem.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Since QUE_get manipulates the queue with interrupts disabled, the queue can be shared by multiple tasks, or by tasks and SWIs or HWIs.

Calling QUE_get with an empty queue returns the queue itself. This provides a means for using a single atomic action to check if a queue is empty, and to remove and return the first element if it is not empty:

```
if ((QUE_Handle)(elem = QUE_get(q)) != q)
    ` process elem `
```

Note: Use QUE_get instead of QUE_dequeue if multiple threads share a queue. QUE_get is never interrupted; QUE_dequeue performs the same action but runs non-atomically. You can use QUE_dequeue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE_dequeue is somewhat faster than QUE_get, but you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.

See Also

QUE_create
QUE_empty
QUE_put

QUE_head

Return element at front of queue

C Interface

Syntax

```
elem = QUE_head(queue);
```

Parameters

```
QUE_Handle          queue;        /* queue object handle */
```

Return Value

```
QUE_Elem            *elem;        /* pointer to first element */
```

Description

QUE_head returns a pointer to the element at the front of queue. The element is not removed from the queue.

The return value, elem, is a pointer to the element at the front of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Calling QUE_head with an empty queue returns the queue itself.

See Also

QUE_create
QUE_empty
QUE_put

QUE_insert

Insert in middle of queue (non-atomically)

C Interface

Syntax

```
QUE_insert(qelem, elem);
```

Parameters

```
Ptr          qelem;     /* element already in queue */
Ptr          elem;      /* element to be inserted in queue */
```

Return Value

Void

Description

QUE_insert inserts elem in the queue in front of qelem.

The qelem parameter is a pointer to an existing element of the QUE. The elem parameter is a pointer to an element to be placed in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Note: If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_insert should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/SEM_post, TSK_disable/ TSK_enable).

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

Description

QUE_new adjusts a queue object to make the queue empty. This operation is not atomic. A typical use of QUE_new is to initialize a queue object that has been statically declared instead of being created with QUE_create. Note that if the queue is not empty, the element(s) in the queue are not freed or otherwise handled, but are simply abandoned.

If you created a queue by declaring a variable of type QUE_Obj, you can initialize the queue with QUE_new.

See Also

QUE_create
QUE_delete
QUE_empty

QUE_next *Return next element in queue (non-atomically)*

C Interface

Syntax

```
elem = QUE_next(qelem);
```

Parameters

```
Ptr              qelem;      /* element in queue */
```

Return Value

```
Ptr              elem;       /* next element in queue */
```

Description

QUE_next returns elem which points to the element in the queue after qelem.

The qelem parameter is a pointer to an existing element of the QUE. The return value, elem, is a pointer to the next element in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE_next to return a pointer to the queue itself. Be careful not to call QUE_remove(elem) in this case.

Note: If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_next should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/SEM_post, TSK_disable/ TSK_enable).

See Also

QUE_get
QUE_insert
QUE_prev
QUE_remove

QUE_prev

Return previous element in queue (non-atomically)

C Interface

Syntax

```
elem = QUE_prev(qelem);
```

Parameters

```
Ptr                           qelem;          /* element in queue */
```

Return Value

```
Ptr                           elem;           /* previous element in queue */
```

Description

QUE_prev returns elem which points to the element in the queue before qelem.

The qelem parameter is a pointer to an existing element of the QUE. The return value, elem, is a pointer to the previous element in the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE_prev to return a pointer to the queue itself. Be careful not to call QUE_remove(elem) in this case.

Note: If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_prev should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/SEM_post, TSK_disable/ TSK_enable).

See Also

QUE_head
QUE_insert
QUE_next
QUE_remove

QUE_put

Put element at end of queue (atomically)

C Interface

Syntax

```
QUE_put(queue, elem);
```

Parameters

```
QUE_Handle        queue;      /* queue object handle */
Void              *elem;      /* pointer to new queue element */
```

Return Value

Void

Description

QUE_put puts elem at the end of queue.

The elem parameter is a pointer to an element to be placed at the end of the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Since QUE_put manipulates queues with interrupts disabled, queues can be shared by multiple tasks, or by tasks and SWIs or HWIs.

Note: Use QUE_put instead of QUE_enqueue if multiple threads share a queue. QUE_put is never interrupted; QUE_enqueue performs the same action but runs non-atomically. You can use QUE_enqueue if you disable interrupts or use a synchronization mechanism such as LCK or SEM to protect the queue.

QUE_enqueue is somewhat faster than QUE_put, but you should not use it unless you know your QUE operation cannot be preempted by another thread that operates on the same queue.

See Also

QUE_get
QUE_head

QUE_remove *Remove from middle of queue (non-atomically)*

C Interface

Syntax

QUE_remove(qelem);

Parameters

Ptr	qelem;	/* element in queue */

Return Value

Void

Description

QUE_remove removes qelem from the queue.

The qelem parameter is a pointer to an existing element to be removed from the QUE. Such elements have a structure defined similarly to that in the example in the QUE Module topic. The first field in the structure must be of type QUE_Elem and is used as an internal pointer.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, be careful not to remove the header node. This can happen when qelem is the return value of QUE_next or QUE_prev. The following code sample shows how qelem should be verified before calling QUE_remove.

```
QUE_Elem *qelem;.

/* get pointer to first element in the queue */
qelem = QUE_head(queue);

/* scan entire queue for desired element */
while (qelem != queue) {
    if(‘ qelem is the elem we’re looking for ‘) {
         break;
    }
    qelem = QUE_next(qelem);
}
/* make sure qelem is not the queue itself */
if (qelem != queue) {
    QUE_remove(qelem);
}
```

Note: If the queue is shared by multiple tasks, or tasks and SWIs or HWIs, QUE_remove should be used in conjunction with some mutual exclusion mechanism (for example, SEM_pend/SEM_post, TSK_disable/ TSK_enable).

Constraints and Calling Context

QUE_remove should not be called when qelem is equal to the queue itself.

See Also

QUE_head
QUE_insert
QUE_next
QUE_prev

2.21 RTDX Module

The RTDX modules manage the real-time data exchange settings.

RTDX Data Declaration Macros

- RTDX_CreateInputChannel
- RTDX_CreateOutputChannel

Function Macros

- RTDX_disableInput
- RTDX_disableOutput
- RTDX_enableInput
- RTDX_enableOutput
- RTDX_read
- RTDX_readNB
- RTDX_sizeofInput
- RTDX_write

Channel Test Macros

- RTDX_channelBusy
- RTDX_isInputEnabled
- RTDX_isOutputEnabled

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the RTDX Manager Properties and RTDX Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
ENABLERTDX	Bool	false
MODE	EnumString	"JTAG" ("HSRTDX", "Simulator")
RTDXDATASEG	Reference	prog.get("L0SARAM")
BUFSIZE	Int16	258
INTERRUPTMASK	Int16	0x00000000

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
channelMode	EnumString	"output" ("input")

Description

The RTDX module provides the data types and functions for:

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

Data channels are represented by global structures. A data channel can be used for input or output, but not both. The contents of an input or output structure are not known to the user. A channel structure has two states: enabled and disabled. When a channel is enabled, any data written to the channel is sent to the host. Channels are initially disabled.

The RTDX assembly interface, *rtdx.i*, is a macro interface file that can be used to interface to RTDX at the assembly level.

RTDX Manager Properties

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

- **Enable Real-Time Data Exchange (RTDX)**. This property should be set to true if you want to link RTDX support into your application.

  Tconf Name: ENABLERTDX Type: Bool

  Example: `bios.RTDX.ENABLERTDX = true;`

- **RTDX Mode**. Select the port configuration mode RTDX should use to establish communication between the host and target. The default is JTAG for most targets. Set this to simulator if you use a simulator. The HS-RTDX emulation technology is also available. If this property is set incorrectly, a message says “RTDX target application does not match emulation protocol“ when you load the program.

  Tconf Name: MODE Type: EnumString

  Options: "JTAG", "HSRTDX", "Simulator"

  Example: `bios.RTDX.MODE = "JTAG";`

- **RTDX Data Segment (.rtdx_data)**. The memory segment used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this segment.

  Tconf Name: RTDXDATASEG Type: Reference

  Example: `bios.RTDX.RTDXDATASEG = prog.get("myMEM");`

- **RTDX Buffer Size (MADUs)**. The size of the RTDX target-to-host message buffer, in minimum addressable data units (MADUs). The default size is 1032 to accommodate a 1024-byte block and two control words. HST channels using RTDX are limited by this value.

  Tconf Name: BUFSIZE Type: Int16

  Example: `bios.RTDX.BUFSIZE = 258;`

- **RTDX Interrupt Mask**. This mask interrupts to be temporarily disabled inside critical RTDX sections. The default value of zero (0) disables all interrupts within critical RTDX sections. Such sections are short (usually <100 cycles). Disabling interrupts also temporarily disables other RTDX clients and prevents other RTDX function calls.

  You should allow all interrupts to be disabled inside critical RTDX sections if your application makes any RTDX calls from SWI or TSK threads. If your application does not make RTDX calls from SWI or TSK threads, you may modify bits in this mask to enable specific high-priority interrupts. See the RTDX documentation for details.

  Tconf Name: INTERRUPTMASK Type: Int16

  Example: `bios.RTDX.INTERRUPTMASK = 0x00000000;`

RTDX Object Properties

To create an RTDX object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myRtdx = bios.RTDX.create("myRtdx");
```

The following properties can be set for an RTDX object in the RTDX Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `myRtdx.comment = "my RTDX";`

- **Channel Mode**. Select output if the RTDX channel handles output from the DSP to the host. Select input if the RTDX channel handles input to the DSP from the host.

  Tconf Name: channelMode Type: EnumString

  Options: "input", "output"

  Example: `myRtdx.channelMode = "output";`

Note: Programs must be linked with C run-time libraries and contain the symbol _main.

RTDX_channelBusy *Return status indicating whether data channel is busy*

C Interface

Syntax

```
int RTDX_channelBusy( RTDX_inputChannel *pichan );
```

Parameters

pichan	/* Identifier for the input data channel */

Return Value

int	/* Status: 0 = Channel is not busy. */ /* non-zero = Channel is busy. */

Reentrant

yes

Description

RTDX_channelBusy is designed to be used in conjunction with RTDX_readNB. The return value indicates whether the specified data channel is currently in use or not. If a channel is busy reading, the test/control flag (TC) bit of status register 0 (STO) is set to 1. Otherwise, the TC bit is set to O.

Constraints and Calling Context

- RTDX_channelBusy cannot be called by an HWI function.

See Also

RTDX_readNB

RTDX_CreateInputChannel *Declare input channel structure*

C Interface

Syntax

```
RTDX_CreateInputChannel( ichan );
```

Parameters

ichan	/* Label for the input channel */

Return Value

none

Reentrant

no

Description

This macro declares and initializes to 0, the RTDX data channel for input.

Data channels must be declared as global objects. A data channel can be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.

A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.

Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer or its COM interface.

Constraints and Calling Context

- RTDX_CreateInputChannel cannot be called by an HWI function.

See Also

RTDX_CreateOutputChannel

RTDX_CreateOutputChannel *Declare output channel structure*

C Interface

Syntax

RTDX_CreateOutputChannel(ochan);

Parameters

ochan	/* Label for the output channel */

Return Value

none

Reentrant

no

Description

This macro declares and initializes the RTDX data channels for output.

Data channels must be declared as global objects. A data channel can be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.

A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.

Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer Studio or its OLE interface.

Constraints and Calling Context

- RTDX_CreateOutputChannel cannot be called by an HWI function.

See Also

RTDX_CreateInputChannel

RTDX_disableInput *Disable an input data channel*

C Interface

Syntax

```
void RTDX_disableInput( RTDX_inputChannel *ichan );
```

Parameters

```
ichan                                    /* Identifier for the input data channel */
```

Return Value

void

Reentrant

yes

Description

A call to a disable function causes the specified input channel to be disabled.

Constraints and Calling Context

- RTDX_disableInput cannot be called by an HWI function.

See Also

RTDX_disableOutput
RTDX_enableInput
RTDX_read

RTDX_disableOutput *Disable an output data channel*

C Interface

Syntax

```
void RTDX_disableOutput( RTDX_outputChannel *ochan );
```

Parameters

ochan	/* Identifier for an output data channel */

Return Value

void

Reentrant

yes

Description

A call to a disable function causes the specified data channel to be disabled.

Constraints and Calling Context

- RTDX_disableOutput cannot be called by an HWI function.

See Also

RTDX_disableInput
RTDX_enableOutput
RTDX_read

RTDX_enableInput *Enable an input data channel*

C Interface

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

Reentrant

yes

Description

A call to an enable function causes the specified data channel to be enabled.

Constraints and Calling Context

- RTDX_enableInput cannot be called by an HWI function.

See Also

RTDX_disableInput
RTDX_enableOutput
RTDX_read

RTDX_enableOutput *Enable an output data channel*

C Interface

Syntax

```
void RTDX_enableOutput( RTDX_outputChannel *ochan );
```

Parameters

```
ochan                    /* Identifier for an output data channel */
```

Return Value

void

Reentrant

yes

Description

A call to an enable function causes the specified data channel to be enabled.

Constraints and Calling Context

- RTDX_enableOutput cannot be called by an HWI function.

See Also

RTDX_disableOutput
RTDX_enableInput
RTDX_write

RTDX_isInputEnabled *Return status of the input data channel*

C Interface

Syntax

RTDX_isInputEnabled(ichan);

Parameter

ichan	/* Identifier for an input channel. */

Return Value

0	/* Not enabled. */
non-zero	/* Enabled. */

Reentrant

yes

Description

The RTDX_isInputEnabled macro tests to see if an input channel is enabled and sets the test/control flag (TC bit) of status register 0 to 1 if the input channel is enabled. Otherwise, it sets the TC bit to 0.

Constraints and Calling Context

- RTDX_isInputEnabled cannot be called by an HWI function.

See Also

RTDX_isOutputEnabled

RTDX_isOutputEnabled *Return status of the output data channel*

C Interface

Syntax

RTDX_isOutputEnabled(ohan);

Parameter

ochan	/* Identifier for an output channel. */

Return Value

0	/* Not enabled. */
non-zero	/* Enabled. */

Reentrant

yes

Description

The RTDX_isOutputEnabled macro tests to see if an output channel is enabled and sets the test/control flag (TC bit) of status register 0 to 1 if the output channel is enabled. Otherwise, it sets the TC bit to 0.

Constraints and Calling Context

- RTDX_isOutputEnabled cannot be called by an HWI function.

See Also

RTDX_isInputEnabled

RTDX_read *Read from an input channel*

C Interface

Syntax

```
int RTDX_read( RTDX_inputChannel *ichan, void *buffer, int bsize );
```

Parameters

```
ichan                   /* Identifier for the input data channel */
buffer                  /* A pointer to the buffer that receives the data */
bsize                   /* The size of the buffer in address units */
```

Return Value

```
> 0                     /* The number of address units of data */
                        /* actually supplied in buffer. */
0                       /* Failure. Cannot post read request */
                        /* because target buffer is full. */
RTDX_READ_ERROR         /* Failure. Channel currently busy or
                        not enabled. */
```

Reentrant

yes

Description

RTDX_read causes a read request to be posted to the specified input data channel. If the channel is enabled, RTDX_read waits until the data has arrived. On return from the function, the data has been copied into the specified buffer and the number of address units of data actually supplied is returned. The function returns RTDX_READ_ERROR immediately if the channel is currently busy reading or is not enabled.

When RTDX_read is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host Library to write data to the target buffer. When the data is received, the target application continues execution.

The specified data is to be written to the specified output data channel, provided that channel is enabled. On return from the function, the data has been copied out of the specified user buffer and into the RTDX target buffer. If the channel is not enabled, the write operation is suppressed. If the RTDX target buffer is full, failure is returned.

When RTDX_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data, but the target application does not wait. Execution of the target application continues immediately. Use RTDX_channelBusy and RTDX_sizeofInput to determine when the RTDX Host Library has written data to the target buffer.

Constraints and Calling Context

- RTDX_read cannot be called by an HWI function.

See Also

RTDX_channelBusy
RTDX_readNB

RTDX_readNB *Read from input channel without blocking*

C Interface

Syntax

```
int RTDX_readNB( RTDX_inputChannel *ichan, void *buffer, int bsize );
```

Parameters

ichan	/* Identifier for the input data channel */
buffer	/* A pointer to the buffer that receives the data */
bsize	/* The size of the buffer in address units */

Return Value

RTDX_OK	/* Success.*/
0 (zero)	/* Failure. The target buffer is full. */
RTDX_READ_ERROR	/*Channel is currently busy reading. */

Reentrant

yes

Description

RTDX_readNB is a nonblocking form of the function RTDX_read. RTDX_readNB issues a read request to be posted to the specified input data channel and immediately returns. If the channel is not enabled or the channel is currently busy reading, the function returns RTDX_READ_ERROR. The function returns 0 if it cannot post the read request due to lack of space in the RTDX target buffer.

When the function RTDX_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data but the target application does not wait. Execution of the target application continues immediately. Use the RTDX_channelBusy and RTDX_sizeofInput functions to determine when the RTDX Host Library has written data into the target buffer.

When RTDX_read is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host Library to write data into the target buffer. When the data is received, the target application continues execution.

Constraints and Calling Context

- RTDX_readNB cannot be called by an HWI function.

See Also

RTDX_channelBusy
RTDX_read
RTDX_sizeofInput

RTDX_sizeofInput *Return the number of MADUs read from a data channel*

C Interface

Syntax

```
int RTDX_sizeofInput( RTDX_inputChannel *pichan );
```

Parameters

```
pichan                                          /* Identifier for the input data channel */
```

Return Value

```
int                                             /* Number of sizeof units of data actually */
                                                /* supplied in buffer */
```

Reentrant

yes

Description

RTDX_sizeofInput is designed to be used in conjunction with RTDX_readNB after a read operation has completed. The function returns the number of sizeof units actually read from the specified data channel into the accumulator (register A).

Constraints and Calling Context

- RTDX_sizeofInput cannot be called by an HWI function.

See Also

RTDX_readNB

RTDX_write

Write to an output channel

C Interface

Syntax

```
int RTDX_write( RTDX_outputChannel *ochan, void *buffer, int bsize );
```

Parameters

ochan	/* Identifier for the output data channel */
buffer	/* A pointer to the buffer containing the data */
bsize	/* The size of the buffer in address units */

Return Value

int	/* Status: non-zero = Success. 0 = Failure. */

Reentrant

yes

Description

RTDX_write causes the specified data to be written to the specified output data channel, provided that channel is enabled. On return from the function, the data has been copied out of the specified user buffer and into the RTDX target buffer. If the channel is not enabled, the write operation is suppressed. If the RTDX target buffer is full, Failure is returned.

Constraints and Calling Context

- RTDX_write cannot be called by an HWI function.

See Also

RTDX_read

2.22 SEM Module

The SEM module is the semaphore manager.

Functions

- SEM_count. Get current semaphore count
- SEM_create. Create a semaphore
- SEM_delete. Delete a semaphore
- SEM_new. Initialize a semaphore
- SEM_pend. Wait for a counting semaphore
- SEM_pendBinary. Wait for a binary semaphore
- SEM_post. Signal a counting semaphore
- SEM_postBinary. Signal a binary semaphore
- SEM_reset. Reset semaphore

Constants, Types, and Structures

```
typedef struct SEM_Obj  *SEM_Handle;
                   /* handle for semaphore object */

struct SEM_Attrs { /* semaphore attributes */
   String  name;  /* printable name */
};

SEM_Attrs SEM_ATTRS = { /* default attribute values */
   "",            /* name */
};
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SEM Manager Properties and SEM Object Properties topics. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default
comment	String	"<add comments here>"
count	Int16	0

Description

The SEM module provides a set of functions that manipulate semaphore objects accessed through handles of type SEM_Handle. Semaphores can be used for task synchronization and mutual exclusion.

Semaphores can be counting semaphores or binary semaphores. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

- **Counting semaphores** keep track of the number of times the semaphore has been posted with SEM_post. This is useful, for example, if you have a group of resources that are shared between tasks. Such tasks might call SEM_pend to see if a resource is available before using one. SEM_pend and SEM_post are for use with counting semaphores.
- **Binary semaphores** can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to SEM_pendBinary clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not. SEM_pendBinary and SEM_postBinary are for use with binary semaphores.

The MBX module uses a counting semaphore internally to manage the count of free (or full) mailbox elements. Another example of a counting semaphore is an ISR that might fill multiple buffers of data for consumption by a task. After filling each buffer, the ISR puts the buffer on a queue and calls SEM_post. The task waiting for the data calls SEM_pend, which simply decrements the semaphore count and returns or blocks if the count is 0. The semaphore count thus tracks the number of full buffers available for the task. The GIO and SIO modules follow this model and use counting semaphores.

The internal data structures used for binary and counting semaphores are the same; the only change is whether semaphore values are incremented and decremented or simply set to zero and non-zero.

SEM_pend and SEM_pendBinary are used to wait for a semaphore. The timeout parameter allows the task to wait until a timeout, wait indefinitely, or not wait at all. The return value is used to indicate if the semaphore was signaled successfully.

SEM_post and SEM_postBinary are used to signal a semaphore. If a task is waiting for the semaphore, SEM_post/SEM_postBinary removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, SEM_post simply increments the semaphore count and returns. (SEM_postBinary sets the semaphore count to non-zero and returns.)

SEM Manager Properties

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

- **Object Memory**. The memory segment that contains the SEM objects created with Tconf.

  Tconf Name: OBJMEMSEG Type: Reference

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

SEM Object Properties

To create a SEM object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySem = bios.SEM.create("mySem");
```

The following properties can be set for a SEM object in the SEM Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `mySem.comment = "my SEM";`

- **Initial semaphore count**. Set this property to the desired initial semaphore count.

  Tconf Name: count Type: Int16

  Example: `mySem.count = 0;`

SEM_count *Get current semaphore count*

C Interface

Syntax

```
count = SEM_count(sem);
```

Parameters

```
SEM_Handle          sem;          /* semaphore handle */
```

Return Value

```
Int                 count;        /* current semaphore count */
```

Description

SEM_count returns the current value of the semaphore specified by sem.

SEM_create *Create a semaphore*

C Interface

Syntax

sem = SEM_create(count, attrs);

Parameters

Int	count;	/* initial semaphore count */
SEM_Attrs	*attrs;	/* pointer to semaphore attributes */

Return Value

SEM_Handle	sem;	/* handle for new semaphore object */

Description

SEM_create creates a new semaphore object which is initialized to count. If successful, SEM_create returns the handle of the new semaphore. If unsuccessful, SEM_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

If attrs is NULL, the new semaphore is assigned a default set of attributes. Otherwise, the semaphore's attributes are specified through a structure of type SEM_Attrs.

```
struct SEM_Attrs { /* semaphore attributes */
   String  name;  /* printable name */
};
```

Default attribute values are contained in the constant SEM_ATTRS, which can be assigned to a variable of type SEM_Attrs before calling SEM_create.

```
SEM_Attrs SEM_ATTRS = { /* default attribute values */
   "",            /* name */
};
```

SEM_create calls MEM_alloc to dynamically create the object's data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module.

Constraints and Calling Context

- count must be greater than or equal to 0.
- SEM_create cannot be called from a SWI or HWI.
- You can reduce the size of your application by creating objects with Tconf rather than XXX_create functions.

See Also

MEM_alloc
SEM_delete

SEM_delete *Delete a semaphore*

C Interface

Syntax

```
SEM_delete(sem);
```

Parameters

```
SEM_Handle              sem;          /* semaphore object handle */
```

Return Value

Void

Description

SEM_delete uses MEM_free to free the semaphore object referenced by sem.

SEM_delete calls MEM_free to delete the SEM object. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Constraints and Calling Context

- No tasks should be pending on sem when SEM_delete is called.
- SEM_delete cannot be called from a SWI or HWI.
- No check is performed to prevent SEM_delete from being used on a statically-created object. If a program attempts to delete a semaphore object that was created using Tconf, SYS_error is called.

See Also

SEM_create

SEM_new *Initialize semaphore object*

C Interface

Syntax

```
Void SEM_new(sem, count);
```

Parameters

```
SEM_Handle        sem;      /* pointer to semaphore object */
Int               count;    /* initial semaphore count */
```

Return Value

Void

Description

SEM_new initializes the semaphore object pointed to by sem with count. The function should be used on a statically created semaphore for initialization purposes only. No task switch occurs when calling SEM_new.

Constraints and Calling Context

- count must be greater than or equal to 0
- no tasks should be pending on the semaphore when SEM_new is called

See Also

QUE_new

SEM_pend *Wait for a semaphore*

C Interface

Syntax

status = SEM_pend(sem, timeout);

Parameters

SEM_Handle	sem;	/* semaphore object handle */
Uns	timeout;	/* return after this many system clock ticks */

Return Value

Bool	status;	/* TRUE if successful, FALSE if timeout */

Description

SEM_pend and SEM_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. In contrast, SEM_pendBinary and SEM_postBinary are for use with binary semaphores, which can have only an available or unavailable state. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

If the semaphore count is greater than zero (available), SEM_pend decrements the count and returns TRUE. If the semaphore count is zero (unavailable), SEM_pend suspends execution of the current task until SEM_post is called or the timeout expires.

If timeout is SYS_FOREVER, a task stays suspended until SEM_post is called on this semaphore. If timeout is 0, SEM_pend returns immediately. If timeout expires (or timeout is 0) before the semaphore is available, SEM_pend returns FALSE. Otherwise SEM_pend returns TRUE.

If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

A task switch occurs when calling SEM_pend if the semaphore count is 0 and timeout is not zero.

Constraints and Calling Context

- SEM_pend can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- SEM_pend cannot be called from the program's main() function.
- If you need to call SEM_pend within a TSK_disable/TSK_enable block, you must use a timeout of 0.
- SEM_pend should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.

See Also

SEM_pendBinary
SEM_post

SEM_pendBinary *Wait for a binary semaphore*

C Interface

Syntax

```
status = SEM_pendBinary(sem, timeout);
```

Parameters

SEM_Handle	sem;	/* semaphore object handle */
Uns	timeout;	/* return after this many system clock ticks */

Return Value

Bool	status;	/* TRUE if successful, FALSE if timeout */

Description

SEM_pendBinary and SEM_postBinary are for use with binary semaphores. These are semaphores that can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to SEM_pendBinary clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not.

In contrast, SEM_pend and SEM_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

If the semaphore count is non-zero (available), SEM_pendBinary sets the count to zero (unavailable) and returns TRUE.

If the semaphore count is zero (unavailable), SEM_pendBinary suspends execution of this task until SEM_post is called or the timeout expires.

If timeout is SYS_FOREVER, a task remains suspended until SEM_postBinary is called on this semaphore. If timeout is 0, SEM_pendBinary returns immediately.

If timeout expires (or timeout is 0) before the semaphore is available, SEM_pendBinary returns FALSE. Otherwise SEM_pendBinary returns TRUE.

If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

A task switch occurs when calling SEM_pendBinary if the semaphore count is 0 and timeout is not zero.

Constraints and Calling Context

- This API can be called from a TSK with any timeout value, but if called from an HWI or SWI the timeout must be 0.
- This API cannot be called from the program's main() function.
- If you need to call this API within a TSK_disable/TSK_enable block, you must use a timeout of 0.
- This API should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.

See Also

SEM_pend
SEM_postBinary

SEM_post *Signal a semaphore*

C Interface

Syntax

SEM_post(sem);

Parameters

SEM_Handle	sem;	/* semaphore object handle */

Return Value

Void

Description

SEM_pend and SEM_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks.

In contrast, SEM_pendBinary and SEM_postBinary are for use with binary semaphores, which can have only an available or unavailable state. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

SEM_post readies the first task waiting for the semaphore. If no task is waiting, SEM_post simply increments the semaphore count and returns.

A task switch occurs when calling SEM_post if a higher priority task is made ready to run.

Constraints and Calling Context

- When called within an HWI, the code sequence calling SEM_post must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.
- If SEM_post is called from within a TSK_disable/TSK_enable block, the semaphore operation is not processed until TSK_enable is called.

See Also

SEM_pend
SEM_postBinary

SEM_postBinary *Signal a binary semaphore*

C Interface

Syntax

```
SEM_postBinary(sem);
```

Parameters

```
SEM_Handle          sem;          /* semaphore object handle */
```

Return Value

Void

Description

SEM_pendBinary and SEM_postBinary are for use with binary semaphores. These are semaphores that can have only two states: available and unavailable. They can be used to share a single resource between tasks. They can also be used for a basic signaling mechanism, where the semaphore can be posted multiple times and a subsequent call to SEM_pendBinary clears the count and returns. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not.

In contrast, SEM_pend and SEM_post are for use with counting semaphores, which keep track of the number of times the semaphore has been posted. This is useful, for example, if you have a group of resources that are shared between tasks. The APIs for binary and counting semaphores cannot be mixed for a single semaphore.

SEM_postBinary readies the first task in the list if one or more tasks are waiting. SEM_postBinary sets the semaphore count to non-zero (available) if no tasks are waiting.

A task switch occurs when calling SEM_postBinary if a higher priority task is made ready to run.

Constraints and Calling Context

- When called within an HWI, the code sequence calling this API must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.
- If this API is called from within a TSK_disable/TSK_enable block, the semaphore operation is not processed until TSK_enable is called.

See Also

SEM_post
SEM_pendBinary

SEM_reset *Reset semaphore count*

C Interface

Syntax

```
SEM_reset(sem, count);
```

Parameters

SEM_Handle	sem;	/* semaphore object handle */
Int	count;	/* semaphore count */

Return Value

Void

Description

SEM_reset resets the semaphore count to count.

No task switch occurs when calling SEM_reset.

Constraints and Calling Context

- count must be greater than or equal to 0.
- No tasks should be waiting on the semaphore when SEM_reset is called.
- SEM_reset cannot be called by an HWI or a SWI.

See Also

SEM_create

2.23 SIO Module

The SIO module is the stream input and output manager.

Functions

- SIO_bufsize. Size of the buffers used by a stream
- SIO_create. Create stream
- SIO_ctrl. Perform a device-dependent control operation
- SIO_delete. Delete stream
- SIO_flush. Idle a stream by flushing buffers
- SIO_get. Get buffer from stream
- SIO_idle. Idle a stream
- SIO_issue. Send a buffer to a stream
- SIO_put. Put buffer to a stream
- SIO_ready. Determine if device is ready
- SIO_reclaim. Request a buffer back from a stream
- SIO_reclaimx. Request a buffer and frame status back from a stream
- SIO_segid. Memory segment used by a stream
- SIO_select. Select a ready device
- SIO_staticbuf. Acquire static buffer from stream

Constants, Types, and Structures

```
#define SIO_STANDARD     0 /* open stream for */
                           /* standard streaming model */
#define SIO_ISSUERECLAIM 1 /* open stream for */
                         /* issue/reclaim streaming model */

#define SIO_INPUT       0  /* open for input */
#define SIO_OUTPUT      1  /* open for output */

typedef SIO_Handle;        /* stream object handle */

typedef DEV_Callback SIO_Callback;

struct SIO_Attrs { /* stream attributes */
   Int    nbufs;     /* number of buffers */
   Int    segid;     /* buffer segment ID */
   size_t align;     /* buffer alignment */
   Bool   flush;  /* TRUE->don't block in DEV_idle*/
   Uns    model;  /* SIO_STANDARD,SIO_ISSUERECLAIM*/
   Uns    timeout;   /* passed to DEV_reclaim */
   SIO_Callback *callback;
                   /* initializes callback in DEV_Obj */
} SIO_Attrs;
```

```
SIO_Attrs SIO_ATTRS = {
    2,                      /* nbufs */
    0,                      /* segid */
    0,                      /* align */
    FALSE,                  /* flush */
    SIO_STANDARD,           /* model */
    SYS_FOREVER             /* timeout */
    NULL                    /* callback */
};
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SIO Manager Properties and SIO Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")
USEISSUERECLAIM	Bool	false

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
deviceName	Reference	prog.get("dev-name")
controlParameter	String	""
mode	EnumString	"input" ("output")
bufSize	Int16	0x80
numBufs	Int16	2
bufSegId	Reference	prog.get("SIO.OBJMEMSEG")
bufAlign	EnumInt	1 (2, 4, 8, 16, 32, 64, ..., 32768)
flush	Bool	false
modelName	EnumString	"Standard" ("Issue/Reclaim")
allocStaticBuf	Bool	false
timeout	Int16	-1
useCallBackFxn	Bool	false
callBackFxn	Extern	prog.extern("FXN_F_nop")
arg0	Arg	0
arg1	Arg	0

Description

The stream manager provides efficient real-time device-independent I/O through a set of functions that manipulate stream objects accessed through handles of type SIO_Handle. The device independence is afforded by having a common high-level abstraction appropriate for real-time applications, continuous streams of data, that can be associated with a variety of devices. All I/O programming is done in a high-level manner using these stream handles to the devices and the stream manager takes care of dispatching into the underlying device drivers.

For efficiency, streams are treated as sequences of fixed-size buffers of data rather than just sequences of MADUs.

Streams can be opened and closed during program execution using the functions SIO_create and SIO_delete, respectively.

The SIO_issue and SIO_reclaim function calls are enhancements to the basic DSP/BIOS device model. These functions provide a second usage model for streaming, referred to as the issue/reclaim model. It is a more flexible streaming model that allows clients to supply their own buffers to a stream, and to get them back in the order that they were submitted. The SIO_issue and SIO_reclaim functions also provide a user argument that can be used for passing information between the stream client and the stream devices.

Both SWI and TSK threads can be used with the SIO module. However, SWI threads can be used only with the issue/reclaim model, and only then if the timeout parameter is 0. TSK threads can be used with either model.

SIO Manager Properties

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

- **Object Memory**. The memory segment that contains the SIO objects created with Tconf.

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Use Only Issue/Reclaim Model**. Enable this option if you want the SIO module to use only the issue/reclaim model. If this option is false (the default) you can also use the standard model.

  Tconf Name: USEISSUERECLAIM Type: Bool

  Example: `bios.SIO.USEISSUERECLAIM = false;`

SIO Object Properties

To create an SIO object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySio = bios.SIO.create("mySio");
```

The following properties can be set for an SIO object in the SIO Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `mySio.comment = "my SIO";`

- **Device**. Select the device to which you want to bind this SIO object. User-defined devices are listed along with DGN and DPI devices.

  Tconf Name: deviceName Type: Reference

  Example: `mySio.deviceName = prog.get("UDEV0");`

- **Device Control String**. Type the device suffix to be passed to any devices stacked below the device connected to this stream.

  Tconf Name: controlParameter Type: String

  Example: `mySio.controlParameter = "/split4/codec";`

- **Mode**. Select input if this stream is to be used for input to the application program and output if this stream is to be used for output.

  Tconf Name: mode Type: EnumString

  Options: "input", "output"

  Example: `mySio.mode = "input";`

- **Buffer size**. If this stream uses the Standard model, this property controls the size of buffers (in MADUs) allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle buffers of any size.

  Tconf Name: bufSize Type: Int16

  Example: `mySio.bufSize = 0x80;`

- **Number of buffers**. If this stream uses the Standard model, this property controls the number of buffers allocated for use by the stream. If this stream uses the Issue/Reclaim model, the stream can handle up to the specified Number of buffers.

  Tconf Name: numBufs Type: Int16

  Example: `mySio.numBufs = 2;`

- **Place buffers in memory segment**. Select the memory segment to contain the stream buffers if Model is Standard.

  Tconf Name: bufSegId Type: Reference

  Example: `mySio.bufSegId = prog.get("myMEM");`

- **Buffer alignment**. Specify the memory alignment to use for stream buffers if Model is Standard. For example, if you select 16, the buffer must begin at an address that is a multiple of 16. The default is 1, which means the buffer can begin at any address.

  Tconf Name: bufAlign Type: EnumInt

  Options: 1, 2, 4, 8, 16, 32, 64, ..., 32768

  Example: `mySio.bufAlign = 1;`

- **Flush**. Check this box if you want the stream to discard all pending data and return without blocking if this object is idled at run-time with SIO_idle.

  Tconf Name: flush Type: Bool

  Example: `mySio.flush = false;`

- **Model**. Select Standard if you want all buffers to be allocated when the stream is created. Select Issue/Reclaim if your program is to allocate the buffers and supply them using SIO_issue. Both SWI and TSK threads can be used with the SIO module. However, SWI threads can be used only with the issue/reclaim model, and only then if the timeout parameter is 0. TSK threads can be used with either model.

  Tconf Name: modelName Type: EnumString

  Options: "Standard", "Issue/Reclaim"

  Example: `mySio.modelName = "Standard";`

- **Allocate Static Buffer(s)**. If this property is set to true, the configuration allocates stream buffers for the user. The SIO_staticbuf function is used to acquire these buffers from the stream. When the Standard model is used, checking this box causes one buffer more than the Number of buffers property to be allocated. When the Issue/Reclaim model is used, buffers are not normally allocated. Checking this box causes the number of buffers specified by the Number of buffers property to be allocated.

  Tconf Name: allocStaticBuf Type: Bool

  Example: `mySio.allocStaticBuf = false;`

- **Timeout for I/O operation**. This parameter specifies the length of time the I/O operations SIO_get, SIO_put, and SIO_reclaim wait for I/O. The device driver's Dxx_reclaim function typically uses this timeout while waiting for I/O. If the timeout expires before a buffer is available, the I/O operation returns (-1 * SYS_ETIMEOUT) and no buffer is returned.

  Tconf Name: timeout Type: Int16

  Example: `mySio.timeout = -1;`

- **use callback function**. Check this box if you want to use this SIO object with a callback function. In most cases, the callback function is SWI_andnHook or a similar function that posts a SWI. Checking this box allows the SIO object to be used with SWI threads.

  Tconf Name: useCallBackFxn Type: Bool

  Example: `mySio.useCallBackFxn = false;`

- **callback function**. A function for the SIO object to call. In most cases, the callback function is SWI_andnHook or a similar function that posts a SWI. This function gets called by the class driver (see the DIO Adapter) in the class driver's callback function. This callback function in the class driver usually gets called in the mini-driver code as a result of the HWI.

  Tconf Name: callBackFxn Type: Extern

  Example: `mySio.callBackFxn = prog.extern("SWI_andnHook");`

- **argument 0**. The first argument to pass to the callback function. If the callback function is SWI_andnHook, this argument should be a SWI object handle.

  Tconf Name: arg0 Type: Arg

  Example: `mySio.arg0 = prog.get("mySwi");`

- **argument 1**. The second argument to pass to the callback function. If the callback function is SWI_andnHook, this argument should be a value mask.

  Tconf Name: arg1 Type: Arg

  Example: `mySio.arg1 = 2;`

SIO_bufsize *Return the size of the buffers used by a stream*

C Interface

Syntax

size = SIO_bufsize(stream);

Parameters

SIO_Handle	stream;

Return Value

size_t	size;

Description

SIO_bufsize returns the size of the buffers used by stream.

This API can be used only if the model is SIO_STANDARD.

See Also

SIO_segid

SIO_create *Open a stream*

C Interface

Syntax

stream = SIO_create(name, mode, bufsize, attrs);

Parameters

String	name;	/* name of device */
Int	mode;	/* SIO_INPUT or SIO_OUTPUT */
size_t	bufsize;	/* stream buffer size */
SIO_Attrs	*attrs;	/* pointer to stream attributes */

Return Value

SIO_Handle	stream;	/* stream object handle */

Description

SIO_create creates a new stream object and opens the device specified by name. If successful, SIO_create returns the handle of the new stream object. If unsuccessful, SIO_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

Internally, SIO_create calls Dxx_open to open a device.

The mode parameter specifies whether the stream is to be used for input (SIO_INPUT) or output (SIO_OUTPUT).

If the stream is being opened in SIO_STANDARD mode, SIO_create allocates buffers of size bufsize for use by the stream. Initially these buffers are placed on the device todevice queue for input streams, and the device fromdevice queue for output streams.

If the stream is being opened in SIO_ISSUERECLAIM mode, SIO_create does not allocate any buffers for the stream. In SIO_ISSUERECLAIM mode all buffers must be supplied by the client via the SIO_issue call. It does, however, prepare the stream for a maximum number of buffers of the specified size.

If the attrs parameter is NULL, the new stream is assigned the default set of attributes specified by SIO_ATTRS. The following stream attributes are currently supported:

```
struct SIO_Attrs { /* stream attributes */
    Int    nbufs;     /* number of buffers */
    Int    segid;     /* buffer segment ID */
    size_t align;     /* buffer alignment */
    Bool   flush; /* TRUE->don't block in DEV_idle */
    Uns    model; /* SIO_STANDARD,SIO_ISSUERECLAIM */
    Uns    timeout;  /* passed to DEV_reclaim */
    SIO_Callback  *callback;
                /* initialize callback in DEV_Obj */
} SIO_Attrs;
```

- **nbufs.** Specifies the number of buffers allocated by the stream in the SIO_STANDARD usage model, or the number of buffers to prepare for in the SIO_ISSUERECLAIM usage model. The default value of nbufs is 2. In the SIO_ISSUERECLAIM usage model, nbufs is the maximum number of buffers that can be outstanding (that is, issued but not reclaimed) at any point in time.
- **segid.** Specifies the memory segment for stream buffers. Use the memory segment names defined in the configuration. The default value is 0, meaning that buffers are to be allocated from the "Segment for DSP/BIOS objects" property in the MEM Manager Properties.

- **align.** Specifies the memory alignment for stream buffers. The default value is 0, meaning that no alignment is needed.
- **flush.** Indicates the desired behavior for an output stream when it is deleted. If flush is TRUE, a call to SIO_delete causes the stream to discard all pending data and return without blocking. If flush is FALSE, a call to SIO_delete causes the stream to block until all pending data has been processed. The default value is FALSE.
- **model.** Indicates the usage model that is to be used with this stream. The two usage models are SIO_ISSUERECLAIM and SIO_STANDARD. The default usage model is SIO_STANDARD.
- **timeout.** Specifies the length of time the device driver waits for I/O completion before returning an error (for example, SYS_ETIMEOUT). timeout is usually passed as a parameter to SEM_pend by the device driver. The default is SYS_FOREVER which indicates that the driver waits forever. If timeout is SYS_FOREVER, the task remains suspended until a buffer is available to be returned by the stream. The timeout attribute applies to the I/O operations SIO_get, SIO_put, and SIO_reclaim. If timeout is 0, the I/O operation returns immediately. If the timeout expires before a buffer is available to be returned, the I/O operation returns the value of (-1 * SYS_ETIMEOUT). Otherwise the I/O operation returns the number of valid MADUs in the buffer, or -1 multiplied by an error code.
- **callback.** Specifies a pointer to channel-specific callback information. The SIO_Callback structure is defined by the SIO module to match the DEV_Callback structure. This structure contains the callback function and two function arguments. The callback function is typically SWI_andnHook or a similar function that posts a SWI. Callbacks can only be used with the SIO_ISSUERECLAIM model.

  Existing DEV drivers do not use this callback function. While DEV drivers can be modified to use this callback, it is not recommended. Instead, the IOM device driver model is recommended for drivers that need the SIO callback feature. IOM drivers use the DIO module to interface with the SIO functions.

SIO_create calls MEM_alloc to dynamically create the object's data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is set by the "Segment for DSP/BIOS objects" property in the MEM Manager Properties.

Constraints and Calling Context

- A stream can only be used by one task simultaneously. Catastrophic failure can result if more than one task calls SIO_get (or SIO_issue/ SIO_reclaim) on the same input stream, or more than one task calls SIO_put (or SIO_issue / SIO_reclaim) on the same output stream.
- SIO_create creates a stream dynamically. Do not call SIO_create on a stream that was created with Tconf.
- You can reduce the size of your application program by creating objects with Tconf rather than using the XXX_create functions. However, streams that are to be used with stacking drivers must be created dynamically with SIO_create.
- SIO_create cannot be called from a SWI or HWI.

See Also

Dxx_open
MEM_alloc
SEM_pend
SIO_delete
SIO_issue
SIO_reclaim
SYS_error

SIO_ctrl

Perform a device-dependent control operation

C Interface

Syntax

```
status = SIO_ctrl(stream, cmd, arg);
```

Parameters

```
SIO_Handle          stream;      /* stream handle */
Uns                 cmd;         /* command to device */
Arg                 arg;         /* arbitrary argument */
```

Return Value

```
Int                 status;      /* device status */
```

Description

SIO_ctrl causes a control operation to be issued to the device associated with stream. cmd and arg are passed directly to the device.

SIO_ctrl returns SYS_OK if successful, and a non-zero device-dependent error value if unsuccessful.

Internally, SIO_ctrl calls Dxx_ctrl to send control commands to a device.

Constraints and Calling Context

- SIO_ctrl cannot be called from an HWI.

See Also

Dxx_ctrl

SIO_delete

Close a stream and free its buffers

C Interface

Syntax

```
status = SIO_delete(stream);
```

Parameters

```
SIO_Handle          stream;     /* stream object */
```

Return Value

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

Description

SIO_delete idles the device before freeing the stream object and buffers.

If the stream being deleted was opened for input, then any pending input data is discarded. If the stream being deleted was opened for output, the method for handling data is determined by the value of the flush field in the SIO_Attrs structure (passed in with SIO_create). If flush is TRUE, SIO_delete discards all pending data and returns without blocking. If flush is FALSE, SIO_delete blocks until all pending data has been processed by the stream.

SIO_delete returns SYS_OK if and only if the operation is successful.

SIO_delete calls MEM_free to delete a stream. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Internally, SIO_delete first calls Dxx_idle to idle the device. Then it calls Dxx_close.

Constraints and Calling Context

- SIO_delete cannot be called from a SWI or HWI.
- No check is performed to prevent SIO_delete from being used on a statically-created object. If a program attempts to delete a stream object that was created using Tconf, SYS_error is called.
- In SIO_ISSUERECLAIM mode, all buffers issued to a stream must be reclaimed before SIO_delete is called. Failing to reclaim such buffers causes a memory leak.

See Also

SIO_create
SIO_flush
SIO_idle
Dxx_idle
Dxx_close

SIO_flush *Flush a stream*

C Interface

Syntax

```
status = SIO_flush(stream);
```

Parameters

SIO_Handle	stream;	/* stream handle */

Return Value

Int	status;	/* result of operation */

Description

SIO_flush causes all pending data to be discarded regardless of the mode of the stream. SIO_flush differs from SIO_idle in that SIO_flush never suspends program execution to complete processing of data, even for a stream created in output mode.

The underlying device connected to stream is idled as a result of calling SIO_flush. In general, the interrupt is disabled for the device.

One of the purposes of this function is to provide synchronization with the external environment.

SIO_flush returns SYS_OK if and only if the stream is successfully idled.

Internally, SIO_flush calls Dxx_idle and flushes all pending data.

If a callback was specified in the SIO_Attrs structure used with SIO_create, then SIO_flush performs no processing and returns SYS_OK.

Constraints and Calling Context

- SIO_flush cannot be called from an HWI.
- If SIO_flush is called from a SWI, no action is performed.

See Also

Dxx_idle
SIO_create
SIO_idle

SIO_get *Get a buffer from stream*

C Interface

Syntax

nmadus = SIO_get(stream, bufp);

Parameters

SIO_Handle	stream	/* stream handle */
Ptr	*bufp;	/* pointer to a buffer */

Return Value

Int	nmadus;	/* number of MADUs read or error if negative */

Description

SIO_get exchanges an empty buffer with a non-empty buffer from stream. The bufp is an input/output parameter which points to an empty buffer when SIO_get is called. When SIO_get returns, bufp points to a new (different) buffer, and nmadus indicates success or failure of the call.

SIO_get blocks until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO_create). If a timeout occurs, the value (-1 * SYS_ETIMEOUT) is returned. If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO_get returns a positive value for nmadus. As a success indicator, nmadus is the number of MADUs received from the stream. To indicate failure, SIO_get returns a negative value for nmadus. As a failure indicator, nmadus is the actual error code multiplied by -1.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- **If the actual buffer size is less than/equal to the maximum positive Int value (15 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. Size_t is the same as unsigned long.

For other architectures, size_t is:

- ’C28x - unsigned long
- ’C54x/’C55x/’C6x - unsigned int

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO_get.

A task switch occurs when calling SIO_get if there are no non-empty data buffers in stream.

Internally, SIO_get calls Dxx_issue and Dxx_reclaim for the device.

Constraints and Calling Context

- The stream must not be created with attrs.model set to SIO_ISSUERECLAIM. The results of calling SIO_get on a stream created for the issue/reclaim streaming model are undefined.
- SIO_get cannot be called from a SWI or HWI.
- This API is callable from the program’s main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

See Also

Dxx_issue
Dxx_reclaim
SIO_put

SIO_idle *Idle a stream*

C Interface

Syntax

```
status = SIO_idle(stream);
```

Parameters

SIO_Handle	stream;	/* stream handle */

Return Value

Int	status;	/* result of operation */

Description

If stream is being used for output, SIO_idle causes any currently buffered data to be transferred to the output device associated with stream. SIO_idle suspends program execution for as long as is required for the data to be consumed by the underlying device.

If stream is being used for input, SIO_idle causes any currently buffered data to be discarded. The underlying device connected to stream is idled as a result of calling SIO_idle. In general, the interrupt is disabled for this device.

If discarding of unrendered output is desired, use SIO_flush instead.

One of the purposes of this function is to provide synchronization with the external environment.

SIO_idle returns SYS_OK if and only if the stream is successfully idled.

Internally, SIO_idle calls Dxx_idle to idle the device.

If a callback was specified in the SIO_Attrs structure used with SIO_create, then SIO_idle performs no processing and returns SYS_OK.

Constraints and Calling Context

- SIO_idle cannot be called from an HWI.
- If SIO_idle is called from a SWI, no action is performed.

See Also

Dxx_idle
SIO_create
SIO_flush

SIO_issue *Send a buffer to a stream*

C Interface

Syntax

```
status = SIO_issue(stream, pbuf, nmadus, arg);
```

Parameters

```
SIO_Handle         stream;      /* stream handle */
Ptr                pbuf;        /* pointer to a buffer */
size_t             nmadus;      /* number of MADUs in the buffer */
Arg                arg;         /* user argument */
```

Return Value

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

Description

SIO_issue is used to send a buffer and its related information to a stream. The buffer-related information consists of the logical length of the buffer (nmadus), and the user argument to be associated with that buffer. SIO_issue sends a buffer to the stream and return to the caller without blocking. It also returns an error code indicating success (SYS_OK) or failure of the call.

Internally, SIO_issue calls Dxx_issue after placing a new input frame on the driver's device->todevice queue.

Failure of SIO_issue indicates that the stream was not able to accept the buffer being issued or that there was a device error when the underlying Dxx_issue was called. In the first case, the application is probably issuing more frames than the maximum MADUs allowed for the stream, before it reclaims any frames. In the second case, the failure reveals an underlying device driver or hardware problem. If SIO_issue fails, SIO_idle should be called for an SIO_INPUT stream, and SIO_flush should be called for an SIO_OUTPUT stream, before attempting more I/O through the stream.

The interpretation of nmadus, the logical size of a buffer, is direction-dependent. For a stream opened in SIO_OUTPUT mode, the logical size of the buffer indicates the number of valid MADUs of data it contains. For a stream opened in SIO_INPUT mode, the logical length of a buffer indicates the number of MADUs being requested by the client. In either case, the logical size of the buffer must be less than or equal to the physical size of the buffer.

The argument arg is not interpreted by DSP/BIOS, but is offered as a service to the stream client. DSP/BIOS and all DSP/BIOS-compliant device drivers preserve the value of arg and maintain its association with the data that it was issued with. arg provides a user argument as a method for a client to associate additional information with a particular buffer of data.

SIO_issue is used in conjunction with SIO_reclaim to operate a stream opened in SIO_ISSUERECLAIM mode. The SIO_issue call sends a buffer to a stream, and SIO_reclaim retrieves a buffer from a stream. In normal operation each SIO_issue call is followed by an SIO_reclaim call. Short bursts of multiple SIO_issue calls can be made without an intervening SIO_reclaim call, but over the life of the stream SIO_issue and SIO_reclaim must be called the same number of times.

At any given point in the life of a stream, the number of SIO_issue calls can exceed the number of SIO_reclaim calls by a maximum of nbufs. The value of nbufs is determined by the SIO_create call or by setting the Number of buffers property for the object in the configuration.

Note: An SIO_reclaim call should not be made without at least one outstanding SIO_issue call. Calling SIO_reclaim with no outstanding SIO_issue calls has undefined results.

Constraints and Calling Context

- The stream must be created with attrs.model set to SIO_ISSUERECLAIM.
- SIO_issue cannot be called from an HWI.

See Also

Dxx_issue
SIO_create
SIO_reclaim

SIO_put

Put a buffer to a stream

C Interface

Syntax

nmadus = SIO_put(stream, bufp, nmadus);

Parameters

SIO_Handle	stream;	/* stream handle */
Ptr	*bufp;	/* pointer to a buffer */
size_t	nmadus;	/* number of MADUs in the buffer */

Return Value

Int	nmadus;	/* number of MADUs, negative if error */

Description

SIO_put exchanges a non-empty buffer with an empty buffer. The bufp parameter is an input/output parameter that points to a non-empty buffer when SIO_put is called. When SIO_put returns, bufp points to a new (different) buffer, and nmadus indicates success or failure of the call.

SIO_put blocks until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO_create). If a timeout occurs, the value (-1 * SYS_ETIMEOUT) is returned. If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO_put returns a positive value for nmadus. As a success indicator, nmadus is the number of valid MADUs in the buffer returned by the stream (usually zero). To indicate failure, SIO_put returns a negative value (the actual error code multiplied by -1).

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- **If the actual buffer size is less than/equal to the maximum positive Int value (15 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. Size_t is the same as unsigned long.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO_put.

A task switch occurs when calling SIO_put if there are no empty data buffers in the stream.

Internally, SIO_put calls Dxx_issue and Dxx_reclaim for the device.

Constraints and Calling Context

- The stream must not be created with attrs.model set to SIO_ISSUERECLAIM. The results of calling SIO_put on a stream created for the issue/reclaim model are undefined.
- SIO_put cannot be called from a SWI or HWI.

- This API is callable from the program’s main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

See Also

Dxx_issue
Dxx_reclaim
SIO_get

SIO_ready

Determine if device for stream is ready

C Interface

Syntax

```
status = SIO_ready(stream);
```

Parameters

```
SIO_Handle         stream;
```

Return Value

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

Description

SIO_ready returns TRUE if a stream is ready for input or output.

If you are using SIO objects with SWI threads, you may want to use SIO_ready to avoid calling SIO_reclaim when it may fail because no buffers are available.

SIO_ready is similar to SIO_select, except that it does not block. You can prevent SIO_select from blocking by setting the timeout to zero, however, SIO_ready is more efficient because SIO_select performs SEM_pend with a timeout of zero. SIO_ready simply polls the stream to see if the device is ready.

See Also

SIO_select

SIO_reclaim

Request a buffer back from a stream

C Interface

Syntax

```
nmadus = 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                 nmadus;      /* number of MADUs or error if negative */
```

Description

SIO_reclaim is used to request a buffer back from a stream. It returns a pointer to the buffer, the number of valid MADUs in the buffer, and a user argument (parg). After the SIO_reclaim call parg points to the same value that was passed in with this buffer using the SIO_issue call.

If you want to return a frame-specific status along with the buffer, use SIO_reclaimx instead of SIO_reclaim.

Internally, SIO_reclaim calls Dxx_reclaim, then it gets the frame from the driver's device->fromdevice queue.

If a stream was created in SIO_OUTPUT mode, then SIO_reclaim returns an empty buffer, and nmadus is zero, since the buffer is empty. If a stream was opened in SIO_INPUT mode, SIO_reclaim returns a non-empty buffer, and nmadus is the number of valid MADUs of data in the buffer.

If SIO_reclaim is called from a TSK thread, it blocks (in either mode) until a buffer can be returned to the caller, or until the stream's timeout attribute expires (see SIO_create), and it returns a positive number or zero (indicating success), or a negative number (indicating an error condition). If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If SIO_reclaim is called from a SWI thread, it returns an error if it is called when no buffer is available. SIO_reclaim never blocks when called from a SWI.

To indicate success, SIO_reclaim returns a positive value for nmadus. As a success indicator, nmadus is the number of valid MADUs in the buffer. To indicate failure, SIO_reclaim returns a negative value for nmadus. As a failure indicator, nmadus is the actual error code multiplied by -1.

Failure of SIO_reclaim indicates that no buffer was returned to the client. Therefore, if SIO_reclaim fails, the client should not attempt to de-reference pbufp, since it is not guaranteed to contain a valid buffer pointer.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size_t, APIs that return a buffer size return a type of Int. The inconsistency is due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- **If the actual buffer size is less than/equal to the maximum positive Int value (15 bits).** Check the return value for negative values, which should be treated as errors. Positive values reflect the correct size.
- **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. Size_t is the same as unsigned long.

SIO_reclaim is used in conjunction with SIO_issue to operate a stream opened in SIO_ISSUERECLAIM mode. The SIO_issue call sends a buffer to a stream, and SIO_reclaim retrieves a buffer from a stream. In normal operation each SIO_issue call is followed by an SIO_reclaim call. Short bursts of multiple SIO_issue calls can be made without an intervening SIO_reclaim call, but over the life of the stream SIO_issue and SIO_reclaim must be called the same number of times. The number of SIO_issue calls can exceed the number of SIO_reclaim calls by a maximum of nbufs at any given time. The value of nbufs is determined by the SIO_create call or by setting the Number of buffers property for the object in the configuration.

Note: An SIO_reclaim call should not be made without at least one outstanding SIO_issue call. Calling SIO_reclaim with no outstanding SIO_issue calls has undefined results.

SIO_reclaim only returns buffers that were passed in using SIO_issue. It also returns the buffers in the same order that they were issued.

A task switch occurs when calling SIO_reclaim if timeout is not set to 0, and there are no data buffers available to be returned.

Constraints and Calling Context

- The stream must be created with attrs.model set to SIO_ISSUERECLAIM.
- There must be at least one outstanding SIO_issue when an SIO_reclaim call is made.
- SIO_reclaim returns an error if it is called from a SWI when no buffer is available. SIO_reclaim does not block if called from a SWI.
- All frames issued to a stream must be reclaimed before closing the stream.
- SIO_reclaim cannot be called from a HWI.
- This API is callable from the program’s main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

See Also

Dxx_reclaim
SIO_issue
SIO_create
SIO_reclaimx

SIO_reclaimx *Request a buffer back from a stream, including frame status*

C Interface

Syntax

```
nmadus = SIO_reclaimx(stream, *pbufp, *parg, *pfstatus);
```

Parameters

```
SIO_Handle     stream;      /* stream handle */
Ptr            *pbufp;      /* pointer to the buffer */
Arg            *parg;       /* pointer to a user argument */
Int            *pfstatus;   /* pointer to frame status */
```

Return Value

```
Int            nmadus;      /* number of MADUs or error if negative */
```

Description

SIO_reclaimx is identical to SIO_reclaim, except that is also returns a frame-specific status in the Int pointed to by the pfstatus parameter.

The device driver can use the frame-specific status to pass frame-specific status information to the application. This allows the device driver to fill in the status for each frame, and gives the application access to that status.

The returned frame status is valid only if SIO_reclaimx() returns successfully. If the nmadus value returned is negative, the frame status should not be considered accurate.

Constraints and Calling Context

- The stream must be created with attrs.model set to SIO_ISSUERECLAIM.
- There must be at least one outstanding SIO_issue when an SIO_reclaimx call is made.
- SIO_reclaimx returns an error if it is called from a SWI when no buffer is available. SIO_reclaimx does not block if called from a SWI.
- All frames issued to a stream must be reclaimed before closing the stream.
- SIO_reclaimx cannot be called from a HWI.
- This API is callable from the program’s main() function only if the stream's configured timeout attribute is 0, or if it is certain that there is a buffer available to be returned.

See Also

SIO_reclaim

SIO_segid *Return the memory segment used by the stream*

C Interface

Syntax

```
segid = SIO_segid(stream);
```

Parameters

```
SIO_Handle          stream;
```

Return Value

```
Int                 segid;        /* memory segment ID */
```

Description

SIO_segid returns the identifier of the memory segment that stream uses for buffers.

See Also

SIO_bufsize

SIO_select *Select a ready device*

C Interface

Syntax

mask = SIO_select(streamtab, nstreams, timeout);

Parameters

SIO_Handle	streamtab;	/* stream table */
Int	nstreams;	/* number of streams */
Uns	timeout;	/* return after this many system clock ticks */

Return Value

Uns	mask;	/* stream ready mask */

Description

SIO_select waits until one or more of the streams in the streamtab[] array is ready for I/O (that is, it does not block when an I/O operation is attempted).

streamtab[] is an array of streams where nstreams < 16. The timeout parameter indicates the number of system clock ticks to wait before a stream becomes ready. If timeout is 0, SIO_select returns immediately. If timeout is SYS_FOREVER, SIO_select waits until one of the streams is ready. Otherwise, SIO_select waits for up to 1 system clock tick less than timeout due to granularity in system timekeeping.

The return value is a mask indicating which streams are ready for I/O. A 1 in bit position j indicates the stream streamtab[j] is ready.

SIO_select results in a context switch if no streams are ready for I/O.

Internally, SIO_select calls Dxx_ready to determine if the device is ready for an I/O operation.

SIO_ready is similar to SIO_select, except that it does not block. You can prevent SIO_select from blocking by setting the timeout to zero, however, SIO_ready is more efficient in this situation because SIO_select performs SEM_pend with a timeout of zero. SIO_ready simply polls the stream to see if the device is ready.

For the SIO_STANDARD model in SIO_INPUT mode only, if stream I/O has not been started (that is, if SIO_get has not been called), SIO_select calls Dxx_issue for all empty frames to start the device.

Constraints and Calling Context

- streamtab must contain handles of type SIO_Handle returned from prior calls to SIO_create.
- streamtab[] is an array of streams; streamtab[i] corresponds to bit position i in mask.
- SIO_select cannot be called from an HWI.
- SIO_select can only be called from a SWI if the timeout value is zero.

See Also

Dxx_ready
SIO_get
SIO_put
SIO_ready
SIO_reclaim

SIO_staticbuf *Acquire static buffer from stream*

C Interface

Syntax

```
nmadus = SIO_staticbuf(stream, bufp);
```

Parameters

```
SIO_Handle          stream;     /* stream handle */
Ptr                 *bufp;      /* pointer to a buffer */
```

Return Value

```
Int                 nmadus;     /* number of MADUs in buffer */
```

Description

SIO_staticbuf returns buffers for static streams that were configured statically. Buffers are allocated for static streams by checking the Allocate Static Buffer(s) check box for the related SIO object.

SIO_staticbuf returns the size of the buffer or 0 if no more buffers are available from the stream.

An inconsistency exists between the sizes of buffers in a stream and the return types corresponding to these sizes. While all buffer sizes in a stream are of type size_t, APIs that return a buffer size return a type of Int. This due to a change in stream buffer sizes and the need to retain the return type for backward compatibility. Because of this inconsistency, it is not possible to return the correct buffer size when the actual buffer size exceeds the size of an Int type. This issue has the following implications:

- **If the actual buffer size is less than/equal to the maximum positive Int value (15 bits).** Check the return value for negative values, which indicate errors. Positive values reflect the correct size.
- **If the actual buffer size is greater than the maximum positive Int value.** Ignore the return value. Size_t is the same as unsigned long.

SIO_staticbuf can be called multiple times for SIO_ISSUERECLAIM model streams.

SIO_staticbuf must be called to acquire all static buffers before calling SIO_get, SIO_put, SIO_issue or SIO_reclaim.

Constraints and Calling Context

- SIO_staticbuf should only be called for streams that are defined statically using Tconf.
- SIO_staticbuf should only be called for static streams whose "Allocate Static Buffer(s)" property has been set to true.
- SIO_staticbuf cannot be called after SIO_get, SIO_put, SIO_issue or SIO_reclaim have been called for the given stream.
- SIO_staticbuf cannot be called from an HWI.

See Also

SIO_get

2.24 STS Module

The STS module is the 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.

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the STS Manager Properties and STS Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
previousVal	Int32	0
unitType	EnumString	"Not time based" ("High resolution time based", "Low resolution time based")
operation	EnumString	"Nothing" ("A * x", "A * x + B", "(A * x + B) / C")
numA	Int32	1
numB	Int32	0
numC	Int32	1

Description

The STS module manages objects called statistics accumulators. Each STS object accumulates the following statistical information about an arbitrary 32-bit wide data series:

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

- **Total**. The sum of the individual data values in this series
- **Maximum.** The largest value already encountered in this series

Using the count and total, the Statistics View analysis tool calculates the average on the host.

Statistics are accumulated in 32-bit variables on the target and in 64-bit variables on the host. When the host polls the target for real-time statistics, it resets the variables on the target. This minimizes space requirements on the target while allowing you to keep statistics for long test runs.

Default STS Tracing

In the RTA Control Panel, you can enable statistics tracing for the following modules by marking the appropriate checkbox. You can also set the HWI Object Properties to perform various STS operations on registers, addresses, or pointers.

Except for tracing TSK execution, your program does not need to include any calls to STS functions in order to gather these statistics. The default units for the statistics values are shown in Table Table 2-6.

Table 2-6: Statistics Units for HWI, PIP, PRD, and SWI Modules

Module	Units
HWI	Gather statistics on monitored values within HWIs
PIP	Number of frames read from or written to data pipe (count only)
PRD	Number of ticks elapsed from time that the PRD object is ready to run to end of execution
SWI	Instruction cycles elapsed from time posted to completion
TSK	Instruction cycles elapsed from time TSK is made ready to run until the application calls TSK_deltatime.

Custom STS Objects

You can create custom STS objects using Tconf. The STS_add operation updates the count, total, and maximum using the value you provide. The STS_set operation sets a previous value. The STS_delta operation accumulates the difference between the value you pass and the previous value and updates the previous value to the value you pass.

By using custom STS objects and the STS operations, you can do the following:

- **Count the number of occurrences of an event**. You can pass a value of 0 to STS_add. The count statistic tracks how many times your program calls STS_add for this STS object.
- **Track the maximum and average values for a variable in your program**. For example, suppose you pass amplitude values to STS_add. The count tracks how many times your program calls STS_add for this STS object. The total is the sum of all the amplitudes. The maximum is the largest value. The Statistics View calculates the average amplitude.
- **Track the minimum value for a variable in your program**. Negate the values you are monitoring and pass them to STS_add. The maximum is the negative of the minimum value.
- **Time events or monitor incremental differences in a value**. For example, suppose you want to measure the time between hardware interrupts. You would call STS_set when the program begins running and STS_delta each time the interrupt routine runs, passing the result of CLK_gethtime each time. STS_delta subtracts the previous value from the current value. The count tracks how many times the interrupt routine was performed. The maximum is the largest number of clock counts between interrupt routines. The Statistics View also calculates the average number of clock counts.

- **Monitor differences between actual values and desired values**. For example, suppose you want to make sure a value stays within a certain range. Subtract the midpoint of the range from the value and pass the absolute value of the result to STS_add. The count tracks how many times your program calls STS_add for this STS object. The total is the sum of all deviations from the middle of the range. The maximum is the largest deviation. The Statistics View calculates the average deviation.

You can further customize the statistics data by setting the STS Object Properties to apply a printf format to the Total, Max, and Average fields in the Statistics View window and choosing a formula to apply to the data values on the host.

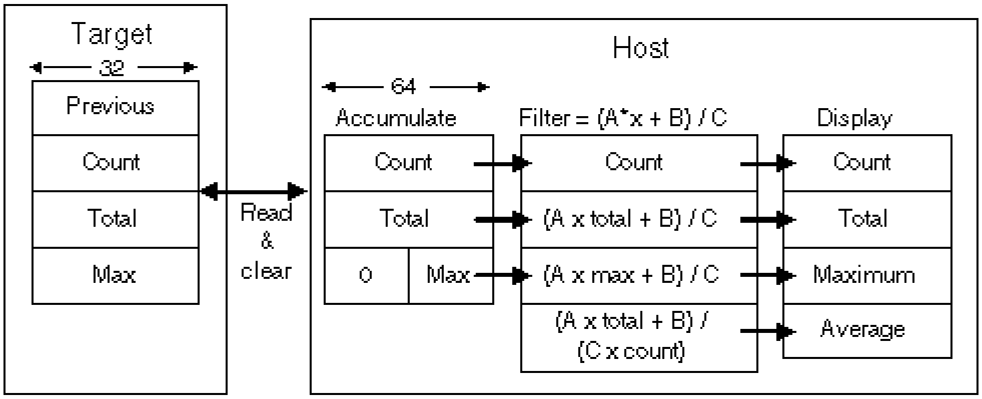
Statistics Data Gathering by the Statistics View Analysis Tool

The statistics manager allows the creation of any number of statistics objects, which in turn can be used by the application to accumulate simple statistics about a time series. This information includes the 32-bit maximum value, the last 32-bit value passed to the object, the number of samples (up to 2^{32} - 1 samples), and the 32-bit sum of all samples.

These statistics are accumulated on the target in real-time until the host reads and clears these values on the target. The host, however, continues to accumulate the values read from the target in a host buffer which is displayed by the Statistics View real-time analysis tool. Provided that the host reads and clears the target statistics objects faster than the target can overflow the 32-bit wide values being accumulated, no information loss occurs.

Using Tconf, you can select a Host Operation for an STS object. The statistics are filtered on the host using the operation and variables you specify. Figure Figure 2-8 shows the effects of the (A x X + B) / C operation.

Figure 2-8. Statistics Accumulation on the Host



STS Manager Properties

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

- **Object Memory**. The memory segment that contains STS objects.

  Tconf Name: OBJMEMSEG Type: Reference

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

STS Object Properties

To create an STS object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySts = bios.STS.create("mySts");
```

The following properties can be set for an STS object in the STS Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `mySts.comment = "my STS";`

- **prev**. The initial 32-bit history value to use in this object.

  Tconf Name: previousVal Type: Int32

  Example: `mySts.previousVal = 0;`

- **unit type**. The unit type property enables you to choose the type of time base units.

  — Not time based. If you select this unit type, the values are displayed in the Statistics View without applying any conversion.

  — High-resolution time based. If you select this type, the Statistics View, by default, presents results in units of instruction cycles.

  — Low-resolution time based. If you select this unit type, the default Statistics View presents results in timer interrupt units.

  Tconf Name: unitType Type: EnumString

  Options: "Not time based", "High resolution time based", "Low resolution time based"

  Example: `mySts.unitType = "Not time based";`

- **host operation**. The expression evaluated (by the host) on the data for this object before it is displayed by the Statistics View real-time analysis tool. The operation can be:

  — A x X
  — A x X + B
  — (A x X + B) / C

  Tconf Name: operation Type: EnumString

  Options: "Nothing", "A * x", "A * x + B", "(A * x + B) / C"

  Example: `mySts.operation = "Nothing";`

- **A, B, C**. The integer parameters used by the expression specified by the Host Operation property above.

  Tconf Name: numA Type: Int32
  Tconf Name: numB Type: Int32
  Tconf Name: numC Type: Int32

  Example:
  ```
  mySts.numA = 1;
  mySts.numB = 0;
  mySts.numC = 1;
  ```

STS_add *Update statistics using the provided value*

C Interface

Syntax

```
STS_add(sts, value);
```

Parameters

```
STS_Handle        sts;      /* statistics object handle */
LgInt              value;   /* new value to update statistics object */
```

Return Value

Void

Reentrant

no

Description

STS_add updates a custom STS object's Total, Count, and Max fields using the data value you provide.

For example, suppose your program passes 32-bit amplitude values to STS_add. The Count field tracks how many times your program calls STS_add for this STS object. The Total field tracks the total of all the amplitudes. The Max field holds the largest value passed to this point. The Statistics View analysis tool calculates the average amplitude.

You can count the occurrences of an event by passing a dummy value (such as 0) to STS_add and watching the Count field.

You can view the statistics values with the Statistics View analysis tool by enabling statistics in the DSP/BIOS→RTA Control Panel window and choosing your custom STS object in the DSP/BIOS→Statistics View window.

See Also

STS_delta
STS_reset
STS_set
TRC_disable
TRC_enable

STS_delta *Update statistics using difference between provided value & setpoint*

C Interface

Syntax

STS_delta(sts,value);

Parameters

STS_Handle	sts;	/* statistics object handle */
LgInt	value;	/* new value to update statistics object */

Return Value

Void

Reentrant

no

Description

Each STS object contains a previous value that can be initialized with Tconf or with a call to STS_set. A call to STS_delta subtracts the previous value from the value it is passed and then invokes STS_add with the result to update the statistics. STS_delta also updates the previous value with the value it is passed.

STS_delta can be used in conjunction with STS_set to monitor the difference between a variable and a desired value or to benchmark program performance. You can benchmark code by using paired calls to STS_set and STS_delta that pass the value provided by CLK_gethtime.

```
STS_set(&sts, CLK_gethtime());
  "processing to be benchmarked"
STS_delta(&sts, CLK_gethtime());
```

Constraints and Calling Context

- Before the first call to STS_delta is made, the previous value of the STS object should be initialized either with a call to STS_set or by setting the prev property of the STS object using Tconf.

Example

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

See Also

STS_add
STS_reset
STS_set
CLK_gethtime
CLK_getltime
PRD_getticks
TRC_disable
TRC_enable

STS_reset *Reset the values stored in an STS object*

C Interface

Syntax

STS_reset(sts);

Parameters

STS_Handle	sts;	/* statistics object handle */

Return Value

Void

Reentrant

no

Description

STS_reset resets the values stored in an STS object. The Count and Total fields are set to 0 and the Max field is set to the largest negative number. STS_reset does not modify the value set by STS_set.

After the Statistics View analysis tool polls statistics data on the target, it performs STS_reset internally. This keeps the 32-bit total and count values from wrapping back to 0 on the target. The host accumulates these values as 64-bit numbers to allow a much larger range than can be stored on the target.

Example

```
STS_reset(&sts);
STS_set(&sts, value);
```

See Also

STS_add
STS_delta
STS_set
TRC_disable
TRC_enable

STS_set *Save a value for STS_delta*

C Interface

Syntax
STS_set(sts, value);

Parameters

STS_Handle	sts;	/* statistics object handle */
LgInt	value;	/* new value to update statistics object */

Return Value
Void

Reentrant

no

Description

STS_set can be used in conjunction with STS_delta to monitor the difference between a variable and a desired value or to benchmark program performance. STS_set saves a value as the previous value in an STS object. STS_delta subtracts this saved value from the value it is passed and invokes STS_add with the result.

STS_delta also updates the previous value with the value it was passed. Depending on what you are measuring, you can need to use STS_set to reset the previous value before the next call to STS_delta.

You can also set a previous value for an STS object in the configuration. 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_getltime());
  "processing to be benchmarked"
STS_delta(&sts, CLK_getltime());
```

This example gathers information about a value’s deviation from the desired value.

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

This example gathers information about a value’s difference from a base value.

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

See Also

STS_add
STS_delta
STS_reset
TRC_disable
TRC_enable

2.25 SWI Module

The SWI module is the software interrupt manager.

Functions

- SWI_andn. Clear bits from SWI's mailbox; post if becomes 0.
- SWI_andnHook. Specialized version of SWI_andn for use as hook function for configured DSP/BIOS objects. Both its arguments are of type (Arg).
- SWI_create. Create a software interrupt.
- SWI_dec. Decrement SWI's mailbox value; post if becomes 0.
- SWI_delete. Delete a software interrupt.
- SWI_disable. Disable software interrupts.
- SWI_enable. Enable software interrupts.
- SWI_getattrs. Get attributes of a software interrupt.
- SWI_getmbox. Return the mailbox value of the SWI when it started running.
- SWI_getpri. Return a SWI’s priority mask.
- SWI_inc. Increment SWI's mailbox value and post the SWI.
- SWI_isSWI. Check current thread calling context.
- SWI_or. Or mask with value contained in SWI's mailbox and post the SWI.
- SWI_orHook. Specialized version of SWI_or for use as hook function for configured DSP/BIOS objects. Both its arguments are of type (Arg).
- SWI_post. Post a software interrupt.
- SWI_raisepri. Raise a SWI’s priority.
- SWI_restorepri. Restore a SWI’s priority.
- SWI_self. Return address of currently executing SWI object.
- SWI_setattrs. Set attributes of a software interrupt.

Constants, Types, and Structures

```
typedef struct SWI_Obj SWI_Handle;

SWI_MINPRI  = 1;  /* Minimum execution priority */
SWI_MAXPRI = 14   /* Maximum execution priority */

struct SWI_Attrs {    /* SWI attributes */
   SWI_Fxn  fxn;      /* address of SWI function */
   Arg      arg0;     /* first arg to function */
   Arg      arg1;     /* second arg to function */
   Bool     iscfxn;   /* TRUE if fxn is in C */
   Int      priority; /* Priority of SWI object */
   Uns      mailbox;  /* check for SWI posting */
};
```

```
SWI_Attrs SWI_ATTRS = {  /* Default attribute values */
    (SWI_Fxn)FXN_F_nop,     /* SWI function */
    0,                      /* arg0 */
    0,                      /* arg1 */
    TRUE,                   /* iscfxn */
    1,                      /* priority */
    0                       /* mailbox */
 };
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SWI Manager Properties and SWI Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
OBJMEMSEG	Reference	prog.get("L0SARAM")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
fxn	Extern	prog.extern("FXN_F_nop")
priority	EnumInt	1 (0 to 14)
mailbox	Int16	0
arg0	Arg	0
arg1	Arg	0

Description

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt service routines.

DSP/BIOS manages four distinct levels of execution threads: hardware interrupt service routines, software interrupt routines, tasks, and background idle functions. A software interrupt is an object that encapsulates a function to be executed and a priority. Software interrupts are prioritized, preempt tasks, and are preempted by hardware interrupt service routines.

Note: SWI functions are called after the processor register state has been saved. SWI functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

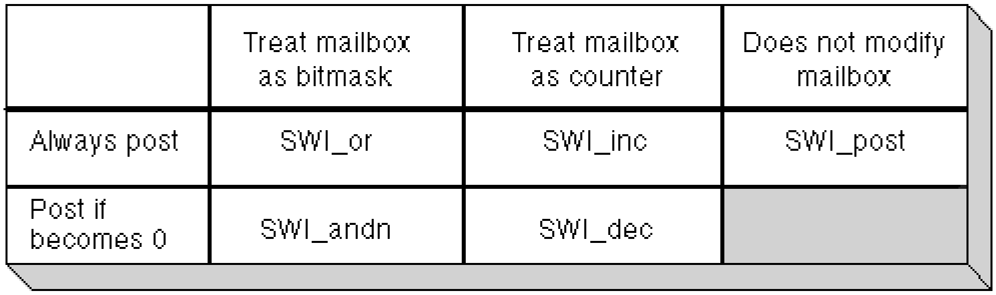
Note: **RTS Functions Callable from TSK Threads Only.** Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK_pend and LCK_post. As a result, RTS functions that call LCK_pend or LCK_post *must not be called in the context of a SWI or HWI thread*. For a list or RTS functions that should not be called from a SWI or an HWI function, see “LCK_pend” on page 169.

The C++ new operator calls malloc, which in turn calls LCK_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

Each software interrupt has a priority level. A software interrupt preempts any lower-priority software interrupt currently executing.

A target program uses an API call to post a SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a SWI is posted by an API call, the SWI object's function is not executed immediately. Instead, the function is scheduled for execution. DSP/BIOS uses the SWI's priority to determine whether to preempt the thread currently running. Note that if a SWI is posted several times before it begins running, (because HWIs and higher priority interrupts are running,) when the SWI does eventually run, it will run only one time.

Software interrupts can be posted for execution with a call to SWI_post or a number of other SWI functions. Each SWI object has a 16-bit mailbox which is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI's function. SWI_andn and SWI_dec post the SWI if the mailbox value transitions to 0. SWI_or and SWI_inc also modify the mailbox value. (SWI_or sets bits, and SWI_andn clears bits.)



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

The SWI_disable and SWI_enable operations allow you to post several SWIs and enable them all for execution at the same time. The SWI priorities then determine which SWI runs first.

All SWIs run to completion; you cannot suspend a SWI while it waits for something (for example, a device) to be ready. So, you can use the mailbox to tell the SWI when all the devices and other conditions it relies on are ready. Within a SWI function, a call to SWI_getmbox returns the value of the mailbox when the SWI started running. Note that the mailbox is automatically reset to its original value when a SWI runs; however, SWI_getmbox will return the saved mailbox value from when the SWI started execution.

Software interrupts can have up to 15 priority levels. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task (TSK) scheduler.

A SWI preempts any currently running SWI with a lower priority. If two SWIs with the same priority level have been posted, the SWI that was posted first runs first. HWIs in turn preempt any currently running SWI, allowing the target to respond quickly to hardware peripherals.

Interrupt threads (including HWIs and SWIs) are all executed using the same stack. A context switch is performed when a new thread is added to the top of the stack. The SWI module automatically saves the processor's registers before running a higher-priority SWI that preempts a lower-priority SWI. After the higher-priority SWI finishes running, the registers are restored and the lower-priority SWI can run if no other higher-priority SWI has been posted. (A separate task stack is used by each task thread.)

See the *Code Composer Studio* online tutorial for more information on how to post SWIs and scheduling issues for the Software Interrupt manager.

SWI Manager Properties

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

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

  Tconf Name: OBJMEMSEG Type: Reference

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

SWI Object Properties

To create a SWI object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var mySwi = bios.SWI.create("mySwi");
```

If you cannot create a new SWI object (an error occurs or the Insert SWI item is inactive in the DSP/BIOS Configuration Tool), try increasing the Stack Size property in the MEM Manager Properties before adding a SWI object or a SWI priority level.

The following properties can be set for a SWI object in the SWI Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

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

  Tconf Name: comment Type: String

  Example: `mySwi.comment = "my SWI";`

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

  Tconf Name: fxn Type: Extern

  Example: `mySwi.fxn = prog.extern("swiFxn");`

- **priority**. This property shows the numeric priority level for this SWI object. SWIs can have up to 15 priority levels. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler. Instead of typing a number in the DSP/BIOS Configuration Tool, you change the relative priority levels of SWI objects by dragging the objects in the ordered collection view.

  Tconf Name: priority Type: EnumInt

  Options: 0 to 14

  Example: `mySwi.priority = 1;`

- **mailbox**. The initial value of the 16-bit word used to determine if this SWI should be posted.

  Tconf Name: mailbox Type: Int16

  Example: `mySwi.mailbox = 7;`

- **arg0, arg1**. Two arbitrary pointer type (Arg) arguments to the above configured user function.

  Tconf Name: arg0 Type: Arg

  Tconf Name: arg1 Type: Arg

  Example: `mySwi.arg0 = 0;`

SWI_andn *Clear bits from SWI’s mailbox and post if mailbox becomes 0*

C Interface

Syntax

SWI_andn(swi, mask);

Parameters

SWI_Handle	swi;	/* SWI object handle*/
Uns	mask	/* inverse value to be ANDed */

Return Value

Void

Reentrant

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 SWI. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

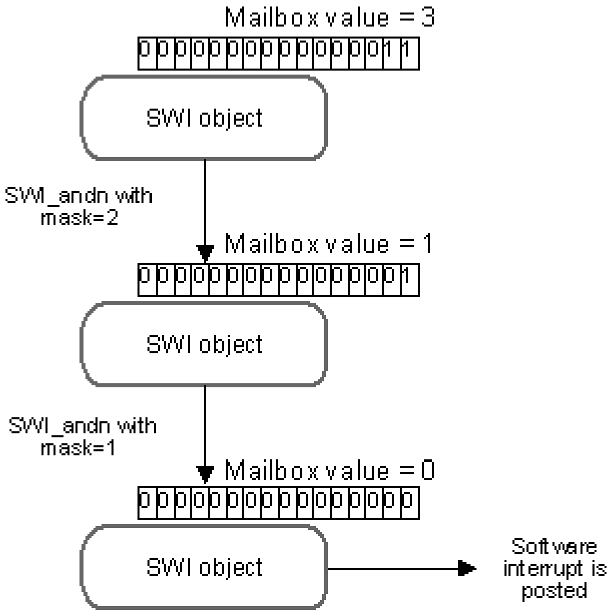
For example, if multiple conditions that all be met before a SWI can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

SWI_andn results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

You specify a SWI’s initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.

Note: Use the specialized version, SWI_andnHook, when SWI_andn functionality is required for a DSP/BIOS object hook function.

The following figure shows an example of how a mailbox with an initial value of 3 can be cleared by two calls to SWI_andn with values of 2 and 1. The entire mailbox could also be cleared with a single call to SWI_andn with a value of 3.



Constraints and Calling Context

- If this function is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_andn must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

```
/* ======== ioReady ======== */

   Void ioReady(unsigned int mask)
   {
      /* clear bits of "ready mask" */
      SWI_andn(&copySWI, mask);
   }
```

See Also

SWI_andnHook
SWI_dec
SWI_getmbox
SWI_inc
SWI_or
SWI_orHook
SWI_post
SWI_self

SWI_andnHook *Clear bits from SWI’s mailbox and post if mailbox becomes 0*

C Interface

Syntax

SWI_andnHook(swi, mask);

Parameters

Arg	swi;	/* SWI object handle*/
Arg	mask	/* value to be ANDed */

Return Value

Void

Reentrant

yes

Description

SWI_andnHook is a specialized version of SWI_andn for use as hook function for configured DSP/BIOS objects. SWI_andnHook clears the bits specified by a mask from SWI’s internal mailbox and also moves the arguments to the correct registers for proper interface with low level DSP/BIOS assembly code. If SWI’s mailbox becomes 0, SWI_andnHook posts the SWI. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

For example, if there are multiple conditions that must all be met before a SWI can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

SWI_andnHook results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

You specify a SWI’s initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_andnHook must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

```
/* ======== ioReady ======== */

   Void ioReady(unsigned int mask)
   {
      /* clear bits of "ready mask" */
      SWI_andnHook(&copySWI, mask);
   }
```

See Also

SWI_andn
SWI_orHook

SWI_create *Create a software interrupt*

C Interface

Syntax

swi = SWI_create(attrs);

Parameters

SWI_Attrs	*attrs;	/* pointer to swi attributes */

Return Value

SWI_Handle	swi;	/* handle for new swi object */

Description

SWI_create creates a new SWI object. If successful, SWI_create returns the handle of the new SWI object. If unsuccessful, SWI_create returns NULL unless it aborts. For example, SWI_create can abort if it directly or indirectly calls SYS_error, and SYS_error is configured to abort.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the object to be created, facilitates setting the SWI object's attributes. The SWI object's attributes are specified through a structure of type SWI_attrs defined as follows:

```
struct SWI_Attrs {
   SWI_Fxn  fxn;
   Arg      arg0;
   Arg      arg1;
   Bool     iscfxn;
   Int      priority;
   Uns      mailbox;
};
```

If attrs is NULL, the new SWI object is assigned the following default attributes.

```
SWI_Attrs SWI_ATTRS = {  /* Default attribute values */
    (SWI_Fxn)FXN_F_nop,     /* SWI function */
    0,                      /* arg0 */
    0,                      /* arg1 */
    TRUE,                   /* iscfxn */
    1,                      /* priority */
    0                       /* mailbox */
 };
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The iscfxn attribute must be TRUE if the fxn attribute references a C function (or an assembly function that expects the C run-time environment). This causes the C preconditions to be applied by the SWI scheduler before calling fxn.

The priority attribute specifies the SWI object's execution priority and must range from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant SWI_ATTRS, which can be assigned to a variable of type SWI_Attrs prior to calling SWI_create.

SWI_create calls MEM_alloc to dynamically create the object’s data structure. MEM_alloc must acquire a lock to the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module, page 2–192.

Constraints and Calling Context

- SWI_create cannot be called from a SWI or HWI.
- The fxn attribute cannot be NULL.
- The priority attribute must be less than or equal to 14 and greater than or equal to 1.

See Also

SWI_delete
SWI_getattrs
SWI_setattrs
SYS_error

SWI_dec *Decrement SWI's mailbox value and post if mailbox becomes 0*

C Interface

Syntax

SWI_dec(swi);

Parameters

SWI_Handle	swi;	/* SWI object handle*/

Return Value

Void

Reentrant

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 SWI. You can increment a mailbox value by using SWI_inc, which always posts the SWI.

For example, you would use SWI_dec if you wanted to post a SWI after a number of occurrences of an event.

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes.

SWI_dec results in a context switch if the SWI's mailbox becomes zero and the SWI has higher priority than the currently executing thread.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_dec must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

```
/* ======== strikeOrBall ======== */

   Void strikeOrBall(unsigned int call)
   {
      if (call == 1) {
         /* initial mailbox value is 3 */
         SWI_dec(&strikeoutSwi);
      }
      if (call == 2) {
         /* initial mailbox value is 4 */
         SWI_dec(&walkSwi);
      }
   }
```

See Also

SWI_inc

SWI_delete *Delete a software interrupt*

C Interface

Syntax

```
SWI_delete(swi);
```

Parameters

```
SWI_Handle          swi;          /* SWI object handle */
```

Return Value

Void

Description

SWI_delete uses MEM_free to free the SWI object referenced by swi.

SWI_delete calls MEM_free to delete the SWI object. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Constraints and Calling Context

- swi cannot be the currently executing SWI object (SWI_self)
- SWI_delete cannot be called from a SWI or HWI.
- SWI_delete must not be used to delete a statically-created SWI object. No check is performed to prevent SWI_delete from being used on a statically-created object. If a program attempts to delete a SWI object that was created using Tconf, SYS_error is called.

See Also

SWI_create
SWI_getattrs
SWI_setattrs
SYS_error

SWI_disable *Disable software interrupts*

C Interface

Syntax
SWI_disable();

Parameters
Void

Return Value
Void

Reentrant

yes

Description

SWI_disable and SWI_enable control software interrupt processing. SWI_disable disables all other SWI functions from running until SWI_enable is called. Hardware interrupts can still run.

SWI_disable and SWI_enable let you ensure that statements that must be performed together during critical processing are not interrupted. In the following example, the critical section is not preempted by any SWIs.

```
SWI_disable();
   `critical section`
SWI_enable();
```

You can also use SWI_disable and SWI_enable to post several SWIs and have them performed in priority order. See the following example.

SWI_disable calls can be nested. The number of nesting levels is stored internally. SWI handling is not reenabled until SWI_enable has been called as many times as SWI_disable.

Constraints and Calling Context

- The calls to HWI_enter and HWI_exit required in any HWIs that schedule SWIs automatically disable and reenable SWI handling. You should not call SWI_disable or SWI_enable within a HWI.
- SWI_disable cannot be called from the program's main() function.
- Do not call SWI_enable when SWIs are already enabled. If you do, a subsequent call to SWI_disable does not disable SWI processing.

Example

```
/* ======== postEm ======== */
   Void postEm
   {
      SWI_disable();
      SWI_post(&encoderSwi);
      SWI_andn(&copySwi, mask);
      SWI_dec(&strikeoutSwi);
      SWI_enable();
   }
```

See Also

HWI_disable
SWI_enable

SWI_enable *Enable software interrupts*

C Interface

Syntax

SWI_enable();

Parameters

Void

Return Value

Void

Reentrant

yes

Description

SWI_disable and SWI_enable control software interrupt processing. SWI_disable disables all other SWI functions from running until SWI_enable is called. Hardware interrupts can still run. See the SWI_disable section for details.

SWI_disable calls can be nested. The number of nesting levels is stored internally. SWI handling is not be reenabled until SWI_enable has been called as many times as SWI_disable.

SWI_enable results in a context switch if a higher-priority SWI is ready to run.

Constraints and Calling Context

- The calls to HWI_enter and HWI_exit are required in any HWI that schedules SWIs. They automatically disable and reenable SWI handling. You should not call SWI_disable or SWI_enable within a HWI.
- SWI_enable cannot be called from the program's main() function.
- Do not call SWI_enable when SWIs are already enabled. If you do so, the subsequent call to SWI_disable will not disable SWI processing.

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

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;
   Bool     iscfxn;
   Int      priority;
   Uns      mailbox;
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The iscfxn attribute is TRUE if the fxn attribute references a C function (or an assembly function that expects the C run-time environment).

The priority attribute specifies the SWI object's execution priority and ranges from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

The following example uses SWI_getattrs:

```
extern  SWI_Handle swi;
SWI_Attrs attrs;

SWI_getattrs(swi, &attrs);
attrs.priority = 5;
SWI_setattrs(swi, &attrs);
```

Constraints and Calling Context

- SWI_getattrs cannot be called from a SWI or HWI.
- The attrs parameter cannot be NULL.

See Also

SWI_create
SWI_delete
SWI_setattrs

SWI_getmbox *Return a SWI's mailbox value*

C Interface

Syntax
num = Uns SWI_getmbox();

Parameters
Void

Return Value

Uns	num	/* mailbox value */

Reentrant

yes

Description

SWI_getmbox returns the value that SWI's mailbox had when the SWI started running. DSP/BIOS saves the mailbox value internally so that SWI_getmbox can access it at any point within a SWI object's function. DSP/BIOS then automatically resets the mailbox to its initial value (defined with Tconf) so that other threads can continue to use the SWI's mailbox.

SWI_getmbox should only be called within a function run by a SWI object.

When called from with the context of a SWI, the value returned by SWI_getmbox is zero if the SWI was posted by a call to SWI_andn, SWI_andnHook, or SWI_dec. Therefore, SWI_getmbox provides relevant information only if the SWI was posted by a call to SWI_inc, SWI_or, SWI_orHook, or SWI_post.

Constraints and Calling Context

- SWI_getmbox cannot be called from the context of an HWI or TSK.
- SWI_getmbox cannot be called from a program's main() function.

Example

This call could be used within a SWI object's function to use the mailbox value within the function. For example, if you use SWI_or or SWI_inc to post a SWI, different mailbox values can require different processing.

```
swicount = SWI_getmbox();
```

See Also

SWI_andn
SWI_andnHook
SWI_dec
SWI_inc
SWI_or
SWI_orHook
SWI_post
SWI_self

SWI_getpri

Return a SWI's priority mask

C Interface

Syntax

key = SWI_getpri(swi);

Parameters

SWI_Handle	swi;	/* SWI object handle*/

Return Value

Uns	key	/* Priority mask of swi */

Reentrant

yes

Description

SWI_getpri returns the priority mask of the SWI passed in as the argument.

Example

```
/* 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 and post the SWI*

C Interface

Syntax

SWI_inc(swi);

Parameters

SWI_Handle	swi;	/* SWI object handle*/

Return Value

Void

Reentrant

no

Description

SWI_inc increments the value in SWI's mailbox by 1 and posts the SWI regardless of the resulting mailbox value. You can decrement a mailbox value using SWI_dec, which only posts the SWI if the mailbox value is 0.

If a SWI is posted several times before it has a chance to begin executing, because HWIs and higher priority SWIs are running, the SWI only runs one time. If this situation occurs, you can use SWI_inc to post the SWI. Within the SWI's function, you could then use SWI_getmbox to find out how many times this SWI has been posted since the last time it was executed.

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes. To get the mailbox value, use SWI_getmbox.

SWI_inc results in a context switch if the SWI is higher priority than the currently executing thread.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_inc must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

Example

```
extern SWI_ObjMySwi;
/* ======== AddAndProcess ======== */
Void AddAndProcess(int count)

  int i;
  for (i = 1; I <= count; ++i)
     SWI_inc(&MySwi);
}
```

See Also

SWI_dec
SWI_getmbox

SWI_isSWI *Check to see if called in the context of a SWI*

C Interface

Syntax

```
result = SWI_isSWI(Void);
```

Parameters

Void

Return Value

Bool	result;	/* TRUE if in SWI context, FALSE otherwise */

Reentrant

yes

Description

This macro returns TRUE when it is called within the context of a SWI or PRD function. This applies no matter whether the SWI was posted by an HWI, TSK, or IDL thread. This macro returns FALSE in all other contexts.

In previous versions of DSP/BIOS, calling SWI_isSWI() from a task switch hook resulted in TRUE. This is no longer the case; task switch hooks are identified as part of the TSK context.

See Also

HWI_isHWI
TSK_isTSK

SWI_or

OR mask with the value contained in SWI's mailbox field

C Interface

Syntax

SWI_or(swi, mask);

Parameters

SWI_Handle	swi;	/* SWI object handle*/
Uns	mask;	/* value to be ORed */

Return Value

Void

Reentrant

no

Description

SWI_or is used to post a software interrupt. SWI_or sets the bits specified by a mask in SWI's mailbox. SWI_or posts the SWI regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes. To get the mailbox value, use SWI_getmbox.

For example, you might use SWI_or to post a SWI if any of three events should cause a SWI to be executed, but you want the SWI's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

SWI_or results in a context switch if the SWI is higher priority than the currently executing thread.

Note: Use the specialized version, SWI_orHook, when SWI_or functionality is required for a DSP/BIOS object hook function.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_or must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

See Also

SWI_andn
SWI_orHook

SWI_orHook *OR mask with the value contained in SWI's mailbox field*

C Interface

Syntax

SWI_orHook(swi, mask);

Parameters

Arg	swi;	/* SWI object handle*/
Arg	mask;	/* value to be ORed */

Return Value

Void

Reentrant

no

Description

SWI_orHook is used to post a software interrupt, and should be used when hook functionality is required for DSP/BIOS hook objects. SWI_orHook sets the bits specified by a mask in SWI's mailbox and also moves the arguments to the correct registers for interfacing with low level DSP/BIOS assembly code. SWI_orHook posts the SWI regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a SWI's initial mailbox value in the configuration. The mailbox value is automatically reset when the SWI executes. To get the mailbox value, use SWI_getmbox.

For example, you might use SWI_orHook to post a SWI if any of three events should cause a SWI to be executed, but you want the SWI's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

SWI_orHook results in a context switch if the SWI is higher priority than the currently executing thread.

Note: Use the specialized version, SWI_orHook, when SWI_or functionality is required for a DSP/BIOS object hook function.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_orHook must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

See Also

SWI_andnHook
SWI_or

SWI_post *Post a software interrupt*

C Interface

Syntax

```
SWI_post(swi);
```

Parameters

```
SWI_Handle          swi;          /* SWI object handle*/
```

Return Value

Void

Reentrant

yes

Description

SWI_post is used to post a software interrupt regardless of the mailbox value. No change is made to the SWI object's mailbox value.

To have a PRD object post a SWI object's function, you can set _SWI_post as the function property of a PRD object and the name of the SWI object you want to post its function as the arg0 property.

SWI_post results in a context switch if the SWI is higher priority than the currently executing thread.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an HWI, interrupts must be enabled.
- When called within an HWI, the code sequence calling SWI_post must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

See Also

SWI_andn
SWI_dec
SWI_getmbox
SWI_inc
SWI_or
SWI_self

SWI_raisepri *Raise a SWI's priority*

C Interface

Syntax
key = SWI_raisepri(mask);

Parameters

Uns	mask;	/* mask of desired priority level */

Return Value

Uns	key;	/* key for use with SWI_restorepri */

Reentrant

yes

Description

SWI_raisepri is used to raise the priority of the currently running SWI to the priority mask passed in as the argument. SWI_raisepri can be used in conjunction with SWI_restorepri to provide a mutual exclusion mechanism without disabling SWIs.

SWI_raisepri should be called before a shared resource is accessed, and SWI_restorepri should be called after the access to the shared resource.

A call to SWI_raisepri not followed by a SWI_restorepri keeps the SWI's priority for the rest of the processing at the raised level. A SWI_post of the SWI posts the SWI at its original priority level.

A SWI object's execution priority must range from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). Priority zero (0) is reserved for the KNL_swi object, which runs the task scheduler.

SWI_raisepri never lowers the current SWI priority.

Constraints and Calling Context

- SWI_raisepri cannot be called from an HWI or TSK level.

Example

```
/* raise priority to the priority of swi_1 */
key = SWI_raisepri(SWI_getpri(&swi_1));
---  access shared resource ---
SWI_restore(key);
```

See Also

SWI_getpri
SWI_restorepri

SWI_restorepri *Restore a SWI’s priority*

C Interface

Syntax

SWI_restorepri(key);

Parameters

Uns	key;	/* key to restore original priority level */

Return Value

Void

Reentrant

yes

Description

SWI_restorepri restores the priority to the SWI's priority prior to the SWI_raisepri call returning the key. SWI_restorepri can be used in conjunction with SWI_raisepri to provide a mutual exclusion mechanism without disabling all SWIs.

SWI_raisepri should be called right before the shared resource is referenced, and SWI_restorepri should be called after the reference to the shared resource.

Constraints and Calling Context

- SWI_restorepri cannot be called from an HWI or TSK level.
- SWI_restorepri must be called with interrupts (HWI and SWI) enabled.
- SWI_restorepri cannot be called from the program’s main() function.

Example

```
/* raise priority to the priority of swi_1 */
key = SWI_raisepri(SWI_getpri(&swi_1));
---  access shared resource ---
SWI_restore(key);
```

See Also

SWI_getpri
SWI_raisepri

SWI_self

Return address of currently executing SWI object

C Interface

Syntax

curswi = SWI_self();

Parameters

Void

Return Value

SWI_Handle	swi;	/* handle for current swi object */

Reentrant

yes

Description

SWI_self returns the address of the currently executing SWI.

Constraints and Calling Context

- SWI_self cannot be called from an HWI or TSK level.
- SWI_self cannot be called from the program's main() function.

Example

You can use SWI_self if you want a SWI to repost itself:

```
SWI_post(SWI_self());
```

See Also

SWI_andn
SWI_getmbox
SWI_post

SWI_setattrs *Set attributes of a software interrupt*

C Interface

Syntax

SWI_setattrs(swi, attrs);

Parameters

SWI_Handle	swi;	/* handle of the swi */
SWI_Attrs	*attrs;	/* pointer to swi attributes */

Return Value

Void

Description

SWI_setattrs sets attributes of an existing SWI object.

The swi parameter specifies the address of the SWI object whose attributes are to be set.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the SWI object, facilitates setting the attributes of the SWI object. If attrs is NULL, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type SWI_attrs defined as follows:

```
struct SWI_Attrs {
   SWI_Fxn  fxn;
   Arg      arg0;
   Arg      arg1;
   Bool     iscfxn;
   Int      priority;
   Uns      mailbox;
};
```

The fxn attribute, which is the address of the swi function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the swi function, fxn.

The iscfxn attribute must be TRUE if the fxn attribute references a C function (or an assembly function that expects the C run-time environment). This causes the C preconditions to be applied by the SWI scheduler before calling fxn.

The priority attribute specifies the SWI object's execution priority and must range from 1 to 14. Priority 14 is the highest priority. You cannot use a priority of 0; that priority is reserved for the system SWI that runs the TSK scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant SWI_ATTRS, which can be assigned to a variable of type SWI_Attrs prior to calling SWI_setattrs.

The following example uses SWI_setattrs:

```
extern  SWI_Handle swi;
SWI_Attrs attrs;

SWI_getattrs(swi, &attrs);
attrs.priority = 5;
SWI_setattrs(swi, &attrs);
```

Constraints and Calling Context

- SWI_setattrs must not be used to set the attributes of a SWI that is preempted or is ready to run.
- The fxn attribute cannot be NULL.
- The priority attribute must be less than or equal to 14 and greater than or equal to 1.

See Also

SWI_create
SWI_delete
SWI_getattrs

2.26 SYS Module

The SYS modules manages system settings.

Functions

- SYS_abort. Abort program execution
- SYS_atexit. Stack an exit handler
- SYS_error. Flag error condition
- SYS_exit. Terminate program execution
- SYS_printf. Formatted output
- SYS_putchar. Output a single character
- SYS_sprintf. Formatted output to string buffer
- SYS_vprintf. Formatted output, variable argument list
- SYS_vsprintf. Output formatted data

Constants, Types, and Structures

```
#define SYS_FOREVER  (Uns)-1 /* wait forever */
#define SYS_POLL     (Uns)0  /* don't wait */

#define SYS_OK          0  /* no error */
#define SYS_EALLOC      1  /* memory alloc error */
#define SYS_EFREE       2  /* memory free error */
#define SYS_ENODEV      3  /* dev driver not found */
#define SYS_EBUSY       4  /* device driver busy */
#define SYS_EINVAL      5  /* invalid parameter */
#define SYS_EBADIO      6  /* I/O failure */
#define SYS_EMODE       7  /* bad mode for driver */
#define SYS_EDOMAIN     8  /* domain error */
#define SYS_ETIMEOUT    9  /* call timed out */
#define SYS_EE0F       10 /* end-of-file */
#define SYS_EDEAD      11 /* deleted obj */
#define SYS_EBADOBJ    12 /* invalid object */
#define SYS_ENOTIMPL   13 /* action not implemented */
#define SYS_ENOTFOUND 14 /* resource not found */

#define SYS_EUSER  256  /* user errors start here */

#define SYS_NUMHANDLERS  8 /* # of atexit handlers */

extern String SYS_errors[]; /* error string array */
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the SYS Manager Properties heading. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default
TRACESIZE	Numeric	512
TRACESEG	Reference	prog.get("L0SARAM")

Name	Type	Default
ABORTFXN	Extern	prog.extern("UTL_doAbort")
ERRORFXN	Extern	prog.extern("UTL_doError")
EXITFXN	Extern	prog.extern("UTL_halt")
PUTCFXN	Extern	prog.extern("UTL_doPutc")

Description

The SYS module makes available a set of general-purpose functions that provide basic system services, such as halting program execution and printing formatted text. In general, each SYS function is patterned after a similar function normally found in the standard C library.

SYS does not directly use the services of any other DSP/BIOS module and therefore resides at the bottom of the system. Other DSP/BIOS modules use the services provided by SYS in lieu of similar C library functions. The SYS module provides hooks for binding system-specific code. This allows programs to gain control wherever other DSP/BIOS modules call one of the SYS functions.

SYS Manager Properties

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

- **Trace Buffer Size**. The size of the buffer that contains system trace information. This system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the CCS Memory view. For example, by default the Putc function writes to the trace buffer.

  Tconf Name: TRACESIZE Type: Numeric

  Example: `bios.SYS.TRACESIZE = 512;`

- **Trace Buffer Memory**. The memory segment that contains system trace information.

  Tconf Name: TRACESEG Type: Reference

  Example: `bios.SYS.TRACESEG = prog.get("myMEM");`

- **Abort Function**. The function to run if the application aborts by calling SYS_abort. The default function is _UTL_doAbort, which logs an error message and calls _halt. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally. The prototype for this function should be:

  ```
  Void myAbort(String fmt, va_list ap);
  ```

  Tconf Name: ABORTFXN Type: Extern

  Example: `bios.SYS.ABORTFXN = prog.extern("myAbort");`

- **Error Function**. The function to run if an error flagged by SYS_error occurs. The default function is _UTL_doError, which logs an error message and returns. The prototype for this function should be:

  ```
  Void myError(String s, Int errno, va_list ap);
  ```

  Tconf Name: ERRORFXN Type: Extern

  Example: `bios.SYS.ERRORFXN = prog.extern("myError");`

- **Exit Function**. The function to run when the application exits by calling SYS_exit. The default function is UTL_halt, which loops forever with interrupts disabled and prevents other processing. The prototype for this function should be:

  ```
  Void myExit(Int status);
  ```

  Tconf Name: EXITFXN Type: Extern

  Example: `bios.SYS.EXITFXN = prog.extern("myExit");`

- **Putc Function.** The function to run if the application calls SYS_putchar, SYS_printf, or SYS_vprintf. The default function is _UTL_doPutc, which writes a character to the system trace buffer. This system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the CCS Memory view. The prototype for this function should be:

```
Void myPutc(Char c);
```

Tconf Name:	PUTCFXN	Type: Extern
Example:	`bios.SYS.PUTCFXN = prog.extern("myPutc");`	

SYS Object Properties

The SYS module does not support the creation of individual SYS objects.

SYS_abort *Abort program execution*

C Interface

Syntax

SYS_abort(format, [arg,] ...);

Parameters

String	format;	/* format specification string */
Arg	arg;	/* optional argument */

Return Value

Void

Description

SYS_abort aborts program execution by calling the function bound to the configuration parameter Abort function, where vargs is of type va_list (a void pointer which can be interpreted as an argument list) and represents the sequence of arg parameters originally passed to SYS_abort.

```
(*(Abort_function))(format, vargs)
```

The function bound to Abort function can elect to pass the format and vargs parameters directly to SYS_vprintf or SYS_vsprintf prior to terminating program execution.

The default Abort function for the SYS manager is _UTL_doAbort, which logs an error message and calls UTL _halt, which is defined in the boot.c file. The UTL_halt function performs an infinite loop with all processor interrupts disabled.

Constraints and Calling Context

- If the function bound to Abort function is not reentrant, SYS_abort must be called atomically.

See Also

SYS_exit
SYS_printf

SYS_atexit *Stack an exit handler*

C Interface

Syntax

success = SYS_atexit(handler);

Parameters

Fxn	handler	/* exit handler function */

Return Value

Bool	success	/* handler successfully stacked */

Description

SYS_atexit pushes handler onto an internal stack of functions to be executed when SYS_exit is called. Up to SYS_NUMHANDLERS(8) functions can be specified in this manner. SYS_exit pops the internal stack until empty and calls each function as follows, where status is the parameter passed to SYS_exit:

```
(*handler)(status)
```

SYS_atexit returns TRUE if handler has been successfully stacked; FALSE if the internal stack is full.

The handlers on the stack are called only if either of the following happens:

- SYS_exit is called.
- All tasks for which the Don’t shut down system while this task is still running property is TRUE have exited. (By default, this includes the TSK_idle task, which manages communication between the target and analysis tools.)

Constraints and Calling Context

- handler cannot be NULL.

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

Description

SYS_error is used to flag DSP/BIOS error conditions. Application programs should call SYS_error to handle program errors. Internal functions also call SYS_error.

SYS_error calls a function to handle errors. The default error function for the SYS manager is _UTL_doError, which logs an error message and returns. The default function can be replaced with your own error function by setting the SYS.ERRORFXN configuration property.

The default error function or an alternate configured error function is called as follows, where vargs is of type va_list (a void pointer which can be interpreted as an argument list) and represents the sequence of arg parameters originally passed to SYS_error.

```
(*(Error_function))(s, errno, vargs)
```

Constraints and Calling Context

- The only valid error numbers are the error constants defined in sys.h (SYS_E*) or numbers greater than or equal to SYS_EUSER. Passing any other error values to SYS_error can cause DSP/BIOS to crash.

SYS_exit *Terminate program execution*

C Interface

Syntax

SYS_exit(status);

Parameters

Int	status;	/* termination status code */

Return Value

Void

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 */
Arg	arg;	/* optional argument */

Return Value

Void

Description

SYS_printf provides a subset of the capabilities found in the standard C library function printf.

Note: SYS_printf and the related functions are code-intensive. If possible, applications should use the LOG Module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_printf are limited to the characters shown in Table Table 2-7.

Table 2-7: Conversion Characters Recognized by SYS_printf

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the ’C67x and 283xx).

Between the % and the conversion character, the following symbols or specifiers contained in square brackets can appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters via the Putc function configured in the SYS Manager Properties. The default Putc function is _UTL_doPutc, which writes a character to the system trace buffer. The size and memory segment for the system trace buffer can also be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the CCS Memory view.

Constraints and Calling Context

- On a DSP with floating-point support, SYS_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- On a DSP with floating-point support, SYS_printf only prints four digits after the decimal point for floating point numbers. Since SYS_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS_printf.
- The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_sprintf
SYS_vprintf
SYS_vsprintf

SYS_sprintf *Output formatted data*

C Interface

Syntax

SYS_sprintf (buffer, format, [arg,] ...);

Parameters

String	buffer;	/* output buffer */
String	format;	/* format specification string */
Arg	arg;	/* optional argument */

Return Value

Void

Description

SYS_sprintf provides a subset of the capabilities found in the standard C library function printf.

Note: SYS_sprintf and the related functions are code-intensive. If possible, applications should use LOG Module module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_sprintf are limited to the characters in Table Table 2-8.

Table 2-8: Conversion Characters Recognized by SYS_sprintf

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the ’C67x and 283xx).

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- On a DSP with floating-point support, SYS_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- On a DSP with floating-point support, SYS_printf only prints four digits after the decimal point for floating point numbers. Since SYS_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS_printf.
- The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_printf
SYS_vprintf
SYS_vsprintf

SYS_vprintf *Output formatted data*

C Interface

Syntax

SYS_vprintf(format, vargs);

Parameters

String	format;	/* format specification string */
va_list	vargs;	/* variable argument list reference */

Return Value

Void

Description

SYS_vprintf provides a subset of the capabilities found in the standard C library function printf.

Note: SYS_vprintf and the related functions are code-intensive. If possible, applications should use LOG Module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_vprintf are limited to the characters in Table Table 2-9.

Table 2-9: Conversion Characters Recognized by SYS_vprintf

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the ’C67x and 283xx).

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters via the Putc function configured in the SYS Manager Properties. The default Putc function is _UTL_doPutc, which writes a character to the system trace buffer. The size and memory segment for the system trace buffer can also be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the CCS Memory view.

Constraints and Calling Context

- On a DSP with floating-point support, SYS_printf prints an error for floating point numbers whose absolute value is greater than the maximum long int (defined as LONG_MAX in the <limits.h> ANSI header). This is because the integer part is computed by simply casting the float parameter to a long int local variable.
- On a DSP with floating-point support, SYS_printf only prints four digits after the decimal point for floating point numbers. Since SYS_printf does not support %e, floating point numbers have to be scaled approximately before being passed to SYS_printf.
- The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_printf
SYS_sprintf
SYS_vsprintf

SYS_vsprintf *Output formatted data*

C Interface

Syntax
SYS_vsprintf(buffer, format, vargs);

Parameters

String	buffer;	/* output buffer */
String	format;	/* format specification string */
va_list	vargs;	/* variable argument list reference */

Return Value
Void

Description

SYS_vsprintf provides a subset of the capabilities found in the standard C library function printf.

Note: SYS_vsprintf and the related functions are code-intensive. If possible, applications should use LOG Module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_vsprintf are limited to the characters in Table Table 2-10.

Table 2-10: Conversion Characters Recognized by SYS_vsprintf

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string
p	data pointer

Note that the %f conversion character is supported only on devices that have a native floating point type (for example, the ’C67x and 283xx).

Between the % and the conversion character, the following symbols or specifiers contained within square brackets can appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- 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

Syntax

SYS_putchar(c);

Parameters

Char	c;	/* next output character */

Return Value

Void

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 system trace buffer. The size and memory segment for the system trace buffer can be set in the SYS Manager Properties. This system trace buffer can be viewed only by looking for the SYS_PUTCBEG symbol in the CCS Memory view.

SYS_putchar is also used internally by SYS_printf and SYS_vprintf when generating their output.

Constraints and Calling Context

- If the function bound to Putc function is not reentrant, SYS_putchar must be called atomically.

See Also

SYS_printf

2.27 TRC Module

The TRC module is the trace manager.

Functions

- TRC_disable. Disable trace class(es)
- TRC_enable. Enable trace type(s)
- TRC_query. Query trace class(es)

Description

The TRC module manages a set of trace control bits which control the real-time capture of program information through event logs and statistics accumulators. For greater efficiency, the target does not store log or statistics information unless tracing is enabled.

Table Table 2-11 lists events and statistics that can be traced. The constants defined in trc.hand trc.h28are shown in the left column.

Table 2-11: Events and Statistics Traced by TRC

Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Log timer interrupts	off
TRC_LOGPRD	Log periodic ticks and start of periodic functions	off
TRC_LOGSWI	Log events when a SWI is posted and completes	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes execution,	off
TRC_STSHWI	Gather statistics on monitored values within HWIs	off
TRC_STSPIP	Count number of frames read from or written to data pipe	off
TRC_STSPRD	Gather statistics on number of ticks elapsed during execution	off
TRC_STSSWI	Gather statistics on length of SWI execution	off
TRC_STSTSK	Gather statistics on length of TSK execution. Statistics are gathered from the time TSK is made ready to run until the application calls TSK_deltatime.	off
TRC_USER0 and TRC_USER1	Your program can use these bits to enable or disable sets of explicit instrumentation actions. You can use TRC_query to check the settings of these bits and either perform or omit instrumentation calls based on the result. 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
TRC_GBLTARG	This bit must also be set for any implicit instrumentation to be performed. This bit can only be set by the target program and is enabled by default.	on
TRC_STSSWI	Gather statistics on length of SWI execution	off

All trace constants except TRC_GBLTARG are switched off initially. To enable tracing you can use calls to TRC_enable or the DSP/BIOS→RTA Control Panel, which uses the TRC module internally. You do not need to enable tracing for messages written with LOG_printf or LOG_event and statistics added with STS_add or STS_delta.

Your program can call the TRC_enable and TRC_disable operations to explicitly start and stop event logging or statistics accumulation in response to conditions encountered during real-time execution. This enables you to preserve the specific log or statistics information you need to see.

TRC_disable *Disable trace class(es)*

C Interface

Syntax

```
TRC_disable(mask);
```

Parameters

Uns	mask;	/* trace type constant mask */

Return Value

Void

Reentrant

no

Description

TRC_disable disables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module topic for a list of constants to use in the mask.)

The following C code would disable tracing of statistics for software interrupts and periodic functions:

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

Internally, DSP/BIOS uses a bitwise AND NOT operation to disable multiple trace types.

For example, you might want to use TRC_disable with a circular log and disable tracing when an unwanted condition occurs. This allows test equipment to retrieve the log events that happened just before this condition started.

See Also

TRC_enable
TRC_query
LOG_printf
LOG_event
STS_add
STS_delta

TRC_enable *Enable trace type(s)*

C Interface

Syntax

TRC_enable(mask);

Parameters

Uns	mask;	/* trace type constant mask */

Return Value

Void

Reentrant

no

Description

TRC_enable enables tracing of one or more trace types. Trace types are specified with a 32-bit mask. (See the TRC Module topic for a list of constants to use in the mask.)

The following C code would enable tracing of statistics for software interrupts and periodic functions:

```
TRC_enable(TRC_STSSWI | TRC_STSPRD);
```

Internally, DSP/BIOS uses a bitwise OR operation to enable multiple trace types.

For example, you might want to use TRC_enable with a fixed log to enable tracing when a specific condition occurs. This allows test equipment to retrieve the log events that happened just after this condition occurred.

See Also

TRC_disable
TRC_query
LOG_printf
LOG_event
STS_add
STS_delta

TRC_query *Query trace class(es)*

C Interface

Syntax
result = TRC_query(mask);

Parameters

Uns	mask;	/* trace type constant mask */

Return Value

Int	result	/* indicates whether all trace types enabled */

Reentrant

yes

Description

TRC_query determines whether particular trace types are enabled. TRC_query returns 0 if all trace types in the mask are enabled. If any trace types in the mask are disabled, TRC_query returns a value with a bit set for each trace type in the mask that is disabled. (See the TRC Module topic for a list of constants to use in the mask.)

Trace types are specified with a 16-bit mask. The full list of constants you can use is included in the description of the TRC module.

For example, the following C code returns 0 if statistics tracing for the PRD class is enabled:

```
result = TRC_query(TRC_STSPRD);
```

The following C code returns 0 if both logging and statistics tracing for the SWI class are enabled:

```
result = TRC_query(TRC_LOGSWI | TRC_STSSWI);
```

Note that TRC_query does not return 0 unless the bits you are querying and the TRC_GBLHOST and TRC_GBLTARG bits are set. TRC_query returns non-zero if either TRC_GBLHOST or TRC_GBLTARG are disabled. This is because no tracing is done unless these bits are set.

For example, if the TRC_GBLHOST, TRC_GBLTARG, and TRC_LOGSWI bits are set, this C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI); /* returns 0 */
result = TRC_query(TRC_LOGPRD); /* returns non-zero */
```

However, if only the TRC_GBLHOST and TRC_LOGSWI bits are set, the same C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI); /* returns non-zero */
result = TRC_query(TRC_LOGPRD); /* returns non-zero */
```

See Also

TRC_enable
TRC_disable

2.28 TSK Module

The TSK module is the task manager.

Functions

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

Task Hook Functions

```
Void TSK_createFxn(TSK_Handle task);

Void TSK_deleteFxn(TSK_Handle task);

Void TSK_exitFxn(Void);

Void TSK_readyFxn(TSK_Handle newtask);

Void TSK_switchFxn(TSK_Handle oldtask,
                   TSK_Handle newtask);
```

Constants, Types, and Structures

```
typedef struct TSK_OBJ *TSK_Handle; /* task object handle*/

struct TSK_Attrs {     /* task attributes */
   Int    priority;    /* execution priority */
   Ptr     stack;      /* pre-allocated stack */
   size_t stacksize;   /* stack size in MADUs */
   Int     stackseg;   /* mem seg for stack allocation */
   Ptr     environ;    /* global environment data struct */
   String name;        /* printable name */
   Bool    exitflag;   /* program termination requires */
                       /* this task to terminate */
   Bool    initstackflag;  /* initialize task stack? */
};

Int TSK_pid;           /* MP processor ID */

Int TSK_MAXARGS = 8;   /* max number of task arguments */
Int TSK_IDLEPRI = 0;   /* used for idle task */
Int TSK_MINPRI = 1;    /* minimum execution priority */
Int TSK_MAXPRI = 15;   /* maximum execution priority */
Int TSK_STACKSTAMP =
TSK_Attrs TSK_ATTRS = { /* default attribute values */
    TSK->PRIORITY,       /* priority */
    NULL,                /* stack */
    TSK->STACKSIZE,      /* stacksize */
    TSK->STACKSEG,       /* stackseg */
    NULL,                /* environ */
    "",                  /* name */
     TRUE,               /* exitflag */
     TRUE,               /* initstackflag */
};

enum TSK_Mode {    /* task execution modes */
  TSK_RUNNING,     /* task currently executing */
  TSK_READY,       /* task scheduled for execution */
  TSK_BLOCKED,     /* task suspended from execution */
  TSK_TERMINATED, /* task terminated from execution */
};
struct TSK_Stat {        /* task status structure */
    TSK_Attrs  attrs;    /* task attributes */
    TSK_Mode   mode;     /* task execution mode */
    Ptr        sp;       /* task stack pointer */
    size_t     used;     /* task stack used */
};
```

Configuration Properties

The following list shows the properties that can be configured in a Tconf script, along with their types and default values. For details, see the TSK Manager Properties and TSK Object Properties headings. For descriptions of data types, see Section 1.4, *DSP/BIOS Tconf Overview*, page 1-10.

Module Configuration Parameters

Name	Type	Default (Enum Options)
ENABLETSK	Bool	true
OBJMEMSEG	Reference	prog.get("L0SARAM")
STACKSIZE	Int16	256
STACKSEG	Reference	prog.get("MEM_NULL")

Name	Type	Default (Enum Options)
PRIORITY	EnumInt	1 (1 to 15)
DRIVETSKTICK	EnumString	"PRD" ("User")
CREATEFXN	Extern	prog.extern("FXN_F_nop")
DELETEFXN	Extern	prog.extern("FXN_F_nop")
EXITFXN	Extern	prog.extern("FXN_F_nop")
CALLSWITCHFXN	Bool	false
SWITCHFXN	Extern	prog.extern("FXN_F_nop")
CALLREADYFXN	Bool	false
READYFXN	Extern	prog.extern("FXN_F_nop")

Instance Configuration Parameters

Name	Type	Default (Enum Options)
comment	String	"<add comments here>"
autoAllocateStack	Bool	true
manualStack	Extern	prog.extern("null","asm")
stackSize	Int16	256
stackMemSeg	Reference	prog.get("L0SARAM")
priority	EnumInt	0 (-1, 0, 1 to 15)
fxn	Extern	prog.extern("FXN_F_nop")
arg0	Arg	0
arg7	Arg	0
envPointer	Arg	0x00000000
exitFlag	Bool	true
allocateTaskName	Bool	false
order	Int16	0

Description

The TSK module makes available a set of functions that manipulate task objects accessed through handles of type TSK_Handle. Tasks represent independent threads of control that conceptually execute functions in parallel within a single C program; in reality, concurrency is achieved by switching the processor from one task to the next.

When you create a task, it is provided with its own run-time stack, used for storing local variables as well as for further nesting of function calls. The TSK_STACKSTAMP value is used to initialize the run-time stack. When creating a task dynamically, you need to initialize the stack with TSK_STACKSTAMP only if the stack is allocated manually and TSK_checkstacks or TSK_stat is to be called. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher-priority task. All tasks executing within a single program share a common set of global variables, accessed according to the standard rules of scope defined for C functions.

Each task is in one of four modes of execution at any point in time: running, ready, blocked, or terminated. By design, there is always one (and only one) task currently running, even if it is a dummy idle task managed internally by TSK. The current task can be suspended from execution by calling certain TSK functions, as well as functions provided by other modules like the SEM Module and the SIO Module; the current task can also terminate its own execution. In either case, the processor is switched to the next task that is ready to run.

You can assign numeric priorities to tasks through TSK. Tasks are readied for execution in strict priority order; tasks of the same priority are scheduled on a first-come, first-served basis. As a rule, the priority of the currently running task is never lower than the priority of any ready task. Conversely, the running task is preempted and re-scheduled for execution whenever there exists some ready task of higher priority.

You can use Tconf to specify one or more sets of application-wide hook functions that run whenever a task state changes in a particular way. For the TSK module, these functions are the Create, Delete, Exit, Switch, and Ready functions. The HOOK module adds an additional Initialization function.

A single set of hook functions can be specified for the TSK module itself. To create additional sets of hook functions, use the HOOK Module. When you create the first HOOK object, any TSK module hook functions you have specified are automatically placed in a HOOK object called HOOK_KNL. To set any properties of this object other than the Initialization function, use the TSK module properties. To set the Initialization function property of the HOOK_KNL object, use the HOOK object properties. If you configure only a single set of hook functions using the TSK module, the HOOK module is not used.

The TSK_create topic describes the Create function. The TSK_delete topic describes the Delete function. The TSK_exit topic describes the Exit function.

If a Switch function is specified, it is invoked when a new task becomes the TSK_RUNNING task. The Switch function gives the application access to both the current and next task handles at task switch time. The function should use these argument types:

```
Void mySwitchFxn(TSK_Handle currTask,
                 TSK_Handle nextTask);
```

This function can be used to save/restore additional task context (for example, external hardware registers), to check for task stack overflow, to monitor the time used by each task, etc.

If a Ready function is specified, it is invoked whenever a task is made ready to run. Even if a higher-priority thread is running, the Ready function runs. The Ready function is called with a handle to the task being made ready to run as its argument. This example function prints the name of both the task that is ready to run and the task that is currently running:

```
Void myReadyFxn(TSK_Handle task)
{
   String      nextName, currName;
   TSK_Handle  currTask = TSK_self();

   nextName = TSK_getname(task);
   LOG_printf(&trace, "Task %s Ready", nextName);

   currName = TSK_getname(currTask);
   LOG_printf(&trace, "Task %s Running", currName);
}
```

The Switch function and Ready function are called in such a way that they can use only functions allowed within a SWI handler. See Appendix A, Function Callability Table, for a list of functions that can be called by SWI handlers. There are no real constraints on what functions are called via the Create function, Delete function, or Exit function.

TSK Manager Properties

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

- **Enable TSK Manager**. If no tasks are used by the program other than TSK_idle, you can optimize the program by disabling the task manager. The program must then not use TSK objects created with either Tconf or the TSK_create function. If the task manager is disabled, the idle loop still runs and uses the system stack instead of a task stack.

  Tconf Name: ENABLETSK Type: Bool

  Example: `bios.TSK.ENABLETSK = true;`

- **Object Memory**. The memory segment that contains the TSK objects created with Tconf.

  Tconf Name: OBJMEMSEG Type: Reference

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

- **Default stack size**. The default size of the stack (in MADUs) used by tasks. You can override this value for an individual task you create with Tconf or TSK_create. The estimated minimum task size is shown in the status bar of the DSP/BIOS Configuration Tool. This property applies to TSK objects created both with Tconf and with TSK_create.

  Tconf Name: STACKSIZE Type: Int16

  Example: `bios.TSK.STACKSIZE = 256;`

- **Stack segment for dynamic tasks**. The default memory segment to contain task stacks created at run-time with the TSK_create function. The TSK_Attrs structure passed to the TSK_create function can override this default. If you select MEM_NULL for this property, creation of task objects at run-time is disabled.

  Tconf Name: STACKSEG Type: Reference

  Example: `bios.TSK.STACKSEG = prog.get("myMEM");`

- **Default task priority**. The default priority level for tasks that are created dynamically with TSK_create. This property applies to TSK objects created both with Tconf and with TSK_create.

  Tconf Name: PRIORITY Type: EnumInt

  Options: 1 to 15

  Example: `bios.TSK.PRIORITY = 1;`

- **TSK tick driven by**. Choose whether you want the system clock to be driven by the PRD module or by calls to TSK_tick and TSK_itick. This clock is used by TSK_sleep and functions such as SEM_pend that accept a timeout argument.

  Tconf Name: DRIVETSKTICK Type: EnumString

  Options: "PRD", "User"

  Example: `bios.TSK.DRIVETSKTICK = "PRD";`

- **Create function**. The name of a function to call when any task is created. This includes tasks that are created statically and those created dynamically using TSK_create. If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally. The TSK_create topic describes the Create function.

  Tconf Name: CREATEFXN Type: Extern

  Example: `bios.TSK.CREATEFXN = prog.extern("tskCreate");`

- **Delete function.** The name of a function to call when any task is deleted at run-time with TSK_delete. The TSK_delete topic describes the Delete function.

  Tconf Name: DELETEFXN Type: Extern

  Example: `bios.TSK.DELETEFXN = prog.extern("tskDelete");`

- **Exit function**. The name of a function to call when any task exits. The TSK_exit topic describes the Exit function.

  Tconf Name: EXITFXN Type: Extern

  Example: `bios.TSK.EXITFXN = prog.extern("tskExit");`

- **Call switch function**. Check this box if you want a function to be called when any task switch occurs.

  Tconf Name: CALLSWITCHFXN Type: Bool

  Example: `bios.TSK.CALLSWITCHFXN = false;`

- **Switch function**. The name of a function to call when any task switch occurs. This function can give the application access to both the current and next task handles. The TSK Module topic describes the Switch function.

  Tconf Name: SWITCHFXN Type: Extern

  Example: `bios.TSK.SWITCHFXN = prog.extern("tskSwitch");`

- **Call ready function**. Check this box if you want a function to be called when any task becomes ready to run.

  Tconf Name: CALLREADYFXN Type: Bool

  Example: `bios.TSK.CALLREADYFXN = false;`

- **Ready function**. The name of a function to call when any task becomes ready to run. The TSK Module topic describes the Ready function.

  Tconf Name: READYFXN Type: Extern

  Example: `bios.TSK.READYFXN = prog.extern("tskReady");`

TSK Object Properties

To create a TSK object in a configuration script, use the following syntax. The Tconf examples that follow assume the object has been created as shown here.

```
var myTsk = bios.TSK.create("myTsk");
```

The following properties can be set for a TSK object in the TSK Object Properties dialog of the DSP/BIOS Configuration Tool or in a Tconf script:

General tab

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

  Tconf Name: comment Type: String

  Example: `myTsk.comment = "my TSK";`

- **Automatically allocate stack**. Check this box if you want the task's private stack space to be allocated automatically when this task is created. The task's context is saved in this stack before any higher-priority task is allowed to block this task and run.

  Tconf Name: autoAllocateStack Type: Bool

  Example: `myTsk.autoAllocateStack = true;`

- **Manually allocated stack**. If you did not check the box to Automatically allocate stack, type the name of the manually allocated stack to use for this task.

  Tconf Name: manualStack Type: Extern

  Example: `myTsk.manualStack = prog.extern("myStack");`

- **Stack size**. Enter the size (in MADUs) of the stack space to allocate for this task. You must enter the size whether the application allocates the stack manually or automatically. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher priority task.

  Tconf Name: stackSize Type: Int16

  Example: `myTsk.stackSize = 256;`

- **Stack Memory Segment**. If you set the "Automatically allocate stack" property to true, specify the memory segment to contain the stack space for this task.

  Tconf Name: stackMemSeg Type: Reference

  Example: `myTsk.stackMemSeg = prog.get("myMEM");`

- **Priority**. The priority level for this task. A priority of -1 causes a task to be suspended until its priority is raised programmatically.

  Tconf Name: priority Type: EnumInt

  Options: -1, 0, 1 to 15

  Example: `myTsk.priority = 1;`

Function tab

- **Task function**. The function to be executed when the task runs. If this function is written in C and you are using the DSP/BIOS Configuration Tool, use a leading underscore before the C function name. (The DSP/BIOS Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.) If you are using Tconf, do not add an underscore before the function name; Tconf adds the underscore needed to call a C function from assembly internally. If you compile C programs with the -pm or -op2 options, you should precede C functions called by task threads with the FUNC_EXT_CALLED pragma. See the online help for the C compiler for details.

  Tconf Name: fxn Type: Extern

  Example: `myTsk.fxn = prog.extern("tskFxn");`

- **Task function argument 0-7**. The arguments to pass to the task function. Arguments can be integers or labels.

  Tconf Name: arg0 to arg7 Type: Arg

  Example: `myTsk.arg0 = 0;`

Advanced tab

- **Environment pointer**. A pointer to a globally-defined data structure this task can access. The task can get and set the task environment pointer with the TSK_getenv and TSK_setenv functions. If your program uses multiple HOOK objects, HOOK_setenv allows you to set individual environment pointers for each HOOK and TSK object combination.

  Tconf Name: envPointer Type: Arg

  Example: `myTsk.envPointer = 0;`

- **Don’t shut down system while this task is still running**. Check this box if you do not want the application to be able to end if this task is still running. The application can still abort. For example, you might clear this box for a monitor task that collects data whenever all other tasks are blocked. The application does not need to explicitly shut down this task.

  Tconf Name: exitFlag Type: Bool

  Example: `myTsk.exitFlag = true;`

- **Allocate Task Name on Target**. Check this box if you want the name of this TSK object to be retrievable by the TSK_getname function. Clearing this box saves a small amount of memory. The task name is available in analysis tools in either case.

  Tconf Name: allocateTaskName Type: Bool

  Example: `myTsk.allocateTaskName = false;`

- **order**. Set this property for all TSK objects so that the numbers match the sequence in which TSK functions with the same priority level should be executed.

  Tconf Name: order Type: Int16

  Example: `myTsk.order = 2;`

TSK_checkstacks *Check for stack overflow*

C Interface

Syntax

TSK_checkstacks(oldtask, newtask);

Parameters

TSK_Handle	oldtask;	/* handle of task switched from */
TSK_Handle	newtask;	/* handle of task switched to */

Return Value

Void

Description

TSK_checkstacks calls SYS_abort with an error message if either oldtask or newtask has a stack in which the last location no longer contains the initial value TSK_STACKSTAMP. The presumption in one case is that oldtask's stack overflowed, and in the other that an invalid store has corrupted newtask's stack.

TSK_checkstacks requires that the stack was initialized by DSP/BIOS. For dynamically-created tasks, initialization is controlled by the initstackflag attribute in the TSK_Attrs structure passed to TSK_create. Statically configured tasks always initialize the stack.

You can call TSK_checkstacks directly from your application. For example, you can check the current task's stack integrity at any time with a call like the following:

```
TSK_checkstacks(TSK_self(), TSK_self());
```

However, it is more typical to call TSK_checkstacks in the task Switch function specified for the TSK manager in your configuration file. This provides stack checking at every context switch, with no alterations to your source code.

If you want to perform other operations in the Switch function, you can do so by writing your own function (myswitchfxn) and then calling TSK_checkstacks from it.

```
Void myswitchfxn(TSK_Handle oldtask,
                 TSK_Handle newtask)
{
    `your additional context switch operations`
    TSK_checkstacks(oldtask, newtask);
    ...
}
```

Constraints and Calling Context

- TSK_checkstacks cannot be called from an HWI or SWI.

TSK_create *Create a task ready for execution*

C Interface

Syntax

task = TSK_create(fxn, attrs, [arg,] ...);

Parameters

Fxn	fxn;	/* pointer to task function */
TSK_Attrs	*attrs;	/* pointer to task attributes */
Arg	arg;	/* task arguments */

Return Value

TSK_Handle	task;	/* task object handle */

Description

TSK_create creates a new task object. If successful, TSK_create returns the handle of the new task object. If unsuccessful, TSK_create returns NULL unless it aborts (for example, because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

The fxn parameter uses the Fxn type to pass a pointer to the function the TSK object should run. For example, if myFxn is a function in your program, you can create a TSK object to call that function as follows:

```
task = TSK_create((Fxn)myFxn, NULL);
```

You can use Tconf to specify an application-wide Create function that runs whenever a task is created. This includes tasks that are created statically and those created dynamically using TSK_create. The default Create function is a no-op function.

For TSK objects created statically, the Create function is called during the BIOS_start portion of the program startup process, which runs after the main() function and before the program drops into the idle loop.

For TSK objects created dynamically, the Create function is called after the task handle has been initialized but before the task has been placed on its ready queue.

Any DSP/BIOS function can be called from the Create function. DSP/BIOS passes the task handle of the task being created to the Create function. The Create function declaration should be similar to this:

```
Void myCreateFxn(TSK_Handle task);
```

The new task is placed in TSK_READY mode, and is scheduled to begin concurrent execution of the following function call:

```
(*fxn)(arg1, arg2, ... argN) /* N = TSK_MAXARGS = 8 */
```

As a result of being made ready to run, the task runs the application-wide Ready function if one has been specified.

TSK_exit is automatically called if and when the task returns from fxn.

If attrs is NULL, the new task is assigned a default set of attributes. Otherwise, the task's attributes are specified through a structure of type TSK_Attrs, which is defined as follows.

```
struct TSK_Attrs { /* task attributes */
  Int    priority;  /* execution priority */
  Ptr    stack;      /* pre-allocated stack */
  size_t stacksize; /* stack size in MADUs */
  Int    stackseg; /* mem seg for stack alloc */
  Ptr    environ;  /* global environ data struct */
  String name;     /* printable name */
  Bool   exitflag; /* prog termination requires */
                   /* this task to terminate */
  Bool   initstackflag;  /* initialize task stack? */
};
```

The priority attribute specifies the task's execution priority and must be less than or equal to TSK_MAXPRI (15); this attribute defaults to the value of the configuration parameter Default task priority (preset to TSK_MINPRI). If priority is less than 0, the task is barred from execution until its priority is raised at a later time by TSK_setpri. A priority value of 0 is reserved for the TSK_idle task defined in the default configuration. You should not use a priority of 0 for any other tasks.

The stack attribute specifies a pre-allocated block of stacksize MADUs to be used for the task's private stack; this attribute defaults to NULL, in which case the task's stack is automatically allocated using MEM_alloc from the memory segment given by the stackseg attribute. The stack attribute can take a value less than 0xFFFF as the stack pointer is 16 bits wide.

The stacksize attribute specifies the number of MADUs to be allocated for the task's private stack; this attribute defaults to the value of the configuration parameter Default stack size. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher priority task.

The stackseg attribute specifies the memory segment to use when allocating the task stack with MEM_alloc; this attribute defaults to the value of the configuration parameter Default stack segment.

The environ attribute specifies the task's global environment through a generic pointer that references an arbitrary application-defined data structure; this attribute defaults to NULL.

The name attribute specifies the task's printable name, which is a NULL-terminated character string; this attribute defaults to the empty string "". This name can be returned by TSK_getname.

The exitflag attribute specifies whether the task must terminate before the program as a whole can terminate; this attribute defaults to TRUE.

The initstackflag attribute specifies whether the task stack is initialized to enable stack depth checking by TSK_checkstacks. This attribute applies both in cases where the stack attribute is NULL (stack is allocated by TSK_create) and where the stack attribute is used to specify a pre-allocated stack. If your application does not call TSK_checkstacks, you can reduce the time consumed by TSK_create by setting this attribute to FALSE.

All default attribute values are contained in the constant TSK_ATTRS, which can be assigned to a variable of type TSK_Attrs prior to calling TSK_create.

A task switch occurs when calling TSK_create if the priority of the new task is greater than the priority of the current task.

TSK_create calls MEM_alloc to dynamically create an object's data structure. MEM_alloc must lock the memory before proceeding. If another thread already holds a lock to the memory, then there is a context switch. The segment from which the object is allocated is described by the DSP/BIOS objects property in the MEM Module, page 2–192.

Constraints and Calling Context

- TSK_create cannot be called from a SWI or HWI.
- The fxn parameter and the name attribute cannot be NULL.
- The priority attribute must be less than or equal to TSK_MAXPRI and greater than or equal to TSK_MINPRI. The priority can be less than zero (0) for tasks that should not execute.
- The string referenced through the name attribute cannot be allocated locally.
- The stackseg attribute must identify a valid memory segment.
- You can reduce the size of your application program by creating objects with Tconf rather than using the XXX_create functions.

See Also

MEM_alloc
SYS_error
TSK_delete
TSK_exit

TSK_delete *Delete a task*

C Interface

Syntax

TSK_delete(task);

Parameters

TSK_Handle	task;	/* task object handle */

Return Value

Void

Description

TSK_delete removes the task from all internal queues and calls MEM_free to free the task object and stack. task should be in a state that does not violate any of the listed constraints.

If all remaining tasks have their exitflag attribute set to FALSE, DSP/BIOS terminates the program as a whole by calling SYS_exit with a status code of 0.

You can use Tconf to specify an application-wide Delete function that runs whenever a task is deleted. The default Delete function is a no-op function. The Delete function is called before the task object has been removed from any internal queues and its object and stack are freed. Any DSP/BIOS function can be called from the Delete function. DSP/BIOS passes the task handle of the task being deleted to your Delete function. Your Delete function declaration should be similar to the following:

```
Void myDeleteFxn(TSK_Handle task);
```

TSK_delete calls MEM_free to delete the TSK object. MEM_free must acquire a lock to the memory before proceeding. If another task already holds a lock to the memory, then there is a context switch.

Note: Unless the mode of the deleted task is TSK_TERMINATED, TSK_delete should be called with care. For example, if the task has obtained exclusive access to a resource, deleting the task makes the resource unavailable.

Constraints and Calling Context

- The task cannot be the currently executing task (TSK_self).
- TSK_delete cannot be called from a SWI or HWI.
- No check is performed to prevent TSK_delete from being used on a statically-created object. If a program attempts to delete a task object that was created using Tconf, SYS_error is called.

See Also

MEM_free
TSK_create

TSK_deltatime *Update task statistics with time difference*

C Interface

Syntax
TSK_deltatime(task);

Parameters

TSK_Handle	task;	/* task object handle */

Return Value
Void

Description

This function accumulates the time difference from when a task is made ready to the time TSK_deltatime is called. These time differences are accumulated in the task's internal STS object and can be used to determine whether or not a task misses real-time deadlines.

If TSK_deltatime is not called by a task, its STS object is never updated in the Statistics View, even if TSK accumulators are enabled in the RTA Control Panel.

TSK statistics are handled differently than other statistics because TSK functions typically run an infinite loop that blocks when waiting for other threads. In contrast, HWI and SWI functions run to completion without blocking. Because of this difference, DSP/BIOS allows programs to identify the “beginning” of a TSK function's processing loop by calling TSK_settime and the “end” of the loop by calling TSK_deltatime.

For example, if a task waits for data and then processes the data, you want to ensure that the time from when the data is made available until the processing is complete is always less than a certain value. A loop within the task can look something like the following:

```
Void task
{
  'do some startup work'

  /* Initialize time in task's
     STS object to current time */
  TSK_settime(TSK_self());

  for (;;) {
    /* Get data */
    SIO_get(...);

    'process data'

    /* Get time difference and
       add it to task's STS object */
    TSK_deltatime(TSK_self());
  }
}
```

In the example above, the task blocks on SIO_get and the device driver posts a semaphore that readies the task. DSP/BIOS sets the task's statistics object with the current time when the semaphore becomes available and the task is made ready to run. Thus, the call to TSK_deltatime effectively measures the processing time of the task.

Constraints and Calling Context

- The results of calls to TSK_deltatime and TSK_settime are displayed in the Statistics View only if Enable TSK accumulators is selected in the RTA Control Panel.

See Also

TSK_getsts
TSK_settime

TSK_disable

Disable DSP/BIOS task scheduler

C Interface

Syntax

```
TSK_disable();
```

Parameters

Void

Return Value

Void

Description

TSK_disable disables the DSP/BIOS task scheduler. The current task continues to execute (even if a higher priority task can become ready to run) until TSK_enable is called.

TSK_disable does not disable interrupts, but is instead used before disabling interrupts to make sure a context switch to another task does not occur when interrupts are disabled.

TSK_disable maintains a count which allows nested calls to TSK_disable. Task switching is not reenabled until TSK_enable has been called as many times as TSK_disable. Calls to TSK_disable can be nested.

Since TSK_disable can prohibit ready tasks of higher priority from running it should not be used as a general means of mutual exclusion. SEM Module semaphores should be used for mutual exclusion when possible.

Constraints and Calling Context

- Do not call any function that can cause the current task to block or otherwise affect the state of the scheduler within a TSK_disable/TSK_enable block. For example, SEM_pend (if timeout is non-zero), TSK_sleep, TSK_yield, and MEM_alloc can all cause blocking. Similarly, any MEM module call and any call that dynamically creates or deletes an object (*XXX*_create or *XXX*_delete) can affect the state of the scheduler. For a complete list, see the "Possible Context Switch" column in Section A.1, *Function Callability Table*.
- TSK_disable cannot be called from a SWI or HWI.
- TSK_disable cannot be called from the program’s main() function.
- Do not call TSK_enable when TSKs are already enabled. If you do so, the subsequent call to TSK_disable will not disable TSK processing.

See Also

SEM Module
TSK_enable

TSK_enable *Enable DSP/BIOS task scheduler*

C Interface

Syntax

```
TSK_enable();
```

Parameters

Void

Return Value

Void

Description

TSK_enable is used to reenable the DSP/BIOS task scheduler after TSK_disable has been called. Since TSK_disable calls can be nested, the task scheduler is not enabled until TSK_enable is called the same number of times as TSK_disable.

A task switch occurs when calling TSK_enable only if there exists a TSK_READY task whose priority is greater than the currently executing task.

Constraints and Calling Context

- Do not call any function that can cause the current task to block or otherwise affect the state of the scheduler within a TSK_disable/TSK_enable block. For example, SEM_pend (if timeout is non-zero), TSK_sleep, TSK_yield, and MEM_alloc can all cause blocking. Similarly, any MEM module call and any call that dynamically creates or deletes an object (*XXX*_create or *XXX*_delete) can affect the state of the scheduler. For a complete list, see the "Possible Context Switch" column in Section A.1, *Function Callability Table*.
- TSK_enable cannot be called from a SWI or HWI.
- TSK_enable cannot be called from the program's main() function.
- Do not call TSK_enable when TSKs are already enabled. If you do so, the subsequent call to TSK_disable will not disable TSK processing.

See Also

SEM Module
TSK_disable

TSK_exit

Terminate execution of the current task

C Interface

Syntax
TSK_exit();

Parameters
Void

Return Value
Void

Description

TSK_exit terminates execution of the current task, changing its mode from TSK_RUNNING to TSK_TERMINATED. If all tasks have been terminated, or if all remaining tasks have their exitflag attribute set to FALSE, then DSP/BIOS terminates the program as a whole by calling the function SYS_exit with a status code of 0.

TSK_exit is automatically called whenever a task returns from its top-level function.

You can use Tconf to specify an application-wide Exit function that runs whenever a task is terminated. The default Exit function is a no-op function. The Exit function is called before the task has been blocked and marked TSK_TERMINATED. Any DSP/BIOS function can be called from an Exit function. Calling TSK_self within an Exit function returns the task being exited. Your Exit function declaration should be similar to the following:

```
Void myExitFxn(Void);
```

A task switch occurs when calling TSK_exit unless the program as a whole is terminated.

Constraints and Calling Context

- TSK_exit cannot be called from a SWI or HWI.
- TSK_exit cannot be called from the program's main() function.

See Also

MEM_free
TSK_create
TSK_delete

TSK_getenv *Get task environment pointer*

C Interface

Syntax

```
environ = TSK_getenv(task);
```

Parameters

```
TSK_Handle             task;          /* task object handle */
```

Return Value

```
Ptr                    environ;       /* task environment pointer */
```

Description

TSK_getenv returns the environment pointer of the specified task. The environment pointer, environ, references an arbitrary application-defined data structure.

If your program uses multiple HOOK objects, HOOK_getenv allows you to get environment pointers you have set for a particular HOOK and TSK object combination.

See Also

HOOK_getenv
HOOK_setenv
TSK_setenv
TSK_seterr
TSK_setpri

TSK_geterr *Get task error number*

C Interface

Syntax

```
errno = TSK_geterr(task);
```

Parameters

```
TSK_Handle          task;         /* task object handle */
```

Return Value

```
Int                 errno;        /* error number */
```

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 */

Description

TSK_getname returns the task's name.

For tasks created with Tconf, the name is available to this function only if the "Allocate Task Name on Target" property is set to true for this task. For tasks created with TSK_create, TSK_getname returns the attrs.name field value, or an empty string if this attribute was not specified.

See Also

TSK_setenv
TSK_seterr
TSK_setpri

TSK_getpri *Get task priority*

C Interface

Syntax

```
priority = TSK_getpri(task);
```

Parameters

```
TSK_Handle          task;          /* task object handle */
```

Return Value

```
Int                 priority;      /* task priority */
```

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 */
```

Description

This function provides access to the task's internal STS object. For example, you can want the program to check the maximum value to see if it has exceeded some value.

See Also

TSK_deltatime
TSK_settime

TSK_isTSK *Check to see if called in the context of a TSK*

C Interface

Syntax

```
result = TSK_isTSK(Void);
```

Parameters

Void

Return Value

```
Bool                    result;         /* TRUE if in TSK context, FALSE otherwise */
```

Reentrant

yes

Description

This macro returns TRUE when it is called within the context of a TSK or IDL function. It returns FALSE in all other contexts.

TSK_isTSK() API returns TRUE when the current thread is neither a HWI nor a SWI. Thus, TSK_isTSK() returns TRUE when it is invoked within a task thread, main(), or a task switch hook.

In previous versions of DSP/BIOS, calling the context checking functions from main() resulted in TRUE for HWI_isHWI(). And, calling the context checking functions from a task switch hook resulted in TRUE for SWI_isSWI(). This is no longer the case; they are identified as part of the TSK context.

In applications that contain no task threads, TSK_isTSK() now returns TRUE from main() and from the IDL threads.

See Also

HWI_isHWI
SWI_isSWI

TSK_itick *Advance the system alarm clock (interrupt use only)*

C Interface

Syntax

```
TSK_itick();
```

Parameters

Void

Return Value

Void

Description

TSK_itick increments the system alarm clock, and readies any tasks blocked on TSK_sleep or SEM_pend whose timeout intervals have expired.

Constraints and Calling Context

- TSK_itick cannot be called by a TSK object.
- TSK_itick cannot be called from the program's main() function.
- When called within an HWI, the code sequence calling TSK_itick must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

See Also

SEM_pend
TSK_sleep
TSK_tick

TSK_self

Returns handle to the currently executing task

C Interface

Syntax

```
curtask = TSK_self();
```

Parameters

Void

Return Value

```
TSK_Handle        curtask;      /* handle for current task object */
```

Description

TSK_self returns the object handle for the currently executing task. This function is useful when inspecting the object or when the current task changes its own priority through TSK_setpri.

No task switch occurs when calling TSK_self.

See Also

TSK_setpri

TSK_setenv *Set task environment*

C Interface

Syntax

```
TSK_setenv(task, environ);
```

Parameters

```
TSK_Handle        task;        /* task object handle */
Ptr               environ;     /* task environment pointer */
```

Return Value

Void

Description

TSK_setenv sets the task environment pointer to environ. The environment pointer, environ, references an arbitrary application-defined data structure.

If your program uses multiple HOOK objects, HOOK_setenv allows you to set individual environment pointers for each HOOK and TSK object combination.

See Also

HOOK_getenv
HOOK_setenv
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

Description

Each task carries a task-specific error number. This number is initially SYS_OK, but can be changed to errno by calling TSK_seterr. TSK_geterr returns the current value of this number.

See Also

TSK_getenv
TSK_geterr

TSK_setpri *Set a task's execution priority*

C Interface

Syntax

```
oldpri = TSK_setpri(task, newpri);
```

Parameters

```
TSK_Handle          task;       /* task object handle */
Int                 newpri;     /* task's new priority */
```

Return Value

```
Int                 oldpri;     /* task's old priority */
```

Description

TSK_setpri sets the execution priority of task to newpri, and returns that task's old priority value. Raising or lowering a task's priority does not necessarily force preemption and re-scheduling of the caller: tasks in the TSK_BLOCKED mode remain suspended despite a change in priority; and tasks in the TSK_READY mode gain control only if their (new) priority is greater than that of the currently executing task.

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

The current task can change its own priority (and possibly preempt its execution) by passing the output of TSK_self as the value of the task parameter.

A context switch occurs when calling TSK_setpri if a task makes its own priority lower than the priority of another currently ready task, or if the currently executing task makes a ready task's priority higher than its own priority. TSK_setpri can be used for mutual exclusion.

Constraints and Calling Context

- newpri must be less than or equal to TSK_MAXPRI.
- The task cannot be TSK_TERMINATED.
- The new priority should not be zero (0). This priority level is reserved for the TSK_idle task.

See Also

TSK_self
TSK_sleep

TSK_settime *Reset task statistics previous value to current time*

C Interface

Syntax
TSK_settime(task);

Parameters

TSK_Handle	task;	/* task object handle */

Return Value
Void

Description

Your application can call TSK_settime before a task enters its processing loop in order to ensure your first call to TSK_deltatime is as accurate as possible and doesn't reflect the time difference since the time the task was created. However, it is only necessary to call TSK_settime once for initialization purposes. After initialization, DSP/BIOS sets the time value of the task's STS object every time the task is made ready to run.

TSK statistics are handled differently than other statistics because TSK functions typically run an infinite loop that blocks when waiting for other threads. In contrast, HWI and SWI functions run to completion without blocking. Because of this difference, DSP/BIOS allows programs to identify the “beginning” of a TSK function's processing loop by calling TSK_settime and the “end” of the loop by calling TSK_deltatime.

For example, a loop within the task can look something like the following:

```
Void task
{
  'do some startup work'

  /* Initialize task's STS object to current time */
  TSK_settime(TSK_self());

  for (;;) {
    /* Get data */
    SIO_get(...);

    'process data'

    /* Get time difference and
       add it to task's STS object */
    TSK_deltatime(TSK_self());
  }
}
```

In the previous example, the task blocks on SIO_get and the device driver posts a semaphore that readies the task. DSP/BIOS sets the task's statistics object with the current time when the semaphore becomes available and the task is made ready to run. Thus, the call to TSK_deltatime effectively measures the processing time of the task.

Constraints and Calling Context

- TSK_settime cannot be called from the program's main() function.
- The results of calls to TSK_deltatime and TSK_settime are displayed in the Statistics View only if Enable TSK accumulators is selected within the RTA Control Panel.

See Also

TSK_deltatime
TSK_getsts

TSK_sleep *Delay execution of the current task*

C Interface

Syntax

```
TSK_sleep(nticks);
```

Parameters

```
Uns              nticks;       /* number of system clock ticks to sleep */
```

Return Value

Void

Description

TSK_sleep changes the current task's mode from TSK_RUNNING to TSK_BLOCKED, and delays its execution for nticks increments of the system clock. The actual time delayed can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

After the specified period of time has elapsed, the task reverts to the TSK_READY mode and is scheduled for execution.

A task switch always occurs when calling TSK_sleep if nticks > 0.

Constraints and Calling Context

- TSK_sleep cannot be called from a SWI or HWI, or within a TSK_disable / TSK_enable block.
- TSK_sleep cannot be called from the program's main() function.
- TSK_sleep should not be called from within an IDL function. Doing so prevents analysis tools from gathering run-time information.
- nticks cannot be SYS_FOREVER.

TSK_stat

Retrieve the status of a task

C Interface

Syntax

TSK_stat(task, statbuf);

Parameters

TSK_Handle	task;	/* task object handle */
TSK_Stat	*statbuf;	/* pointer to task status structure */

Return Value

Void

Description

TSK_stat retrieves attribute values and status information about a task.

Status information is returned through statbuf, which references a structure of type TSK_Stat defined as follows:

```
struct TSK_Stat {      /* task status structure */
    TSK_Attrs  attrs; /* task attributes */
    TSK_Mode   mode;  /* task execution mode */
    Ptr        sp;    /* task stack pointer */
    size_t     used;  /* task stack used */
};
```

When a task is preempted by a software or hardware interrupt, the task execution mode returned for that task by TSK_stat is still TSK_RUNNING because the task runs when the preemption ends.

The current task can inquire about itself by passing the output of TSK_self as the first argument to TSK_stat. However, the task stack pointer (sp) in the TSK_Stat structure is the value from the previous context switch.

TSK_stat has a non-deterministic execution time. As such, it is not recommended to call this API from SWIs or HWIs.

Constraints and Calling Context

- statbuf cannot be NULL.

See Also

TSK_create

TSK_tick

Advance the system alarm clock

C Interface

Syntax

TSK_tick();

Parameters

Void

Return Value

Void

Description

TSK_tick increments the system clock, and readies any tasks blocked on TSK_sleep or SEM_pend whose timeout intervals have expired. TSK_tick can be invoked by an HWI or by the currently executing task. The latter is particularly useful for testing timeouts in a controlled environment.

A task switch occurs when calling TSK_tick if the priority of any of the readied tasks is greater than the priority of the currently executing task.

Constraints and Calling Context

- When called within an HWI, the code sequence calling TSK_tick must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.

See Also

CLK Module
SEM_pend
TSK_itick
TSK_sleep

TSK_time *Return current value of system clock*

C Interface

Syntax

```
curtime = TSK_time();
```

Parameters

Void

Return Value

```
Uns                     curtime;        /* current time */
```

Description

TSK_time returns the current value of the system alarm clock.

Note that since the system clock is usually updated asynchronously via TSK_itick or TSK_tick, curtime can lag behind the actual system time. This lag can be even greater if a higher priority task preempts the current task between the call to TSK_time and when its return value is used. Nevertheless, TSK_time is useful for getting a rough idea of the current system time.

TSK_yield

Yield processor to equal priority task

C Interface

Syntax

TSK_yield();

Parameters

Void

Return Value

Void

Description

TSK_yield yields the processor to another task of equal priority.

A task switch occurs when you call TSK_yield if there is an equal priority task ready to run.

Tasks of higher priority preempt the currently running task without the need for a call to TSK_yield. If only lower-priority tasks are ready to run when you call TSK_yield, the current task continues to run. Control does not pass to a lower-priority task.

Constraints and Calling Context

- When called within an HWI, the code sequence calling TSK_yield must be either wrapped within an HWI_enter/HWI_exit pair or invoked by the HWI dispatcher.
- TSK_yield cannot be called from the program's main() function.

See Also

TSK_sleep

2.29 std.h and stdlib.h functions

This section contains descriptions of special utility macros found in std.h and DSP/BIOS standard library functions found in stdlib.h.

Macros

- **ArgToInt.** Cast an Arg type parameter as an integer type.
- **ArgToPtr.** Cast an Arg type parameter as a pointer type.

Functions

- **atexit.** Register an exit function.
- ***calloc.** Allocate and clear memory.
- **exit.** Call the exit functions registered by atexit.
- **free.** Free memory.
- ***getenv.** Get environmental variable.
- ***malloc.** Allocate memory.
- ***realloc.** Reallocate a memory packet.

Syntax

```
#include <std.h>
ArgToInt(arg)
ArgToPtr(arg)


#include <stdlib.h>
int  atexit(void (*fcn)(void));
void *calloc(size_t nobj, size_t size);
void exit(int status);
void free(void *p);
char *getenv(char *name);
void *malloc(size_t size);
void *realloc(void *p, size_t size);
```

Description

The DSP/BIOS library contains some C standard library functions which supersede the library functions bundled with the C compiler. These functions follow the ANSI C specification for parameters and return values. Consult Kernighan and Ritchie for a complete description of these functions.

The functions calloc, free, malloc, and realloc use MEM_alloc and MEM_free (with segid = Segment for malloc/free) to allocate and free memory.

getenv uses the _environ variable defined and initialized in the boot file to search for a matching environment string.

exit calls the exit functions registered by atexit before calling SYS_exit.

Note: RTS Functions Callable from TSK Threads Only. Many runtime support (RTS) functions use lock and unlock functions to prevent reentrancy. However, DSP/BIOS SWI and HWI threads cannot call LCK_pend and LCK_post. As a result, RTS functions that call LCK_pend or LCK_post *must not be called in the context of a SWI or HWI thread*. For a list or RTS functions that should not be called from a SWI or an HWI function, see “LCK_pend” on page 169.

To determine whether a particular RTS function uses LCK_pend, refer to the source code for that function shipped with Code Composer Studio. The following table shows some of the RTS functions that call LCK_pend in certain versions of Code Composer Studio:

fprintf	printf	vfprintf	sprintf
vprintf	vsprintf	clock	strftime
minit	malloc	realloc	free
calloc	rand	srand	getenv

The C++ new operator calls malloc, which in turn calls LCK_pend. As a result, the new operator cannot be used in the context of a SWI or HWI thread.

Function Callability and Error Tables

This appendix provides tables describing TMS320C28x errors and function callability.

A.1 Function Callability Table

The following table indicates what types of threads can call each of the DSP/BIOS functions. The Possible Context Switch column indicates whether another thread may be run as a result of this function. For example, the function may block on a resource or it may make another thread ready to run. The Possible Context Switch column does not indicate whether the function disables interrupts that might schedule higher-priority threads.

Table A-1 Function Callability

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
ATM_andi	Yes	Yes	Yes	No	Yes
ATM_andu	Yes	Yes	Yes	No	Yes
ATM_cleari	Yes	Yes	Yes	No	Yes
ATM_clearu	Yes	Yes	Yes	No	Yes
ATM_deci	Yes	Yes	Yes	No	Yes
ATM_decu	Yes	Yes	Yes	No	Yes
ATM_inci	Yes	Yes	Yes	No	Yes
ATM_incu	Yes	Yes	Yes	No	Yes
ATM_ori	Yes	Yes	Yes	No	Yes
ATM_oru	Yes	Yes	Yes	No	Yes
ATM_seti	Yes	Yes	Yes	No	Yes
ATM_setu	Yes	Yes	Yes	No	Yes
BUF_alloc	Yes	Yes	Yes	No	Yes
BUF_create	Yes	No	No	Yes	Yes
BUF_delete	Yes	No	No	Yes	Yes
BUF_free	Yes	Yes	Yes	No	Yes

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
BUF_maxbuff	Yes	No	No	No	Yes
BUF_stat	Yes	Yes	Yes	No	Yes
C28_disableIER	Yes	Yes	Yes	No	Yes
C28_enableIER	Yes	Yes	Yes	No	Yes
C28_plug	Yes	Yes	Yes	No	Yes
CLK_countspms	Yes	Yes	Yes	No	Yes
CLK_cpuCyclesPerHtime	Yes	Yes	Yes	No	Yes
CLK_cpuCyclesPerLtime	Yes	Yes	Yes	No	Yes
CLK_gethtime	Yes	Yes	Yes	No	No
CLK_getltime	Yes	Yes	Yes	No	No
CLK_getprd	Yes	Yes	Yes	No	Yes
CLK_reconfig	Yes	Yes	Yes	No	Yes
CLK_start	Yes	Yes	Yes	No	No
CLK_stop	Yes	Yes	Yes	No	No
DEV_createDevice	Yes	No	No	Yes*	Yes
DEV_deleteDevice	Yes	No	No	Yes*	Yes
DEV_match	Yes	Yes	Yes	No	Yes
GBL_getClkin	Yes	Yes	Yes	No	Yes
GBL_getFrequency	Yes	Yes	Yes	No	Yes
GBL_getProcId	Yes	Yes	Yes	No	Yes
GBL_getVersion	Yes	Yes	Yes	No	Yes
GBL_setFrequency	No	No	No	No	Yes
GBL_setProcId	No	No	No	No	No*
GIO_abort	Yes	No*	No*	Yes	No
GIO_control	Yes	No*	No*	Yes	Yes
GIO_create	Yes	No	No	No	Yes
GIO_delete	Yes	No	No	Yes	Yes
GIO_flush	Yes	No*	No*	Yes	No
GIO_new	Yes	Yes	Yes	No	Yes
GIO_read	Yes	No*	No*	Yes	Yes*
GIO_submit	Yes	Yes*	Yes*	Yes	Yes*
GIO_write	Yes	No*	No*	Yes	Yes*
HOOK_getenv	Yes	Yes	Yes	No	Yes
HOOK_setenv	Yes	Yes	Yes	No	Yes
HST_getpipe	Yes	Yes	Yes	No	Yes
HWI_disable	Yes	Yes	Yes	No	Yes
HWI_dispatchPlug	Yes	Yes	Yes	No	Yes
HWI_enable	Yes	Yes	Yes	Yes*	No
HWI_enter	No	No	Yes	No	No
HWI_exit	No	No	Yes	Yes	No

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
HWI_isHWI	Yes	Yes	Yes	No	Yes
HWI_restore	Yes	Yes	Yes	Yes*	Yes
IDL_run	Yes	No	No	No	No
LCK_create	Yes	No	No	Yes*	Yes
LCK_delete	Yes	No	No	Yes*	No
LCK_pend	Yes	No	No	Yes*	No
LCK_post	Yes	No	No	Yes*	No
LOG_disable	Yes	Yes	Yes	No	Yes
LOG_enable	Yes	Yes	Yes	No	Yes
LOG_error	Yes	Yes	Yes	No	Yes
LOG_event	Yes	Yes	Yes	No	Yes
LOG_message	Yes	Yes	Yes	No	Yes
LOG_printf	Yes	Yes	Yes	No	Yes
LOG_reset	Yes	Yes	Yes	No	Yes
MBX_create	Yes	No	No	Yes*	Yes
MBX_delete	Yes	No	No	Yes*	No
MBX_pend	Yes	Yes*	Yes*	Yes*	No
MBX_post	Yes	Yes*	Yes*	Yes*	Yes*
MEM_alloc	Yes	No	No	Yes*	Yes
MEM_calloc	Yes	No	No	Yes*	Yes
MEM_define	Yes	No	No	Yes*	Yes
MEM_free	Yes	No	No	Yes*	Yes
MEM_getBaseAddress	Yes	Yes	Yes	No	Yes
MEM_increaseTableSize	Yes	No	No	Yes*	Yes
MEM_redefine	Yes	No	No	Yes*	Yes
MEM_stat	Yes	No	No	Yes*	Yes
MEM_undefine	Yes	No	No	Yes*	Yes
MEM_valloc	Yes	No	No	Yes*	Yes
MSGQ_alloc	Yes	Yes	Yes	No	Yes
MSGQ_close	Yes	Yes	Yes	No	Yes
MSGQ_count	Yes	Yes*	Yes*	No	No
MSGQ_free	Yes	Yes	Yes	No	Yes
MSGQ_get	Yes	Yes*	Yes*	Yes*	No
MSGQ_getAttrs	Yes	Yes	Yes	No	Yes
MSGQ_getDstQueue	Yes	Yes	Yes	No	No
MSGQ_getMsgId	Yes	Yes	Yes	No	Yes
MSGQ_getMsgSize	Yes	Yes	Yes	No	Yes
MSGQ_getSrcQueue	Yes	Yes	Yes	No	No
MSGQ_isLocalQueue	Yes	Yes	Yes	No	Yes
MSGQ_locate	Yes	No	No	Yes	No

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
MSGQ_locateAsync	Yes	Yes	Yes	No	No
MSGQ_open	Yes	Yes*	Yes*	Yes*	Yes
MSGQ_put	Yes	Yes	Yes	No	No
MSGQ_release	Yes	Yes	Yes	No	No
MSGQ_setErrorHandler	Yes	Yes	Yes	No	Yes
MSGQ_setMsgId	Yes	Yes	Yes	No	Yes
MSGQ_setSrcQueue	Yes	Yes	Yes	No	Yes
PIP_alloc	Yes	Yes	Yes	Yes	Yes
PIP_free	Yes	Yes	Yes	Yes	Yes
PIP_get	Yes	Yes	Yes	Yes	Yes
PIP_getReaderAddr	Yes	Yes	Yes	No	Yes
PIP_getReaderNumFrames	Yes	Yes	Yes	No	Yes
PIP_getReaderSize	Yes	Yes	Yes	No	Yes
PIP_getWriterAddr	Yes	Yes	Yes	No	Yes
PIP_getWriterNumFrames	Yes	Yes	Yes	No	Yes
PIP_getWriterSize	Yes	Yes	Yes	No	Yes
PIP_peek	Yes	Yes	Yes	No	Yes
PIP_put	Yes	Yes	Yes	Yes	Yes
PIP_reset	Yes	Yes	Yes	Yes	Yes
PIP_setWriterSize	Yes	Yes	Yes	No	Yes
PRD_getticks	Yes	Yes	Yes	No	Yes
PRD_start	Yes	Yes	Yes	No	Yes
PRD_stop	Yes	Yes	Yes	No	Yes
PRD_tick	Yes	Yes	Yes	Yes	No
QUE_create	Yes	No	No	Yes*	Yes
QUE_delete	Yes	No	No	Yes*	Yes
QUE_dequeue	Yes	Yes	Yes	No	Yes
QUE_empty	Yes	Yes	Yes	No	Yes
QUE_enqueue	Yes	Yes	Yes	No	Yes
QUE_get	Yes	Yes	Yes	No	Yes
QUE_head	Yes	Yes	Yes	No	Yes
QUE_insert	Yes	Yes	Yes	No	Yes
QUE_new	Yes	Yes	Yes	No	Yes
QUE_next	Yes	Yes	Yes	No	Yes
QUE_prev	Yes	Yes	Yes	No	Yes
QUE_put	Yes	Yes	Yes	No	Yes
QUE_remove	Yes	Yes	Yes	No	Yes
RTDX_channelBusy	Yes	Yes	No	No	Yes
RTDX_CreateInputChannel	Yes	Yes	No	No	Yes
RTDX_CreateOutputChannel	Yes	Yes	No	No	Yes

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
RTDX_disableInput	Yes	Yes	No	No	Yes
RTDX_disableOutput	Yes	Yes	No	No	Yes
RTDX_enableInput	Yes	Yes	No	No	Yes
RTDX_enableOutput	Yes	Yes	No	No	Yes
RTDX_isInputEnabled	Yes	Yes	No	No	Yes
RTDX_isOutputEnabled	Yes	Yes	No	No	Yes
RTDX_read	Yes	Yes	No	No	No
RTDX_readNB	Yes	Yes	No	No	No
RTDX_sizeofInput	Yes	Yes	No	No	Yes
RTDX_write	Yes	Yes	No	No	No
SEM_count	Yes	Yes	Yes	No	Yes
SEM_create	Yes	No	No	Yes*	Yes
SEM_delete	Yes	Yes*	No	Yes*	No
SEM_new	Yes	Yes	Yes	No	Yes
SEM_pend	Yes	Yes*	Yes*	Yes*	No
SEM_pendBinary	Yes	Yes*	Yes*	Yes*	No
SEM_post	Yes	Yes	Yes	Yes*	Yes
SEM_postBinary	Yes	Yes	Yes	Yes*	Yes
SEM_reset	Yes	No	No	No	Yes
SIO_bufsize	Yes	Yes	Yes	No	Yes
SIO_create	Yes	No	No	Yes*	Yes
SIO_ctrl	Yes	Yes	No	No	Yes
SIO_delete	Yes	No	No	Yes*	Yes
SIO_flush	Yes	Yes*	No	No	No
SIO_get	Yes	No	No	Yes*	Yes*
SIO_idle	Yes	Yes*	No	Yes*	No
SIO_issue	Yes	Yes	No	No	Yes
SIO_put	Yes	No	No	Yes*	Yes*
SIO_ready	Yes	Yes	Yes	No	No
SIO_reclaim	Yes	Yes*	No	Yes*	Yes*
SIO_reclaimx	Yes	Yes*	No	Yes*	Yes*
SIO_segid	Yes	Yes	Yes	No	Yes
SIO_select	Yes	Yes*	No	Yes*	No
SIO_staticbuf	Yes	Yes	No	No	Yes
STS_add	Yes	Yes	Yes	No	Yes
STS_delta	Yes	Yes	Yes	No	Yes
STS_reset	Yes	Yes	Yes	No	Yes
STS_set	Yes	Yes	Yes	No	Yes
SWI_andn	Yes	Yes	Yes	Yes*	No
SWI_andnHook	Yes	Yes	Yes	Yes*	No

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
SWI_create	Yes	No	No	Yes*	Yes
SWI_dec	Yes	Yes	Yes	Yes*	No
SWI_delete	Yes	No	No	Yes*	Yes
SWI_disable	Yes	Yes	No	No	No
SWI_enable	Yes	Yes	No	Yes*	No
SWI_getattrs	Yes	Yes	Yes	No	Yes
SWI_getmbox	No	Yes	No	No	No
SWI_getpri	Yes	Yes	Yes	No	Yes
SWI_inc	Yes	Yes	Yes	Yes*	No
SWI_isSWI	Yes	Yes	Yes	No	Yes
SWI_or	Yes	Yes	Yes	Yes*	No
SWI_orHook	Yes	Yes	Yes	Yes*	No
SWI_post	Yes	Yes	Yes	Yes*	No
SWI_raisepri	No	Yes	No	No	No
SWI_restorepri	No	Yes	No	Yes	No
SWI_self	No	Yes	No	No	No
SWI_setattrs	Yes	Yes	Yes	No	Yes
SYS_abort	Yes	Yes	Yes	No	Yes
SYS_atexit	Yes	Yes	Yes	No	Yes
SYS_error	Yes	Yes	Yes	No	Yes
SYS_exit	Yes	Yes	Yes	No	Yes
SYS_printf	Yes	Yes	Yes	No	Yes
SYS_putchar	Yes	Yes	Yes	No	Yes
SYS_sprintf	Yes	Yes	Yes	No	Yes
SYS_vprintf	Yes	Yes	Yes	No	Yes
SYS_vsprintf	Yes	Yes	Yes	No	Yes
TRC_disable	Yes	Yes	Yes	No	Yes
TRC_enable	Yes	Yes	Yes	No	Yes
TRC_query	Yes	Yes	Yes	No	Yes
TSK_checkstacks	Yes	No	No	No	No
TSK_create	Yes	No	No	Yes*	Yes
TSK_delete	Yes	No	No	Yes*	No
TSK_deltatime	Yes	Yes	Yes	No	No
TSK_disable	Yes	No	No	No	No
TSK_enable	Yes	No	No	Yes*	No
TSK_exit	Yes	No	No	Yes*	No
TSK_getenv	Yes	Yes	Yes	No	Yes
TSK_geterr	Yes	Yes	Yes	No	Yes
TSK_getname	Yes	Yes	Yes	No	Yes
TSK_getpri	Yes	Yes	Yes	No	Yes

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?	Callable from main()?
TSK_getsts	Yes	Yes	Yes	No	Yes
TSK_isTSK	Yes	Yes	Yes	No	Yes
TSK_itick	No	Yes	Yes	Yes	No
TSK_self	Yes	Yes	Yes	No	No
TSK_setenv	Yes	Yes	Yes	No	Yes
TSK_seterr	Yes	Yes	Yes	No	Yes
TSK_setpri	Yes	Yes	Yes	Yes*	Yes
TSK_settime	Yes	Yes	Yes	No	No
TSK_sleep	Yes	No	No	Yes*	No
TSK_stat	Yes	Yes*	Yes*	No	Yes
TSK_tick	Yes	Yes	Yes	Yes*	No
TSK_time	Yes	Yes	Yes	No	No
TSK_yield	Yes	Yes	Yes	Yes*	No

Note: *See the appropriate API reference page for more information.

Table A-2 RTS Function Calls

Function	Callable by TSKs?	Callable by SWIs?	Callable by HWIs?	Possible Context Switch?
calloc	Yes	No	No	Yes*
clock	Yes	No	No	Yes*
fprintf	Yes	No	No	Yes*
free	Yes	No	No	Yes*
getenv	Yes	No	No	Yes*
malloc	Yes	No	No	Yes*
minit	Yes	No	No	Yes*
printf	Yes	No	No	Yes*
rand	Yes	No	No	Yes*
realloc	Yes	No	No	Yes*
sprintf	Yes	No	No	Yes*
srand	Yes	No	No	Yes*
strftime	Yes	No	No	Yes*
vfprintf	Yes	No	No	Yes*
vprintf	Yes	No	No	Yes*
vsprintf	Yes	No	No	Yes*

Note: *See Section 2.29, *std.h and stdlib.h functions*, page 2-432 for more information.

A.2 DSP/BIOS Error Codes

Table A-3 Error Codes

Name	Value	SYS_Errors[Value]
SYS_OK	0	`"(SYS_OK)”`
SYS_EALLOC	1	`"(SYS_EALLOC): segid = %d, size = %u, align = %u"` Memory allocation error.
SYS_EFREE	2	`"(SYS_EFREE): segid = %d, ptr = ox%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.
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_ENOTIMPL	13	`"(SYS_ENOTIMPL): action not implemented"` An attempt was made to use an action that is not implemented.
SYS_ENOTFOUND	14	`"(SYS_ENOTFOUND): resource not found"` An attempt was made to use a resource that could not be found.
SYS_EUSER	>=256	`"(SYS EUSER): <user-defined string>"` User-defined error.

Appendix B

SPRU625L—August 2012

C28x DSP/BIOS Register Usage

This appendix provides tables describing the TMS320C28x^{TM} register conventions in terms of preservation across multi-threaded context switching and preconditions.

Topic **Page**

B.1 Overview

In a multi-threaded application using DSP/BIOS, it is necessary to know which registers can or cannot be modified. Furthermore, users need to understand which registers need to be saved/restored across a function call or an interrupt.

The following definitions describe the various possible register handling behaviors:

- **Scratch register.** These registers are saved/restored by the HWI dispatcher or HWI_enter/HWI_exit with temporary register bit masks.
- **Preserved register.** These registers are saved/restored during a TSK context switch.
- **Initialized register.** These registers are set to a particular value during HWI processing and restored to their incoming value upon exiting to the interrupt routine.
- **Read-Only register.** These registers may be read but must not be modified.
- **Global register.** These registers are shared across all threads in the system. To make a temporary change, save the register, make the change, and then restore it.
- **Don't care.** These registers are not changed by DSP/BIOS. You can use them freely.
- **Other.** These registers do not fit into one of the categories above.

B.2 Register Conventions

Table 2–12. Register and Status Bit Handling

Register	Status Bit	Register or Status Bit Name	Type	Notes
ACC (AL, AH)		Accumulator	Scratch	Hardware saves during ISR
XAR0 (AR0, AR0H)		Auxiliary register 0	Scratch	Hardware saves AR0 during ISR
XAR1 (AR1, AR1H)		Auxiliary register 1	Preserved	Hardware saves AR1 during ISR
XAR2 (AR2, AR2H)		Auxiliary register 2	Preserved	
XAR3 (AR3, AR3H)		Auxiliary register 3	Preserved	
XAR4 (AR4, AR4H)		Auxiliary register 4	Scratch	
XAR5 (AR5, AR5H)		Auxiliary register 5	Scratch	
XAR6 (AR6, AR6H)		Auxiliary register 6	Scratch	
XAR7 (AR7, AR7H)		Auxiliary register 7	Scratch	
DP		Data-page pointer	Don't care	Hardware saves during ISR, DSP/BIOS not used
IFR		Interrupt flag register	Don't care	DSP/BIOS not used
IER		Interrupt enable register	Don't care	Hardware saves during ISR
DBGIER		Debug interrupt enable register	Don't care	
DBGSTAT		Debug status register	Read-Only	Hardware saves during ISR
P (PL, PH)		Product register	Scratch	Hardware saves during ISR
RPC		Return program counter	Preserved	
SP		Stack pointer	Initialized	HWI sets to HWI stack before calling ISR
ST0		Status register 0	Scratch	Hardware saves during ISR
	OVC/OVCU	Overflow counter	Don't care	
	PM	Product shift mode bits	Initialized (001 No shift)	
	V	Overflow flag	Don't care	
	N	Negative flag	Don't care	
	Z	Zero flag	Don't care	
	C	Carry bit	Don't care	
	TC	Test/control flag	Don't care	

Register	Status Bit	Register or Status Bit Name	Type	Notes
	OVM	Overflow mode bit	Initialized (0: Normal)	
	SXM	Sign-extension mode bit	Don't care	
ST1		Status register 1	Scratch	Hardware saves during ISR
	ARP	Auxiliary register pointer	Don't care	
	XF	XF status bit	Other	Do not use with DSP/BIOS. Use GPIO instead
	M0M1MAP	M0 and M1 mapping mode bit	Read-Only (1: 28x mode)	DSP/BIOS not support ‘C27x , DSP/BIOS initialized during boot process
	OBJMODE	Object compatibility mode bit	Read-Only (1: 28x mode)	DSP/BIOS not support ‘C27x, DSP/BIOS initialized during boot process
	AMODE	Address mode bit	Initialized (0: C28ADDR)	
	IDLESTAT	IDLE status bit	Read-Only	
	EALLOW	Emulation access enable bit	Don't care	
	LOOP	Loop instruction status bit	Don't care	
	SPA	Stack pointer alignment bit	Preserved	
	VMAP	Vector map bit	Read-Only	DSP/BIOS initialized to configured value during boot process
	PAGE0	PAGE0 addressing mode configuration bit	Initialized (0: Stack addressing)	
	DBGM	Debug enable mask bit	Don't care	
	INTM	Interrupt global mask bit	Don't care	
XT (T, TL)		Multiplicand register	Scratch	Hardware saves T during ISR

Appendix C

SPRU625L—August 2012

C28x Real-Time Mode Emulation

This appendix describes DSP/BIOS support for ’C28x real-time mode.

C.1 Real-Time Mode Background

The ’C28x provides for debugging in real-time mode. In this mode, time-critical interrupts (also called foreground code) continue to be serviced while non-critical code (also call background code) is halted as usual at breakpoints.

This mode is intended for use in applications that run critical tasks (such as driving motors) while other tasks have lesser importance. In order to debug such applications, developers often want to be able to leave the critical portion running while debugging other parts of the application.

For information about this real-time mode, see Chapter 7 of the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430).

C.2 Related Configuration Properties

To identify an interrupt as time-critical, you enable that interrupt in the debug interrupt enable register (DBGIER) and the interrupt enable register (IER). While in real-time mode, the debug enable mask bit (DBGM) enables or disables time-critical interrupts; the interrupt global mask bit (INTM) is ignored. The NMI and RS interrupts are always considered time-critical, and are always serviced once requested.

If you use the DSP/BIOS timer interrupt as a time-critical interrupt, you should set the CLK Module property “Continue to run on SW breakpoint” (FREERUN) to true.

Do not set the HWI Module property “Interrupt Mask” (interruptMask) to block time-critical interrupts in the HWI dispatcher. If you use HWI_enter and HWI_exit instead of the HWI dispatcher, you should likewise not block time-critical interrupts.

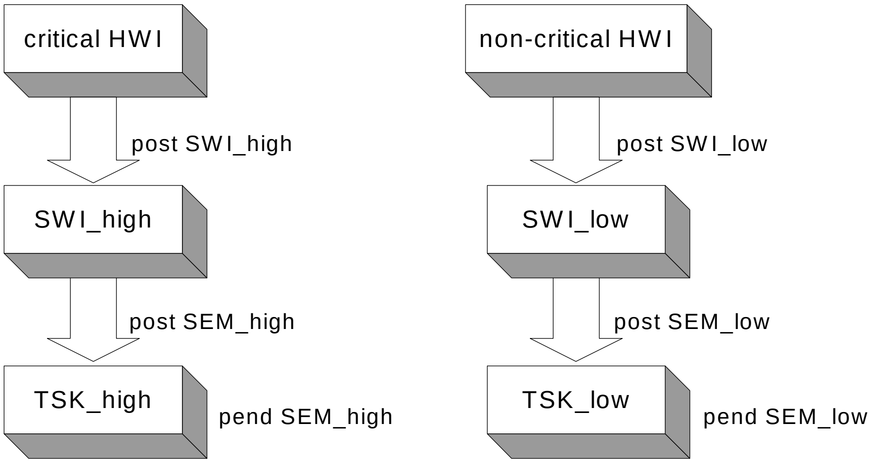
Multiple time-critical interrupts can occur and be serviced while the device is otherwise halted while debugging.

C.3 Thread Interaction Issues

There are thread interaction issues in a DSP/BIOS application that you need to be aware of if you identify any interrupt functions as time-critical.

When a breakpoint occurs, a time-critical HWI function continues running unless the breakpoint was within the code run by that time-critical HWI. Whether any SWIs or TSKs posted by the time-critical HWI continue to run depends upon where the breakpoint occurred.

To illustrate the possible results, suppose we have an application designed as follows:



In this example, the time-critical HWI posts a high-priority SWI, which posts a semaphore to allow a high-priority TSK to run. The non-critical HWI posts a low-priority SWI, which posts a semaphore to allow a low-priority TSK to run.

The following table shows which threads can run depending on where a breakpoint occurs. All other threads are halted and behave as usual at breakpoints.

Table C-1 Effects of Breakpoint Location on Thread Running

Breakpoint Location	Which Threads Continue Running
critical HWI	none
SWI_high	critical HWI
TSK_high	critical HWI, SWI_high
non-critical HWI	critical HWI
SWI_low	critical HWI, SWI_high
TSK_low	critical HWI, SWI_high, TSK_high
SWI scheduler code	critical HWI

For example, if a breakpoint occurs in SWI_low, the TSK_high function cannot run because SWI_low has higher priority than TSK_high.

When determining which threads continue running, recall that PRD threads run as SWI threads, and IDL threads run as TSK threads.

Index

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its semiconductor products and services per JESD46C and to discontinue any product or service per JESD48B. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

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 relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components which meet ISO/TS16949 requirements, mainly for automotive use. Components which have not been so designated are neither designed nor intended for automotive use; and TI will not be responsible for any failure of such components to meet such requirements.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303 Dallas, Texas 75265