TMS320C54x DSP/BIOS Application Programming Interface (API) Reference Guide

Literature Number: SPRU404A
May 2000

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

Preface

Read This First

About This Manual

DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320C54x DSP chips the ability to develop embedded real-time software. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

You should read and become familiar with the TMS320C54x DSP/BIOS User's Guide (literature number SPRU326C), the companion volume to this reference guide.

Before you read this manual, you should follow the tutorials in the TMS320C54x Code Composer Studio Tutorial (literature number SPRU327C) to get an overview of DSP/BIOS. This manual discusses various aspects of DSP/BIOS in depth and assumes that you have at least a basic understanding of other aspects of DSP/BIOS.

Notational Conventions

This document uses the following conventions:

- The TMS320C54x core is also referred to as 'C5000.
- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

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

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

Related Documentation From Texas Instruments

The following books describe the TMS320C54x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

TMS320C54x DSP/BIOS User's Guide (literature number SPRU326b) gives developers of applications for DSP chips the information necessary to develop and analyze embedded real-time software using the DSP/BIOS software. The ***User's Guide*** is the companion to this ***API Reference Guide***.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C5000 generation of devices.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C5000 C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C5000 generation of devices.

TMS320C54x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the 'C5000. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

TMS320C54x Evaluation Module Technical Reference (literature number SPRU135) describes the 'C5000 evaluation module, its features, design details and external interfaces.

TMS320C54x Simulator Getting Started Guide (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the 'C5000. The installation for Windows 3.1, SunOS™, and HP-UX™ systems is covered.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C5000 devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x Simulator Addendum (literature number SPRU170) tells you how to define and use a memory map to simulate ports for the 'C5000. This addendum to the *TMS320C5xx C Source Debugger User's Guide* discusses standard serial ports, buffered serial ports, and time division multiplexed (TDM) serial ports.

TMS320C5x C Source Debugger User's Guide (literature number SPRU055) tells you how to invoke the 'C5x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C5000 emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Composer Studio Tutorial (literature number SPRU327a) introduces the Code Composer Studio integrated development environment and software tools.

Related Documentation

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

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

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

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

Trademarks

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

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, XDS, Code Composer, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, BIOSuite, and SPOX.

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

Contents

Chapter 1

API Functions

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

Topic Page

1.1 DSP/BIOS Modules

The DSP/BIOS modules are:

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

1.2 Naming Conventions

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

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

1.3 List of Operations

The DSP/BIOS operations are:

Function	Operation
ATM_andi, ATM_andu	Atomically AND two memory locations and return previous value of the second
ATM_cleari, ATM_clearu	Atomically clear memory location and return previous value
ATM_deci, ATM_decu	Atomically decrement memory and return new value
ATM_inci, ATM_incu	Atomically increment memory and return new value
ATM_ori, ATM_oru	Atomically OR memory location and return previous value
ATM_seti, ATM_setu	Atomically set memory and return previous value
C54_disableIMR	Disable certain maskable interrupts
C54_enableIMR	Enable certain maskable interrupts
C54_plug	C function to plug an interrupt vector
CLK_countspms	Number of hardware timer counts per millisecond
CLK_gethtime	Get high-resolution time
CLK_getltime	Get low-resolution time
CLK_getprd	Get period register value

Function	Operation
DEV_match	Match a device name with a driver
Dxx_close	Close device
Dxx_ctrl	Device control operation
Dxx_idle	Idle device
Dxx_init	Initialize device
Dxx_issue	Send a buffer to the device
Dxx_open	Open device
Dxx_ready	Check if device is ready for I/O
Dxx_reclaim	Retrieve a buffer from a device
HST_getpipe	Get corresponding pipe object
HWI_disable	Globally disable hardware interrupts
HWI_enable	Globally enable hardware interrupts
HWI_enter	Hardware interrupt service routine prolog
HWI_exit	Hardware interrupt service routine epilog
HWI_restore	Restore global interrupt enable state
IDL_run	Make one pass through idle functions
LCK_create	Create a resource lock
LCK_delete	Delete a resource lock
LCK_pend	Acquire ownership of a resource lock
LCK_post	Relinquish ownership of a resource lock
LOG_disable	Disable a log
LOG_enable	Enable a log

Function	Operation
LOG_error/LOG_message	Write a message to the system log
LOG_event	Append an unformatted message to a log
LOG_printf	Append a formatted message to a message log
LOG_reset	Reset a log
MBX_create	Create a mailbox
MBX_delete	Delete a mailbox
MBX_pend	Wait for a message from mailbox
MBX_post	Post a message to mailbox
MEM_alloc, MEM_valloc, MEM_calloc	Allocate from a memory section
MEM_define	Define a new memory section
MEM_free	Free a block of memory
MEM_redefine	Redefine an existing memory section
MEM_stat	Return the status of a memory section
PIP_alloc	Get an empty frame from a pipe
PIP_free	Recycle a frame that has been read back into a pipe
PIP_get	Get a full frame from a pipe
PIP_getReaderAddr	Get the value of the readerAddr pointer of the pipe
PIP_getReaderNumFrames	Get the number of pipe frames available for reading
PIP_getReaderSize	Get the number of words of data in a pipe frame
PIP_getWriterAddr	Get the value of the writerAddr pointer of the pipe
PIP_getWriterNumFrames	Get the number of pipe frames available to be written to
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

Function	Operation
PIP_put	Put a full frame into a pipe
PIP_reset	Reset all fields of a pipe object to their original values
PIP_setWriterSize	Set the number of valid words written to a pipe frame
PRD_getticks	Get the current tick counter
PRD_start	Arm a periodic function for one-time execution
PRD_stop	Stop a periodic function from execution
PRD_tick	Advance tick counter, dispatch periodic functions
QUE_create	Create an empty queue
QUE_delete	Delete an empty queue
QUE_dequeue	Remove from front of queue (non-atomically)
QUE_empty	Test for an empty queue
QUE_enqueue	Insert at end of queue (non-atomically)
QUE_get	Get element from front of queue (atomically)
QUE_head	Return element at front of queue
QUE_insert	Insert in middle of queue (non-atomically)
QUE_new	Set a queue to be empty
QUE_next	Return next element in queue (non-atomically)
QUE_prev	Return previous element in queue (non-atomically)
QUE_put	Put element at end of queue (atomically)
QUE_remove	Remove from middle of queue (non-atomically)
RTDX_channelBusy	Return status indicating whether a channel is busy
RTDX_disableInput	Disable an input channel
RTDX_disableOutput	Disable an output channel

Function	Operation
RTDX_enableInput	Enable an input channel
RTDX_enableOutput	Enable an output channel
RTDX_read	Read from an input channel
RTDX_readNB	Read from an input channel without blocking
RTDX_sizeofInput	Return the number of bytes read from an input channel
RTDX_write	Write to an output channel
SEM_count	Get current semaphore count
SEM_create	Create a semaphore
SEM_delete	Delete a semaphore
SEM_ipost	Signal a semaphore (interrupt only)
SEM_new	Initialize a semaphore
SEM_pend	Wait for a semaphore
SEM_post	Signal a semaphore
SEM_reset	Reset semaphore
SIO_bufsize	Size of the buffers used by a stream
SIO_create	Create stream
SIO_ctrl	Perform a device-dependent control operation
SIO_delete	Delete stream
SIO_flush	Idle a stream by flushing buffers
SIO_get	Get buffer from stream
SIO_idle	Idle a stream
SIO_issue	Send a buffer to a stream
SIO_put	Put buffer to a stream
SIO_reclaim	Request a buffer back from a stream

Function	Operation
SIO_segid	Memory section used by a stream
SIO_select	Select a ready device
SIO_staticbuf	Acquire static buffer from stream
STS_add	Add a value to a statistics object
STS_delta	Add computed value of an interval to object
STS_reset	Reset the values stored in an STS object
STS_set	Store initial value of an interval to object
SWI_andn	Clear bits from SWI's mailbox and post if becomes 0
SWI_create	Create a software interrupt
SWI_dec	Decrement SWI's mailbox and post if becomes 0
SWI_delete	Delete a software interrupt
SWI_disable	Disable software interrupts
SWI_enable	Enable software interrupts
SWI_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_or	Set or mask in an SWI's mailbox and post
SWI_post	Post a software interrupt
SWI_raisepri	Raise an SWI's priority
SWI_restorepri	Restore an SWI's priority
SWI_self	Return address of currently executing SWI object
SWI_setattrs	Set attributes of a software interrupt
SYS_abort	Abort program execution

Function	Operation
SYS_atexit	Stack an exit handler
SYS_error	Flag error condition
SYS_exit	Terminate program execution
SYS_printf, SYS_sprintf, SYS_vprintf, SYS_vsprintf	Formatted output
SYS_putchar	Output a single character
TRC_disable	Disable a set of trace controls
TRC_enable	Enable a set of trace controls
TRC_query	Test whether a set of trace controls is enabled
TSK_checkstacks	Check for stack overflow
TSK_create	Create a task ready for execution
TSK_delete	Delete a task
TSK_deltatime	Update task STS with time difference
TSK_disable	Disable DSP/BIOS task scheduler
TSK_enable	Enable DSP/BIOS task scheduler
TSK_exit	Terminate execution of the current task
TSK_getenv	Get task environment
TSK_geterr	Get task error number
TSK_getname	Get task name
TSK_getpri	Get task priority
TSK_getsts	Get task STS object
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

Function	Operation
TSK_setpri	Set a task execution priority
TSK_settime	Set task STS previous time
TSK_sleep	Delay execution of the current task
TSK_stat	Retrieve the status of a task
TSK_tick	Advance system alarm clock
TSK_time	Return current value of system clock
TSK_yield	Yield processor to equal priority task

1.4 Assembly Language Interface

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

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

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

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

The Postconditions section lists registers set by the macro.

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

Example:

Assembly Interface

Syntax HWI_enter MASK IMRDISABLEMASK

Preconditions intm = 1

Postconditions dp = GBL_A_SYSPAGE
cpl = ovm = c16 = frct = cmpt = 0

Modifies c, cpl, dp, sp

Reentrant yes

Assembly functions can call C functions. Remember that the C compiler adds an underscore prefix to function names, so when calling a C function from assembly, add an underscore to the beginning of the C function name. For example, call _myfunction instead of myfunction. See the "*TMS320C5400 Optimizing C Compiler User's Guide*" for more details.

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

ATM Module

Atomic functions written in assembly language

Functions

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

Description

ATM provides a set of atomic functions which are used to manipulate variables with interrupts disabled. These functions may therefore be used on data shared between tasks, and on data shared between tasks and interrupt routines.

ATM_andi

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

C Interface

Syntax

ival = ATM_andi(idst, isrc);

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst;

*dst = val & src;

‘interrupt enable‘

return(val);
```

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

See Also

ATM_andu
ATM_ori

ATM_andu

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

C Interface

Syntax

```
uval = ATM_andu(udst, usrc);
```

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

```
'interrupt disable'

val = *dst;

*dst = val & src;

'interrupt enable'

return(val);
```

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

See Also

ATM_andi
ATM_oru

ATM_cleari

Atomically clear Int memory location and return previous value

C Interface

Syntax ival = ATM_cleari(idst);

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

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

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst;
*dst = 0;

‘interrupt enable‘

return (val);
```

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

See Also

ATM_clearu
ATM_seti

ATM_clearu

Atomically clear Uns memory location and return previous value

C Interface

Syntax uval = ATM_clearu(udst);

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

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

Assembly Interface none

Description

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

```
'interrupt disable'

val = *dst;
*dst = 0;

'interrupt enable'

return (val);
```

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

See Also

ATM_clearu
ATM_setu

ATM_deci

Atomically decrement Int memory and return new value

C Interface

Syntax ival = ATM_deci(idst);

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

Return Value Int ival; /* new value after decrement */

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst - 1;

*dst = val;

‘interrupt enable‘

return (val);
```

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

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

See Also

ATM_decu
ATM_inci

ATM_decu

Atomically decrement Uns memory and return new value

C Interface

Syntax uval = ATM_decu(udst);

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

Return Value Uns uval; /* new value after decrement */

Assembly Interface none

Description

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

```
'interrupt disable'

val = *dst - 1;

*dst = val;

'interrupt enable'

return (val);
```

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

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

See Also

ATM_deci
ATM_incu

ATM_inci

Atomically increment Int memory and return new value

C Interface

Syntax ival = ATM_inci(idst);

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

Return Value Int ival; /* new value after increment */

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst + 1;

*dst = val;

‘interrupt enable‘

return (val);
```

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

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

See Also

ATM_deci
ATM_incu

ATM_incu *Atomically increment Uns memory and return new value*

C Interface

Syntax uval = ATM_incu(udst);

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

Return Value Uns uval; /* new value after increment */

Assembly Interface none

Description

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

```
‘interrupt disable‘

val = *dst + 1;

*dst = val;

‘interrupt enable‘

return (val);
```

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

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

See Also

ATM_decu
ATM_inci

ATM_ori

Atomically OR Int memory location and return previous value

C Interface

Syntax

```
ival = ATM_ori(idst, isrc);
```

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst;

*dst = val | src;

‘interrupt enable‘

return(val);
```

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

See Also

ATM_andi
ATM_oru

ATM_oru

Atomically OR Uns memory location and return previous value

C Interface

Syntax

uval = ATM_oru(udst, usrc);

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

```
'interrupt disable'

val = *dst;

*dst = val | src;

'interrupt enable'

return(val);
```

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

See Also

ATM_andu
ATM_ori

ATM_seti

Atomically set Int memory and return previous value

C Interface

Syntax iold = ATM_seti(idst, inew);

Parameters

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

Return Value

```
Int         iold;    /* previous value of *idst */
```

Assembly Interface

none

Description

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

```
‘interrupt disable‘

val = *dst;

*dst = new;

‘interrupt enable‘

return (val);
```

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

See Also

ATM_setu
ATM_cleari

ATM_setu

Atomically set Uns memory and return previous value

C Interface

Syntax uold = ATM_setu(udst, unew);

Parameters

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

Return Value

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

Assembly Interface none

Description

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

```
'interrupt disable'

val = *dst;

*dst = new;

'interrupt enable'

return (val);
```

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

See Also

ATM_clearu
ATM_setu

C54 Module

Target-specific functions for the TMS320C54x family

Functions

- C54_disableIMR. ASM macro to disable selected interrupts in the IMR
- C54_enableIMR. ASM macro to enable selected interrupts in the IMR
- C54_plug. Plug interrupt vector

Description

The C54 module provides certain target-specific functions and definitions for the TMS320C54x family of processors. See c54.h for the complete list of definitions for hardware flags for C. c54.h contains C language macros, #defines for various TMS320C54x registers, and structure definitions.

c54.h54 contains assembly language macros for saving and restoring registers in interrupt service routines.

C54_disableIMR

Disable certain maskable interrupts

C Interface

Syntax

```
oldmask = C54_disableIMR(mask); /
```

Parameters

```
Uns  mask;     /* disable mask */
```

Return Value

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

Assembly Interface

Syntax

```
C54_disableIMR IEMASK, REG0, REG1
```

Parameters

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

Return Value

none

Description

C54_disableIMR disables interrupts by clearing the bits specified by mask in the Interrupt Mask Register (IMR).

The C version of C54_disableIMR returns a mask of bits actually cleared. This return value should be passed to C54_enableIMR to re-enable interrupts.

See C54_enableIMR for a description and code examples for safely protecting a critical section of code from interrupts.

See Also

C54_enableIMR

C54_enableIMR

Enable certain maskable interrupts

C Interface

Syntax C54_enableIMR(oldmask);

Parameters

```
Uns    oldmask;    /* enable mask */
```

Return Value Void

Assembly Interface

Syntax C54_enableIMR IEMASK, REG0, REG1

Parameters

```
IEMASK  ; interrupt enable mask
REG0    ; temporary register that can be modified
REG1    ; temporary register that can be modified
```

Return Value none

Description

C54_disableIMR and C54_enableIMR are used to disable and enable specific internal interrupts by modifying the Interrupt Mask Register (IMR). C54_disableIMR clears the bits specified by the mask parameter in the IMR and returns a mask of the bits it cleared. C54_enableIMR sets the bits specified by the oldmask parameter in the IMR.

C54_disableIMR and C54_enableIMR are usually used in tandem to protect a critical section of code from interrupts. The following code examples show a region protected from all interrupts:

```
; ASM example

.include c54.h54
...

; disable interrupts specified by IEMASK
C54_disableIMR IEMASK, b0, b1

‘do some critical operation‘

; enable interrupts specified by IEMASK
C54_enableIMR IEMASK, b0, b1

/* C example */
Uns    oldmask;

oldmask = C54_disableIMR(~0);
  ‘do some critical operation; ‘
  ‘do not call TSK_sleep(), SEM_post(), etc.‘
C54_enableIMR(oldmask);
```

Note:

DSP/BIOS kernel calls that may cause rescheduling of tasks (e.g., SEM_post, TSK_sleep) should be avoided within a C54_disableIR / C54_enableIR block since the interrupts may be disabled for an indeterminate amount of time if a task switch occurs.

Alternatively, you may disable DSP/BIOS task scheduling for this block by enclosing it with TSK_disable / TSK_enable. You may also use C54_disableIMR / C54_enableIMR to disable selected interrupts, allowing other interrupts to occur. However, if another ISR does occur during this region, it could cause a task switch. You can prevent this by using TSK_disable / TSK_enable around the entire region:

```
Uns    oldmask;

TSK_disable();
oldmask = C54_disableIMR(INTMASK);
  ‘do some critical operation;‘
  ‘OK to be interrupted or call TSK_sleep(), SEM_post(), etc.‘
C54_enableIMR(oldmask);
TSK_enable();
```

Note:

If you use C54_disableIMR / C54_enableIMR to disable only some interrupts, you must surround this region with TSK_disable / TSK_enable, to prevent an intervening ISR from causing a task switch.

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

See Also

C54_disableIMR

C54_plug

C function to plug an interrupt vector

C Interface

Syntax

```
C54_plug(vecid, fxn);
```

Parameters

```
Int   vecid;    /* interrupt id */
Fxn   fxn;      /* pointer to ISR */
```

Return Value

Void

Assembly Interface

none

Description

C54_plug writes an branch vector into the interrupt vector table, at the address corresponding to vecid. The op-codes written in the branch vector create a branch to the function entry point specified by fxn:

```
b       fxn
```

C54_plug does not enable the interrupt. Use C54_enableIMR to enable specific interrupts.

Constraints and Calling Context

- ❑ vecid must be a valid interrupt ID in the range of 2-32.

See Also

C54_enableIMR

CLK Module

System clock manager

Functions

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

Description

The CLK module provides a method for invoking functions periodically.

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

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

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

```
CLKOUT / (TDDR + 1)
```

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

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

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

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

rate, the 16-bit timer counter register wraps around quite fast. The 32-bit high-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding the difference between the value in the period register and the current value of the timer counter register. You can use the CLK_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 CLK functions performed when a timer interrupt occurs are performed in the context of the hardware interrupt that caused the system clock to tick. Therefore, the amount of processing performed within CLK functions should be minimized and these functions may only invoke DSP/BIOS calls that are allowable from within a hardware ISR. (They should not call HWI_enter and HWI_exit as these are called internally before and after CLK functions.)

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

CLK Manager Properties

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

- **Object Memory**. The memory section that contains the CLK objects created with the Configuration Tool.
- **Enable CLK Manager**. If checked, the on-chip timer hardware is used to drive the high- and low-resolution times and to trigger execution of CLK functions.
- **Use high resolution time for internal timings**. If checked, the high-resolution timer is used to monitor internal periods; otherwise the less intrusive, low-resolution timer is used.
- **Microseconds/Int**. The number of microseconds between timer interrupts. The period register is set to a value that achieves the desired period as closely as possible.
- **Directly configure on-chip timer registers**. If checked, the timer's hardware registers, PRD and TDDR, can be directly set to the desired values. In this case, the Microseconds/Int field is computed based on the values in PRD and TDDR and the CPU clock speed.
- **Fix TDDR**. If checked, the value in the TDDR field is not modified by changes to the Microseconds/Int field.
- **TDDR Register**. The on-chip timer divide-down register.
- **PRD Register**. The on-chip timer period register.

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

- **Instructions/Int**. The number of instruction cycles represented by the period specified above.

CLK Object Properties

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

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

- **comment**. Type a comment to identify this CLK object.
- **function**. The function to be executed when the timer hardware interrupt occurs. This function must be written like an HWI function; it must be written in assembly and must save and restore any registers this function modifies. However, this function may not call HWI_enter or HWI_exit because DSP/BIOS calls them internally before and after this function runs.
  These functions should be very short as they are performed frequently. Since all functions are performed using the same periodic rate, functions that need to occur at a multiple of that rate should count the number of interrupts and perform their activities when the counter reaches the appropriate value.
  If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)

CLK - Code Composer Studio Interface

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

CLK_countspms

Number of hardware timer counts per millisecond

C Interface

Syntax ncounts = CLK_countspms();

Parameters Void

Return Value Uns ncounts;

Assembly Interface

Syntax CLK_countspms

Preconditions none

Postconditions a

Modifies ag, ah, al, c

Reentrant yes

Description

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

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

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

See Also

CLK_gethtime
CLK_getprd
STS_delta

CLK_gethtime

Get high-resolution time

C Interface

Syntax

```
currtime = CLK_gethtim();
```

Parameters

Void

Return Value

```
LgUns currtime      /* high-resolution time */
```

Assembly Interface

Syntax

```
CLK_gethtime
```

Preconditions

intm = 1
cpl = ovm = c16 = frct = cmpt = 0

Postconditions

ah = bits 32 - 16 of high-resolution time
al = bits 15 - 0 of high-resolution time

Modifies

ag, ah, al, ar5, bg, bh, bl, c, dp, t, tc

Reentrant

no

Description

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

High-resolution time is the number of times the timer counter register has been decremented. When the CLK manager is enabled in the Configuration Tool, the time counter is decremented at the following rate, where CLKOUT is the DSP clock speed in MIPS (see the Global Settings Property dialog) and TDDR is the value of the timer divide-down register (see the CLK Manager Property dialog):

```
CLKOUT / (TDDR + 1)
```

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

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

High-resolution time is actually calculated by multiplying the low-resolution time by the value of the period register property and adding the current value

of the timer counter. Although the CLK_gethtime uses the period register value to calculate the high-resolution time, the value of the high-resolution time is independent of the actual value in the period register. This is because the timer counter register is divided by the period register value when incrementing the low-resolution time, and the result is multiplied by the same period register value to calculate the low-resolution time.

CLK_gethtime provides a value with greater accuracy than CLK_getltime, but which wraps back to 0 more frequently. For example, if the chip's clock rate is 200 MHz, then regardless of the period register value, the CLK_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.

Example

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

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

Assembly Interface

Syntax

```
CLK_getltime
```

Preconditions

none

Postconditions

ah = bits 32 - 16 of low-resolution time
al = bits 15 - 0 of low-resolution time

Modifies

ag, ah, al, c

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.

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

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

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

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

Therefore, CLK_gethtime provides a value with greater accuracy than CLK_getltime, but which wraps back to 0 more frequently. For example, if the chip's clock 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.

Example

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

   Void showTicks()
   {
      LOG_printf(&trace, "time = %d", (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 */

Assembly Interface

Syntax CLK_getprd

Preconditions none

Postconditions a

Modifies ag, ah, al, c

Reentrant yes

Description

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

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

See Also

CLK_countspms
CLK_gethtime
STS_delta

DEV Module

Device driver interface

Functions

- ❏ DEV_match. Match device name with driver
- ❏ Dxx_close. Close device
- ❏ Dxx_ctrl. Device control
- ❏ Dxx_idle. Idle device
- ❏ Dxx_init. Initialize device
- ❏ Dxx_issue. Send frame to device
- ❏ Dxx_open. Open device
- ❏ Dxx_ready. Device ready
- ❏ Dxx_reclaim. Retrieve frame from device

Constants, Types, and Structures

```
#define DEV_INPUT       0
#define DEV_OUTPUT      1

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

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

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

```
} DEV_Fxns;

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

Description

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

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

UDEV Manager Properties

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

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

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

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

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

UDEV Object Properties

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

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

DEV_match

Match a device name with a driver

C Interface

Syntax

substr = DEV_match(name, device);

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

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

See Also

SIO_create

Dxx_close

Close device

C Interface

Syntax status = Dxx_close(device);

Parameters DEV_Handle device; /* device handle */

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

Assembly Interface none

Description

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

SIO_delete calls Dxx_idle to idle the device before calling Dxx_close.

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

Constraints and Calling Context

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

See Also

Dxx_idle
Dxx_open
SIO_delete

Dxx_ctrl

Device control operation

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

Dxx_ctrl performs a control operation on the device associated with device and returns an error code indicating success (SYS_OK) or failure. The actual control operation is designated through cmd and arg, which are interpreted in a driver-dependent manner.

Dxx_ctrl is called by SIO_ctrl to send control commands to a device.

Constraints and Calling Context

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

See Also

SIO_ctrl

Dxx_idle

Idle device

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

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

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

Constraints and Calling Context

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

See Also

SIO_delete
SIO_idle
SIO_flush

Dxx_init

Initialize device

C Interface

Syntax Dxx_init();

Parameters Void

Return Value Void

Assembly Interface none

Description

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

Dxx_init is called at system startup, before the application's main function is called.

Dxx_issue

Send a buffer to the device

C Interface

Syntax status = Dxx_issue(device);

Parameters DEV_Handle device; /* device handle */

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

Assembly Interface none

Description

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

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

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

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

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

Dxx_issue must preserve and maintain buffers sent to the device so they can be returned in the order they were received, by a call to Dxx_reclaim.

Constraints and Calling Context

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

See Also

Dxx_reclaim
SIO_issue

Dxx_open

Open device

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

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

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

See Also

Dxx_close
SIO_create

Dxx_ready

Check if device is ready for I/O

C Interface

Syntax

status = Dxx_ready(device, sem);

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

If the device is an input device that is not started, Dxx_ready starts the device.

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

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

See Also

SIO_select

Dxx_reclaim

Retrieve a buffer from a device

C Interface

Syntax status = Dxx_reclaim(device);

Parameters DEV_Handle device; /* device handle */

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

Assembly Interface none

Description

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

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

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

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

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

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

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

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

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

Constraints and Calling Context

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

See Also

Dxx_issue
SIO_issue

DAX Driver

Generic streaming device

Description

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

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

Configuring a DAX Device

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

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

DAX_Params Elements

DAX_Params is defined in dax.h as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- arg is an optional parameter which may be used by the controller for configuration parameters specific for that controller. The default is 0, meaning there are no controller-specific parameters.

- **outputdelay** is an optional parameter that specifies the number of output buffers to delay before actually starting the output. The default is 1, meaning that output is double-buffered and does not start until the second SIO_put call.

Data Streaming

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

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

Example

The following example declares DAX_PRMS as a DAX_Params structure:

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

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

DGN Driver

Software generator driver

Comments

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

Configuring a DGN Device

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

Data Streaming

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

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

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

Generating Sine Values

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

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

The sine wave magnitude (maximum and minimum) value is approximated to the nearest power of two. This is done by computing a shift value by which each entry in the table is right-shifted before being copied into the input buffer.

DGN Driver Properties

There are no global properties for the DGN driver manager.

DGN Object Properties

The following properties can be set for a DGN device:

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

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

A user function must have the following form:

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

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

DGS Driver

Stackable gather/scatter driver

Description

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

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

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

Configuring a DGS Device

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

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

DGS_Params is defined in dgs.h as follows:

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

The device parameters are:

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

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

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

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

Transform Functions

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

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

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

Data Streaming

DGS devices can be opened for input or output. DGS_open allocates buffers for use by the underlying device. For input devices, the size of these buffers is (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 the Configuration Tool. To do that, add two new SIO objects. Enter /codec (or any other configured terminal device) as the Device Control Parameter for the first stream. Then select the DGS device configured to use u32tou8 in the Device property. For the second stream, enter /port as the Device Control Parameter. Then select the DGS device configured to use u32tou8 in the Device property.

Example

The following example declares DGS_PRMS as a DGS_Params structure:

```
#include <dgs.h>

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

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

See Also

DTR

DHL Driver

Host link driver

Description

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

Configuring a DHL Device

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

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

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

Data Streaming

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

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

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

```
stream = SIO_create("/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:

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

DHL Object Properties

The following properties can be set for a DHL device:

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

DNL Driver

Null driver

Description

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

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

Configuring a DNL Device

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

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

Data Streaming

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

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

Description

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

Configuring a DOV Device

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

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

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

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

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

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

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

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

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

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

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

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

```
#include <dov.h>

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

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

Data Streaming

DOV devices may only be opened for input.

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

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

You can use the same stacking device in more 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 with the Configuration Tool (rather than dynamically with SIO_create), add SIO objects with the Configuration Tool. Enter the string that identifies the terminating device preceded by “/” in the SIO object’s Device Control Parameters (e.g., /codec, /port). Then select the stacking device (overlap, overlapio) from the Device property.

See Also

DTR
DGS

DPI Driver

Pipe driver

Description

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

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

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

Configuring a DPI Device

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

Data Streaming

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

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

And in the function for the other task:

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

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

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

and from the writer task call:

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

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

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

DPI and the SIO_ISSUERECLAIM Streaming Model

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

A more efficient version of the driver that exchanges the buffers across both sides of the stream, rather than copying them, is also provided. To use this variant of the pipe driver for ISSUERECLAIM, edit the C source file dpi.c provided in the C:\ti\c5400\bios\src\drivers folder. Comment out the following line:

```
#define COPYBUFS
```

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

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

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

Converting a Single-Processor Application to a Multiprocessor Application

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

Constraints

- Only one reader and one writer may open the same pipe.

DPI Driver Properties

There are no global properties for the DPI driver manager.

DPI Object Properties

The following property can be set for a DPI device:

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

DST Driver

Stackable “split” driver

Description

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

Configuring a DST Device

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

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

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

Example 1:

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

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

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

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

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

Example 2:

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

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

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

Example 3:

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

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

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

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

Data Streaming

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

Constraints

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

DTR Driver

Stackable streaming transformer driver

Description

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

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

Configuring a DTR Device

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

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

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

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

  If you type _DTR_multiply, a data scaling operation is performed on the data stream to multiply the contents of the buffer by the scale.value of the device parameters.
- **Init Fxn**: Type 0 (zero).
- **Stacking Device**: Put a checkmark in this box.

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

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

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

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

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

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

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

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

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

```
if (user.fxn != NULL) {

   (*user.fxn)(user.arg, buffer, size);

}
```

Data Streaming

DTR transformer devices can be opened for input or output and use the same mode of I/O with the underlying streaming device. If a transformer is used as

a data source, it inputs a buffer from the underlying streaming device and then transforms this data in place. If the transformer is used as a data sink, it outputs a given buffer to the underlying device after transforming this data in place.

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

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

Global Settings

Global settings manager

Functions

None

Description

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

Global Settings Properties

The following Global Settings can be made:

- **Target Board Name**. The type of board on which your target chip is mounted.
- **DSP MIPS (CLKOUT)**. This number, times 1000000, is the number of instructions the processor can execute in 1 second. This value is used by the CLK manager to calculate register settings for the on-chip timers.
- **DSP Type**. The target CPU type. If you are using a custom board, you can type a value in this field. Type the number after the C in the chip model. For example, type 54 for a ’C5000 chip.
- **PMST(6-0)**. The low seven bits of the PMST register (MP/MC, OVLY, AVIS, DROM, CLKOFF, SMUL, and SST). Only the low seven bits can be directly modified. The high nine bits (IPTR) of the PMST are computed based on the base address of the VECT memory section.
- **PMST(15-0)**. The entire PMST register. PMST(6-0) can be modified directly. PMST(15-7) are computed based on the base address of the VECT memory section.
- **SWWSR**. The value for the Software Wait-State Register, which controls the software-programmable wait-state generator.

  The SWWSR, BSCR, and CLKMD registers are initialized during the boot initialization (via BIOS_init) before the program's main function is called. See *Volume 1: CPU and Peripherals* of the TMS320C54x DSP Reference Set for details on the SWWSR, BSCR, and CLKMD.
- **BSCR**. The value for the Bank-Switching Control Register, which allows switching between external memory banks without requiring external wait states.
- **Modify CLKMD**. Put a check mark in this box if you want to modify the value of the Clock Mode Register, which is used to program the PLL (phase-locked loop).
- **CLKMD - (PLL) Clock Mode Register**. The value of the Clock Mode Register.

- **Function Call Model**. This setting controls which libraries are used to link the application. If you change this setting, you must set the compiler and linker options to correspond. Use the far option only with 'C5000 chips that support extended addressing (e.g., 5402, 549, 5410).
- **C Autoinitialization Model**. Select the run-time initialization model.
- **Call user init function**. Put a checkmark in this box if you want an initialization function to be called early during program initialization—after .cinit processing and before the main function. This function can perform special hardware setup. The code in this function should not use any DSP/BIOS API calls.
- **User init function**. Type the name of the initialization function.
- **Enable Real Time Analysis**. Remove the checkmark from this box if you want to remove support for DSP/BIOS implicit instrumentation from the program. This optimizes a program by reducing code size, but removes support for the DSP/BIOS plug-ins and the LOG, STS, and TRC module APIs.
- **FARMODE**. This informational field shows which Function Call Model is selected.

HST Module

Host Channel manager

Functions

- HST_getpipe. Get corresponding pipe object

Description

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

Note:

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

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

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

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

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

HST Manager Properties

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

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

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

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

HST Object Properties

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

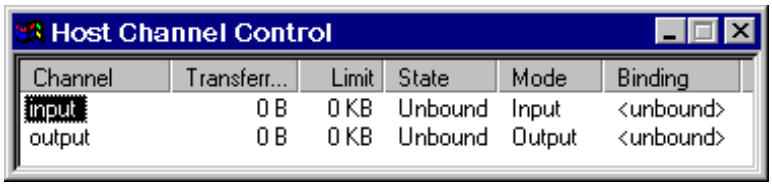
The following properties can be set for a host file object:

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

HST - Host Channel Control Interface

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

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



2) Right-click on a channel and choose Bind from the pop-up menu.

3) Select the file to which you want to bind this channel. For an input channel, select the file that contains the input data. For an output channel, you can type the name of a file that does not exist or choose any file that you want to overwrite.

4) Right-click on a channel and choose Start from the pop-up menu. For an input channel, this causes the host to transfer the first frame of data and causes the target to run the function for this HST object. For an output channel, this causes the target to run the function for this HST object.

HST_getpipe

Get corresponding pipe object

C Interface

Syntax pipe = HST_getpipe(hst);

Parameters Standee hst /* host object handle */

Return Value PIP_Handle pipe /* pipe object handle*/

Assembly Interface

Syntax HST_getpipe

Preconditions ar2 = address of the host channel object

Postconditions ar2 = address of the pipe object

Modifies ar2, c

Reentrant yes

Description

HST_getpipe gets the address of the pipe object for the specified host channel object.

Example

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

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

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

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

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

    size = PIP_getReaderSize;
    out->writerSize = size;

    for (; size > 0; size--) {
```

```
        *dst++ = *src++;
    }

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

See Also

PIP_alloc
PIP_free
PIP_get
PIP_put

HWI Module

Hardware interrupt manager

Functions

- ❏ HWI_disable. Disable hardware interrupts
- ❏ HWI_enable. Enable hardware interrupts
- ❏ HWI_enter. Hardware ISR prolog
- ❏ HWI_exit. Hardware ISR epilog
- ❏ HWI_restore. Restore hardware interrupt state

Description

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

Interrupt routines can be written in assembly or in a mix of assembly and C.

The HWI_enter assembly macro must be called prior to any DSP/BIOS API calls that could post or affect a software interrupt or semaphore, and the HWI_exit assembly macro must be called at the very end of the function's code.

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

Do not call SWI_disable or SWI_enable within an HWI function.

Note:

You must use HWI_disable and HWI_enable to bracket a block of code that atomically makes DSP/BIOS API calls.

Note:

Do not call HWI_enter, HWI_exit, or any other DSP/BIOS functions from a non-maskable interrupt (NMI) service routine.

The DSP/BIOS API calls that require an HWI function to use HWI_enter and HWI_exit are:

- SWI_andn
- SWI_dec
- SWI_inc
- SWI_or
- SWI_post
- PIP_alloc
- PIP_free
- PIP_get
- PIP_put
- PRD_tick
- SEM_post

Note:

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

Note:

An HWI function must use HWI_enter if it indirectly runs a function containing any of the API calls listed above.

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

The mask argument to HWI_enter and HWI_exit allows you to save and restore registers used within the function.

Hardware interrupts always interrupt software interrupts unless hardware interrupts have been disabled with HWI_disable.

Note:

By using HWI_enter and HWI_exit as an HWI function's prolog and epilog, an HWI function can be interrupted; i.e., a hardware interrupt can interrupt another interrupt. You can use the IMRDISABLEMASK parameter for the HWI_enter API to prevent this from occurring.

HWI Manager Properties

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

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

- **Function Stub Memory**. Select the memory section where the dispatch code should be placed for interrupt service routines that are configured to be monitored.

HWI Object Properties

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

- **comment** A comment is provided to identify each HWI object.
- **function** The function to execute. Interrupt routines must be written at least partially in assembly language. Within an HWI function, the HWI_enter assembly macro must be called prior to any DSP/BIOS API calls that could post or affect a software interrupt. HWI functions can post software interrupts, but they do not run until your HWI function calls the HWI_exit assembly macro, which must be the last statement in any HWI function that calls HWI_enter.
- **monitor** If set to anything other than Nothing, an STS object is created for this ISR that is passed the specified value on every invocation of the interrupt service routine. The STS update occurs just before entering the ISR.
- **addr** If the monitor field above is set to Data Address, this field lets you specify a data memory address to be read; the word-sized value is read and passed to the STS object associated with this HWI object.
- **type**. The type of the value to be monitored: unsigned or signed. Signed quantities are sign extended when loaded into the accumulator; unsigned quantities are treated as word-sized positive values.
- **operation**. The operation to be performed on the value monitored. You can choose one of several STS operations.

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

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

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

HWI interrupts for the TMS320C54x:

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

HWI - Execution Graph Interface

Time spent performing HWI functions is not directly traced for performance reasons. However, the Other Threads row in the Execution Graph, which you

can open by choosing Tools→DSP/BIOS→Execution Graph, includes time spent performing both HWI and IDL functions.

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

HWI_disable

Disable hardware interrupts

C Interface

Syntax oldST1 = HWI_disable();

Parameters Void

Return Value Uns oldST1;

Assembly Interface

Syntax HWI_disable

Preconditions none

Postconditions intm = 1 (with 2 cycles of latency)

Modifies c, intm

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_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.

Example

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

See Also

HWI_enable
SWI_disable
SWI_enable

HWI_enable

Enable interrupts

C Interface

Syntax HWI_enable();

Parameters Void

Return Value Void

Assembly Interface

Syntax HWI_enable

Preconditions none

Postconditions intm = 0 (with 2 cycles of latency)

Modifies c, intm, tc

Reentrant yes

Description

HWI_enable enables hardware interrupts by clearing the intm bit in the status register.

Hardware interrupts are enabled unless a call to HWI_disable disables them.

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

Any call to HWI_enable enables interrupts, even if HWI_disable has been called several times.

Example

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

See Also

HWI_disable
SWI_disable
SWI_enable

HWI_enter *Hardware ISR prolog*

C Interface

Syntax none

Parameters none

Return Value none

Assembly Interface

Syntax HWI_enter MASK IMRDISABLEMASK

Preconditions intm = 1

Postconditions dp = GBL_A_SYSPAGE
cpl = ovm = c16 = frct = cmpt = 0

Modifies c, cpl, dp, sp

Reentrant yes

Description

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

HWI_enter must be used in the ISR before any DSP/BIOS API calls that could trigger a software interrupt; e.g., SWI_post. HWI_enter is used in tandem with HWI_exit to ensure that the DSP/BIOS SWI manager is called at the appropriate time. Normally, HWI_enter and HWI_exit must surround all statements in any DSP/BIOS assembly language ISRs.

One common mask, C54_CNOTPRESERVED, is defined in c54.h54. This mask specifies the C temporary registers and should be used when saving the context for an ISR that is written in C.

Constraints and Calling Context

- ❏ This API should not be used for the NMI HWI function.
- ❏ This API must be called within any hardware interrupt function (except NMI's HWI function) before the first operation in an ISR that uses any DSP/BIOS API calls that might post or affect a software interrupt or semaphore. Such functions must be written in assembly language.
- ❏ If an interrupt function calls HWI_enter, it must end by calling HWI_exit.

Example

CLK_isr:

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

See Also

HWI_exit

HWI_exit

Hardware ISR epilog

C Interface

Syntax none

Parameters none

Return Value none

Assembly Interface

Syntax HWI_exit MASK IMRRESTOREMASK

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
intm = 1 (i.e., interrupts are disabled)

Postconditions intm = 0

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 interrupt service routine (ISR) was invoked.

HWI_exit must be the last statement in an ISR that uses DSP/BIOS API calls which could trigger a software interrupt; e.g., SWI_post.

HWI_exit restores the registers specified by MASK. MASK is used to specify the set of registers that were saved by HWI_enter.

HWI_enter and HWI_exit must surround all statements in any DSP/BIOS assembly language ISRs that call C functions.

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

Of the interrupts in IMRRESTOREMASK, HWI_exit only restores those that were enabled upon entering the ISR. HWI_exit does not affect the status of interrupt bits that are not in IMRRESTOREMASK.

If upon exiting an ISR you do not wish to restore one of the interrupts that were disabled with HWI_enter, do not set that interrupt bit in the IMRRESTOREMASK in HWI_exit.

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

Constraints and Calling Context

- ❏ This API should not be used for the NMI HWI function.
- ❏ This API must be the last operation in an ISR that uses any DSP/BIOS API calls that might post or affect a software interrupt or semaphore. Such functions must be written in assembly language.
- ❏ The MASK parameters must match the corresponding parameters used for HWI_enter.

Example

CLK_isr:

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

See Also

HWI_enter

HWI_restore

Restore global interrupt enable state

C Interface

Syntax HWI_restore(oldST1);

Parameters Uns oldST1;

Returns Void

Assembly Interface

Syntax HWI_restore

Preconditions al = mask (intm is set to the value of bit 11)
intm = 1

Postconditions none

Modifies c, intm

Reentrant no

Description

HWI_restore sets the intm bit in the ST1 register using bit 11 of the oldST1 parameter. If bit 11 is 1, the intm bit is not modified. If bit 11 is 0, the intm bit is set to 0, which enables interrupts.

When you call HWI_disable, the previous contents of the ST1 register are returned. You can use this returned value with HWI_restore.

Constraints and Calling Context

- HWI_restore cannot be called from an ISR context.

Example

```
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

IDL Module

Event Log manager

Functions

- IDL_run. Make one pass through idle functions

Description

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

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

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

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

An application always has an IDL_cpuLoad object, which runs a function that provides data about the CPU utilization of the application. In addition, the LNK_dataPump function handles host I/O in the background, and the RTA_dispatch function handles run-time analysis communication.

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

IDL Manager Properties

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

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

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

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

- enabling hardware interrupts or the SWI or TSK schedulers
- using CLK APIs to get the time
- accessing PIP objects
- blocking tasks
- creating dynamic objects

❏ **Idle Loop Instruction Count**. This is the number of instruction cycles required to perform the IDL loop and the default IDL functions (LNK_dataPump, RTA_dispatcher, and IDL_cpuLoad) that communicate with the host. Since these functions are performed whenever no other processing is needed, background processing is subtracted from the CPU load before it is displayed.

IDL Object Properties

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

The following properties can be set for an IDL object:

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

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

❏ **Include in CPU load calibration**. You can remove an individual IDL function from the CPU load calculation by removing the check mark from this box. The CPU load calibration is performed only if the Auto calculate idle loop instruction count box is checked in the IDL Manager Properties. You should remove a function from the calculation if it blocks or depends on variables or structures that are not initialized until the main function runs.

IDL- Execution Graph Interface

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

IDL_run

Make one pass through idle functions

C Interface

Syntax IDL_run();

Parameters Void

Return Value Void

Assembly Interface none

Description

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

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

Note:

BIOS_init and BIOS_start must be called before IDL_run to ensure that DSP/BIOS has been initialized. For example, the DSP/BIOS boot file contains the following system calls around the call to main:

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

LCK Module

Resource lock manager

Functions

- LCK_create. Create a resource lock
- LCK_delete. Delete a resource lock
- LCK_pend. Acquire ownership of a resource lock
- LCK_post. Relinquish ownership of a resource lock

Constants, Types, and Structures

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

/* lock object */
typedef struct LCK_Attrs LCK_Attrs;

struct LCK_Attrs {
    Int dummy;
};

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

Description

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

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

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

LCK Manager Properties

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

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

LCK Object Properties

The following property can be set for a LCK object:

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

LCK_create

Create a resource lock

C Interface

Syntax

```
lock = LCK_create(attrs);
```

Parameters

```
LCK_Attrs    attrs;   /* pointer to lock attributes */
```

Return Value

```
LCK_Handle   lock;    /* handle for new lock object */
```

Assembly Interface

none

Description

LCK_create creates a new lock object and returns its handle. The lock has no current owner and its corresponding resource is available for acquisition through LCK_pend.

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

Note:

At present, no attributes are supported for lock objects.

All default attribute values are contained in the constant LCK_ATTRS, which may be assigned to a variable of type LCK_Attrs prior to calling LCK_create.

Constraints and Calling Context

- ❏ LCK_create cannot be called by ISRs.
- ❏ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions.

See Also

LCK_delete
LCK_pend
LCK_post

LCK_delete

Delete a resource lock

C Interface

Syntax

```
LCK_delete(lock);
```

Parameters

```
LCK_Handle  lock;    /* lock handle */
```

Return Value

Void

Assembly Interface

none

Description

LCK_delete uses MEM_free to free the lock referenced by lock.

Constraints and Calling Context

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

See Also

LCK_create
LCK_post
LCK_pend

LCK_pend

Acquire ownership of a resource lock

C Interface

Syntax

```
status = LCK_pend(lock, timeout);
```

Parameters

```
LCK_Handle    lock;    /* lock handle */
Uns           timeout; /* return after this many system clock ticks */
```

Return Value

```
Bool          status;  /* TRUE if successful, FALSE if timeout */
```

Assembly Interface

none

Description

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

The task owning lock may call LCK_pend any number of times without risk of blocking, although relinquishing ownership of the lock requires a balancing number of calls to LCK_post.

LCK_pend returns TRUE if it successfully acquires ownership of lock, returns FALSE if timeout.

Constraints and Calling Context

- ❏ lock must be a handle for a resource lock object created through a prior call to LCK_create.
- ❏ LCK_pend cannot be called by an ISR.

See Also

LCK_create
LCK_delete
LCK_post

LCK_post

Relinquish ownership of a resource LCK

C Interface

Syntax LCK_post(lock);

Parameters LCK_Handle lock; /* lock handle */

Return Value Void

Assembly Interface none

Description

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

Constraints and Calling Context

- ❏ lock must be a handle for a resource lock object created through a prior call to LCK_create.
- ❏ LCK_post cannot be called by an ISR.

See Also

LCK_create
LCK_delete
LCK_pend

LOG Module

Capture events in real time

Functions

- ❏ LOG_disable. Disable the system log
- ❏ LOG_enable. Enable the system log
- ❏ LOG_error. Write a user error event to the system log
- ❏ LOG_event. Append unformatted message to message log
- ❏ LOG_message. Write a user message event to the system log
- ❏ LOG_printf. Append formatted message to message log
- ❏ LOG_reset. Reset the system log

Description

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

The system log stores messages about system events for the types of log tracing you have enabled. See the TRC Module, page 1–288, for a list of events that can be traced in the system log.

You can add messages to user logs or the system log by using LOG_printf or LOG_event. To reduce execution time, log data is always formatted on the host. Calls that access LOG objects return in less than 2 microseconds.

LOG_error writes a user error event to the system log. This operation is not affected by any TRC trace bits; an error event is always written to the system log. LOG_message writes a user message event to the system log, provided that both TRC_GBLHOST and TRC_GBLTARG (the host and target trace bits, respectively) traces are enabled.

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

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

See the *TMS320C54x Code Composer Studio Tutorial* for examples of how to use the LOG Manager.

LOG Manager Properties

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

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

LOG Object Properties

The following properties can be set for a log object:

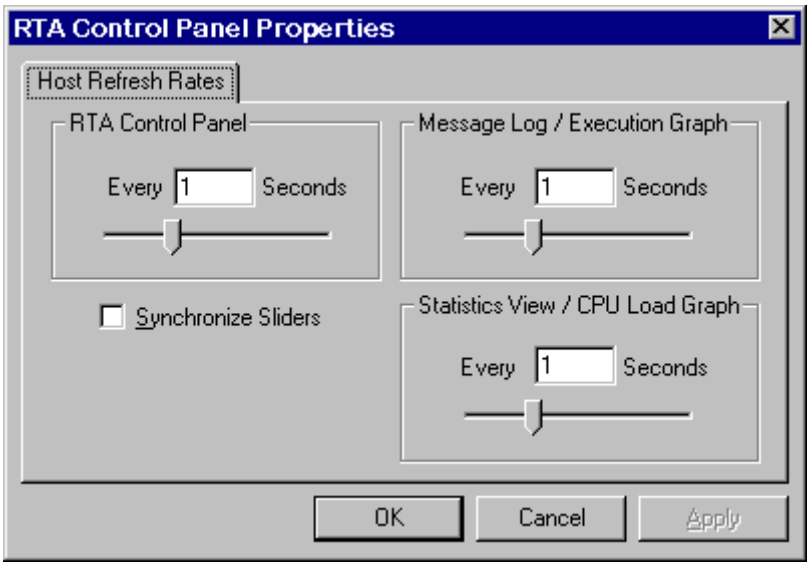
- **comment**. Type a comment to identify this LOG object.
- **bufseg**. The name of a memory section to contain the log buffer.
- **buflen**. The length of the log buffer (in words).
- **logtype**. The type of the log: circular or fixed. Events added to a full circular log overwrite the oldest event in the buffer, whereas events added to a full fixed log are dropped.
  - **Fixed**. The log stores the first messages it receives and stops accepting messages when its message buffer is full.
  - **Circular**. The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.
- **datatype**. Choose printf if you use LOG_printf to write to this log and provide a format string.
  Choose raw data if you want to use LOG_event to write to this log and have the Event Log apply a printf-style format string to all records in the log.
- **format**. If you choose raw data as the datatype, type a printf-style format string in this field. Provide up to three (3) conversion characters (such as %d) to format words two, three, and four in all records in the log. Do not put quotes around the format string. The format string can use %d, %x, %o, %s, and %r conversion characters; it cannot use other types of conversion characters.
  See LOG_printf, page 1–118, and LOG_event, page 1–116, for information about the structure of a log record.

LOG - Code Composer Studio Interface

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

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

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



LOG_disable

Disable a message log

C Interface

Syntax LOG_disable(log);

Parameters LOG_Handle log; /* log object handle */

Return Value Void

Assembly Interface

Syntax LOG_disable

Preconditions ar2 = address of the LOG object

Postconditions none

Modifies c

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

Assembly Interface

Syntax LOG_enable

Preconditions ar2 = address of the LOG object

Postconditions none

Modifies c

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

Assembly Interface

Syntax

LOG_error format [section]

Preconditions

ar2 = format
bh = arg0
dp = GBL_A_SYSPAGE

Postconditions

none (see the description of the section argument below)

Modifies

ag, ah, al, ar0, ar2, ar3, bl, c, t, tc

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 passed to LOG_error may contain any of the conversion characters supported for LOG_printf. See LOG_printf, page 1–118, for details.

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

Example

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

See Also

LOG_event
LOG_message
LOG_printf
TRC_disable
TRC_enable

LOG_message

Write a program-supplied message to the system log

C Interface

Syntax

LOG_message(format, arg0);

Parameters

```
String    format;  /* printf-style format string */
Arg       arg0;    /* copied to second word of log record */
```

Return Value

Void

Assembly Interface

Syntax

LOG_message format [section]

Preconditions

ar2 = format
bh = arg0
dp = GBL_A_SYSPAGE

Postconditions

none (see the description of the section argument below)

Modifies

ag, ah, al, ar0, ar2, ar3, bl, c, t, tc

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 may contain any of the conversion characters supported for LOG_printf. See LOG_printf, page 1–118, for details.

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

Example

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

See Also

LOG_error
LOG_event
LOG_printf
TRC_disable
TRC_enable

LOG_event

Append an unformatted message to a message log

C Interface

Syntax

```
LOG_event(log, arg0, arg1, arg2);
```

Parameters

```
LOG_Handle   log;   /* log objecthandle */
Arg      arg0;      /* copied to second word of log record */
Arg      arg1;      /* copied to third word of log record */
Arg      arg2;        /* copied to fourth word of log record */
```

Return Value

Void

Assembly Interface

Syntax

LOG_event

Preconditions

ar2 = address of the LOG object
bh = arg0
bl = arg1
t = arg2

Postconditions

none

Modifies

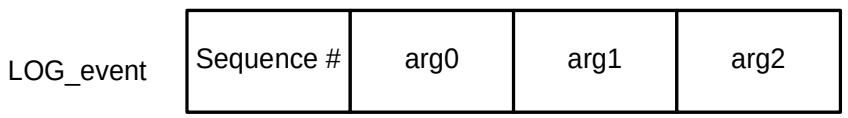
ag, ah, al, ar0, ar2, ar3, c, tc

Reentrant

yes

Description

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



You can format the log by using LOG_printf instead of LOG_event.

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

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

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

Example

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

See Also

LOG_error
LOG_printf
TRC_disable
TRC_enable

LOG_printf

Append a formatted message to a message log

C Interface

Syntax

```
LOG_printf(log, format);
   or
LOG_printf(log, format,arg0);
   or
LOG_printf(log, format, arg0, arg1);
```

Parameters

```
LOG_Handle   log;     /* log object handle */
String     format;      /* printf format string */
Arg        arg0;         /* value for first format string token */
Arg        arg1;        /* value for second format string token */
```

Return Value

Void

Assembly Interface

Syntax

LOG_printf format [section]

Preconditions

ar2 = address of the LOG object
bh = arg0
bl = arg1

Postconditions

none

Modifies

ag, ah, al, ar0, ar2, ar3, c, t, tc

Reentrant

yes

Description

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

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

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

Conversion Character	Description
%d	Signed integer
%x	Unsigned hexadecimal integer
%o	Unsigned octal integer

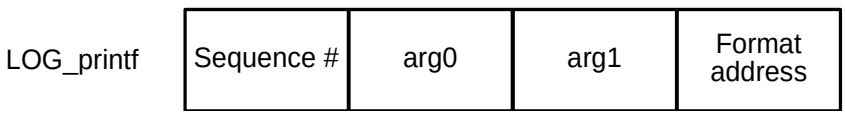
%s	Character string
	This character can only be used with constant string pointers. That is, the string must appear in the source and be passed to LOG_printf. For example, the following is supported:
	`char *msg = "Hello world!";`
	`LOG_printf(&trace, "%s", msg);`
	However, the following example is not 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>.

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

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

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



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

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

Constraints and Calling Context

- ❏ LOG_printf (even the C version) supports 0, 1, or 2 arguments after the format string.

Example

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

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

Assembly Interface

Syntax LOG_reset

Preconditions ar2 = address of the LOG object

Postconditions none

Modifies ag, ah, al, ar3, ar4, c

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 ISR or other thread. It is therefore possible for the log to contain inconsistent data if LOG_reset is preempted by an ISR or other thread that uses the same log.

Example

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

See Also

LOG_disable
LOG_enable

MBX Module

Mailbox manager

Functions

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

Constants, Types, and Structures

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

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

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

Description

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

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

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

MBX Manager Properties

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

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

MBX Object Properties

The following properties can be set for an MBX object:

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

MBX Code Composer Studio Interface

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

MBX_create

Create a mailbox

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

MBX_create creates a mailbox object which is initialized to contain up to mbxlength messages of size msgsize. If successful, MBX_create returns the handle of the new mailbox object. If unsuccessful, MBX_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS_error, and SYS_error causes an abort).

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

All default attribute values are contained in the constant MBX_ATTRS, which may be assigned to a variable of type MBX_Attrs prior to calling MBX_create.

No task switch occurs when calling MBX_create.

Constraints and Calling Context

- ❏ MBX_create cannot be called by ISRs.
- ❏ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions.

See Also

MBX_delete
SYS_error

MBX_delete

Delete a mailbox

C Interface

Syntax MBX_delete(mbx);

Parameters MBX_Handle mbx; /* mailbox object handle */

Return Value Void

Assembly Interface none

Description

MBX_delete frees the mailbox object referenced by mbx.

No task switch occurs when calling MBX_delete.

Constraints and Calling Context

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

See Also

MBX_create

MBX_pend

Wait for a message from mailbox

C Interface

Syntax

```
status = MBX_pend(mbx, msg, timeout);
```

Parameters

```
MBX_Handle   mbx;       /* mailbox object handle */
Ptr          msg;       /* message pointer */
Uns          timeout;   /* return after this many system clock ticks */
```

Return Value

```
Bool    status;    /* TRUE if successful, FALSE if timeout */
```

Assembly Interface

none

Description

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

If timeout is SYS_FOREVER, the task remains suspended until MBX_post is called on this mailbox. If timeout is 0, MBX_pend returns immediately.

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

A task switch occurs when calling MBX_pend if the mailbox is empty and timeout is not 0, or if a higher priority task is blocked on MBX_post.

Constraints and Calling Context

- ❏ MBX_pend may only be called from an ISR if timeout is 0.
- ❏ MBX_pend cannot be called within a TSK_disable / TSK_enable block.

See Also

MBX_post

MBX_post

Post a message to mailbox

C Interface

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 */
```

Assembly Interface

none

Description

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

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

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

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

Constraints and Calling Context

- ❏ MBX_post may only be called from an ISR if timeout is 0.
- ❏ MBX_post cannot be called within a TSK_disable / TSK_enable block.

See Also

MBX_pend

MEM Module

Memory section manager

Functions

- ❏ MEM_alloc. Allocate from a memory section
- ❏ MEM_calloc. Allocate and initialize to 0
- ❏ MEM_define. Define a new memory section
- ❏ MEM_free. Free a block of memory
- ❏ MEM_redefine. Redefine an existing memory section
- ❏ MEM_stat. Return the status of a memory section
- ❏ MEM_valloc. Allocate and initialize to a value

Constants, Types, and Structures

```
MEM->MALLOCSEG = 0;      /* segid for malloc(), free() */

#define MEM_HEADERSIZE   /* free block header size */
#define MEM_HEADERMASK   /* mask to align on MEM_HEADERSIZE */
#define MEM_ILLEGAL      /* illegal memory address */

MEM_Attrs MEM_ATTRS ={   /* default attribute values */
    0
};

typedef struct MEM_Segment {
    Ptr    base;         /* base of the section */
    Uns    length;       /* size of the section */
    Uns    space;        /* memory space */
} MEM_Segment;

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

Description

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

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

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

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

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

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

MEM Manager Properties

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

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

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

  Stack size is shown as a hex value in Minimum Addressable Units (MAUs). An MAU is the smallest unit of data storage that can be read or written by the CPU. For the 'C5000 this is a 16-bit word.

- **Stack Section (.stack)**. The memory section containing the software stack. This section should be located in RAM.
- **BIOS Code Section (.bios)**. The memory section containing the DSP/BIOS code.
- **Startup Code Section (.sysinit)**. The memory section containing DSP/BIOS startup initialization code; this memory may be reused after main starts executing.
- **DSP/BIOS Init Tables (.gblinit, .trcinit)**. The memory section containing the DSP/BIOS global and instrumentation initialization tables.
- **DSP/BIOS Kernel State (.sysdata)**. The memory section containing system data about the DSP/BIOS kernel state.
- **DSP/BIOS Conf Sections (.obj)**. The memory section containing configuration properties that can be read by the target program.
- **No Dynamic Memory Heaps**. Put a checkmark in this box to completely disable the ability to dynamically allocate memory and the ability to dynamically create and delete objects. If this box is checked, your program may not call the MEM_alloc, MEM_valloc, and MEM_calloc functions or the XXX_create function for any DSP/BIOS module. If this box is checked and the program calls one of these functions, an error occurs when the program is linked.
- **Segment for DSP/BIOS objects**. The default memory section that will contain objects created at run-time with an XXX_create function.
- **Segment for malloc / free**. The memory section from which space is allocated when a program calls malloc and from which space is freed when a program calls free.
- **User .cmd file for non-DSP/BIOS sections**. Put a checkmark in this box if you want to have full control over the memory used for the sections that follow. You will need to create your own linker command file that begins by including the linker command file created by the Configuration Tool. Your linker command file should then assign memory for the items normally handled by the following properties. See the *TMS320C54x Optimizing C Compiler User's Guide* for more details.
- **Text Section (.text)**. The memory section containing the executable code, string literals, and compiler-generated constants. This section may be located in ROM or RAM.
- **Switch jump tables (.switch)**. The memory section containing the switch statement tables. This section may be located in ROM or RAM.
- **C variables (.bss)**. The memory section containing global and static C variables. At boot or load time, the data in the .cinit section is copied to this section. This section should be located in RAM.

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

MEM Object Properties

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

- **comment**. Type a comment to identify this MEM object.
- **base**. The address at which this memory section begins. This value is shown in hex.
- **len**. The length of this memory section in words. This value is shown in hex.
- **Create a heap in this memory**. If this box is checked, a heap is created in this memory section. Memory can by allocated dynamically from a heap. In order to remove the heap from a memory section, you may need to select another memory section that contains a heap for properties that dynamically allocate memory in this memory section. The properties you should check are in the Memory Section Manager (the Segment for DSP/BIOS objects and Segment for malloc/free properties) and the Task Manager (the Default stack segment for dynamic tasks property). If you disable dynamic memory allocation in the Memory Section Manager, you cannot create a heap in any memory section.
- **Heap size**. The size of the heap to be created in this memory section.
- **space**. Type of memory section. This is set to code for memory sections that store programs, and data for memory sections that store program data.

The predefined memory sections in a configuration file, particularly those for external memory, are dependent on the board template you select. In general, the following sections may be defined for the ’C5000:

Name	Memory Section Type
USERREGS	User scratchpad memory
BIOSREGS	Scratchpad memory reserved for use by DSP/BIOS
VECT	Interrupt vector table
IDATA	Internal data RAM
IPROG	Internal program RAM
EDATA	External data memory
EPROG	External program memory

MEM Code Composer Studio Interface

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

MEM_alloc *Allocate from a memory section*

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

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

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

The block contains size MAUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

MEM_alloc does not initialize the allocated memory locations.

If the memory request cannot be satisfied, MEM_alloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

Constraints and Calling Context

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

See Also

MEM_calloc
MEM_free
MEM_valloc
SYS_error
C library stdlib.h

MEM_calloc *Allocate from a memory section and set value to 0*

C Interface

Syntax

```
addr = MEM_calloc(segid, size, align)
```

Parameters

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

Return Value

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

Assembly Interface

none

Description

MEM_calloc is functionally equivalent to calling MEM_valloc with value set to 0.

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

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

The block contains size MAUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM_calloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

Constraints and Calling Context

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

See Also

MEM_alloc
MEM_free
MEM_valloc
SYS_error
C library stdlib.h

MEM_define

Define a new memory section

C Interface

Syntax

```
segid = MEM_define(base, length, attrs);
```

Parameters

```
Ptr       base;      /* base address of new section */
Uns       length;    /* length (in MAUs) of new section */
MEM_Attrs  *attrs;   /* section attributes */
```

Return Value

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

Assembly Interface

none

Description

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

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

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

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

Note:

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

MEM_define and MEM_redefine must not be called when a task switch is possible. To guard against a task switch, these functions should be used only in the main function, and should be preceded with a call to TSK_disable and followed with a call to TSK_enable.

Note:

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

Constraints and Calling Context

- ❑ At least one section must exist at the time MEM_define is called.

See Also

MEM_redefine

MEM_free

Free a block of memory

C Interface

Syntax

```
status = MEM_free(segid, addr, size);
```

Parameters

```
Int    segid;    /* memory section identifier */
Ptr    addr;     /* block address pointer */
Uns    size;     /* block length */
```

Return Value

```
Bool   status;   /* TRUE if successful */
```

Assembly Interface

none

Description

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

Constraints and Calling Context

- ❏ addr must be a valid pointer returned from a call to MEM_alloc.
- ❏ segid and size are those values used in a previous call to MEM_alloc.

See Also

MEM_alloc
C library stdlib.h

MEM_redefine

Redefine an existing memory section

C Interface

Syntax

```
MEM_redefine(segid, base, length);
```

Parameters

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

Return Value

Void

Assembly Interface

none

Description

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

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

MEM_define and MEM_redefine must not be called when a task switch is possible. To guard against a task switch, these functions should be used only in the main function, and should be preceded with a call to TSK_disable and followed with a call to TSK_enable.

Note:

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

See Also

MEM_define

MEM_stat

Return the status of a memory section

C Interface

Syntax

```
status = MEM_stat(segid, statbuf);
```

Parameters

```
Int       segid;      /* memory section identifier */
MEM_Stat  *statbuf;   /* pointer to stat buffer */
```

Return Value

```
Bool      status;     /* TRUE if successful */
```

Assembly Interface

none

Description

MEM_stat returns the status of the memory section specified by segid in the status structure pointed to by statbuf.

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

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

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

MEM_valloc *Allocate from a memory section and set value*

C Interface

Syntax

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

Parameters

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

Return Value

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

Assembly Interface

none

Description

MEM_valloc uses MEM_alloc to allocate the memory before initializing it to value.

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

The block contains size MAUs and starts at an address that is a multiple of align. If align is 0 or 1, there is no alignment constraint.

If the memory request cannot be satisfied, MEM_valloc calls SYS_error with SYS_EALLOC and returns MEM_ILLEGAL.

Constraints and Calling Context

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

See Also

MEM_alloc
MEM_calloc
MEM_free
SYS_error
C library stdlib.h

PIP Module

Buffered pipe manager

Functions

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

PIP_Obj Structure Members

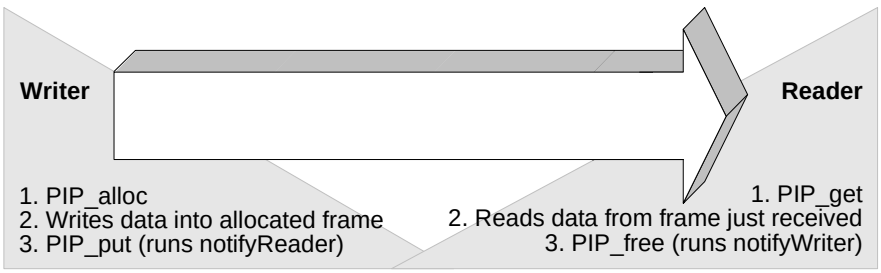
- **Ptr readerAddr**. Pointer to the address to begin reading from after calling PIP_get
- **Uns readerSize**. Number of words of data in the frame read with PIP_get
- **Uns readerNumFrames**. Number of frames available to be read
- **Ptr writerAddr**. Pointer to the address to begin writing to after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

Description

The PIP module manages data pipes, which are used to buffer streams of input and output data. These data pipes provide a consistent software data structure you can use to drive I/O between the DSP chip and all kinds of real-time peripheral devices.

Each pipe object maintains a buffer divided into a fixed number of fixed length frames, specified by the numframes and framesize properties. All I/O operations on a pipe deal with one frame at a time; although each frame has a fixed length, the application may put a variable amount of data in each frame up to the length of the frame.

A pipe has two ends, as shown in the following figure. The writer end (also called the producer) is where your program writes frames of data. The reader end (also called the consumer) is where your program reads frames of data.



Internally, pipes are implemented as a circular list; frames are reused at the writer end of the pipe after PIP_free releases them.

The notifyReader and notifyWriter functions are called from the context of the code that calls PIP_put or PIP_free. These functions may be written in C or assembly. To avoid problems with recursion, the notifyReader and notifyWriter functions should not directly call any of the PIP module functions for the same pipe. Instead, they should post a software interrupt that uses the PIP module functions.

Note:

When DSP/BIOS starts up, it calls the notifyWriter function internally for each created pipe object to initiate the pipe's I/O.

The code that calls PIP_free or PIP_put should preserve any necessary registers.

Often one end of a pipe is controlled by a hardware ISR and the other end is controlled by a SWI function.

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

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

PIP Manager Properties

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

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

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

PIP Object Properties

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

The following properties can be set for a pipe object:

- **comment**. Type a comment to identify this PIP object.
- **bufseg**. The memory section that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- **bufalign**. The alignment (in words) of the buffer allocated within the specified memory section.
- **framesize**. The length of each frame (in words)
- **numframes**. The number of frames
- **monitor**. The end of the pipe to be monitored by a hidden STS object. Can be set to reader, writer, or nothing. In the Statistics View plug-in, your choice determines whether the STS display for this pipe shows a count of the number of frames handled at the reader or writer end of the pipe.
- **notifyWriter**. The function to execute when a frame of free space is available. This function should notify (e.g., by calling SWI_andn) the object that writes to this pipe that an empty frame is available.
  The notifyWriter function is performed as part of the thread that called PIP_free or PIP_alloc. To avoid problems with recursion, the notifyWriter function should not directly call any of the PIP module functions for the same pipe.
- **nwarg0, nwarg1**. Two 16-bit arguments passed to notifyWriter; these arguments can each be either an unsigned 16-bit constant or a symbolic label.
- **notifyReader**. The function to execute when a frame of data is available. This function should notify (e.g., by calling SWI_andn) the object that

reads from this pipe that a full frame is ready to be processed.
The notifyReader function is performed as part of the thread that called PIP_put or PIP_get. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.

- **nrarg0, nrarg1**. Two 16-bit arguments passed to notifyReader; these arguments can each be either an unsigned 16-bit constant or a symbolic label.

PIP - Code Composer Studio Interface

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

PIP_alloc

Allocate an empty frame from a pipe

C Interface

Syntax PIP_alloc(pipe);

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Void

Assembly Interface

Syntax PIP_alloc

Preconditions ar2 = address of the pipe object
the pipe must contain empty frames before calling PIP_alloc

Postconditions none

Modifies ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm

Reentrant no

Description

PIP_alloc allocates an empty frame from the pipe object you specify. You can write to this frame and then use PIP_put to put the frame into the pipe.

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

Constraints and Calling Context

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

Example

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

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

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

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

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

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

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

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

The example for HST_getpipe, page 1–84, also uses a pipe with host channel objects.

See Also

PIP_free
PIP_get
PIP_put
HST_getpipe

PIP_free

Recycle a frame that has been read to a pipe

C Interface

Syntax

```
PIP_free(pipe);
```

Parameters

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

Return Value

Void

Assembly Interface

Syntax

```
PIP_free
```

Preconditions

ar2 = address of the pipe object

Postconditions

none

Modifies

ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm, and any registers modified by the notifyWriter function

Reentrant

no

Description

PIP_free releases a frame after you have read the frame with PIP_get. The frame is recycled so that PIP_alloc can reuse it.

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

Example

See the example for PIP_alloc, page 1–145. The example for HST_getpipe, page 1–84, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_get
PIP_put
HST_getpipe

PIP_get

Get a full frame from the pipe

C Interface

Syntax PIP_get(pipe);

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Void

Assembly Interface

Syntax PIP_get

Preconditions ar2 = address of the pipe object
the pipe must contain full frames before calling PIP_get

Postconditions none

Modifies ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm

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 (e.g., by calling SWI_andn) the object that reads from this pipe that a full frame is available. The notifyReader function is performed as part of the thread that calls PIP_get or PIP_put. To avoid problems with recursion, the notifyReader function should not directly call any PIP module functions for the same pipe.

Constraints and Calling Context

- Before calling PIP_get, a function should check the readerNumFrames member of the PIP_Obj structure by calling PIP_getReaderNumFrames to make sure it is greater than 0 (i.e., at least one full frame is available).
- PIP_get can only be called one time before calling PIP_free. You cannot operate on two frames from the same pipe simultaneously.

Example

See the example for PIP_alloc, page 1–145. The example for HST_getpipe, page 1–84, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_free
PIP_put
HST_getpipe

PIP_getReaderAddr

Get the value of the readerAddr pointer of the pipe

C Interface

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

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Ptr readerAddr

Assembly Interface none

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Description

PIP_getReaderAddr is a C function that returns the value of the readerAddr pointer of a pipe object.

The readerAddr pointer is normally used following a call to PIP_get, as the address to begin reading from.

Example

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

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

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

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

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

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

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

PIP_getReaderNumFrames

Get the number of pipe frames available for reading

C Interface

Syntax

```
num = PIP_getReaderNumFrames(pipe);
```

Parameters

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

Return Value

```
Uns    num;        /* number of filled frames to be read */
```

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Description

PIP_getReaderNumFrames is a C function that returns the value of the readerNumFrames element of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 1–150.

PIP_getReaderSize

Get the number of words of data in a pipe frame

C Interface

Syntax num = PIP_getReaderSize(pipe);

Parameters PIP_Handle pipe; /* pipe object handle*/

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

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Description

PIP_getReaderSize is a C function that returns the value of the readerSize element of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 1–150.

PIP_getWriterAddr

Get the value of the writerAddr pointer of the pipe

C Interface

Syntax writerAddr = PIP_getWriterAddr(pipe);

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Ptr writerAddr;

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Description

PIP_getWriterAddr is a C function that returns the value of the writerAddr pointer of a pipe object.

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

Example

See the example for PIP_getReaderAddr, page 1–150.

PIP_getWriterNumFrames

Get number of pipe frames available to be written to

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 */

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

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 1–150.

PIP_getWriterSize

Get the number of words that can be written to a pipe frame

C Interface

Syntax num = PIP_getWriterSize(pipe);

Parameters PIP_Handle pipe; /* pipe object handle*/

Return Value Uns num; /* number of words to be written in empty frame */

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

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 the pipe frame size and address without actually claiming the pipe frame

C Interface

Syntax

```
framesize = PIP_peek(pipe, addr, rw);
```

Parameters

```
PIP_Handle    pipe;     /* pipe object handle */
Ptr           *addr;    /* the address of the variable that keeps the frame
                           address */
Uns           rw;       /* the flag that indicates the reader or writer side */
```

Return Value

```
Int    framesize;       /* the frame size */
```

Assembly Interface

none

Description

PIP_peek can be used before calling PIP_alloc or PIP_get to get the pipe frame size and address without actually claiming the pipe frame.

The pipe parameter is the pipe object handle, the addr parameter is the address of the variable that keeps the retrieved frame address, and the rw parameter is the flag that indicates what side of the pipe PIP_peek is to operate on. If rw is PIP_READER, then PIP_peek operates on the reader side of the pipe. If rw is PIP_WRITER, then PIP_peek operates on the writer side of the pipe.

PIP_getReaderNumFrames or PIP_getWriterNumFrames can be called to ensure that a frame exists before calling PIP_peek, although PIP_peek returns –1 if no pipe frame exists.

PIP_peek returns the frame size, or –1 if no pipe frames are available. If the return value of PIP_peek in frame size is not –1, then *addr is the location of the frame address.

See Also

PIP_alloc
PIP_free
PIP_get
PIP_put
PIP_reset

PIP_put

Put a full frame into the pipe

C Interface

Syntax PIP_put(pipe);

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Void

Assembly Interface

Syntax PIP_put

Preconditions ar2 = address of the pipe object

Postconditions none

Modifies ag, ah, al, ar2, ar3, ar4, ar5, asm, bg, bh, bl, braf, brc, c, ovb, rea, rsa, sxm, and any registers modified by the notifyReader function

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 (e.g., by calling SWI_andn) the object that reads from this pipe that a full frame is ready to be processed. The notifyReader function is performed as part of the thread that called PIP_get or PIP_put. To avoid problems with recursion, the notifyReader function should not directly call any of the PIP module functions for the same pipe.

Example

See the example for PIP_alloc, page 1–145. The example for HST_getpipe, page 1–84, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_free
PIP_get
HST_getpipe

PIP_reset

Reset all fields of a pipe object to their original values

C Interface

Syntax PIP_reset(pipe);

Parameters PIP_Handle pipe; /* pipe object handle */

Return Value Void

Assembly Interface none

Description

PIP_reset resets all fields of a pipe object to their original values.

The pipe parameter specifies the address of the pipe object that is to be reset.

Constraints and Calling Context

- PIP_reset should not be called between the PIP_alloc call and the PIP_put call or between the PIP_get call and the PIP_free call.
- PIP_reset should be called when interrupts are disabled to avoid the race condition.

See Also

PIP_alloc
PIP_free
PIP_get
PIP_peek
PIP_put

PIP_setWriterSize

Set the number of valid words written to a pipe frame

C Interface

Syntax

```
PIP_setWriterSize(pipe, size);
```

Parameters

```
PIP_Handle   pipe;       /* pipe object handle */
Uns          size;       /* size to be set */
```

Return Value

Void

Assembly Interface

Syntax

none

Preconditions

none

Postconditions

none

Modifies

none

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 1–150.

PRD Module

Periodic function manager

Functions

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

Description

While some applications can schedule functions based on a real-time clock, many applications need to schedule functions based on I/O availability or some other programmatic event.

The PRD module allows you to create PRD objects that schedule periodic execution of program functions. The period may be driven by the CLK module or by calls to PRD_tick whenever a specific event occurs. There can be several PRD objects, but all are driven by the same period counter. Each PRD object can execute its functions at different intervals based on the period counter.

- **To schedule functions based on a real-time clock**. Set the clock interrupt rate you want to use in the Clock Manager property sheet. Put a check mark in the Use On-chip Clock (CLK) box for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object.
- **To schedule functions based on I/O availability or some other event**. Remove the check mark from the Use On-chip Clock (CLK) property field for the Periodic Function Manager. Set the frequency of execution (in number of ticks) in the period field for the individual period object. Your program should call PRD_tick to increment the tick counter.

The function executed by a PRD object is statically defined in the Configuration Tool. PRD functions are called from the context of the function run by the PRD_swi SWI object. PRD functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

The PRD module uses an SWI object (called PRD_swi by default) which itself is triggered on a periodic basis to manage execution of period objects. Normally, this SWI object should have the highest software interrupt priority to allow this software interrupt to be performed once per tick. This software interrupt is automatically created (or deleted) by the Configuration Tool if one or more (or no) PRD objects exist.

See the *TMS320C54x Code Composer Studio Tutorial* for an example that demonstrates the interaction between the PRD module and the SWI module.

When the PRD_swi object runs its function, the following actions occur:

```
for ("Loop through period objects") {
   if ("time for a periodic function")
       "run that periodic function";
}
```

PRD Manager Properties

The DSP/BIOS Periodic Function Manager allows the creation of an arbitrary number of objects that encapsulate a function, two arguments, and a period specifying the time between successive invocations of the function. The period is expressed in ticks, where a tick is defined as a single invocation of the PRD_tick operation. The time between successive invocations of PRD_tick defines the period represented by a tick.

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

- **Object Memory**. The memory section that contains the PRD objects.
- **Use CLK Manager to drive PRD**. If this field is checked, the on-chip timer hardware (managed by CLK) is used to advance the tick count; otherwise, the application must invoke PRD_tick on a periodic basis.
- **Microseconds/Tick**. The number of microseconds between ticks. If the Use CLK Manager to drive PRD field above is checked, this field is automatically set by the CLK module; otherwise, you must explicitly set this field.

PRD Object Properties

The following properties can be set for each PRD object:

- **comment**. Type a comment to identify this PRD object.
- **period (ticks)**. The function executes after period ticks have elapsed.
- **mode**. If continuous is selected the function executes every period ticks; otherwise it executes just once after each call to PRD_tick.
- **function**. The function to be executed
- **arg0, arg1**. Two 16-bit arguments passed to function; these arguments can be either an unsigned 16-bit constant or a symbolic label.

The following informational property is also displayed for each PRD object:

- **period (ms)**. The number of milliseconds represented by the period specified above

PRD - Code Composer Studio Interface

To enable PRD logging, choose Tools→DSP/BIOS→RTA Control Panel and put a check in the appropriate box. You see indicators for PRD ticks in the PRD ticks row of the Execution Graph, which you can open by choosing Tools→DSP/BIOS→Execution Graph. In addition, you see a graph of activity, including PRD function execution.

You can also enable PRD accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a PRD object, you see statistics about the number of ticks that elapsed during execution of the PRD function.

PRD_getticks

Get the current tick count

C Interface

Syntax

num = PRD_getticks();

Parameters

Void

Return Value

LgUns num /* current tick counter */

Assembly Interface

Syntax

PRD_getticks

Preconditions

cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE

Postconditions

ah = upper 16 bits of the 32-bit tick counter
al = lower 16 bits of the 32-bit tick counter

Modifies

ag, ah, al, c

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-chip timer, the tick value is the number of low resolution clock ticks that have occurred since the program started running. When the number of ticks reaches the maximum value that can be stored in 32 bits, the value wraps back to 0. See the CLK Module, page 1–31, for more details.

If the periodic functions are being driven programmatically, the tick value is the number of times PRD_tick has been called.

Example

```
/* ======== showTicks ======== */
Void showTicks()
   {
      LOG_printf(&trace, "ticks = %d", PRD_getticks());
   }
```

See Also

PRD_start
PRD_tick
CLK_gethtime
CLK_getltime
STS_delta

PRD_start

Arm a periodic function for one-time (or continuous) execution

C Interface

Syntax PRD_start(prd);

Parameters PRD_Handle prd; /* prd object handle*/

Return Value Void

Assembly Interface

Syntax PRD_start

Preconditions ar2 = address of the PRD object

Postconditions none

Modifies c

Reentrant no

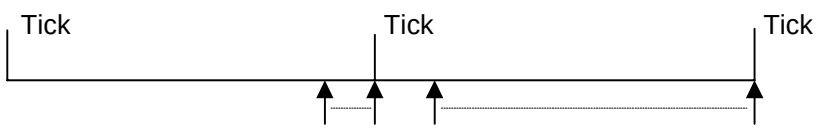
Description

PRD_start starts a period object that has its mode property set to one-shot in the Configuration Tool.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified number of ticks have occurred after a call to PRD_start.

For example, you might have a function that should be executed a certain number of periodic ticks after some condition is met.

When you use PRD_start to start a period object, the exact time the function runs can vary by nearly one tick cycle. As this figure shows, PRD ticks occur at a fixed rate and the call to PRD_start may occur at any point between ticks:



Time to first tick after PRD_start is called.

Due to implementation details, if a PRD function calls PRD_start for a PRD object that is lower in the list of PRD objects, the function sometimes runs a full tick cycle early.

Example

```
/* ======== startClock ======== */
Void startPrd(Int periodID)
    {
       if ("condition met") {
          PRD_start(&periodID);
       }
    }
```

See Also

PRD_tick
PRD_getticks

PRD_stop

Stop a period object to prevent its function execution

C Interface

Syntax PRD_stop(prd);

Parameters PRD_Handle prd; /* prd object handle*/

Return Value Void

Assembly Interface

Syntax PRD_stop

Preconditions ar2 = address of the PRD object

Postconditions none

Modifies c

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 Tool.

Unlike PRD objects that are configured as continuous, one-shot PRD objects do not automatically continue to run. A one-shot PRD object runs its function only after the specified numbers of ticks have occurred after a call to PRD_start.

PRD_stop is the way to stop those one-shot PRD objects once started and before their period counters have run out.

Example

```
PRD_stop(&prd);
```

See Also

PRD_getticks
PRD_start
PRD_tick

PRD_tick

Advance tick counter, enable periodic functions

C Interface

Syntax PRD_tick();

Parameters Void

Return Value Void

Assembly Interface

Syntax PRD_tick

Preconditions intm = 1
cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE

Postconditions dp = GBL_A_SYSPAGE

Modifies ag, ah, al, bg, bh, bl, c, tc

Reentrant no

Description

PRD_tick advances the period counter by one tick. Unless you are driving PRD functions using the on-chip clock, PRD objects execute their functions at intervals based on this counter.

For example, a hardware ISR could perform PRD_tick to notify a periodic function when data is available for processing.

Constraints and Calling Context

- This API should be invoked from interrupt service routines. All the registers that are modified by this API should be saved and restored, before and after the API is invoked, respectively.

See Also

PRD_start
PRD_getticks

QUE Module

Atomic queue manager

Functions

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

Constants, Types, and Structures

```
typedef struct QUE_Obj *QUE_Handle;
                          /* handle for queue object */

struct QUE_Attrs{         /* queue attributes */
    Int    dummy;         /* DUMMY */
};

QUE_Attrs QUE_ATTRS = {   /* default attribute values */
    0,
};

typedef QUE_Elem;         /* queue element */
```

Description

The QUE module makes available a set of functions that manipulate queue objects accessed through handles of type QUE_Handle. Each queue contains an ordered sequence of zero or more elements referenced through variables of type QUE_Elem, which are generally embedded as the first field within some struct.

For example, the DEV_Frame structure which is used by SIO and DEV to enqueue and dequeue I/O buffers is defined as follows:

```
struct DEV_Frame {
    QUE_Elem   link;    /* must be first field! */
    Ptr        addr;
    Uns        size;
}
```

The functions QUE_put and QUE_get are atomic in that they manipulate the queue with interrupts disabled. These functions may therefore be used to safely share queues between tasks, or between tasks and ISRs. All other QUE functions should only be called by tasks, or by tasks and ISRs when they are used in conjunction with some mutual exclusion mechanism (e.g., SEM_pend / SEM_post, TSK_disable / TSK_enable).

QUE Manager Properties

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

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

QUE Object Properties

The following property can be set for a QUE object:

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

QUE_create

Create an empty queue

C Interface

Syntax

```
queue = QUE_create(attrs);
```

Parameters

```
QUE_Attrs     *attrs;    /* pointer to queue attributes */
```

Return Value

```
QUE_Handle    queue;     /* handle for new queue object */
```

Assembly Interface

none

Description

QUE_create creates a new queue which is initially empty. If successful, QUE_create returns the handle of the new queue. If unsuccessful, QUE_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

If attrs is NULL, the new queue is assigned a default set of attributes. Otherwise, the queue's attributes are specified through a structure of type QUE_Attrs.

Note:

At present, no attributes are supported for queue objects, and the type QUE_Attrs is defined as a dummy structure.

All default attribute values are contained in the constant QUE_ATTRS, which may be assigned to a variable of type QUE_Attrs prior to calling QUE_create.

Constraints and Calling Context

- ❏ QUE_create cannot be called by ISRs.
- ❏ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions.

See Also

MEM_alloc
QUE_empty
QUE_delete
SYS_error

QUE_delete

Delete an empty queue

C Interface

Syntax QUE_delete(queue);

Parameters QUE_Handle queue; /* queue handle */

Return Value Void

Assembly Interface

none

Description

QUE_delete uses MEM_free to free the queue object referenced by queue.

Constraints and Calling Context

- ❏ queue must be empty.
- ❏ QUE_delete cannot be called by ISRs.
- ❏ No check is performed to prevent QUE_delete from being used on a statically-created object. If a program attempts to delete a queue object that was created using the Configuration Tool, SYS_error is called.

See Also

QUE_create
QUE_empty

QUE_dequeue

Remove from front of queue (non-atomically)

C Interface

Syntax

```
elem = QUE_dequeue(queue);
```

Parameters

```
QUE_Handle  queue;      /* queue object handle */
```

Return Value

```
Ptr         elem;       /* pointer to former first element */
```

Assembly Interface

none

Description

QUE_dequeue removes the element from the front of queue and returns elem.

Note:

QUE_get must be used instead of QUE_dequeue if queue is shared by multiple tasks, or tasks and ISRs (unless another mutual exclusion mechanism is used).

See Also

QUE_get

QUE_empty

Test for an empty queue

C Interface

Syntax

```
empty = QUE_empty(queue);
```

Parameters

```
QUE_Handle  queue;       /* queue object handle */
```

Return Value

```
Bool        empty;       /* TRUE if queue is empty */
```

Assembly Interface

none

Description

QUE_empty returns TRUE if there are no elements in queue, and FALSE otherwise.

See Also

QUE_get

QUE_enqueue

Insert at end of queue (non-atomically)

C Interface

Syntax

QUE_enqueue(queue, elem);

Parameters

```
QUE_Handle  queue;      /* queue object handle */
Ptr         elem;       /* pointer to queue elem */
```

Return Value

Void

Assembly Interface

none

Description

QUE_enqueue inserts elem at the end of queue.

Note:

QUE_put must be used instead of QUE_enqueue if queue is shared by multiple tasks, or tasks and ISRs (unless another mutual exclusion mechanism is used).

See Also

QUE_put

QUE_get

Get element from front of queue (atomically)

C Interface

Syntax

elem = QUE_get(queue);

Parameters

QUE_Handle queue; /* queue object handle */

Return Value

Void *elem; /* pointer to former first element */

Assembly Interface

none

Description

QUE_get removes the element from the front of queue and returns elem.

Since QUE_get manipulates queue with interrupts disabled, queue may be shared by multiple tasks, or by tasks and ISRs.

Calling QUE_get with an empty queue returns the queue itself. This provides a means for using a single atomic action to check if a queue is empty, and to remove and return the first element if it is not empty:

```
if ((QUE_Handle)(elem = QUE_get(q)) != q)
    ‘ process elem ‘
```

See Also

QUE_create
QUE_empty
QUE_put

QUE_head

Return element at front of queue

C Interface

Syntax

```
elem = QUE_head(queue);
```

Parameters

```
QUE_Handle  queue;       /* queue object handle */
```

Return Value

```
QUE_Elem    *elem;       /* pointer to first element */
```

Assembly Interface

none

Description

QUE_head returns a pointer to the element at the front of queue. The element is not removed from the queue.

Calling QUE_head with an empty queue returns the queue itself.

See Also

QUE_create
QUE_empty
QUE_put

QUE_insert

Insert in middle of queue (non-atomically)

C Interface

Syntax

```
QUE_insert(qelem, elem);
```

Parameters

```
Ptr qelem;      /* element already in queue */
Ptr elem;       /* element to be inserted in queue */
```

Return Value

Void

Assembly Interface

none

Description

QUE_insert inserts elem in the queue in front of qelem.

Note:

If the queue is shared by multiple tasks, or tasks and ISRs, QUE_insert should be used in conjunction with some mutual exclusion mechanism (e.g., SEM_pend/ SEM_post, TSK_disable / TSK_enable).

See Also

QUE_head
QUE_next
QUE_prev
QUE_remove

QUE_new

Set a queue to be empty

C Interface

Syntax

```
QUE_new(queue);
```

Parameters

```
QUE_Handle    queue;    /* pointer to queue object */
```

Return Value

Void

Assembly Interface

none

Description

QUE_new adjusts a queue object to make the queue empty. This operation is not atomic. A typical use of QUE_new is to initialize a queue object that has been statically declared instead of being created with QUE_create. Note that if the queue is not empty, the element(s) in the queue are not freed or otherwise handled, but are simply abandoned.

See Also

QUE_create
QUE_delete
QUE_empty

QUE_next

Return next element in queue (non-atomically)

C Interface

Syntax

```
elem = QUE_next(qelem);
```

Parameters

```
Ptr qelem;      /* element in queue */
```

Return Value

```
Ptr elem;       /* next element in queue */
```

Assembly Interface

none

Description

QUE_next returns elem which points to the element in the queue after qelem.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE_next to return a pointer to the queue itself. Be careful not to call QUE_remove(elem) in this case.

Note:

If the queue is shared by multiple tasks, or tasks and ISRs, QUE_next should be used in conjunction with some mutual exclusion mechanism (e.g., SEM_pend/ SEM_post, TSK_disable / TSK_enable).

See Also

QUE_get
QUE_insert
QUE_prev
QUE_remove

QUE_prev

Return previous element in queue (non-atomically)

C Interface

Syntax

```
elem = QUE_prev(qelem);
```

Parameters

```
Ptr qelem;      /* element in queue */
```

Return Value

```
Ptr elem;       /* previous element in queue */
```

Assembly Interface

none

Description

QUE_prev returns elem which points to the element in the queue before qelem.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, it is possible for QUE_prev to return a pointer to the queue itself. Be careful not to call QUE_remove(elem) in this case.

Note:

If the queue is shared by multiple tasks, or tasks and ISRs, QUE_prev should be used in conjunction with some mutual exclusion mechanism (e.g., SEM_pend/ SEM_post, TSK_disable / TSK_enable).

See Also

QUE_head
QUE_insert
QUE_next
QUE_remove

QUE_put

Put element at end of queue (atomically)

C Interface

Syntax

```
QUE_put(queue, elem);
```

Parameters

```
QUE_Handle  queue;       /* queue object handle */
Void        *elem;       /* pointer to new queue element */
```

Return Value

Void

Assembly Interface

none

Description

QUE_put puts elem at the end of queue.

Since QUE_put manipulates queue with interrupts disabled, queue may be shared by multiple tasks, or by tasks and ISRs.

See Also

QUE_get
QUE_head

QUE_remove

Remove from middle of queue (non-atomically)

C Interface

Syntax QUE_remove(qelem);

Parameters Ptr qelem; /* element in queue */

Return Value Void

Assembly Interface none

Description

QUE_remove removes qelem from the queue.

Since QUE queues are implemented as doubly linked lists with a dummy node at the head, be careful not to remove the header node. This can happen when qelem is the return value of QUE_next or QUE_prev. The following code sample shows how qelem should be verified before calling QUE_remove.

```
QUE_Elem *qelem;.

/* get pointer to first element in the queue */
qelem = QUE_head(queue);

/* scan entire queue for desired element */
while (qelem != queue) {
    if(‘ qelem is the elem we’re looking for ‘) {
         break;
    }
    qelem = QUE_next(queue);
}

/* make sure qelem is not the queue itself */
if (qelem != queue) {
    QUE_remove(qelem);
}
```

Note:

If the queue is shared by multiple tasks, or tasks and ISRs, QUE_remove should be used in conjunction with some mutual exclusion mechanism (e.g., SEM_pend/ SEM_post and TSK_disable / TSK_enable).

Constraints and Calling Context

- ❑ QUE_remove should not be called when qelem is equal to the queue itself.

See Also

QUE_head
QUE_insert
QUE_next
QUE_prev

RTDX Module

Real-Time Data Exchange Settings

RTDX Data Declaration Macros

- RTDX_CreateInputChannel
- RTDX_CreateOutputChannel

Functions

- RTDX_channelBusy
- RTDX_disableInput
- RTDX_disableOutput
- RTDX_enableInput
- RTDX_enableOutput
- RTDX_read
- RTDX_readNB
- RTDX_sizeofInput
- RTDX_write

Macros

- RTDX_isInputEnabled
- RTDX_isOutputEnabled

Description

The RTDX module provides the data types and functions for:

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

Data channels are represented by globally declared structures. A data channel may be used either for input or output, but not both. The contents of an input or output structure are not known to the user. A channel structure contains two states: enabled and disabled. When a channel is enabled, any data written to the channel is sent to the host. Channels are initialized to be disabled.

RTDX Manager Properties

The following settings refer to target configuration parameters:

- **Enable Real-Time Data Exchange (RTDX)**. This box should be checked if you want to link RTDX support into your application.
- **RTDX Data Segment**. The memory section used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this section.
- **RTDX Buffer Size (MAUs)**. The size of the RTDX target-to-host message buffer, in minimum addressable units (MAUs). The default size is 1032 to accommodate a full 1024 byte block and two control words. HST channels that use RTDX are limited by this parameter.

- **RTDX Text Segment**. The code sections for the RTDX module are placed in this section.
- **RTDX Interrupt Mask**. This mask identifies RTDX clients and protect RTDX critical sections. The mask specifies the interrupts to be temporarily disabled inside RTDX critical sections. This also temporarily disables other RTDX clients and prevents another RTDX function call. See the RTDX on-line help for details.

RTDX Object Properties

The following properties can be set for an RTDX object:

- **comment**. Type a comment to identify this RTDX object.
- **Channel Mode**. Select output if the RTDX channel handles output from the DSP to the host. Select input if the RTDX channel handles input to the DSP from the host.

RTDX_CreateInputChannel *Declare input channel structure*

C Interface

Syntax RTDX_CreateInputChannel(ichan);

Parameters ichan /* Label for the input channel */

Return Value none

Assembly Interface none

Description

This macro declares and initializes the RTDX data channel for input.

Data channels must be declared as global objects. A data channel may be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.

A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.

Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer or its OLE interface.

See Also

RTDX_CreateOutputChannel

RTDX_CreateOutputChannel *Declare output channel structure*

C Interface

Syntax RTDX_CreateOutputChannel(ochan);

Parameters ochan /* Label for the output channel */

Return Value none

Assembly Interface none

Description

This macro declares and initializes the RTDX data channels for output.

Data channels must be declared as global objects. A data channel may be used either for input or output, but not both. The contents of an input or output data channel are unknown to the user.

A channel can be in one of two states: enabled or disabled. Channels are initialized as disabled.

Channels can be enabled or disabled via a User Interface function. They can also be enabled or disabled remotely from Code Composer or its OLE interface.

See Also

RTDX_CreateInputChannel

RTDX_channelBusy *Return status indicating whether data channel is busy*

C Interface

Syntax int RTDX_channelBusy(RTDX_inputChannel *pichan);

Parameters pichan /* Identifier for the input data channel */

Return Value int /* Status: 0 = Channel is not busy. non-zero = Channel is busy. */

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Note:

No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

RTDX_channelBusy is designed to be used in conjunction with RTDX_readNB. The return value indicates whether the specified data channel is currently in use or not.

See Also

RTDX_readNB

RTDX_disableInput *Disable an input data channel*

C Interface

Syntax void RTDX_disableInput(RTDX_inputChannel *ichan);

Parameters ichan /* Identifier for the input data channel */

Return Value void

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

A call to a disable function causes the specified channel to be disabled.

See Also

RTDX_disableOutput
RTDX_enableInput
RTDX_read

RTDX_disableOutput *Enable or disable a data channel*

C Interface

Syntax void RTDX_disableOutput(RTDX_outputChannel *ochan);

Parameters ochan /* Identifier for an output data channel */

Return Value void

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

A call to an disable function causes the specified data channel to be disabled.

See Also

RTDX_disableInput
RTDX_enableOutput
RTDX_write

RTDX_enableInput *Enable or disable a data channel*

C Interface

Syntax void RTDX_enableInput(RTDX_inputChannel *ichan);

Parameters
```
ochan   /* Identifier for an output data channel */
ichan   /* Identifier for the input data channel */
```

Return Value void

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

A call to an enable function causes the specified data channel to be enabled.

See Also

RTDX_disableInput
RTDX_enableOutput
RTDX_read

RTDX_enableOutput *Enable or disable a data channel*

C Interface

Syntax void RTDX_enableOutput(RTDX_outputChannel *ochan);

Parameters ochan /* Identifier for an output data channel */

Return Value void

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

A call to an enable function causes the specified data channel to be enabled.

See Also

RTDX_disableOutput
RTDX_enableInput
RTDX_write

RTDX_isInputEnabled

Return status of the input data channel

C Interface

Syntax

```
RTDX_isInputEnabled( c );
```

Parameter

```
c            /* Identifier for an input channel. */
```

Return Value

```
0            /* Not enabled. */
non-zero     /* Enabled. */
```

Description

The RTDX_isInputEnabled macro returns the enabled status of a data channel.

See Also

RTDX_isOutputEnabled

RTDX_isOutputEnabled *Return status of the output data channel*

C Interface

Syntax RTDX_isOutputEnabled(c);

Parameter c /* Identifier for an output channel. */

Return Value 0 /* Not enabled. */
non-zero /* Enabled. */

Description

The RTDX_isOutputEnabled macro returns the enabled status of a data channel.

See Also

RTDX_isInputEnabled

RTDX_read

Read from an input channel

C Interface

Syntax

```
int RTDX_read( RTDX_inputChannel *ichan, void *buffer, int bsize );
```

Parameters

```
ichan     /* Identifier for the input data channel */
buffer    /* A pointer to the buffer that receives the data */
bsize     /* The size of the buffer in address units */
```

Return Value

```
> 0   /* The number of address units of data actually supplied in buffer. */
0     /* Failure. Cannot post read request because target buffer is full. */
RTDX_READ_ERROR/* Failure. Channel currently busy or not enabled. */
```

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

RTDX_read causes a read request to be posted to the specified input data channel. If the channel is enabled, RTDX_read busy waits until the data has arrived. On return from the function, the data has been copied into the specified buffer and the number of address units of data actually supplied is returned. The function returns RTDX_READ_ERROR immediately if the channel is currently busy reading or is not enabled.

When RTDX_read is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host Library to write data into the target buffer. When the data is received, the target application continues execution.

When the function RTDX_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data but the target application does not wait. Execution of the target application continues immediately. Use

the RTDX_channelBusy and RTDX_sizeofInput functions to determine when the RTDX Host Library has written data into the target buffer.

See Also

RTDX_channelBusy
RTDX_readNB
RTDX_sizeofInput

RTDX_readNB *Read from input channel without blocking*

C Interface

Syntax

```
int RTDX_readNB( RTDX_inputChannel *ichan, void *buffer, int bsize );
```

Parameters

```
ichan    /* Identifier for the input data channel */
buffer   /* A pointer to the buffer that receives the data */
bsize    /* The size of the buffer in address units */
```

Return Value

RTDX_OK Success.
0 (zero) Failure. The target buffer is full.
RTDX_READ_ERRORChannel is currently busy reading.

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

RTDX_readNB is a nonblocking form of the function RTDX_read. RTDX_readNB issues a read request to be posted to the specified input data channel and immediately returns. If the channel is not enabled or the channel is currently busy reading, the function returns RTDX_READ_ERROR. The function returns 0 if it cannot post the read request due to lack of space in the RTDX target buffer.

When the function RTDX_readNB is used, the target application notifies the RTDX Host Library that it is ready to receive data but the target application does not wait. Execution of the target application continues immediately. Use the RTDX_channelBusy and RTDX_sizeofInput functions to determine when the RTDX Host Library has written data into the target buffer.

When RTDX_read is used, the target application notifies the RTDX Host Library that it is ready to receive data and then waits for the RTDX Host

Library to write data into the target buffer. When the data is received, the target application continues execution.

See Also

RTDX_channelBusy
RTDX_read
RTDX_sizeofInput

RTDX_sizeofInput *Return the number of bytes read from a data channel*

C Interface

Syntax int RTDX_sizeofInput(RTDX_inputChannel *pichan);

Parameters pichan /* Identifier for the input data channel */

Return Value int /* Number of sizeof units of data actually supplied in buffer */

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

Note:

No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

RTDX_sizeofInput is designed to be used in conjunction with RTDX_readNB after a read operation has completed. The function returns the number of sizeof units actually read from the specified data channel.

See Also

RTDX_readNB

RTDX_write

Write to an output channel

C Interface

Syntax

```
int RTDX_write( RTDX_outputChannel *ochan, void *buffer, int bsize );
```

Parameters

```
ochan     /* Identifier for the output data channel */
buffer    /* A pointer to the buffer containing the data */
bsize     /* The size of the buffer in address units */
```

Return Value

```
int       /* Status: non-zero = Success. 0 = Failure. */
```

Assembly Interface

Syntax none

Preconditions none

Postconditions none

Modifies none

Reentrant yes

> **Note:**
>
> No assembly macro is provided for this API. See the *TMS320C5400 Optimizing C Compiler User's Guide* for more information.

Description

RTDX_write causes the specified data to be written to the specified output data channel, provided that channel is enabled. On return from the function, the data has been copied out of the specified user buffer and into the RTDX target buffer. If the channel is not enabled, the write operation is suppressed. If the RTDX target buffer is full, Failure is returned.

See Also

RTDX_read

SEM Module

Semaphore Manager

Functions

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

Constants, Types, and Structures

```
typedef struct SEM_Obj  *SEM_Handle;
                        /* handle for semaphore object */

struct SEM_Attrs {      /* semaphore attributes */
    Int    dummy;       /* DUMMY */
};

SEM_Attrs SEM_ATTRS = { /* default attribute values */
    0,
};
```

Description

The SEM module makes available a set of functions that manipulate semaphore objects accessed through handles of type SEM_Handle. SEM semaphores are counting semaphores that may be used for both task synchronization and mutual exclusion.

SEM_pend is used to wait for a semaphore. The timeout parameter to SEM_pend allows the task to wait until a timeout, wait indefinitely, or not wait at all. SEM_pend's return value is used to indicate if the semaphore was signaled successfully.

SEM_post is used to signal a semaphore. If a task is waiting for the semaphore, SEM_post removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, SEM_post simply increments the semaphore count and returns.

SEM Manager Properties

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

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

SEM Object Properties

The following properties can be set for a SEM object:

- **comment**. Type a comment to identify this SEM object.
- **Initial semaphore count**. Set this property to the desired initial semaphore count.

SEM - Code Composer Studio Interface

The SEM tab of the Kernel/Object View shows information about semaphore objects.

SEM_count

Get current semaphore count

C Interface

Syntax count = SEM_count(sem);

Parameters SEM_Handle sem; /* semaphore handle */

Return Value Int count; /* current semaphore count */

Assembly Interface none

Description

SEM_count returns the current value of the semaphore specified by sem.

SEM_create

Create a semaphore

C Interface

Syntax

```
sem = SEM_create(count, attrs);
```

Parameters

```
Int          count;    /* initial semaphore count */
SEM_Attrs    *attrs;   /* pointer to semaphore attributes */
```

Return Value

```
SEM_Handle  sem;      /* handle for new semaphore object */
```

Assembly Interface

none

Description

SEM_create creates a new semaphore object which is initialized to count. If successful, SEM_create returns the handle of the new semaphore. If unsuccessful, SEM_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

If attrs is NULL, the new semaphore is assigned a default set of attributes. Otherwise, the semaphore's attributes are specified through a structure of type SEM_Attrs.

Note:

At present, no attributes are supported for semaphore objects, and the type SEM_Attrs is defined as a dummy structure.

All default attribute values are contained in the constant SEM_ATTRS, which may be assigned to a variable of type SEM_Attrs prior to calling SEM_create.

No task switch occurs when calling SEM_create.

Constraints and Calling Context

- ❏ count must be greater than or equal to 0.
- ❏ SEM_create cannot be called by ISRs.
- ❏ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions.

See Also

MEM_alloc
SEM_delete
SYS_error

SEM_delete

Delete a semaphore

C Interface

Syntax

```
SEM_delete(sem);
```

Parameters

```
SEM_Handle    sem;    /* semaphore object handle */
```

Return Value

Void

Assembly Interface

none

Description

SEM_delete uses MEM_free to free the semaphore object referenced by sem.

No task switch occurs when calling SEM_delete.

Constraints and Calling Context

- ❏ No tasks should be pending on sem when SEM_delete is called.
- ❏ SEM_delete cannot be called by ISRs.
- ❏ No check is performed to prevent SEM_delete from being used on a statically-created object. If a program attempts to delete a semaphore object that was created using the Configuration Tool, SYS_error is called.

See Also

SEM_create

SEM_ipost

Signal a semaphore (interrupt use only)

C Interface

Syntax

```
SEM_ipost(sem);
```

Parameters

```
SEM_Handle    sem;    /* semaphore object handle */
```

Return Value

Void

Assembly Interface

none

Description

SEM_ipost readies the first task waiting for the semaphore. If no task is waiting, SEM_ipost simply increments the semaphore count and returns.

SEM_ipost is the same as SEM_post in the DSP/BIOS environment. You may call either in an ISR if you have disabled task switching or prepared the ISR for a task switch to occur. SEM_ipost is provided for source compatibility reasons only. For portable code, use SEM_ipost within an ISR and SEM_post within a task.

See Also

SEM_pend
SEM_post

SEM_new

Initialize semaphore object

C Interface

Syntax

Void SEM_new(sem, count);

Parameters

```
SEM_Handle    sem;      /* pointer to semaphore object */
Int           count;    /* initial semaphore count */
```

Return Value

Void

Assembly Interface

none

Description

SEM_new initializes the semaphore object pointed to by sem with count. No task switch occurs when calling SEM_new.

Constraints and Calling Context

- ❏ count must be greater than or equal to 0.

See Also

QUE_new

SEM_pend

Wait for a semaphore

C Interface

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 */
```

Assembly Interface

none

Description

If the semaphore count is greater than zero, SEM_pend decrements the count and returns TRUE. Otherwise, SEM_pend suspends the execution of the current task until SEM_post is called or the timeout expires. If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

If timeout is SYS_FOREVER, the task remains suspended until SEM_post is called on this semaphore. If timeout is 0, SEM_pend returns immediately.

If timeout expires (or timeout is 0) before the semaphore is available, SEM_pend returns FALSE. Otherwise SEM_pend returns TRUE.

A task switch occurs when calling SEM_pend if the semaphore count is 0 and timeout is not zero.

Constraints and Calling Context

- ❏ SEM_pend may only be called from an ISR if timeout is 0.
- ❏ SEM_pend cannot be called from within a TSK_disable / TSK_enable block.
- ❏ SEM_pend should not be called from within an IDL function. Doing so prevents DSP/BIOS plug-ins from gathering run-time information.

See Also

SEM_post

SEM_post

Signal a semaphore

C Interface

Syntax

SEM_post(sem);

Parameters

SEM_Handle sem; /* semaphore object handle */

Return Value

Void

Assembly Interface

none

Description

SEM_post readies the first task waiting for the semaphore. If no task is waiting, SEM_post simply increments the semaphore count and returns.

A task switch occurs when calling SEM_post if a higher priority task is made ready to run.

See Also

SEM_ipost
SEM_pend

SEM_reset

Reset semaphore count

C Interface

Syntax

SEM_reset(sem, count);

Parameters

```
SEM_Handle  sem;      /* semaphore object handle */
Int         count;    /* semaphore count */
```

Return Value

Void

Assembly Interface

none

Description

SEM_reset resets the semaphore count to count.

No task switch occurs when calling SEM_reset.

Constraints and Calling Context

- ❏ count must be greater than or equal to 0.
- ❏ No tasks should be waiting for sem when SEM_reset is called.

See Also

SEM_create

SIO Module

Stream input and output manager

Functions

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

Constants, Types, and Structures

```
#define SIO_STANDARD     0 /* open stream for */
                           /* standard streaming model */
#define SIO_ISSUERECLAIM 1 /* open stream for */
                           /* issue/reclaim streaming model */

#define SIO_INPUT       0  /* open for input */
#define SIO_OUTPUT      1  /* open for output */
typedef SIO_Handle;        /* stream object handle */

struct SIO_Attrs {      /* stream attributes */
    Int     nbufs;      /* number of buffers */
    Int     segid;      /* buffer section ID */
    Int     align;      /* buffer alignment */
    Bool    flush       /* TRUE = don't block in DEV_idle() */
    Uns     model;      /* usage model: */
                        /* SIO_STANDARD/SIO_ISSUERECLAIM */
    Uns     timeout;    /* timeout value used by device*/
};

SIO_Attrs SIO_ATTRS = {
    2,                      /* nbufs */
    0,                      /* segid */
    0,                      /* align */
    FALSE,                  /* flush */
    SIO_STANDARD,           /* model */
    SYS_FOREVER             /* timeout */
};
```

Description

The stream manager provides efficient real-time device-independent I/O through a set of functions that manipulate stream objects accessed through handles of type SIO_Handle. The device independence is afforded by having a common high-level abstraction appropriate for real-time applications—continuous streams of data—that can be associated with a variety of devices. All I/O programming is done in a high-level manner using these stream handles to the devices and the stream manager takes care of dispatching into the underlying device drivers.

For efficiency, streams are treated as sequences of fixed-size buffers of data rather than just sequences of bytes.

Streams can be opened and closed at any point during program execution using the functions SIO_create and SIO_delete, respectively.

The SIO_issue and SIO_reclaim function calls are enhancements to the basic DSP/BIOS device model. These functions provide a second usage model for streaming, referred to as the issue/reclaim model. It is a more flexible streaming model that allows clients to supply their own buffers to a stream, and to get them back in the order that they were submitted. The SIO_issue and SIO_reclaim functions also provide a user argument that can be used for passing information between the stream client and the stream devices.

SIO Manager Properties

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

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

SIO Object Properties

The following properties can be set for an SIO object:

- **comment**. Type a comment to identify this SIO object.
- **Device**. Select the device to which you want to bind this SIO object. User-defined devices are listed along with DGN and DPI devices.
- **Device Control Parameter**. Type the device suffix to be passed to any devices stacked below the device connected to this stream.
- **Mode**. Select input if this stream is to be used for input to the application program and output if this stream is to be used for output.
- **Buffer size**. If this stream uses the Standard model, this property controls the size of buffers allocated for use by the steam. If this stream uses the Issue/Reclaim model, the stream can handle buffers of any size.

- **Number of buffers**. If this stream uses the Standard model, this property controls the number of buffers allocated for use by the steam. If this stream uses the Issue/Reclaim model, the stream can handle up to the specified Number of buffers.
- **Place buffers in memory segment**. Select the memory section to contain the stream buffers if Model is Standard.
- **Buffer alignment**. Specify the memory alignment to use for stream buffers if Model is Standard. For example, if you select 16, the buffer must begin at an address that is a multiple of 16. The default is 1, which means the buffer can begin at any address.
- **Flush**. Check this box if you want the stream to discard all pending data and return without blocking if this object is idled at run-time with SIO_idle.
- **Model**. Select Standard if you want all buffers to be allocated when the stream is created. Select Issue/Reclaim if your program is to allocate the buffers and supply them using SIO_issue.
- **Allocate Static Buffer(s)**. If this box is checked, the Configuration Tool allocates stream buffers for the user. The SIO_staticbuf function is used to acquire these buffers from the stream. When the Standard model is used, checking this box causes one buffer more than the Number of buffers property to be allocated. When the Issue/Reclaim model is used, buffers are not normally allocated. Checking this box causes the number of buffers specified by the Number of buffers property to be allocated.
- **Timeout for I/O operation**. This parameter specifies the length of time SIO_reclaim waits for I/O. SIO_reclaim passes this value to the driver's Dxx_reclaim function. If the timeout expires before a buffer is available, SIO_reclaim returns SYS_ETIMEOUT and no buffer is returned.

SIO_bufsize

Return the size of the buffers used by a stream

C Interface

Syntax

```
size = SIO_bufsize(stream);
```

Parameters

```
SIO_Handle stream;
```

Return Value

```
Uns      size;
```

Assembly Interface

none

Description

SIO_bufsize returns the size of the buffers used by stream.

See Also

SIO_segid

SIO_create

Open a stream

C Interface

Syntax

```
stream = SIO_create(name, mode, bufsize, attrs);
```

Parameters

```
String      name;     /* name of device */
Int         mode;     /* SIO_INPUT or SIO_OUTPUT */
Uns         bufsize;  /* stream buffer size */
SIO_Attrs  *attrs;    /* pointer to stream attributes */
```

Return Value

```
SIO_Handle stream;    /* stream object handle */
```

Assembly Interface

none

Description

SIO_create creates a new stream object and opens the device specified by name. If successful, SIO_create returns the handle of the new stream object. If unsuccessful, SIO_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

The mode parameter specifies whether the stream is to be used for input (SIO_INPUT) or output (SIO_OUTPUT).

If the stream is being opened in SIO_STANDARD mode, SIO_create allocates buffers of size bufsize for use by the stream. Initially these buffers are placed on the device todevice queue for input streams, and the device fromdevice queue for output streams.

If the stream is being opened in SIO_ISSUERECLAIM mode, SIO_create does not allocate any buffers for the stream. In SIO_ISSUERECLAIM mode all buffers must be supplied by the client via the SIO_issue call. It does, however, prepare the stream for a maximum number of buffers of the specified size.

If the attrs parameter is NULL, the new stream is assigned the default set of attributes specified by SIO_ATTRS. The following stream attributes are currently supported:

```
struct SIO_Attrs {
   Int    nbufs;
   Int    segid;
   Int    align;
   Bool   flush;
   Uns    model;
   Uns    timeout;
};
```

The nbufs attribute specifies the number of buffers allocated by the stream in the SIO_STANDARD usage model, or the number of buffers to prepare for in

the SIO_ISSUERECLAIM usage model. The default value of nbufs is 2. In the SIO_ISSUERECLAIM usage model, nbufs is the maximum number of buffers that can be outstanding (i.e., issued but not reclaimed) at any point in time.

The segid attribute specifies the memory section for stream buffers. Use the memory section names defined using the Configuration Tool. The default value is 0, meaning that buffers are to be allocated from the Segment for DSP/BIOS objects defined for the MEM manager.

The align attribute specifies the memory alignment for stream buffers. The default value is 0, meaning that no alignment is needed.

The flush attribute indicates the desired behavior for an output stream when it is deleted. If flush is TRUE, a call to SIO_delete or SIO_idle causes the stream to discard all pending data and return without blocking. If flush is FALSE, a call to SIO_delete or SIO_idle causes the stream to block until all pending data has been processed. The default value is FALSE.

The model attribute indicates the usage model that is to be used with this stream. The two usage models are SIO_ISSUERECLAIM and SIO_STANDARD. The default usage model is SIO_STANDARD.

The timeout attribute specifies the length of time the device driver waits for I/O completion before returning an error (e.g., SYS_ETIMEOUT). timeout is usually passed as a parameter to SEM_pend by the device driver. The default is SYS_FOREVER which indicates that the driver waits forever. If timeout is SYS_FOREVER, the task remains suspended until a buffer is available to be returned by the stream. If timeout is 0, SIO_reclaim returns immediately. If the timeout expires before a buffer is available to be returned, SIO_reclaim returns (-1 * SYS_ETIMEOUT). Otherwise SIO_reclaim returns the number of valid bytes in the buffer, or -1 multiplied by an error code.

Constraints and Calling Context

- A stream can only be used by one task simultaneously. Catastrophic failure may result if more than one task calls SIO_get (or SIO_issue / SIO_reclaim) on the same input stream, or more than one task calls SIO_put (or SIO_issue / SIO_reclaim) on the same output stream.
- SIO_create creates a stream dynamically. Do not call SIO_create on a stream that was created with the Configuration Tool.
- You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions. However, streams that are to be used with stacking drivers must be created dynamically with SIO_create.

See Also

MEM_alloc
SEM_pend
SIO_delete
SIO_issue
SIO_reclaim
SYS_error

SIO_ctrl

Perform a device-dependent control operation

C Interface

Syntax

```
status = SIO_ctrl(stream, cmd, arg);
```

Parameters

```
SIO_Handle stream;    /* stream handle */
Uns        cmd;       /* command to device */
Arg        arg;       /* arbitrary argument */
```

Return Value

```
Int    status;    /* device status */
```

Assembly Interface

none

Description

SIO_ctrl causes a control operation to be issued to the device associated with stream. cmd and arg are passed directly to the device.

SIO_ctrl returns SYS_OK if successful, and a non-zero device-dependent error value if unsuccessful.

See Also

Dxx_ctrl

SIO_delete

Close a stream and free its buffers

C Interface

Syntax

```
status = SIO_delete(stream);
```

Parameters

```
SIO_Handle  stream;       /* stream object */
```

Return Value

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

Assembly Interface

none

Description

SIO_delete idles the device before freeing the stream object and buffers.

If the stream being deleted was opened for input, then any pending input data is discarded. If the stream being deleted was opened for output, the method for handling data is determined by the value of the object's Flush property in the Configuration Tool or the flush field in the SIO_Attrs structure (passed in with SIO_create). If flush is TRUE, SIO_delete discards all pending data and return without blocking. If flush is FALSE, SIO_delete blocks until all pending data has been processed by the stream.

SIO_delete returns SYS_OK if and only if the operation is successful.

Constraints and Calling Context

- ❏ SIO_delete cannot be called by ISRs.
- ❏ No check is performed to prevent SIO_delete from being used on a statically-created object. If a program attempts to delete a stream object that was created using the Configuration Tool, SYS_error is called.

See Also

SIO_create
SIO_flush
SIO_idle

SIO_flush

Flush a stream

C Interface

Syntax

```
status = SIO_flush(stream);
```

Parameters

```
SIO_Handle  stream;       /* stream handle */
```

Return Value

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

Assembly Interface

none

Description

SIO_flush causes all pending data to be discarded regardless of the mode of the stream. SIO_flush differs from SIO_idle in that SIO_flush never suspends program execution to complete processing of data, even for a stream created in output mode.

The underlying device connected to stream is idled as a result of calling SIO_flush. In general, the interrupt is disabled for the device.

One of the purposes of this function is to provide synchronization with the external environment.

SIO_flush returns SYS_OK if and only if the stream is successfully idled.

See Also

SIO_create
SIO_idle

SIO_get

Get a buffer from stream

C Interface

Syntax

```
nbytes = SIO_get(stream, bufp);
```

Parameters

```
SIO_Handle   stream;    /* stream handle */
Ptr          *bufp;     /* pointer to a buffer */
```

Return Value

```
Int          nbytes;    /* number of bytes read or error if negative */
```

Assembly Interface

none

Description

SIO_get exchanges an empty buffer with a non-empty buffer from stream. The bufp parameter is an input/output parameter which points to an empty buffer when SIO_get is called. When SIO_get returns, bufp points to a new (different) buffer, and nbytes indicates success or failure of the SIO_get call.

To indicate success, SIO_get returns a positive value for nbytes. As a success indicator, nbytes is the number of bytes received from the stream. To indicate failure, SIO_get returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO_get.

A task switch occurs when calling SIO_get if there are no non-empty data buffers in stream.

Constraints and Calling Context

- The stream must not be created with attrs.model set to SIO_ISSUERECLAIM. The results of calling SIO_get on a stream created for the issue/reclaim streaming model are undefined.

See Also

SIO_create
SIO_put

SIO_idle

Idle a stream

C Interface

Syntax

```
status = SIO_idle(stream);
```

Parameters

```
SIO_Handle  stream;       /* stream handle */
```

Return Value

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

Assembly Interface

none

Description

If stream is being used for output, SIO_idle causes any currently buffered data to be transferred to the output device associated with stream. SIO_idle suspends program execution for as long as is required for the data to be consumed by the underlying device.

If stream is being used for input, SIO_idle causes any currently buffered data to be discarded. The underlying device connected to stream is idled as a result of calling SIO_idle. In general, the interrupt is disabled for this device.

If stream is being used for output, the method for handling data is determined by the value of the object's Flush property in the Configuration Tool or the flush field in the SIO_Attrs structure (passed in with SIO_create). If flush is TRUE, SIO_idle discards all pending data and return without blocking. If flush is FALSE, SIO_idle blocks until all pending data has been processed by the stream.

One of the purposes of this function is to provide synchronization with the external environment.

SIO_idle returns SYS_OK if and only if the stream is successfully idled.

See Also

SIO_create
SIO_flush

SIO_issue

Send a buffer to a stream

C Interface

Syntax

```
status = SIO_issue(stream, pbuf, nbytes, arg);
```

Parameters

```
SIO_Handle  stream;    /* stream handle */
Ptr         pbuf;      /* pointer to a buffer */
Uns         nbytes;    /* number of bytes in the buffer */
Arg         arg;       /* user argument */
```

Return Value

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

Assembly Interface

none

Description

SIO_issue is used to send a buffer and its related information to a stream. The buffer-related information consists of the logical length of the buffer (nbytes), and the user argument to be associated with that buffer. SIO_issue sends a buffer to the stream and return to the caller without blocking. It also returns an error code indicating success (SYS_OK) or failure of the call.

Failure of SIO_issue indicates that the stream was not able to accept the buffer being issued or that there was a device error when the underlying Dxx_issue was called. In the first case, the application is probably issuing more frames than the maximum nbufs allowed for the stream, before it reclaims any frames. In the second case, the failure reveals an underlying device driver or hardware problem. If SIO_issue fails, SIO_idle should be called for an SIO_INPUT stream, and SIO_flush should be called for an SIO_OUTPUT stream, before attempting more I/O through the stream.

The interpretation of nbytes, the logical size of a buffer, is direction-dependent. For a stream opened in SIO_OUTPUT mode, the logical size of the buffer indicates the number of valid bytes of data it contains. For a stream opened in SIO_INPUT mode, the logical length of a buffer indicates the number of bytes being requested by the client. In either case, the logical size of the buffer must be less than or equal to the physical size of the buffer.

The argument arg is not interpreted by DSP/BIOS, but is offered as a service to the stream client. DSP/BIOS and all DSP/BIOS-compliant device drivers preserve the value of arg and maintain its association with the data that it was issued with. arg provides a user argument as a method for a client to associate additional information with a particular buffer of data.

SIO_issue is used in conjunction with SIO_reclaim to operate a stream opened in SIO_ISSUERECLAIM mode. The SIO_issue call sends a buffer to a stream, and SIO_reclaim retrieves a buffer from a stream. In normal operation each SIO_issue call is followed by an SIO_reclaim call. Short

bursts of multiple SIO_issue calls can be made without an intervening SIO_reclaim call, but over the life of the stream SIO_issue and SIO_reclaim must be called the same number of times.

At any given point in the life of a stream, the number of SIO_issue calls can exceed the number of SIO_reclaim calls by a maximum of nbufs. The value of nbufs is determined by the SIO_create call or by setting the Number of buffers property for the object in the Configuration Tool.

Note:

An SIO_reclaim call should not be made without at least one outstanding SIO_issue call. Calling SIO_reclaim with no outstanding SIO_issue calls has undefined results.

Constraints and Calling Context

- The stream must be created with attrs.model set to SIO_ISSUERECLAIM.

See Also

SIO_create
SIO_reclaim

SIO_put

Put a buffer to a stream

C Interface

Syntax

```
nbytes = SIO_put(stream, bufp, nbytes);
```

Parameters

```
SIO_Handle    stream;     /* stream handle */
Ptr           *bufp;      /* pointer to a buffer */
Uns           nbytes;     /* number of bytes in the buffer */
```

Return Value

```
Int       nbytes;     /* number of bytes returned or negative if error */
```

Assembly Interface

none

Description

SIO_put exchanges a non-empty buffer with an empty buffer from stream. The bufp parameter is an input/output parameter which points to a non-empty buffer when SIO_put is called. When SIO_put returns, bufp points to a new (different) buffer, and nbytes indicates success or failure of the SIO_put call.

To indicate success, SIO_put returns a positive value for nbytes. As a success indicator, nbytes is the number of valid bytes in the buffer returned by the stream (usually zero). To indicate failure, SIO_put returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Since this operation is generally accomplished by redirection rather than by copying data, references to the contents of the buffer pointed to by bufp must be recomputed after the call to SIO_put.

A task switch occurs when calling SIO_put if there are no empty data buffers in stream.

Constraints and Calling Context

- ❑ The stream must not be created with attrs.model set to SIO_ISSUERECLAIM. The results of calling SIO_put on a stream created for the issue/reclaim model are undefined.

See Also

SIO_create
SIO_get

SIO_reclaim

Request a buffer back from a stream

C Interface

Syntax

```
nbytes = SIO_reclaim(stream, pbufp, parg);
```

Parameters

```
SIO_Handle  stream;   /* stream handle */
Ptr         *pbufp;   /* pointer to the buffer */
Arg         *parg;    /* pointer to a user argument */
```

Return Value

```
Int     nbytes;   /* number of bytes or error if negative */
```

Assembly Interface

none

Description

SIO_reclaim is used to request a buffer back from a stream. It returns a pointer to the buffer, the number of valid bytes in the buffer, and a user argument (parg). After the SIO_reclaim call parg points to the same value that was passed in with this buffer using the SIO_issue call.

If the stream was created in SIO_OUTPUT mode, then SIO_reclaim returns an empty buffer, and nbytes is zero, since the buffer is empty. If the stream was opened in SIO_INPUT mode, SIO_reclaim returns a non-empty buffer, and nbytes is the number of valid bytes of data in the buffer. In either mode SIO_reclaim blocks until a buffer can be returned to the caller, or until the timeout expires, and it returns a positive number or zero (indicating success), or a negative number (indicating an error condition). If timeout is not equal to SYS_FOREVER or 0, the task suspension time can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

To indicate success, SIO_reclaim returns a positive value for nbytes. As a success indicator, nbytes is the number of valid bytes in the buffer. To indicate failure, SIO_reclaim returns a negative value for nbytes. As a failure indicator, nbytes is the actual error code multiplied by -1.

Failure of SIO_reclaim indicates that no buffer was returned to the client. Therefore, if SIO_reclaim fails, the client should not attempt to de-reference pbufp, since it is not guaranteed to contain a valid buffer pointer.

SIO_reclaim is used in conjunction with SIO_issue to operate a stream opened in SIO_ISSUERECLAIM mode. The SIO_issue call sends a buffer to a stream, and SIO_reclaim retrieves a buffer from a stream. In normal operation each SIO_issue call is followed by an SIO_reclaim call. Short bursts of multiple SIO_issue calls can be made without an intervening SIO_reclaim call, but over the life of the stream SIO_issue and SIO_reclaim must be called the same number of times. The number of SIO_issue calls can exceed the number of SIO_reclaim calls by a maximum of nbufs at any given

time. The value of nbufs is determined by the SIO_create call or by setting the Number of buffers property for the object in the Configuration Tool.

Note:

An SIO_reclaim call should not be made without at least one outstanding SIO_issue call. Calling SIO_reclaim with no outstanding SIO_issue calls has undefined results.

SIO_reclaim only returns buffers that were passed in using SIO_issue. It also returns the buffers in the same order that they were issued.

A task switch occurs when calling SIO_reclaim if timeout is not set to 0, and there are no data buffers available to be returned.

Constraints and Calling Context

- ❏ The stream must be created with attrs.model set to SIO_ISSUERECLAIM.
- ❏ There must be at least one outstanding SIO_issue when an SIO_reclaim call is made.
- ❏ All frames issued to a stream must be reclaimed before closing the stream.

See Also

SIO_issue
SIO_create

SIO_segid

Return the memory segment used by the stream

C Interface

Syntax

```
segid = SIO_segid(stream);
```

Parameters

```
SIO_Handle stream;
```

Return Value

```
Int     segid;      /* memory segment ID */
```

Assembly Interface

none

Description

SIO_segid returns the identifier of the memory section that stream uses for buffers.

See Also

SIO_bufsize

SIO_select

Select a ready device

C Interface

Syntax

```
mask = SIO_select(streamtab, nstreams, timeout);
```

Parameters

```
SIO_Handle streamtab[];   /* stream table */
Int        nstreams;      /* number of streams */
Uns        timeout;       /* return after this many system clock ticks */
```

Return Value

```
Uns    mask;    /* stream ready mask */
```

Assembly Interface

none

Description

SIO_select waits until one or more of the streams in the streamtab[] array is ready for I/O (i.e., it does not block when an I/O operation is attempted).

streamtab[] is an array of streams where nstreams < 16. The timeout parameter indicates the number of system clock ticks to wait before a stream becomes ready. If timeout is 0, SIO_select returns immediately. If timeout is SYS_FOREVER, SIO_select waits until one of the streams is ready. Otherwise, SIO_select waits for up to 1 system clock tick less than timeout due to granularity in system timekeeping.

The return value is a mask indicating which streams are ready for I/O. A 1 in bit position j indicates the stream streamtab[j] is ready.

Constraints and Calling Context

- ❏ streamtab must contain handles of type SIO_Handle returned from prior calls to SIO_create.
- ❏ streamtab[] is an array of streams; streamtab[i] corresponds to bit position i in mask.

See Also

SIO_get
SIO_put
SIO_reclaim

SIO_staticbuf

Acquire static buffer from stream

C Interface

Syntax

```
nbytes = SIO_staticbuf(stream, bufp);
```

Parameters

```
SIO_Handle stream;   /* stream handle */
Ptr        *bufp;    /* pointer to a buffer */
```

Return Value

```
Int        nbytes;   /* number of bytes in buffer */
```

Assembly Interface

none

Description

SIO_staticbuf returns buffers for static streams that were configured using the Configuration Tool. Buffers are allocated for static streams by checking the Allocate Static Buffer(s) check box for the related SIO object.

SIO_staticbuf returns the size of the buffer or 0 if no more buffers are available from the stream.

SIO_staticbuf can be called multiple times for SIO_ISSUERECLAIM model streams.

SIO_staticbuf must be called to acquire all static buffers before calling SIO_get, SIO_put, SIO_issue or SIO_reclaim.

Constraints and Calling Context

- SIO_staticbuf should only be called for streams that are defined statically using the Configuration Tool.
- SIO_staticbuf should only be called for static streams whose Allocate Static Buffer(s) check box has been checked.
- SIO_staticbuf cannot be called after SIO_get, SIO_put, SIO_issue or SIO_reclaim have been called for the given stream.

See Also

SIO_get

STS Module

Statistics Objects Manager

Functions

- ❏ STS_add. Update statistics using provided value
- ❏ STS_delta. Update statistics using difference between provided value and setpoint
- ❏ STS_reset. Reset values stored in STS object
- ❏ STS_set. Save a setpoint value

Constants, Types, and Structures

```
struct STS_Obj {
    LgInt    num;     /* count */
    LgInt    acc;      /* total value */
    LgInt    max;     /* maximum value */
}
```

Note:

STS objects should not be shared across threads. Therefore, STS_add, STS_delta, STS_reset, and STS_set are not reentrant.

Description

The STS module manages objects called statistics accumulators. Each STS object accumulates the following statistical information about an arbitrary 32-bit wide data series:

- ❏ **Count**. The number of values in an application-supplied data series
- ❏ **Total**. The sum of the individual data values in this series
- ❏ **Maximum**. The largest value already encountered in this series

Using the count and total, the Statistics View plug-in calculates the average on the host.

Statistics are accumulated in 32-bit variables on the target and in 64-bit variables on the host. When the host polls the target for real-time statistics, it resets the variables on the target. This minimizes space requirements on the target while allowing you to keep statistics for long test runs.

Default STS Tracing

In the RTA Control Panel, you can enable statistics tracing for the following modules by marking the appropriate checkbox. You can also set the HWI object properties to perform various STS operations on registers, addresses, or pointers.

Your program does not need to include any calls to STS functions in order to gather these statistics. The units for the statistics values are controlled by the Statistics Units property of the manager for the module being traced:

Module	Units
HWI	Gather statistics on monitored values within HWIs
PIP	Number of frames read from or written to data pipe (count only)
PRD	Number of ticks elapsed from start to end of execution
SWI	Instruction cycles elapsed from time posted to completion

Custom STS Objects

You can create custom STS objects using the Configuration Tool. The STS_add operation updates the count, total, and maximum using the value you provide. The STS_set operation sets a previous value. The STS_delta operation accumulates the difference between the value you pass and the previous value and updates the previous value to the value you pass.

By using custom STS objects and the STS operations, you can do the following:

- **Count the number of occurrences of an event**. You can pass a value of 0 to STS_add. The count statistic tracks how many times your program calls STS_add for this STS object.
- **Track the maximum and average values for a variable in your program**. For example, suppose you pass amplitude values to STS_add. The count tracks how many times your program calls STS_add for this STS object. The total is the sum of all the amplitudes. The maximum is the largest value. The Statistics View calculates the average amplitude.
- **Track the minimum value for a variable in your program**. Negate the values you are monitoring and pass them to STS_add. The maximum is the negative of the minimum value.
- **Time events or monitor incremental differences in a value**. For example, suppose you want to measure the time between hardware interrupts. You would call STS_set when the program begins running and STS_delta each time the interrupt routine runs, passing the result of CLK_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.

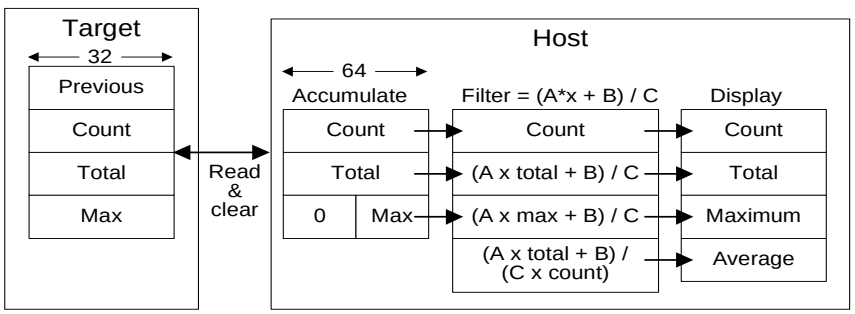
You can further customize the statistics data by setting the STS object properties to apply a printf format to the Total, Max, and Average fields in the Statistics View window and choosing a formula to apply to the data values on the host.

Statistics Data Gathering by the Statistics View Plug-in

The statistics manager allows the creation of any number of statistics objects, which in turn can be used by the application to accumulate simple statistics about a time series. This information includes the 32-bit maximum value, the last 32-bit value passed to the object, the number of samples (up to 232 - 1 samples), and the 32-bit sum of all samples.

These statistics are accumulated on the target in real-time until the host reads and clears these values on the target. The host, however, continues to accumulate the values read from the target in a host buffer which is displayed by the Statistics View real-time analysis tool. Provided that the host reads and clears the target statistics objects faster than the target can overflow the 32-bit wide values being accumulated, no information loss occurs.

Using the Configuration Tool, you can select a Host Operation for an STS object. The statistics are filtered on the host using the operation and variables you specify. This figure shows the effects of the (A x X + B) / C operation.



STS Manager Properties

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

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

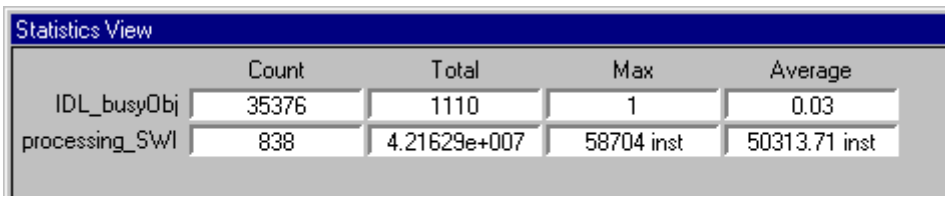
STS Object Properties

The following properties can be set for a statistics object:

- **comment**. Type a comment to identify this STS object
- **prev**. The initial 32-bit history value to use in this object
- **format**. The printf-style format string used to display the data for this object
- **host operation**. The expression evaluated (by the host) on the data for this object before it is displayed by the Statistics View real-time analysis tool. The operation can be:
  - A x X
  - A x X + B
  - (A x X + B) / C
- **A, B, C**. The integer parameters used by the expression specified by the Host Operation field above.

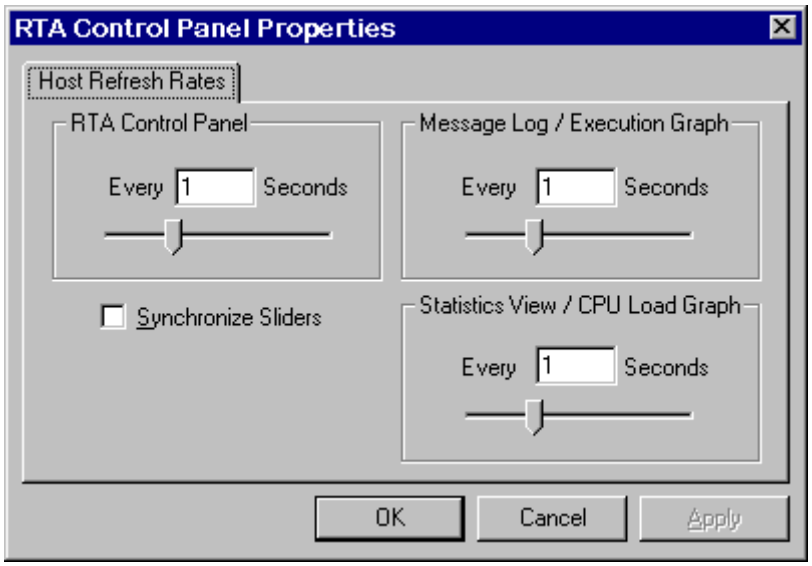
STS - Statistics View Interface

You can view statistics in real-time with the Statistics View plug-in by choosing the Tools→DSP/BIOS→Statistics View menu item.



To pause the display, right-click on this window and choose Pause from the pop-up menu. To reset the values to 0, right-click on this window and choose Clear from the pop-up menu.

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



See the *TMS320C54x Code Composer Studio Tutorial* for more information on how to monitor statistics with the Statistics View plug-in.

STS_add

Update statistics using the provided value

C Interface

Syntax

```
STS_add(sts, value);
```

Parameters

```
STS_Handle  sts;     /* statistics object handle */
LgInt    value;      /* new value to update statistics object */
```

Return Value

Void

Assembly Interface

Syntax

```
STS_add
```

Preconditions

ar2 = address of the STS object
a = 32-bit value
sxm = 1

Postconditions

none

Modifies

ag, ah, al, ar2, bg, bh, bl, c, ovb

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 plug-in calculates the average amplitude.

You can count the occurrences of an event by passing a dummy value (such as 0) to STS_add and watching the Count field.

You can view the statistics values with the Statistics View plug-in by enabling statistics in the Tools→DSP/BIOS→RTA Control Panel window and choosing your custom STS object in the Tools→DSP/BIOS→Statistics View window.

See Also

STS_delta
STS_reset
STS_set
TRC_disable
TRC_enable

STS_delta

Update statistics using the difference between the provided value and the setpoint

C Interface

Syntax STS_delta(sts,value);

Parameters
```
STS_Handle  sts;     /* statistics object handle */
LgInt    value;      /* new value to update statistics object */
```

Return Value Void

Assembly Interface

Syntax STS_delta

Preconditions
ar2 = address of the STS object
a = 32-bit value
sxm = 1

Postconditions none

Modifies ag, ah, al, ar2, bg, bh, bl, c, ovb

Reentrant no

Description

Each STS object contains a previous value that can be initialized with the Configuration Tool or with a call to STS_set. A call to STS_delta subtracts the previous value from the value it is passed and then invokes STS_add with the result to update the statistics. STS_delta also updates the previous value with the value it is passed.

STS_delta can be used in conjunction with STS_set to monitor the difference between a variable and a desired value or to benchmark program performance.

You can benchmark your code by using paired calls to STS_set and STS_delta that pass the value provided by CLK_gethtime.

```
STS_set(&sts, CLK_gethltime());
  "processing to be benchmarked"
STS_delta(&sts, CLK_gethltime());
```

Constraints and Calling Context

- Before the first call to STS_delta is made, the previous value of the STS object should be initialized either with a call to STS_set or by setting the prev property of the STS object using the Configuration Tool.

Example

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

See Also

STS_add
STS_reset
STS_set
CLK_gethtime
CLK_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

Assembly Interface

Syntax STS_reset

Preconditions ar2 = address of the STS object

Postconditions none

Modifies ag, ah, al, ar2, c

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 plug-in polls statistics data on the target, it performs STS_reset internally. This keeps the 32-bit total and count values from wrapping back to 0 on the target. The host accumulates these values as 64-bit numbers to allow a much larger range than can be stored on the target.

Example

```
STS_reset(&sts);
STS_set(&sts, value);
```

See Also

STS_add
STS_delta
STS_set
TRC_disable
TRC_enable

STS_set

Save a value for STS_delta

C Interface

Syntax

STS_set(sts, value);

Parameters

```
STS_Handle  sts;    /* statistics object handle */
LgInt    value;     /* new value to update statistics object */
```

Return Value

Void

Assembly Interface

Syntax

STS_set

Preconditions

ar2 = address of the STS object
a = 32-bit value

Postconditions

none

Modifies

none

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 may need to use STS_set to reset the previous value before the next call to STS_delta.

You can also set a previous value for an STS object in the Configuration Tool. STS_set changes this value.

See STS_delta for details on how to use the value you set with STS_set.

Example

This example gathers performance information for the processing between STS_set and STS_delta.

```
STS_set(&sts, CLK_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

SWI Module

Software interrupt manager

Functions

- ❏ SWI_andn. Clear bits from SWI's mailbox; post if becomes 0
- ❏ SWI_create. Create a software interrupt
- ❏ SWI_dec. Decrement SWI's mailbox value; post if becomes 0
- ❏ SWI_delete. Delete a software interrupt
- ❏ SWI_disable. Disable software interrupts
- ❏ SWI_enable. Enable software interrupts
- ❏ SWI_getattrs. Get attributes of a software interrupt
- ❏ SWI_getmbox. Return an SWI's mailbox value
- ❏ SWI_getpri. Return an SWI's priority mask
- ❏ SWI_inc. Increment SWI's mailbox value
- ❏ SWI_or. Or mask with value contained in SWI's mailbox field
- ❏ SWI_post. Post a software interrupt
- ❏ SWI_raisepri. Raise an SWI's priority
- ❏ SWI_restorepri. Restore an SWI's priority
- ❏ SWI_self. Return address of currently executing SWI object
- ❏ SWI_setattrs. Set attributes of a software interrupt

Description

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt service routines.

DSP/BIOS manages four distinct levels of execution threads: hardware interrupt service routines, software interrupts, tasks, and background idle functions. A software interrupt is an object that encapsulates a function to be executed and a priority. Software interrupts are prioritized, preempt tasks, and are preempted by hardware interrupt service routines.

Note:

SWI functions are called after the processor register state has been saved. SWI functions can be written in C or assembly and must follow the C calling conventions described in the compiler manual.

Note: All processor registers are saved before calling SWI functions. This includes st0, st1, a, b, ar0-ar7, the T registers, bk, brc, rsa, rea, and pmst. The following status register bits are set to 0 before calling the user function: ARP, C16, CMPT, CPL, FRCT, and OVM. If the function is a C function, specified with a leading underscore in the Configuration Tool, CPL is set to 1 before calling the function.

Each software interrupt has a priority level. A software interrupt preempts any lower-priority software interrupt currently executing.

A target program uses an API call to post an SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a software interrupt is posted by an API call, the SWI object's function is not executed immediately. Instead, the function is scheduled for execution. DSP/BIOS uses the software interrupt's priority to determine whether to preempt the thread currently running. Note that if a software interrupt is posted several times before it begins running, because HWIs and higher priority interrupts are running, the software interrupt only runs one time.

Software interrupts can be scheduled for execution with a call to SWI_post or a number of other SWI functions. Each SWI object has a 16-bit mailbox which is used either to determine whether to post the software interrupt or as a value that can be evaluated within the software interrupt's function. SWI_andn and SWI_dec post the software interrupt if the mailbox value transitions to 0. SWI_or and SWI_inc also modify the mailbox value. (SWI_or sets bits, and SWI_andn clears bits.)

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

The SWI_disable and SWI_enable operations allow you to post several software interrupts and enable them all for execution at the same time. The software interrupt priorities then determine which software interrupt runs first.

All software interrupts run to completion; you cannot suspend a software interrupt while it waits for something—e.g., a device—to be ready. So, you can use the mailbox to tell the software interrupt when all the devices and other conditions it relies on are ready. Within a software interrupt processing function, a call to SWI_getmbox returns the value of the mailbox when the

software interrupt started running. The mailbox is automatically reset to its original value when a software interrupt runs.

Software interrupts can have up to 15 priority levels. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

A software interrupt preempts any currently running software interrupt with a lower priority. If two software interrupts with the same priority level have been posted, the software interrupt that was posted first runs first. Hardware interrupts in turn preempt any currently running software interrupt, allowing the target to respond quickly to hardware peripherals. For information about setting software interrupt priorities, you can choose Help–>Help Topics in the Configuration Tool, click the Index tab, and type priority.

Interrupt threads—including hardware interrupts and software interrupts—are all executed using the same stack. A context switch is performed when a new thread is added to the top of the stack. The SWI module automatically saves the processor's registers before running a higher-priority software interrupt that preempts a lower-priority software interrupt. After the higher-priority software interrupt finishes running, the registers are restored and the lower-priority software interrupt can run if no other higher-priority software interrupts have been posted. (A separate task stack is used by each task thread.)

See the *TMS320C54x Code Composer Studio Tutorial* for more information on how to post software interrupts and scheduling issues for the Software Interrupt manager.

SWI Manager Properties

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

- **Object Memory**. The memory section that contains the SWI objects.
- **Statistics Units**. The units used to display the elapsed instruction cycles or time from when a software interrupt is posted to its completion within the Statistics View plug-in. Raw causes the STS Data to display the number of instruction cycles if the CLK module's Use high resolution time for internal timings parameter is set to True (the default). If this CLK parameter is set to False and the Statistics Units is set to Raw, SWI statistics are displayed in units of timer interrupt periods. You can also choose milliseconds or microseconds.

SWI Object Properties

The following properties can be set for an SWI object:

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

- **priority**. This field shows the numeric priority level for this SWI object. Software interrupts can have up to 15 priority levels. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler. Instead of typing a number in this field, you change the relative priority levels of SWI objects by dragging the objects in the ordered collection view.
- **function**. The function to execute
- **mailbox**. The initial value of the 16-bit word used to determine if this software interrupt should be posted.
- **arg0, arg1**. Two 16-bit arguments passed to function; these arguments can be either an unsigned 16-bit constant or a symbolic label.

SWI - Code Composer Studio Interface

The SWI tab of the Kernel/Object View shows information about software interrupt objects.

To enable SWI logging, choose Tools→DSP/BIOS→RTA Control Panel and put a check in the appropriate box. To view a graph of activity that includes SWI function execution, choose Tools→DSP/BIOS→Execution Graph.

You can also enable SWI accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose an SWI object, you see statistics about the number of instruction cycles elapsed from the time the SWI was posted to the SWI function's completion.

SWI_andn

Clear bits from SWI's mailbox and post if mailbox becomes 0

C Interface

Syntax

SWI_andn(swi, mask);

Parameters

```
SWI_Handle    swi;        /* SWI object handle*/
Uns           mask        /* value to be ANDed */
```

Return Value

Void

Assembly Interface

Syntax

SWI_andn

Preconditions

cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
ar2 = address of the SWI object
al = mask
intm = 0 (if called outside the context of an ISR)

Postconditions

none

Modifies

ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

Reentrant

yes

Description

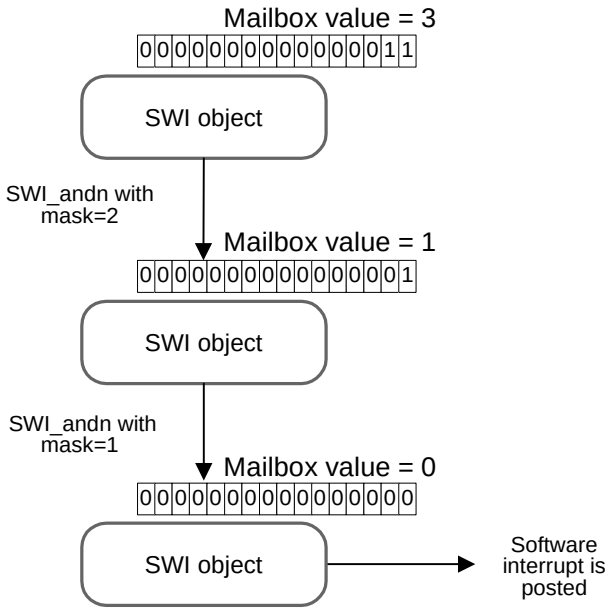
SWI_andn is used to conditionally post a software interrupt. SWI_andn clears the bits specified by a mask from SWI's internal mailbox. If SWI's mailbox becomes 0, SWI_andn posts the software interrupt. The bitwise logical operation performed is:

```
mailbox = mailbox AND (NOT MASK)
```

For example, if there are multiple conditions that must all be met before a software interrupt can run, you should use a different bit in the mailbox for each condition. When a condition is met, clear the bit for that condition.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

The following figure shows an example of how a mailbox with an initial value of 3 can be cleared by two calls to SWI_andn with values of 2 and 1. The entire mailbox could also be cleared with a single call to SWI_andn with a value of 3.



Constraints and Calling Context

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

Example

```
/* ======== ioReady ======== */

   Void ioReady(unsigned int mask)
   {
     SWI_andn(&copySWI, mask);   /* clear bits of "ready mask" */
   }
```

See Also

SWI_dec
SWI_getmbox
SWI_inc
SWI_or
SWI_post
SWI_self

SWI_create

Create a software interrupt

C Interface

Syntax

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

Parameters

```
SWI_Attrs    *attrs;        /* pointer to swi attributes */
```

Return Value

```
SWI_Handle    swi;         /* handle for new swi object */
```

Assembly Interface

none

Description

SWI_create creates a new SWI object. If successful, SWI_create returns the handle of the new SWI object. If unsuccessful, SWI_create returns NULL unless it aborts. For example, SWI_create may abort if it directly or indirectly calls SYS_error, and SYS_error is configured to abort.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the object to be created, facilitates setting the SWI object's attributes. If attrs is NULL, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type SWI_attrs defined as follows:

```
struct SWI_Attrs {
   SWI_Fxn  fxn;
   Arg    arg0;
   Arg    arg1;
   Int    priority;
   Uns    mailbox;
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The priority attribute specifies the SWI object's execution priority and must range from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant SWI_ATTRS, which may be assigned to a variable of type SWI_Attrs prior to calling SWI_create.

Constraints and Calling Context

- ❏ SWI_create cannot be called by ISRs.
- ❏ The fxn attribute cannot be NULL.
- ❏ The priority attribute must be less than or equal to 14 and greater than or equal to 0.

See Also

SWI_delete
SWI_getattrs
SWI_setattrs
SYS_error

SWI_dec

Decrement SWI's mailbox value and post if mailbox becomes 0

C Interface

Syntax SWI_dec(swi);

Parameters SWI_Handle swi; /* SWI object handle*/

Return Value Void

Assembly Interface

Syntax SWI_dec

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
ar2 = address of the SWI object
intm = 0 (if called outside the context of an ISR)

Postconditions none

Modifies ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

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 software interrupt. You can increment a mailbox value by using SWI_inc, which always posts the software interrupt.

For example, you would use SWI_dec if you wanted to post a software interrupt after a number of occurrences of an event.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

Example

```
/* ======== strikeOrBall ======== */

   Void strikeOrBall(unsigned int call)
   {
      if (call == 1) {
        SWI_dec(&strikeoutSwi); /* initial mailbox value is 3 */
      }
      if (call == 2) {
       SWI_dec(&walkSwi);      /* initial mailbox value is 4 */
      }
   }
```

See Also

SWI_delete
SWI_getmbox
SWI_inc
SWI_or
SWI_post
SWI_self

SWI_delete

Delete a software interrupt

C Interface

Syntax SWI_delete(swi);

Parameters SWI_Handle swi; /* SWI object handle */

Return Value Void

Assembly Interface none

Description

SWI_delete uses MEM_free to free the SWI object referenced by swi.

Constraints and Calling Context

- ❏ swi cannot be the currently executing SWI object (SWI_self)
- ❏ SWI_delete cannot be called by ISRs.
- ❏ SWI_delete must not be used to delete a statically-created SWI object. No check is performed to prevent SWI_delete from being used on a statically-created object. If a program attempts to delete a SWI object that was created using the Configuration Tool, SYS_error is called.

See Also

SWI_create
SWI_getattrs
SWI_setattrs
SYS_error

SWI_disable

Disable software interrupts

C Interface

Syntax SWI_disable();

Parameters Void

Return Value Void

Assembly Interface

Syntax SWI_disable

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE

Postconditions none

Modifies c

Reentrant yes

Description

SWI_disable and SWI_enable control SWI software interrupt processing. SWI_disable disables all other SWI functions from running until SWI_enable is called. Hardware interrupts can still run.

SWI_disable and SWI_enable allow you to ensure that statements that must be performed together during critical processing are not interrupted. In the following example, the critical section is not preempted by any software interrupts.

```
SWI_disable();
   ‘critical section‘
SWI_enable();
```

You can also use SWI_disable and SWI_enable to post several software interrupts and allow them to be performed in priority order. See the example that follows.

SWI_disable calls can be nested—the number of nesting levels is stored internally. Software interrupt handling is not reenabled until SWI_enable has been called as many times as SWI_disable.

Constraints and Calling Context

- The calls to HWI_enter and HWI_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenable software interrupt handling. You should not call SWI_disable or SWI_enable within a hardware ISR.

Example

```
/* ======== postEm ======== */

   Void postEm()
   {
      SWI_disable();

      SWI_post(&encoderSwi);
      SWI_andn(&copySwi, mask);
      SWI_dec(&strikeoutSwi);

      SWI_enable();
   }
```

See Also

HWI_disable
HWI_enable
SWI_enable

SWI_enable

Enable software interrupts

C Interface

Syntax SWI_enable();

Parameters Void

Return Value Void

Assembly Interface

Syntax SWI_enable

Preconditions can only be called if SWI_disable was called before
cpl = 0
dp = GBL_A_SYSPAGE

Postconditions none

Modifies ag, ah, al, c

Reentrant yes

Description

SWI_disable and SWI_enable control SWI software interrupt processing. SWI_disable disables all other software interrupt functions from running until SWI_enable is called. Hardware interrupts can still run. See the SWI_disable section for details.

SWI_disable calls can be nested—the number of nesting levels is stored internally. Software interrupt handling is not be reenabled until SWI_enable has been called as many times as SWI_disable.

Constraints and Calling Context

- The calls to HWI_enter and HWI_exit required in any hardware ISRs that schedules software interrupts automatically disable and reenable software interrupt handling. You should not call SWI_disable or SWI_enable within a hardware ISR.

See Also

HWI_disable
HWI_enable
SWI_disable

SWI_getattrs

Get attributes of a software interrupt

C Interface

Syntax

SWI_getattrs(swi, attrs);

Parameters

```
SWI_Handle    swi;      /* handle of the swi */
SWI_Attrs     *attrs;   /* pointer to swi attributes */
```

Return Value

Void

Assembly Interface

none

Description

SWI_getattrs retrieves attributes of an existing SWI object.

The swi parameter specifies the address of the SWI object whose attributes are to be retrieved. The attrs parameter, which is the pointer to a structure that contains the retrieved attributes for the SWI object, facilitates retrieval of the attributes of the SWI object.

The SWI object's attributes are specified through a structure of type SWI_attrs defined as follows:

```
struct SWI_Attrs {
   SWI_Fxn    fxn;
   Arg    arg0;
   Arg    arg1;
   Int    priority;
   Uns    mailbox;
};
```

The fxn attribute, which is the address of the SWI function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the SWI function, fxn.

The priority attribute specifies the SWI object's execution priority and ranges from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

The following example uses SWI_getattrs:

```
extern  SWI_Handle swi;
SWI_Attrs attrs;

SWI_getattrs(swi, &attrs);
attrs.priority = 5;
SWI_setattrs(swi, &attrs);
```

Constraints and Calling Context

- ❏ SWI_getattrs cannot be called by ISRs.
- ❏ The attrs parameter cannot be NULL.

See Also

SWI_create
SWI_delete
SWI_setattrs

SWI_getmbox

Return a SWI's mailbox value

C Interface

Syntax **num =** Uns SWI_getmbox();

Parameters Void

Return Value Uns num /* mailbox value */

Assembly Interface

Syntax SWI_getmbox

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE

Postconditions alaccl = current software interrupt's mailbox value

Modifies ag, ah, al, c

Reentrant yes

Description

SWI_getmbox returns the value that SWI's mailbox had when the software interrupt started running. DSP/BIOS saves the mailbox value internally so that SWI_getmbox can access it at any point within an SWI object's function. DSP/BIOS then automatically resets the mailbox to its initial value (defined with the Configuration Tool) so that other threads can continue to use the software interrupt's mailbox.

SWI_getmbox should only be called within a function run by a SWI object.

The value returned by SWI_getmbox may be non-zero if the SWI was posted by a call to SWI_andn or SWI_dec. Therefore, SWI_getmbox provides relevant information only if the SWI was posted by a call to SWI_or, SWI_inc, or SWI_post.

Example

This example could be used within a SWI object's function to use the value of the mailbox within the function. For example, if you use SWI_or or SWI_inc to post a software interrupt, different mailbox values may require different processing.

```
/* get current SWI mailbox value */
swicount = SWI_getmbox();
```

See Also

SWI_andn
SWI_dec
SWI_inc
SWI_or
SWI_post
SWI_self

SWI_getpri

Return a SWI's priority mask

C Interface

Syntax

```
key =  SWI_getpri(swi);
```

Parameters

```
SWI_Handle    swi;    /* SWI object handle*/
```

Return Value

```
Uns    key          /* Priority mask of swi */
```

Assembly Interface

Syntax

```
SWI_getpri
```

Preconditions

ar2 = address of the SWI object

Postconditions

a = SWI object's priority mask

Modifies

ag, ah, al, c

Reentrant

yes

Description

SWI_getpri returns the priority mask of the SWI passed in as the argument.

Example

```
/* Get the priority key of swi1 */
key = SWI_getpri(&swi1);
/* Get the priorities of swi1 and swi3 */
key = SWI_getpri(&swi1) | SWI_getpri(&swi3);
```

See Also

SWI_raisepri
SWI_restorepri

SWI_inc

Increment SWI's mailbox value

C Interface

Syntax SWI_inc(swi);

Parameters SWI_Handle swi: /* SWI object handle*/

Return Value Void

Assembly Interface

Syntax SWI_inc

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
ar2 = address of the SWI object
intm = 0 (if called outside the context of an ISR)

Postconditions none

Modifies ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

Reentrant no

Description

SWI_inc increments the value in SWI's mailbox by 1 and posts the software interrupt regardless of the resulting mailbox value. You can decrement a mailbox value by using SWI_dec, which only posts the software interrupt if the mailbox value is 0.

If a software interrupt is posted several times before it has a chance to begin executing, because HWIs and higher priority software interrupts are running, the software interrupt only runs one time. If this situation occurs, you can use SWI_inc to post the software interrupt. Within the software interrupt's function, you could then use SWI_getmbox to find out how many times this software interrupt has been posted since the last time it was executed.

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI_getmbox.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

Example

```
/* ======== AddAndProcess ======== */

   Void AddAndProcess(int count)
   {
      int i;

      for (i = 1; I <= count; ++i)
          SWI_inc(&MySwi);
      SWI_post(&MySwi);
   }
```

See Also

SWI_andn
SWI_dec
SWI_getmbox
SWI_or
SWI_post
SWI_self

SWI_or

OR mask with the value contained in SWI's mailbox field

C Interface

Syntax

```
SWI_or(swi, mask);
```

Parameters

```
SWI_Handle    swi;        /* SWI object handle*/
Uns           mask;       /* value to be ORed */
```

Return Value

Void

Assembly Interface

Syntax

```
SWI_or
```

Preconditions

cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
ar2 = address of the SWI object
al = mask
intm = 0 (if called outside the context of an ISR)

Postconditions

none

Modifies

ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

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 software interrupt regardless of the resulting mailbox value. The bitwise logical operation performed on the mailbox value is:

```
mailbox = mailbox OR mask
```

You specify a software interrupt's initial mailbox value in the Configuration Tool. The mailbox value is automatically reset when the software interrupt executes. To get the mailbox value, use SWI_getmbox.

For example, you might use SWI_or to post a software interrupt if any of three events should cause a software interrupt to be executed, but you want the software interrupt's function to be able to tell which event occurred. Each event would correspond to a different bit in the mailbox.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

See Also

SWI_andn
SWI_dec
SWI_getmbox
SWI_inc
SWI_post
SWI_self

SWI_post

Post a software interrupt

C Interface

Syntax SWI_post(swi);

Parameters SWI_Handle swi; /* SWI object handle */

Return Value Void

Assembly Interface

Syntax SWI_post

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
ar2 = address of the SWI object
intm = 0 (if called outside the context of an ISR)

Postconditions none

Modifies ag, ah, al, ar0, ar2, ar3, ar4, ar5, bg, bh, bl, c, dp, t, tc

Reentrant no

Description

SWI_post is used to post a software interrupt regardless of the mailbox value. No change is made to SWI's mailbox value.

To have a PRD object post an SWI object's function, you can set _SWI_post as the function property of a PRD object and the name of the software interrupt object you want to post its function as the arg0 property.

Constraints and Calling Context

- If this macro (API) is invoked outside the context of an interrupt service routine, interrupts must be enabled.

See Also

SWI_andn
SWI_dec
SWI_getmbox
SWI_inc
SWI_or
SWI_self

SWI_raisepri

Raise a SWI's priority

C Interface

Syntax key = SWI_raisepri(mask);

Parameters Uns mask; /* mask of desired priority level */

Return value Uns key; /* key for use with SWI_restorepri */

Assembly Interface

Syntax SWI_raisepri

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
a = priority mask of desired priority level

Postconditions a = old priority mask

Modifies ag, ah, al, bg, bh, bl, c

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 software interrupts.

SWI_raisepri should be called before the shared resource is accessed, and SWI_restorepri should be called after the access to the shared resource.

A call to SWI_raisepri not followed by a SWI_restorepri keeps the SWI's priority for the rest of the processing at the raised level. A SWI_post of the SWI posts the SWI at its original priority level.

A SWI object's execution priority must range from 0 to 14. The highest level is SWI_MAXPRI (14). The lowest is SWI_MINPRI (0). The priority level of 0 is reserved for the KNL_swi object, which runs the task scheduler.

SWI_raisepri never lowers the current SWI priority.

Example

```
/* raise priority to the priority of swi_1 */
key = SWI_raisepri(SWI_getpri(&swi_1));
---  access shared resource ---
SWI_restore(key);
```

See Also

SWI_getpri
SWI_restorepri

SWI_restorepri

Restore a SWI's priority

C Interface

Syntax SWI_restorepri(key);

Parameters Uns key; /* key to restore original priority level */

Return Value Void

Assembly Interface

Syntax SWI_restorepri

Preconditions
cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE
a = old priority mask
intm = 0
SWI_D_lock < 0
not in an ISR

Postconditions none

Modifies ag, ah, al, c, intm, tc

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 software interrupts.

SWI_raisepri should be called right before the shared resource is referenced, and SWI_restorepri should be called after the reference to the shared resource.

Constraints and Calling Context

- This macro (API) must not be invoked from an ISR.

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 */

Assembly Interface

Syntax SWI_self

Preconditions cpl = ovm = c16 = frct = cmpt = 0
dp = GBL_A_SYSPAGE

Postconditions al = address of the current SWI object

Modifies ag, ah, al, c

Reentrant yes

Description

SWI_self returns the address of the currently executing software interrupt.

Within a hardware ISR, SWI_self returns the address of the software interrupt highest in the processing stack—i.e., the software interrupt that yielded to the hardware interrupt. If no software interrupt is running or yielding, SWI_self returns NULL.

Example

You can use SWI_self if you want a software interrupt to repost itself:

```
SWI_post(SWI_self());
```

See Also

SWI_andn
SWI_dec
SWI_getmbox
SWI_inc
SWI_or
SWI_post

SWI_setattrs

Set attributes of a software interrupt

C Interface

Syntax

```
SWI_setattrs(swi, attrs);
```

Parameters

```
SWI_Handle    swi;      /* handle of the swi */
SWI_Attrs     *attrs;   /* pointer to swi attributes */
```

Return Value

Void

Assembly Interface

none

Description

SWI_setattrs sets attributes of an existing SWI object.

The swi parameter specifies the address of the SWI object whose attributes are to be set.

The attrs parameter, which can be either NULL or a pointer to a structure that contains attributes for the SWI object, facilitates setting the attributes of the SWI object. If attrs is NULL, the new SWI object is assigned a default set of attributes. Otherwise, the SWI object's attributes are specified through a structure of type SWI_attrs defined as follows:

```
struct SWI_Attrs {
   SWI_Fxn    fxn;
   Arg    arg0;
   Arg    arg1;
   Int    priority;
   Uns    mailbox;
};
```

The fxn attribute, which is the address of the swi function, serves as the entry point of the software interrupt service routine.

The arg0 and arg1 attributes specify the arguments passed to the swi function, fxn.

The priority attribute specifies the SWI object's execution priority and must range from 2 to 15. Priority 15 is the highest priority. You cannot use a priority of 1; that priority is reserved for the system SWI that runs the TSK scheduler.

The mailbox attribute is used either to determine whether to post the SWI or as a value that can be evaluated within the SWI function.

All default attribute values are contained in the constant SWI_ATTRS, which may be assigned to a variable of type SWI_Attrs prior to calling SWI_setattrs.

The following example uses SWI_setattrs:

```
extern  SWI_Handle swi;
SWI_Attrs attrs;

SWI_getattrs(swi, &attrs);
attrs.priority = 5;
SWI_setattrs(swi, &attrs);
```

Constraints and Calling Context

- ❏ SWI_setattrs cannot be called by ISRs.
- ❏ SWI_setattrs should not be used to set the attributes of a SWI that is preempted or is ready to run.
- ❏ The fxn attribute cannot be NULL.
- ❏ The priority attribute must be less than or equal to 15 and greater than or equal to 2.

See Also

SWI_create
SWI_delete
SWI_getattrs

SYS Module

System Settings

Functions

- SYS_abort. Abort program execution
- SYS_atexit. Stack an exit handler
- SYS_error. Flag error condition
- SYS_exit. Terminate program execution
- SYS_printf. Formatted output
- SYS_putchar. Output a single character
- SYS_sprintf. Formatted output to string buffer
- SYS_vprintf. Formatted output, variable argument list
- SYS_vsprintf. Output formatted data

Constants, Types, and Structures

```
#define SYS_FOREVER  (Uns)-1 /* wait forever */
#define SYS_POLL     (Uns)0  /* don't wait */

#define SYS_OK       0     /* no error */
#define SYS_EALLOC   1     /* memory allocation error */
#define SYS_EFREE    2     /* memory free error */
#define SYS_ENODEV   3     /* device driver not found */
#define SYS_EBUSY    4     /* device driver busy */
#define SYS_EINVAL   5     /* invalid parameter for device */
#define SYS_EBADIO   6     /* I/O failure */
#define SYS_EMODE    7     /* bad mode for device driver */
#define SYS_EDOMAIN  8     /* domain error */
#define SYS_ETIMEOUT 9     /* call timed out */
#define SYS_EE0F     10    /* end-of-file */
#define SYS_EDEAD    11    /* previously deleted object */
#define SYS_EBADOBJ  12    /* invalid object */
#define SYS_EUSER    256   /* user errors start here */

#define SYS_NUMHANDLERS  8 /* number of atexit handlers */

extern String SYS_errors[]; /* array of error strings */
```

Description

The SYS module makes available a set of general-purpose functions that provide basic system services, such as halting program execution and printing formatted text. In general, each SYS function is patterned after a similar function normally found in the standard C library.

SYS does not directly use the services of any other DSP/BIOS module and therefore resides at the bottom of the system. Other DSP/BIOS modules use the services provided by SYS in lieu of similar C library functions. The SYS

module provides hooks for binding system-specific code. This allows programs to gain control wherever other DSP/BIOS modules call one of the SYS functions.

SYS Manager Properties

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

- **Trace Buffer Size**. The size of the buffer that contains system trace information. For example, by default the Putc function writes to the trace buffer.
- **Trace Buffer Memory**. The memory section that contains system trace information.
- **Abort function**. The function to run if the application aborts by calling SYS_abort. The default function is _UTL_doAbort, which logs an error message and calls _halt.
  If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- **Error function**. The function to run if an error flagged by SYS_error occurs. The default function is _UTL_doError, which logs an error message.
  If this function is written in C, use a leading underscore before the C function name.
- **Exit function**. The function to run when the application exits by calling SYS_exit. The default function is UTL_halt, which loops forever with interrupts disabled—preventing other processing.
  If this function is written in C, use a leading underscore before the C function name.
- **Putc function**. The function to run if the application calls SYS_putchar, SYS_printf, or SYS_vprintf. The default function is _UTL_doPutc, which writes a character to the trace buffer.
  If this function is written in C, use a leading underscore before the C function name.

SYS Object Properties

The SYS module does not support the creation of individual SYS objects.

SYS_abort

Abort program execution

C Interface

Syntax

SYS_abort(format, [arg,] ...);

Parameters

```
String  format;   /* format specification string */
Arg     arg;      /* optional argument */
```

Return Value

Void

Assembly Interface

none

Description

SYS_abort aborts program execution by calling the function bound to the configuration parameter Abort function, where vargs is of type va_list and represents the sequence of arg parameters originally passed to SYS_abort.

```
(*(Abort_function))(format, vargs)
```

The function bound to Abort function may elect to pass the format and vargs parameters directly to SYS_vprintf or SYS_vsprintf prior to terminating program execution.

The default Abort function for the SYS manager is _UTL_doAbort, which logs an error message and calls UTL _halt, which is defined in the boot.c file. The UTL_halt function performs an infinite loop with all processor interrupts disabled.

Constraints and Calling Context

- If the function bound to Abort function is not reentrant, SYS_abort must be called atomically.

See Also

SYS_exit
SYS_printf

SYS_atexit

Stack an exit handler

C Interface

Syntax

```
success = SYS_atexit(handler);
```

Parameters

```
Fxn     handler     /* exit handler function */
```

Return Value

```
Bool    success     /* handler successfully stacked */
```

Assembly Interface

none

Description

SYS_atexit pushes handler onto an internal stack of functions to be executed when SYS_exit is called. Up to SYS_NUMHANDLERS(8) functions can be specified in this manner. SYS_exit pops the internal stack until empty and calls each function as follows, where status is the parameter passed to SYS_exit:

```
(*handler)(status)
```

SYS_atexit returns TRUE if handler has been successfully stacked; FALSE if the internal stack is full.

The handlers on the stack are called only if either of the following happens:

- ❏ SYS_exit is called.
- ❏ All tasks for which the Don't shut down system while this task is still running property is TRUE have exited. (By default, this includes the TSK_idle task, which manages communication between the target and DSP/BIOS plug-ins.)

Constraints and Calling Context

- ❏ handler cannot be NULL.
- ❏ SYS_atexit cannot be called from ISRs.

SYS_error

Flag error condition

C Interface

Syntax

```
SYS_error(s, errno, [arg], ...);
```

Parameters

```
String s;       /* error string */
Int    errno;   /* error code */
Arg    arg;     /* optional argument */
```

Return Value

Void

Assembly Interface

none

Description

SYS_error is used to flag DSP/BIOS error conditions. Application programs as well as internal functions use SYS_error to handle program errors.

SYS_error calls the function bound to Error function to handle errors.

The default Error function for the SYS manager is _UTL_doError, which logs an error message, disables interrupts, and then runs in an infinite loop.

Constraints and Calling Context

- The only valid error numbers are the error constants defined in sys.h (SYS_E*) or numbers greater than or equal to SYS_EUSER. Passing any other error values to SYS_error may cause DSP/BIOS to crash.
- The string passed to SYS_error must be non-NULL.

SYS_exit

Terminate program execution

C Interface

Syntax SYS_exit(status);

Parameters Int status; /* termination status code */

Return Value Void

Assembly Interface

none

Description

SYS_exit first pops a stack of handlers registered through the function SYS_atexit, and then terminates program execution by calling the function bound to the configuration parameter Exit function, passing on its original status parameter.

```
(*handlerN)(status)
    ...
(*handler2)(status)
(*handler1)(status)

(*(Exit_function))(status)
```

The default Exit function for the SYS manager is UTL_halt, which performs an infinite loop with all processor interrupts disabled.

Constraints and Calling Context

- If the function bound to Exit function or any of the handler functions is not reentrant, SYS_exit must be called atomically.

See Also

SYS_abort
SYS_atexit

SYS_printf *Output formatted data*

C Interface

Syntax SYS_printf(format, [arg,] ...);

Parameters

```
String  format;    /* format specification string */
String  buffer;    /* output buffer */
Arg     arg;       /* optional argument */
va_list  vargs;    /* variable argument list reference */
```

Return Value Void

Assembly Interface none

Description

SYS_printf provides a subset of the capabilities found in the standard C library function printf.

> **Note:**
>
> SYS_printf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_printf are limited to the characters in the following table.

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- ❏ The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_vprintf
SYS_sprintf
SYS_vsprintf

SYS_sprintf *Output formatted data*

C Interface

Syntax SYS_sprintf (buffer, format, [arg,] ...);

Parameters

```
String  format;    /* format specification string */
String  buffer;    /* output buffer */
Arg     arg;       /* optional argument */
va_list  vargs;    /* variable argument list reference */
```

Return Value Void

Assembly Interface none

Description

SYS_sprintf provides a subset of the capabilities found in the standard C library function printf.

> **Note:**
>
> SYS_sprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_printf are limited to the characters in the following table.

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- ❏ The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_printf
SYS_vprintf
SYS_vsprintf

SYS_vprintf *Output formatted data*

C Interface

Syntax SYS_vprintf(format, vargs);

Parameters

```
String  format;    /* format specification string */
String  buffer;    /* output buffer */
Arg     arg;       /* optional argument */
va_list  vargs;    /* variable argument list reference */
```

Return Value Void

Assembly Interface none

Description

SYS_printf provides a subset of the capabilities found in the standard C library function printf.

> **Note:**
>
> SYS_vprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_printf are limited to the characters in the following table.

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- ❏ The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_printf
SYS_sprintf
SYS_vsprintf

SYS_vsprintf *Output formatted data*

C Interface

Syntax SYS_vsprintf(buffer, format, vargs);

Parameters

```
String  format;    /* format specification string */
String  buffer;    /* output buffer */
Arg     arg;       /* optional argument */
va_list  vargs;    /* variable argument list reference */
```

Return Value Void

Assembly Interface none

Description

SYS_printf provides a subset of the capabilities found in the standard C library function printf.

> **Note:**
>
> SYS_vsprintf and the related functions are code-intensive. If possible, applications should use LOG module functions to reduce code size and execution time.

Conversion specifications begin with a % and end with a conversion character. The conversion characters recognized by SYS_printf are limited to the characters in the following table.

Character	Corresponding Output Format
d	signed decimal integer
u	unsigned decimal integer
f	decimal floating point
o	octal integer
x	hexadecimal integer
c	single character
s	NULL-terminated string

Between the % and the conversion character, the following symbols or specifiers contained within square brackets may appear, in the order shown.

```
%[-][0][width]type
```

A dash (-) symbol causes the converted argument to be left-justified within a field of width characters with blanks following. A 0 (zero) causes the converted argument to be right-justified within a field of size width with leading 0s. If neither a dash nor 0 are given, the converted argument is right-justified in a field of size width, with leading blanks. The width is a decimal integer. The converted argument is not modified if it has more than width characters, or if width is not given.

The length modifier l can precede %d, %u, %o, and %x if the corresponding argument is a long integer.

SYS_vprintf is equivalent to SYS_printf, except that the optional set of arguments is replaced by a va_list on which the standard C macro va_start has already been applied. SYS_sprintf and SYS_vsprintf are counterparts of SYS_printf and SYS_vprintf, respectively, in which output is placed in a specified buffer.

Both SYS_printf and SYS_vprintf internally call the function SYS_putchar to output individual characters in a system-dependent fashion via the configuration parameter Putc function. This parameter is bound to a function that displays output on a debugger if one is running, or places output in an output buffer between PUTCEND and PUTCBEG.

Constraints and Calling Context

- ❏ The function bound to Exit function or any of the handler functions are not reentrant; SYS_exit must be called atomically.

See Also

SYS_printf
SYS_sprintf
SYS_vprintf

SYS_putchar

Output a single character

C Interface

Syntax SYS_putchar(c);

Parameters Char c; /* next output character */

Return Value Void

Assembly Interface none

Description

SYS_putchar outputs the character c by calling the system-dependent function bound to the configuration parameter Putc function.

```
((Putc function))(c)
```

For systems with limited I/O capabilities, the function bound to Putc function might simply place c into a global buffer that can be examined after program termination.

The default Putc function for the SYS manager is _UTL_doPutc, which writes a character to the trace buffer.

SYS_putchar is also used internally by SYS_printf and SYS_vprintf when generating their output.

Constraints and Calling Context

- If the function bound to Putc function is not reentrant, SYS_putchar must be called atomically.

See Also

SYS_printf

TRC Module

Trace manager

Functions

- ❏ TRC_disable. Disable trace class(es)
- ❏ TRC_enable. Enable trace type(s)
- ❏ TRC_query. Query trace class(es)

Description

The TRC module manages a set of trace control bits which control the real-time capture of program information through event logs and statistics accumulators. For greater efficiency, the target does not store log or statistics information unless tracing is enabled.

The following events and statistics can be traced. The constants defined in trc.h and trc.h54 are shown in the left column:

Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Log timer interrupts	off
TRC_LOGPRD	Log periodic ticks and start of periodic functions	off
TRC_LOGSWI	Log events when a software interrupt is posted and completes	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes execution, and terminates	off
TRC_STSHWI	Gather statistics on monitored values within HWIs	off
TRC_STSPIP	Count number of frames read from or written to data pipe	off
TRC_STSPRD	Gather statistics on number of ticks elapsed during execution	off
TRC_STSSWI	Gather statistics on length of SWI execution	off
TRC_STSTSK	Gather statistics on length of TSK execution	off
TRC_USER0 and TRC_USER1	Your program can use these bits to enable or disable sets of explicit instrumentation actions. You can use TRC_query to check the settings of these bits and either perform or omit instrumentation calls based on the result. DSP/BIOS does not use or set these bits.	off
TRC_GBLHOST	This bit must be set in order for any implicit instrumentation to be performed. Simultaneously starts or stops gathering of all enabled types of tracing. This can be important if you are trying to correlate events of different types. This bit is usually set at run time on the host in the RTA Control Panel.	off

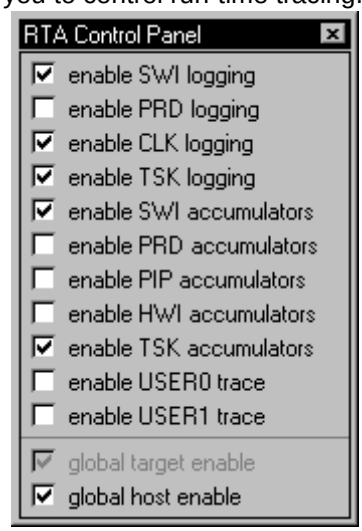
Constant	Tracing Enabled/Disabled	Default
TRC_GBLTARG	This bit must also be set in order for any implicit instrumentation to be performed. This bit can only be set by the target program and is enabled by default.	on
TRC_STSSWI	Gather statistics on length of SWI execution	off

All trace constants except TRC_GBLTARG are switched off initially. To enable tracing you can use calls to TRC_enable or the Tools→DSP/BIOS→RTA Control Panel, which uses the TRC module internally. You do not need to enable tracing for messages written with LOG_printf or LOG_event and statistics added with STS_add or STS_delta.

Your program can call the TRC_enable and TRC_disable operations to explicitly start and stop event logging or statistics accumulation in response to conditions encountered during real-time execution. This enables you to preserve the specific log or statistics information you need to see.

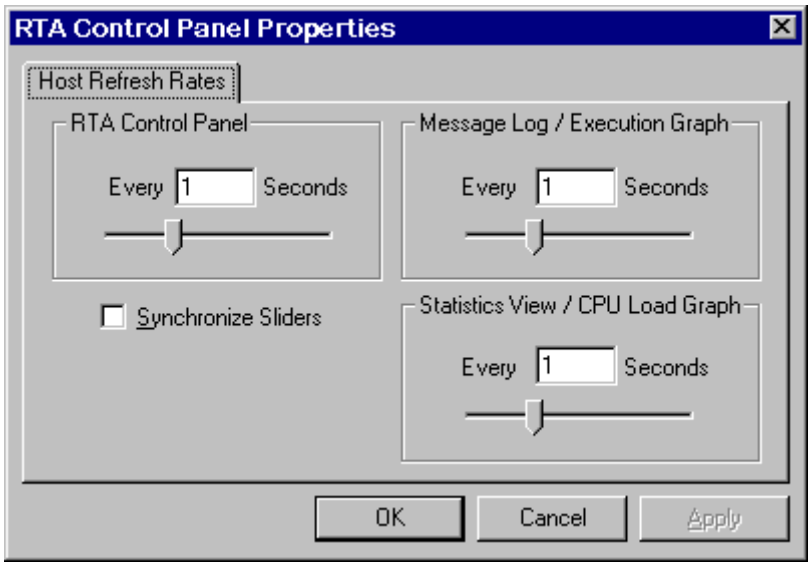
TRC - Code Composer Studio Interface

You can choose Tools→DSP/BIOS→RTA Control Panel to open a window that allows you to control run-time tracing.



Once you have enabled tracing, you can use Tools→DSP/BIOS→Execution Graphand Tools→DSP/BIOS→Event Log to see log information, and Tools→DSP/BIOS→Statistics View to see statistical information.

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



See the *TMS320C54x Code Composer Studio Tutorial* for more information on how to enable tracing in the RTA Control Panel.

TRC_disable

Disable trace class(es)

C Interface

Syntax TRC_disable(mask);

Parameters Uns mask; /* trace type constant mask */

Return Value Void

Assembly Interface

Syntax TRC_disable mask

Inputs mask (see the TRC Module for a list of constants to use in the mask)

Preconditions constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)

Postconditions none

Modifies c

Reentrant no

Description

TRC_disable disables tracing of one or more trace types. Trace types are specified with a 32-bit mask. The following C code would disable tracing of statistics for software interrupts and periodic functions:

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

Internally, DSP/BIOS uses a bitwise AND NOT operation to disable multiple trace types.

The full list of constants you can use to disable tracing is included in the description of the TRC module.

For example, you might want to use TRC_disable with a circular log and disable tracing when an unwanted condition occurs. This allows test equipment to retrieve the log events that happened just before this condition started.

See Also

TRC_enable
TRC_query
LOG_printf
LOG_event
STS_add
STS_delta

TRC_enable

Enable trace type(s)

C Interface

Syntax TRC_enable(mask);

Parameters Uns mask; /* trace type constant mask */

Return Value Void

Assembly Interface

Syntax TRC_enable mask

Inputs mask (see the TRC Module for a list of constants to use in the mask)

Preconditions constant - mask for trace types (TRC_LOGSWI, TRC_LOGPRD, ...)

Postconditions none

Modifies c

Reentrant no

Description

TRC_enable enables tracing of one or more trace types. Trace types are specified with a 32-bit mask. The following C code would enable tracing of statistics for software interrupts and periodic functions:

```
TRC_enable(TRC_STSSWI | TRC_STSPRD);
```

Internally, DSP/BIOS uses a bitwise OR operation to enable multiple trace types.

The full list of constants you can use to enable tracing is included in the description of the TRC module.

For example, you might want to use TRC_enable with a fixed log to enable tracing when a specific condition occurs. This allows test equipment to retrieve the log events that happened just after this condition occurred.

See Also

TRC_disable
TRC_query
LOG_printf
LOG_event
STS_add
STS_delta

TRC_query

Query trace class(es)

C Interface

Syntax

```
result = TRC_query(mask);
```

Parameters

```
Uns mask;     /* trace type constant mask */
```

Return Value

```
Int result    /* indicates whether all trace types enabled */
```

Assembly Interface

Syntax

```
TRC_query mask
```

Inputs

mask (see the TRC Module for a list of constants to use in the mask)

Preconditions

constant - mask for trace types

Postconditions

a == 0 if all trace types in the mask are enabled
a != 0 if any trace type in the mask is disabled

Modifies

ag, ah, al, c

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.

Trace types are specified with a 16-bit mask. The full list of constants you can use is included in the description of the TRC module.

For example, the following C code returns 0 if statistics tracing for the PRD class is enabled:

```
result = TRC_query(TRC_STSPRD);
```

The following C code returns 0 if both logging and statistics tracing for the SWI class are enabled:

```
result = TRC_query(TRC_LOGSWI | TRC_STSSWI);
```

Note that TRC_query does not return 0 unless the bits you are querying and the TRC_GBLHOST and TRC_GBLTARG bits are set. TRC_query returns non-zero if either TRC_GBLHOST or TRC_GBLTARG are disabled. This is because no tracing is done unless these bits are set.

For example, if the TRC_GBLHOST, TRC_GBLTARG, and TRC_LOGSWI bits are set, the following C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)    /* returns 0 */
result = TRC_query(TRC_LOGPRD)    /* returns non-zero */
```

However, if only the TRC_GBLHOST and TRC_LOGSWI bits are set, the same C code returns the results shown:

```
result = TRC_query(TRC_LOGSWI)    /* returns non-zero */
result = TRC_query(TRC_LOGPRD)    /* returns non-zero */
```

See Also

TRC_enable
TRC_disable

TSK Module

Task manager

Functions

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

Constants, Types, and Structures

```
typedef struct TSK_OBJ *TSK_Handle;
                       /* handle for task object */

struct TSK_Attrs {    /* task attributes */
    Int    priority;  /* execution priority */
    Ptr    stack;     /* pre-allocated stack */
    Uns    stacksize; /* stack size in bytes */
    Int    stackseg;  /* memory seg for stack allocation */
```

```
    Ptr    environ;   /* global environment data structure */
    String name;      /* printable name */
    Bool   exitflag;  /* program termination requires this */
                      /* task to terminate */
    TSK_DBG_Mode debug /* indicates enum type TSK_DBG_YES, */
                       /* TSK_DBG_NO or TSK_DBG_MAYBE */
};

Int TSK_pid;          /* MP processor ID */

Int TSK_MAXARGS = 8;  /* maximum number of task arguments */
Int TSK_IDLEPRI = 0;  /* used for idle task */
Int TSK_MINPRI = 1;   /* minimum execution priority */
Int TSK_MAXPRI = 15;  /* maximum execution priority */
Int TRG_STACKSTAMP =
TSK_Attrs TSK_ATTRS = { /* default attribute values */
    TSK->PRIORITY,      /* priority */
    NULL,               /* stack */
    TSK->STACKSIZE,     /* stacksize */
    TSK->STACKSEG,      /* stackseg */
    NULL,               /* environ */
    "",                 /* name */
     TRUE,              /* exitflag */
};

enum TSK_Mode {       /* task execution modes */
    TSK_RUNNING,      /* task is currently executing */
    TSK_READY,        /* task is scheduled for execution */
    TSK_BLOCKED,      /* task is suspended from execution */
    TSK_TERMINATED,   /* task is terminated from execution */
};

struct TSK_Stat {     /* task status structure */
    TSK_Attrs  attrs; /* task attributes */
    TSK_Mode   mode;  /* task execution mode */
    Ptr        sp;    /* task stack pointer */
    Uns        used;  /* task stack used */
};
```

Description

The TSK module makes available a set of functions that manipulate task objects accessed through handles of type TSK_Handle. Tasks represent independent threads of control that conceptually execute functions in parallel within a single C program; in reality, concurrency is achieved by switching the processor from one task to the next.

When you create each task, it is provided with its own run-time stack, used for storing local variables as well as for further nesting of function calls. The TRG_STACKSTAMP value is used to initialize the run-time stack. Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher-priority task. All tasks executing within a single program

share a common set of global variables, accessed according to the standard rules of scope defined for C functions.

Each task is in one of four modes of execution at any point in time: running, ready, blocked, or terminated. By design, there is always one (and only one) task currently running, even if it is a dummy idle task managed internally by TSK. The current task can be suspended from execution by calling certain TSK functions, as well as functions provided by other modules like SEM and SIO; the current task can also terminate its execution. In either case, the processor is switched to the next task that is ready to run.

You can assign numeric priorities to tasks through TSK. Tasks are readied for execution in strict priority order; tasks of the same priority are scheduled on a first-come, first-served basis. As a rule, the priority of the currently running task is never lower than the priority of any ready task. Conversely, the running task is preempted and re-scheduled for execution whenever there exists some ready task of higher priority.

The user-definable function pointer configuration parameters Create function, Delete function, and Exit function are described on the TSK_create, TSK_delete, and TSK_exit manual pages, respectively.

Switch function, if not NULL, is invoked during a task switch giving the application access to both the current and next task handles at task switch time:

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

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

The functions attached to the Switch function and Ready function are called from within the kernel and may make only those function calls allowed from within a software interrupt handler. See Appendix A, Function Callability and Error Tables, for a list of functions that can be called within the Kernel. There are no real constraints on what functions are called via Create function, Delete function, or Exit function since these are invoked outside the kernel.

TSK Manager Properties

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

- **Enable TSK Manager**. If no tasks are used by the program other than TSK_idle, you can optimize the program by disabling the task manager. The program must then not use TSK objects created with either the Configuration Tool or the TSK_create function. If the task manager is disabled, the idle loop still runs and uses the application stack instead of a task stack.
- **Object Memory**. The memory section that contains the TSK objects created with the Configuration Tool.
- **Default stack size**. The default size of the stack (in MAUs) used by tasks. You can override this value for an individual task you create with the Configuration Tool or TSK_create. The estimated minimum task size is shown in the status bar of the Configuration Tool.
  This property applies to TSK objects created both with the Configuration Tool and with TSK_create.
- **Default stack segment for dynamic tasks**. The memory section where the task's stack is placed. You can override this value for an individual task.
  This property applies to TSK objects created both with the Configuration Tool and with TSK_create.
- **Default task priority**. The default priority level for tasks that are created dynamically with TSK_create.
  This property applies to TSK objects created both with the Configuration Tool and with TSK_create.
- **Create function**. The name of a function to call when any task is created at run-time with TSK_create. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- **Delete function**. The name of a function to call when any task is deleted at run-time with TSK_delete. If this function is written in C, use a leading underscore before the C function name.
- **Exit function**. The name of a function to call when any task exits. If this function is written in C, use a leading underscore before the C function name.
- **Call switch function**. Check this box if you want a function to be called when any task switch occurs.
- **Switch function**. The name of a function to call when any task switch occurs. This function can give the application access to both the current

and next task handles. If this function is written in C, use a leading underscore before the C function name.

- **Call ready function**. Check this box if you want a function to be called when any task becomes ready to run.
- **Ready function**. The name of a function to call when any task becomes ready to run. If this function is written in C, use a leading underscore before the C function name.
- **Statistics Units**. The units used within DSP/BIOS plug-ins to display the elapsed instruction cycles or time from when a task is made ready to run until a call like the following is made:

```
TSK_deltatime(TSK_self())
```

Choose Raw causes the STS Data window to display the number of instruction cycles if the CLK module's Use high resolution time for internal timings parameter is set to True (the default). If this CLK parameter is set to False and the Statistics Units is set to Raw, TSK statistics are displayed in units of timer interrupt periods. You can also choose milliseconds or microseconds.

TSK Object Properties

The following properties can be set for a TSK object:

- **comment**. A comment to identify this TSK object.
- **Task function**. The function to be executed when the task runs. If this function is written in C, use a leading underscore before the C function name. (The Configuration Tool generates assembly code which must use the leading underscore when referencing C functions or labels.)
- **Task function argument 0-7**. The arguments to pass to the task function. Arguments may be integers or labels. For labels defined in a C program, add a leading underscore before the label name.
- **Automatically allocate stack**. Check this box if you want the task's private stack space to be allocated automatically when this task is created. The task's context is saved in this stack before any higher-priority task is allowed to block this task and run.
- **Manually allocated stack**. If you did not check the box to Automatically allocate stack, type the name of the manually allocated stack to use for this task. If the stack is defined in a C program, add a leading underscore before the stack name.
- **Stack size**. If you checked the box to Automatically allocate stack, type the size (in MAUs) of the stack space to allocate for this task. Each stack must be large enough to handle normal subroutine calls as well as a

single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher priority task.

- **Stack Memory Segment**. If you checked the box to Automatically allocate stack, select the memory section to contain the stack space for this task.
- **Priority**. The priority level for this task.
- **Environment pointer**. A pointer to a globally defined data structure that this task may access. The task can get and set the task environment pointer with the TSK_getenv and TSK_setenv functions. If this structure is defined in C, use a leading underscore before the structure name.
- **Don't shut down system while this task is still running**. Check this box if you do not want the application to be able to end if this task is still running. The application can still abort. For example, you might clear this box for a monitor task that collects data whenever all other tasks are blocked. The application does not need to explicitly shut down this task.
- **Allocate Task Name on Target**. Check this box if you want the name of this TSK object to be retrievable by the TSK_getname function. Clearing this box saves a small amount of memory. The task name is available in DSP/BIOS plug-ins in either case.

TSK - DSP/BIOS Plug-ins Interface

The TSK tab of the Kernel/Object View shows information about task objects.

To enable TSK logging, choosing Tools→DSP/BIOS→RTA Control Panel and check the appropriate box. Then you can open the system log by choosing View–>System Log. You see a graph of activity that includes TSK function execution states.

Only TSK objects created with the Configuration Tool are traced. The System Log graph includes time spent performing dynamically created TSK functions in the Other Threads row.

You can also enable TSK accumulators in the RTA Control Panel. Then you can choose Tools→DSP/BIOS→Statistics View, which lets you select objects for which you want to see statistics. If you choose a TSK object, you see statistics about the time elapsed from the time the TSK was posted until TSK_deltatime(TSK_self) is called.

TSK_checkstacks

Check for stack overflow

C Interface

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

Assembly Interface

none

Description

TSK_checkstacks calls SYS_abort with an error message if either oldtask or newtask has a stack in which the last location no longer contains the initial value TRG_STACKSTAMP. The presumption in one case is that oldtask's stack overflowed, and in the other that an invalid store has corrupted newtask's stack.

You may call TSK_checkstacks directly from your application. For example, you can check the current task's stack integrity at any time with a call like the following:

```
TSK_checkstacks(TSK_self, TSK_self);
```

However, it is more typical to call TSK_checkstacks in the task Switch function specified for the TSK manager in your configuration file. This provides stack checking at every context switch, with no alterations to your source code.

If you want to perform other operations in Switch function, you may do so by writing your own function (myswitchfxn) and then calling TSK_checkstacks from it.

```
Void myswitchfxn(TSK_Handle oldtask, TSK_Handle newtask)
{
    'your additional context switch operations'
    TSK_checkstacks(oldtask, newtask);
    ...
}
```

TSK_create

Create a task ready for execution

C Interface

Syntax

```
task = TSK_create(fxn, attrs, [arg,] ...);
```

Parameters

```
Fxn          fxn;     /* entry point of the task */
TSK_Attrs   *attrs;   /* pointer to task attributes */
Arg          arg;     /* task arguments */
```

Return Value

```
TSK_Handle  task;     /* task object handle */
```

Assembly Interface

none

Description

TSK_create creates a new task object. If successful, TSK_create returns the handle of the new task object. If unsuccessful, TSK_create returns NULL unless it aborts (e.g., because it directly or indirectly calls SYS_error, and SYS_error is configured to abort).

Create_ function, the system specific task create function, has a default value of SYS_nop. This function is called after the task handle has been initialized but before the task has been placed on its ready queue. It is called with the task handle as its only parameter. Any DSP/BIOS function may be called from Create_function.

```
(*(Create_function))(task)
```

The new task is placed in TSK_READY mode, and is scheduled to begin concurrent execution of the following function call:

```
(*fxn)(arg1, arg2, ... argN)    /* N == TSK_MAXARGS == 8 */
```

TSK_exit is automatically called if and when the task returns from fxn.

If attrs is NULL, the new task is assigned a default set of attributes. Otherwise, the task's attributes are specified through a structure of type TSK_Attrs defined as follows:

```
struct TSK_Attrs {
    Int      priority;
    Ptr      stack;
    Uns      stacksize;
    Uns      stackseg;
    Ptr      environ;
    String   name;
    Bool     exitflag;
};
```

The priority attribute specifies the task's execution priority and must be less than or equal to TSK_MAXPRI (15); this attribute defaults to the value of the

configuration parameter Default task priority (preset to TSK_MINPRI). If priority is less than 0, task is barred from execution until its priority is raised at a later time by another task. A priority value of 0 is reserved for the TSK_idle task defined in the default configuration. You should not use a priority of 0 for any other tasks.

The stack attribute specifies a pre-allocated block of stacksize bytes to be used for the task's private stack; this attribute defaults to NULL, in which case the task's stack is automatically allocated using MEM_alloc from the memory section given by the stackseg attribute.

The stacksize attribute specifies the number of bytes to be allocated for the task's private stack; this attribute defaults to the value of the configuration parameter Default stack size (preset to 1024). Each stack must be large enough to handle normal subroutine calls as well as a single task preemption context. A task preemption context is the context that gets saved when one task preempts another as a result of an interrupt thread readying a higher priority task.

The stackseg attribute specifies the memory section to use when allocating the task stack with MEM_alloc; this attribute defaults to the value of the configuration parameter Default stack segment.

The environ attribute specifies the task's global environment through a generic pointer that references an arbitrary application-defined data structure; this attribute defaults to NULL.

The name attribute specifies the task's printable name, which is a NULL-terminated character string; this attribute defaults to the empty string "". This name can be returned by TSK_getname.

The exitflag attribute specifies whether or not the task must terminate before the program as a whole can terminate; this attribute defaults to TRUE.

All default attribute values are contained in the constant TSK_ATTRS, which may be assigned to a variable of type TSK_Attrs prior to calling TSK_create.

A task switch occurs when calling TSK_create if the priority of the new task is greater than the priority of the current task.

Constraints and Calling Context

- ❏ TSK_create cannot be called by ISRs.
- ❏ The fxn parameter and the name attribute cannot be NULL.
- ❏ The priority attribute must be less than or equal to TSK_MAXPRI and greater than or equal to TSK_MINPRI. The priority may be less than zero (0) for tasks that should not execute.

- ❏ The string referenced through the name attribute cannot be allocated locally.
- ❏ The stackseg attribute must identify a valid memory section.
- ❏ You can reduce the size of your application program by creating objects with the Configuration Tool rather than using the XXX_create functions.

See Also

MEM_alloc
SYS_error
TSK_delete
TSK_exit

TSK_delete

Delete a task

C Interface

Syntax

```
TSK_delete(task);
```

Parameters

```
TSK_Handle    task;    /* task object handle */
```

Return Value

Void

Assembly Interface

none

Description

TSK_delete removes the task from all internal queues and calls MEM_free to free the task object and stack. task should be in a state that does not violate any of the listed constraints.

If all remaining tasks have their exitflag attribute set to FALSE, DSP/BIOS terminates the program as a whole by calling SYS_exit with a status code of 0.

Delete_function, the system specific task delete function, has a default value of SYS_nop. This function is called before the task object has been removed from any internal queues and its object and stack are freed. Any DSP/BIOS function may be called from *(Delete_function).

```
(*(Delete_function))(task)
```

No task switch occurs when calling TSK_delete.

Note:

Unless the mode of the deleted task is TSK_TERMINATED, TSK_delete should be called with care. For example, if the task has obtained exclusive access to a resource, deleting the task makes the resource unavailable.

Constraints and Calling Context

- ❏ The task cannot be the currently executing task (TSK_self).
- ❏ TSK_delete cannot be called by ISRs.
- ❏ No check is performed to prevent TSK_delete from being used on a statically-created object. If a program attempts to delete a task object that was created using the Configuration Tool, SYS_error is called.

See Also

MEM_free
TSK_create

TSK_deltatime

Update task statistics with difference between current time and time task was made ready

C Interface

Syntax

```
TSK_deltatime(task);
```

Parameters

```
TSK_Handle  task;    /* task object handle */
```

Return Value

Void

Assembly Interface

none

Description

This function accumulates the time difference from when a task is made ready to the time TSK_deltatime is called. These time differences are accumulated in the task's internal STS object and can be used to determine whether or not a task misses real-time deadlines.

For example, if a task waits for data and then processes the data, you want to ensure that the time from when the data is made available until the processing is complete is always less than a certain value. A loop within the task may look something like the following:

```
/* Initialize time in task's STS object to current time */
TSK_settime(TSK_self());

for (;;) {
   /* Get data */
   SIO_get(...);
   'process data'
   /* Get time difference and add it to the task's STS object */
   TSK_deltatime(TSK_self());
}
```

Assuming this task blocks on SIO_get, the device driver posts a semaphore that readies the task. DSP/BIOS sets the time value of the STS object every time the task is made ready to run, so TSK_settime is only called once when the task is starting up.

See Also

TSK_getsts
TSK_settime

TSK_disable

Disable DSP/BIOS task scheduler

C Interface

Syntax

```
TSK_disable();
```

Parameters

Void

Return Value

Void

Assembly Interface

none

Description

TSK_disable disables the DSP/BIOS task scheduler. The current task continues to execute (even if a higher priority task may become ready to run) until TSK_enable is called.

TSK_disable does not disable interrupts, but is instead used before disabling interrupts to make sure a context switch to another task does not occur when interrupts are disabled.

TSK_disable handlers are disabled. TSK_disable maintains a count which allows nested calls to TSK_disable. Task switching is not reenabled until TSK_enable has been called as many times as TSK_disable. Calls to TSK_disable may be nested.

Since TSK_disable may prohibit ready tasks of higher priority from running it should not be used as a general means of mutual exclusion—SEM semaphores should be used for mutual exclusion when possible.

Constraints and Calling Context

- ❏ No kernel operations that may cause the current task to block (e.g., SEM_pend, TSK_sleep, TSK_yield) can be made from within a TSK_disable / TSK_enable block.
- ❏ TSK_yield cannot be called within a TSK_disable / TSK_enable block.

See Also

SEM Module
TSK_enable

TSK_enable

Enable DSP/BIOS scheduler

C Interface

Syntax TSK_enable();

Parameters Void

Return Value Void

Assembly Interface none

Description

TSK_enable is used to reenable the DSP/BIOS task scheduler after TSK_disable has been called. Since TSK_disable calls may be nested, the task scheduler is not enabled until TSK_enable is called the same number of times as TSK_disable.

A task switch occurs when calling TSK_enable only if there exists a TSK_READY task whose priority is greater than the currently executing task.

Constraints and Calling Context

- ❑ No kernel operations that may cause the current task to block (e.g., SEM_pend, TSK_sleep, TSK_yield) can be made from within a TSK_disable / TSK_enable block.

See Also

SEM Module
TSK_disable
C54_enableIMR

TSK_exit

Terminate execution of the current task

C Interface

Syntax TSK_exi();

Parameters Void

Return Value Void

Assembly Interface

none

Description

TSK_exit terminates execution of the current task, changing its mode from TSK_RUNNING to TSK_TERMINATED. If all tasks have been terminated, or if all remaining tasks have their exitflag attribute set to FALSE, then DSP/BIOS terminates the program as a whole by calling the function SYS_exit with a status code of 0.

TSK_exit is automatically called whenever a task returns from its top-level function.

Exit_function, the system specific task exit function, has a default value of SYS_nop. This function is called before the task has been blocked and marked TSK_TERMINATED. Any DSP/BIOS function may be called from *(Exit_function).

```
(*(Exit_function))()
```

A task switch occurs when calling TSK_exit unless the program as a whole is terminated.

Constraints and Calling Context

- ❏ TSK_exit cannot be called by ISRs.

See Also

MEM_free
TSK_create
TSK_delete

TSK_getenv

Get task environment pointer

C Interface

Syntax

```
environ = TSK_getenv(task);
```

Parameters

```
TSK_Handle  task;      /* task object handle */
```

Return Value

```
Ptr         environ;   /* task environment pointer */
```

Assembly Interface

none

Description

TSK_getenv returns the environment pointer of task. The environment pointer, environ, references an arbitrary application-defined data structure.

See Also

TSK_setenv
TSK_seterr
TSK_setpri

TSK_geterr

Get task error number

C Interface

Syntax

```
errno = TSK_geterr(task);
```

Parameters

```
TSK_Handle  task;    /* task object handle */
```

Return Value

```
Int         errno;   /* error number */
```

Assembly Interface

none

Description

Each task carries a task-specific error number. This number is initially SYS_OK, but it can be changed by TSK_seterr. TSK_geterr returns the current value of this number.

See Also

SYS_error
TSK_setenv
TSK_seterr
TSK_setpri

TSK_getname

Get task name

C Interface

Syntax

```
name = TSK_getname(task);
```

Parameters

```
TSK_Handle  task;    /* task object handle */
```

Return Value

```
String      name;    /* task name */
```

Assembly Interface

none

Description

TSK_getname returns the task's name.

For tasks created with the Configuration Tool, the name is available to this function only if the Allocate Task Name on Target box is checked in the properties for this task. For tasks created with TSK_create, TSK_getname returns the attrs.name field value, or an empty string if this attribute was not specified.

See Also

TSK_setenv
TSK_seterr
TSK_setpri

TSK_getpri

Get task priority

C Interface

Syntax

```
priority = TSK_getpri(task);
```

Parameters

```
TSK_Handle  task;      /* task object handle */
```

Return Value

```
Int         priority;  /* task priority */
```

Assembly Interface

none

Description

TSK_getpri returns the priority of task.

See Also

TSK_setenv
TSK_seterr
TSK_setpri

TSK_getsts

Get the handle of the task's STS object

C Interface

Syntax

```
sts = TSK_getsts(task);
```

Parameters

```
TSK_Handle  task;    /* task object handle */
```

Return Value

```
STS_Handle     sts;    /* statistics object handle */
```

Assembly Interface

none

Description

This function provides access to the task's internal STS object. For example, you may want the program to check the maximum value to see if it has exceeded some value.

See Also

TSK_deltatime
TSK_settime

TSK_itick

Advance the system alarm clock (interrupt use only)

C Interface

Syntax TSK_itick();

Parameters Void

Return Value Void

Assembly Interface none

Description

TSK_itick increments the system alarm clock, and readies any tasks blocked on TSK_sleep or SEM_pend whose timeout intervals have expired.

Constraints and Calling Context

- TSK_itick may be invoked only inside a hardware ISR.

See Also

SEM_pend
TSK_sleep
TSK_tick

TSK_self

Returns handle to the currently executing task

C Interface

Syntax

```
curtask = TSK_self();
```

Parameters

Void

Return Value

```
TSK_Handle    curtask;    /* handle for current task object */
```

Assembly Interface

none

Description

TSK_self returns the object handle for the currently executing task. This function is useful when inspecting the object or when the current task changes its own priority through TSK_setpri.

No task switch occurs when calling TSK_self.

See Also

TSK_setpri

TSK_setenv

Set task environment

C Interface

Syntax

```
TSK_setenv(task, environ);
```

Parameters

```
TSK_Handle task;        /* task object handle */
Ptr        environ;     /* task environment pointer */
```

Return Value

Void

Assembly Interface

none

Description

TSK_setenv sets the task environment pointer to environ. The environment pointer, environ, references an arbitrary application-defined data structure.

See Also

TSK_getenv
TSK_geterr

TSK_seterr

Set task error number

C Interface

Syntax

```
TSK_seterr(task, errno);
```

Parameters

```
TSK_Handle task;    /* task object handle */
Int        errno;   /* error number */
```

Return Value

Void

Assembly Interface

none

Description

Each task carries a task-specific error number. This number is initially SYS_OK, but can be changed to errno by calling TSK_seterr. TSK_geterr returns the current value of this number.

See Also

TSK_getenv
TSK_geterr

TSK_setpri

Set a task's execution priority

C Interface

Syntax

```
oldpri = TSK_setpri(task, newpri);
```

Parameters

```
TSK_Handle  task;       /* task object handle */
Int         newpri;     /* task's new priority */
```

Return Value

```
Int         oldpri;     /* task's old priority */
```

Assembly Interface

none

Description

TSK_setpri sets the execution priority of task to newpri, and returns that task's old priority value. Raising or lowering a task's priority does not necessarily force preemption and re-scheduling of the caller: tasks in the TSK_BLOCKED mode remain suspended despite a change in priority; and tasks in the TSK_READY mode gain control only if their (new) priority is greater than that of the currently executing task.

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

The current task can change its own priority (and possibly preempt its execution) by passing the output of TSK_self as the value of the task parameter.

A task switch occurs when calling TSK_setpri only if there exists some TSK_READY task whose priority is greater than the currently executing task. This is important and allows TSK_setpri to be used for mutual exclusion.

Constraints and Calling Context

- ❏ newpri must be less than or equal to TSK_MAXPRI.
- ❏ The task cannot be TSK_TERMINATED.
- ❏ The new priority should not be zero (0). This priority level is reserved for the TSK_idle task.
- ❏ TSK_setpri cannot be called from a DSP/BIOS SWI.

See Also

TSK_self
TSK_sleep

TSK_settime

Reset task statistics previous value to current time

C Interface

Syntax

```
TSK_settime(task);
```

Parameters

```
TSK_Handle  task;    /* task object handle */
```

Return Value

Void

Assembly Interface

none

Description

This function initializes the previous time in the task's STS object. It can be used after a task starts running, but before it enters a processing loop. TSK_settime effectively does an STS_set operation to the task's internal STS object using the current time.

DSP/BIOS sets the time value of the STS object every time the task is made ready to run, so TSK_settime should only called once when the task is starting up.

For example, a loop within the task may look something like the following:

```
Void task()
{
  'do some startup work'

  /* Initialize time in task's STS object to current time */
  TSK_settime(TSK_self());

  for (;;) {
    /* Get data */
    SIO_get(...);

    'process data'

    /* Get time difference and add it to task's STS object */
    TSK_deltatime(TSK_self());
  }
```

Assuming this task blocks on SIO_get, the device driver posts a semaphore that readies the task.

See Also

TSK_deltatime
TSK_getsts

TSK_sleep

Delay execution of the current task

C Interface

Syntax

```
TSK_sleep(nticks);
```

Parameters

```
Uns  nticks;  /* number of system clock ticks to sleep */
```

Return Value

Void

Assembly Interface

none

Description

TSK_sleep changes the current task's mode from TSK_RUNNING to TSK_BLOCKED, and delays its execution for nticks increments of the system clock. The actual time delayed can be up to 1 system clock tick less than timeout due to granularity in system timekeeping.

After the specified period of time has elapsed, the task reverts to the TSK_READY mode and is scheduled for execution.

A task switch always occurs when calling TSK_sleep if nticks > 0.

Constraints and Calling Context

- TSK_sleep cannot be called by ISRs, or within a TSK_disable / TSK_enable block.
- nticks cannot be SYS_FOREVER.
- TSK_sleep should not be called from within an IDL function. Doing so prevents DSP/BIOS plug-ins from gathering run-time information.

TSK_stat

Retrieve the status of a task

C Interface

Syntax

TSK_stat(task, statbuf);

Parameters

```
TSK_Handle  task;      /* task object handle */
TSK_Stat    *statbuf;  /* pointer to task status structure */
```

Return Value

Void

Assembly Interface

none

Description

TSK_stat retrieves attribute values and status information about task; the current task can inquire about itself by passing the output of TSK_self as the first argument to TSK_stat.

Status information is returned through statbuf, which references a structure of type TSK_Stat defined as follows:

```
struct TSK_Stat {    /* task status structure */
    TSK_Attrs attrs; /* task attributes */
    TSK_Mode  mode;  /* task execution mode */
    Ptr       sp;    /* task's current stack pointer */
    Uns       used;  /* max number of words ever used on the */
                     /* task stack */
};
```

When a task is preempted by a software or hardware interrupt, the task execution mode returned for that task by TSK_stat is still TSK_RUNNING because the task will run when the preemption ends.

Constraints and Calling Context

- ❏ statbuf cannot be NULL.

See Also

TSK_create

TSK_tick

Advance the system alarm clock

C Interface

Syntax TSK_tick();

Parameters Void

Return Value Void

Assembly Interface none

Description

TSK_tick increments the system clock, and readies any tasks blocked on TSK_sleep or SEM_pend whose timeout intervals have expired. TSK_tick may be invoked by an ISR or by the currently executing task. The latter is particularly useful for testing timeouts in a controlled environment.

A task switch occurs when calling TSK_tick if the priority of any of the readied tasks is greater than the priority of the currently executing task.

TSK_tick may also be called from an ISR but TSK_itick is more efficient. TSK_tick behaves differently dependent upon whether it is called from an ISR or from a task: TSK_itick has reduced overhead since it assumes it is called from an ISR and does not run a check.

See Also

CLK Module
SEM_pend
TSK_itick
TSK_sleep

TSK_time

Return current value of system clock

C Interface

Syntax curtime = TSK_time();

Parameters Void

Return Value Uns curtime; /* current time */

Assembly Interface none

Description

TSK_time returns the current value of the system alarm clock.

Note that since the system clock is usually updated asynchronously by an interrupt service routine (via TSK_itick or TSK_tick), curtime may lag behind the actual system time. This lag may be even greater if a higher priority task preempts the current task between the call to TSK_time and when its return value is used. Nevertheless, TSK_time is useful for getting a rough idea of the current system time.

TSK_yield

Yield processor to equal priority task

C Interface

Syntax TSK_yield();

Parameters Void

Return Value Void

Assembly Interface none

Description

TSK_yield yields the processor to another task of equal priority.

A task switch occurs when you call TSK_yield if there is an equal priority task ready to run.

Constraints and Calling Context

- TSK_yield cannot be called by ISRs.

See Also

TSK_sleep

C library stdlib.h

DSP/BIOS standard C library functions

Syntax

```
#include <stdlib.h>       /* supplied with the C compiler */
```

Functions

```
int  atexit(void (*fcn)(void));
void *calloc(size_t nobj, size_t size);
void exit(int status);
void free(void *p);
char *getenv(char *name);
void *malloc(size_t size);
void *realloc(void *p, size_t size);
```

Description

The DSP/BIOS library contains some C standard library functions which supersede the library functions bundled with the C compiler. These functions follow the ANSI C specification for parameters and return values. Consult Kernighan and Ritchie for a complete description of these functions.

The functions calloc, free, malloc, and realloc use MEM_alloc and MEM_free (with segid = Segment for malloc/free) to allocate and free memory.

getenv uses the _environ variable defined and initialized in the boot file to search for a matching environment string.

exit calls the exit functions registered by atexit before calling SYS_exit.

Chapter 2

Utility Programs

This chapter provides documentation for utilities that can be used to examine various files from the MS-DOS command line. These programs are provided with DSP/BIOS in the bin subdirectory.

Topic Page

2.1 cdbprint

Prints a listing of all parameters defined in a configuration file

Syntax

cdbprint [-a] [-l] [-w] cdb-file

Description

This utility reads a .cdb file created with the Configuration Tool and creates a list of all the objects and parameters. This tool can be used to compare two configuration files or to simply review the values of a single configuration file.

The -a flag causes cdbprint to list all objects and fields including those that are normally not visible (i.e., unconfigured objects and hidden fields). Without this flag, cdbprint ignores unconfigured objects or modules as well as any fields that are hidden.

The -l flag causes cdbprint to list the internal parameter names instead of the labels used by the Configuration Tool. Without this flag, cdbprint lists the labels used by the Configuration Tool.

The -w flag causes cdbprint to list only those parameters that can also be modified in the Configuration Tool. Without this flag, cdbprint lists both read-only and read-write parameters.

Example

The following sequence of commands can be used to compare a configuration file called test54.cdb to the default configuration provided with DSP/BIOS:

```
cdbprint ../../include/bios54.cdb > original.txt
cdbprint test54.cdb > test54.txt
diff original.txt test54.txt
```

2.2 gconfgen

Reads a reads a .cdb file created with the Configuration Tool

Syntax

gconfgen cdb-file

Description

This command line utility reads a .cdb file (e.g. program.cdb) created with the Configuration Tool and generates the target configuration files (programcfg.cmd, programcfg.h<54><62>, programcfg.s<54><62>) that are linked with the rest of the application code.

This utility is useful when the build process is controlled by a scripted mechanism, such as a make file, to generate the configuration source files from the configuration database file (.cdb file). Caution should be used, however, following product upgrades, since gconfgen does not detect revision changes. After a product update, use the graphical Configuration Tool to update your .cdb files to the new version. Once updated, gconfgen can be used again to generate the target configuration files.

Example

You can use gconfgen from a make file, as shown in the following example, from the makefiles provided with the DSP/BIOS examples in the product distribution. To use gconfgen from the command line or makefiles, use its full path (TI_DIR\plugins\bios\gconfgen) or add its folder (TI_DIR\plugins\bios) to your PATH environment variable. (Note that TI_DIR is the root directory of the product distribution).

```
*
*  Makefile for creation of program named by the PROG variable
*
*  The following naming conventions are used by this makefile:
*  <prog>.asm    - C54 assembly language source file
*  <prog>.obj    - C54 object file (compiled/assembled source)
*  <prog>.out    - C54 executable (fully linked program)
*  <prog>cfg.s54 - configuration assembly source file generated
*           by Configuration Tool
*  <prog>cfg.h54 - configuration assembly header file generated
*           by Configuration Tool
*  <prog>cfg.cmd - configuration linker command file generated
*           by Configuration Tool
*

TI_DIR  := $(subst \,/,$(TI_DIR))
include $(TI_DIR)/c5400/bios/include/c54rules.mak
*
*  Compiler, assembler, and linker options.
*
* -g enable symbolic debugging
```

```
CC54OPTS = -g
AS54OPTS =
* -q quiet run
LD54OPTS = -q      * -q quiet run
*  Every BIOS program must be linked with:
*  $(PROG)cfg.o54 - result of assembling $(PROG)cfg.s54
*  $(PROG)cfg.cmd - linker command file generated by Config
*           Tool. If additional liner command files exist,
*           $(PROG)cfg.cmd must appear first.
*
PROG   = tsktest
OBJS   = $(PROG)cfg.obj
LIBS   =
CMDS   = $(PROG)cfg.cmd
*
*  Targets:
*
all:: $(PROG).out
$(PROG).out: $(OBJS) $(CMDS)
$(PROG)cfg.obj: $(PROG)cfg.h54
$(PROG).obj:
$(PROG)cfg.s54 $(PROG)cfg.h54  $(PROG)cfg.cmd :: $(PROG).cdb
$(TI_DIR)/plugins/bios/gconfgen $(PROG).cdb
.clean clean::
   @ echo removing generated configuration files ...
   @$(REMOVE) -f $(PROG)cfg.s54 $(PROG)cfg.h54 $(PROG)cfg.cmd
   @ echo removing object files and binaries ...
   @$(REMOVE) -f *.obj *.out *.lst *.map
```

2.3 nmti

Display symbols and values in a TI COFF file

Syntax

nmti [file1 file2 ...]

Description

nmti prints the symbol table (name list) for each TI executable file listed on the command line. Executable files must be stored as COFF (Common Object File Format) files.

If no files are listed, the file a.out is searched. The output is sent to stdout. Note that both linked (executable) and unlinked (object) files can be examined with nmti.

Each symbol name is preceded by its value (blanks if undefined) and one of the following letters:

A	absolute symbol
B	bss segment symbol
D	data segment symbol
E	external symbol
S	section name symbol
T	text segment symbol
U	undefined symbol

The type letter is upper case if the symbol is external, and lower case if it is local.

2.4 sectti

Display information about sections in TI COFF files

Syntax

sectti [-a] [file1 file2 ...]

Description

sectti displays location and size information for all the sections in a TI executable file. Executable files must be stored as COFF (Common Object File Format) files.

All values are in hexadecimal. If no file names are given, a.out is assumed. Note that both linked (executable) and unlinked (object) files can be examined with sectti.

Using the -a flag causes sectti to display all program sections, including sections used only on the target by the DSP/BIOS plug-ins. If you omit the -a flag, sectti displays only the program sections that are loaded on the target.

2.5 size54

Display the section sizes of an object file

Syntax

size54 [file1 file2 ...]

Description

This utility prints the decimal numer of MAUs required by all code sections, all data sections, and the .bss and .stack sections for each COFF file argument. If no file is specified, a.out is used. Note that both linked (executable) and unlinked (object) files can be examined with size54.

2.6 vers

Display version information for a DSP/BIOS source or library file

Syntax

vers [file1 file2 ...]

Description

The vers utility displays the version number of DSP/BIOS files installed in your system. For example, the following command checks the version number of the bios.a54 file in the lib sub-directory.

```
..\bin\vers bios.a54
bios.a54:
    *** library
    *** "date and time"
    *** bios-c05
    *** "version number"
```

The actual output from vers may contain additional lines of information. To identify your software version number to Technical Support, use the version number shown.

Note that both libraries and source files can be examined with vers.

Appendix A

Function Callability and Error Tables

This appendix provides tables describing errors and function callability.

Topic **Page**

A.1 Functions Callable by Tasks, SWI Handlers, or Hardware ISRs

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
ATM_andi()	Yes	Yes	Yes	No
ATM_andu()	Yes	Yes	Yes	No
ATM_cleari()	Yes	Yes	Yes	No
ATM_clearu()	Yes	Yes	Yes	No
ATM_deci()	Yes	Yes	Yes	No
ATM_decu()	Yes	Yes	Yes	No
ATM_inci()	Yes	Yes	Yes	No
ATM_incu()	Yes	Yes	Yes	No
ATM_ori()	Yes	Yes	Yes	No
ATM_oru()	Yes	Yes	Yes	No
ATM_seti()	Yes	Yes	Yes	No
ATM_setu()	Yes	Yes	Yes	No
C54_disableIMR	Yes	Yes	Yes	No
C54_enableIMR	Yes	Yes	Yes	No
C54_plug	Yes	Yes	Yes	No
CLK_countspms()	Yes	Yes	Yes	No
CLK_gethtime()	Yes	Yes	Yes	No
CLK_getltime()	Yes	Yes	Yes	No
CLK_getprd()	Yes	Yes	Yes	No
DEV_match	Yes	Yes	Yes	No
Dxx_close()	Yes	No	No	Yes (see Note 1)
Dxx_ctrl()	Yes	No	No	Yes (see Note 2)
Dxx_idle()	Yes	No	No	Yes (see Note 2)
Dxx_issue	Yes	No	No	No
Dxx_open()	Yes	No	No	Yes (see Note 2)
Dxx_ready()	Yes	No	No	No
Dxx_reclaim	Yes	No	No	No
HST_getpipe	Yes	Yes	Yes	No
HWI_disable()	Yes	Yes	Yes	No
HWI_enable()	Yes	Yes	Yes	No
HWI_enter	No	No	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
HWI_exit	No	No	Yes	Yes
HWI_restore()	Yes	Yes	Yes	No
IDL_run	Yes	No	No	No
LCK_create()	Yes	No	No	Yes (see Note 2)
LCK_delete()	Yes	No	No	Yes (see Note 2)
LCK_pend()	Yes	No	No	Yes (see Note 3)
LCK_post()	Yes	No	No	Yes (see Note 2)
LOG_disable	Yes	Yes	Yes	No
LOG_enable	Yes	Yes	Yes	No
LOG_error()	Yes	Yes	Yes	No
LOG_message	Yes	Yes	Yes	No
LOG_event()	Yes	Yes	Yes	No
LOG_printf()	Yes	Yes	Yes	No
LOG_reset	Yes	Yes	Yes	No
MBX_create()	Yes	No	No	Yes (see Note 2)
MBX_delete()	Yes	No	No	Yes (see Note 2)
MBX_pend()	Yes	Yes (see Note 4)	Yes (see Note 5)	Yes (see Note 5)
MBX_post()	Yes	Yes (see Note 5)	Yes (see Note 5)	Yes (see Note 6)
MEM_alloc()	Yes	No	No	Yes (see Note 2)
MEM_calloc()	Yes	No	No	Yes (see Note 2)
MEM_define	Yes	No	No	No
MEM_free()	Yes	No	No	Yes (see Note 2)
MEM_redefine	Yes	No	No	No
MEM_stat()	Yes	No	No	Yes (see Note 2)
MEM_valloc()	Yes	No	No	Yes (see Note 2)
PIP_alloc	Yes	Yes	Yes	Yes
PIP_free	Yes	Yes	Yes	Yes
PIP_get	Yes	Yes	Yes	Yes
PIP_getReaderAddr	Yes	Yes	Yes	No
PIP_getReaderNumFrames	Yes	Yes	Yes	No
PIP_getReaderSize	Yes	Yes	Yes	No
PIP_getWriterAddr	Yes	Yes	Yes	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
PIP_getWriterNumFrames	Yes	Yes	Yes	No
PIP_getWriterSize	Yes	Yes	Yes	No
PIP_put	Yes	Yes	Yes	Yes
PIP_setWriterSize	Yes	Yes	Yes	No
PRD_getticks	Yes	Yes	Yes	No
PRD_start	Yes	Yes	Yes	No
PRD_stop	Yes	Yes	Yes	No
PRD_tick	Yes	Yes	Yes	Yes
QUE_create	Yes	No	No	Yes (see Note 2)
QUE_delete	Yes	No	No	Yes (see Note 2)
QUE_dequeue	Yes	Yes	Yes	No
QUE_empty	Yes	Yes	Yes	No
QUE_enqueue	Yes	Yes	Yes	No
QUE_get	Yes	Yes	Yes	No
QUE_head	Yes	Yes	Yes	No
QUE_insert	Yes	Yes	Yes	No
QUE_new	Yes	Yes	Yes	No
QUE_next	Yes	Yes	Yes	No
QUE_prev	Yes	Yes	Yes	No
QUE_put	Yes	Yes	Yes	No
QUE_remove	Yes	Yes	Yes	No
RTDX_channelBusy	Yes	Yes	No	No
RTDX_disableInput	Yes	Yes	No	No
RTDX_disableOutput	Yes	Yes	No	No
RTDX_enableInput	Yes	Yes	No	No
RTDX_enableOutput	Yes	Yes	No	No
RTDX_read	Yes	Yes	No	No
RTDX_readNB	Yes	Yes	No	No
RTDX_sizeofInput	Yes	Yes	No	No
RTDX_write	Yes	Yes	No	No
SEM_count	Yes	Yes	Yes	No
SEM_create	Yes	No	No	Yes (see Note 2)

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
SEM_delete	Yes	No	No	Yes (see Note 2)
SEM_ipost	No	Yes	Yes	Yes (see Note 7)
SEM_new	Yes	Yes	Yes	No
SEM_pend	Yes	Yes (see Note 2)	No (see Note 2)	Yes (see Note 8)
SEM_post	Yes	Yes	Yes	Yes (see Note 8)
SEM_reset	Yes	No	No	No
SIO_bufsize	Yes	Yes	Yes	No
SIO_create	Yes	No	No	Yes (see Note 2)
SIO_ctrl	Yes	No	No	Yes (see Note 8)
SIO_delete	Yes	No	No	Yes (see Note 8)
SIO_flush	Yes	No	No	No
SIO_get	Yes	No	No	Yes (see Note 2)
SIO_idle)	Yes	No	No	Yes (see Note 2)
SIO_issue	Yes	No	No	No
SIO_put	Yes	No	No	Yes
SIO_reclaim	Yes	No	No	Yes
SIO_segid	Yes	Yes	Yes	No
SIO_select	Yes	No	No	Yes (see Note 9)
SIO_staticbuf	Yes	No	No	No
STS_add	Yes	Yes	Yes	No
STS_delta	Yes	Yes	Yes	No
STS_reset	Yes	Yes	Yes	No
STS_set	Yes	Yes	Yes	No
SWI_andn	Yes	Yes	Yes	No
SWI_create	Yes	No	No	Yes (see Note 2)
SWI_dec	Yes	Yes	Yes	No
SWI_delete	Yes	No	No	Yes (see Note 2)
SWI_disable	Yes	Yes	Yes	No
SWI_enable	Yes	Yes	Yes	Yes (see Note 10)
SWI_getattrs	Yes	Yes	Yes	No
SWI_getmbox	No	Yes	No	No
SWI_getpri	Yes	Yes	No	No

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
SWI_inc	Yes	Yes	Yes	Yes
SWI_or	Yes	Yes	Yes	Yes
SWI_post()	Yes	Yes	Yes	Yes (see Note 11)
SWI_raisepri	No	Yes	No	Yes
SWI_restorepri	No	Yes	No	Yes
SWI_self	No	Yes	No	No
SWI_setattrs	Yes	Yes	Yes	No
SYS_abort	Yes	Yes	Yes	No
SYS_atexit	Yes	Yes	Yes	No
SYS_error	Yes	Yes	Yes	No
SYS_exit	Yes	Yes	Yes	No
SYS_printf	Yes	Yes	Yes	No
SYS_putchar	Yes	Yes	Yes	No
SYS_sprintf	Yes	Yes	Yes	No
SYS_vprintf	Yes	Yes	Yes	No
SYS_vsprintf	Yes	Yes	Yes	No
TRC_disable	Yes	Yes	Yes	No
TRC_enable	Yes	Yes	Yes	No
TRC_query	Yes	Yes	Yes	No
TSK_checkstacks	Yes	No	No	No
TSK_create	Yes	No	No	Yes (see Note 2)
TSK_delete	Yes	No	No	Yes (see Note 2)
TSK_deltatime	Yes	Yes	Yes	No
TSK_disable	Yes	No	No	No
TSK_enable	Yes	No	No	Yes (see Note 11)
TSK_exit	Yes	No	No	Yes (see Note 12)
TSK_getenv	Yes	Yes	Yes	No
TSK_geterr	Yes	Yes	Yes	No
TSK_getname	Yes	Yes	Yes	No
TSK_getpri	Yes	Yes	Yes	No
TSK_getsts	Yes	Yes	Yes	No
TSK_itick	No	Yes	Yes	Yes (see Note 8)

Function	Callable by Tasks?	Callable by SWI Handlers?	Callable by Hardware ISRs?	Possible Context Switch?
TSK_self	Yes	Yes	Yes	No
TSK_setenv	Yes	Yes	Yes	No
TSK_seterr	Yes	Yes	Yes	No
TSK_setpri	Yes	No	No	Yes (see Note 8)
TSK_settime	Yes	Yes	Yes	No
TSK_sleep	Yes	No	No	Yes (see Note 13)
TSK_stat	Yes	Yes	Yes	No
TSK_tick	Yes	Yes	Yes	Yes
TSK_time	Yes	Yes	Yes	No
TSK_yield	Yes	Yes	No	Yes (see Note 13)

Notes:
1) Task switch if memory manager blocks.
2) Task switch if higher priority task is waiting for lock.
3) Task switch if resource already locked by another task.
4) May only be called by SWI handlers or hardware ISRs if timeout parameter equals 0.
5) Task switch if mailbox empty and timeout parameter not 0, or if higher priority task blocked in MBX_post().
6) Task switch if higher priority task made ready, or mailbox is full and `timeout` parameter is not equal to 0.
7) Task switch if higher priority task made ready.
8) Task switch if semaphore count equals 0 and timeout parameter is not equal to 0.
9) Task switch if no streams are ready for I/O.
10) SWI_enable() will allow pending SWI handlers to run. A task switch will occur if one of these routines makes a higher priority task ready.
11) Task switch if higher priority task became ready while within a TSK_disable()/TSK_enable() block.
12) Task switch will always occur.
13) Task switch if there is an equal priority task ready.

A.2 DSP/BIOS Error Codes

Name	Value	SYS_Errors[Value]
SYS_OK	0	"(SYS_OK)"
SYS_EALLOC	1	"(SYS_EALLOC): segid = %d, size = %u, align = %u" Memory allocation error.
SYS_EFREE	2	"(SYS_EFREE): segid = %d, ptr = 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 to the device.
SYS_EBADIO	6	"(SYS_EBADIO): device failure" The device was unable to support the I/O operation.
SYS_EMODE	7	"(SYS_EMODE): invalid mode" An attempt was made to open a device in an improper mode; e.g., an attempt to open an input device for output.
SYS_EDOMAIN	8	"(SYS_EDOMAIN): domain error" Used by SPOX-MATH when type of operation does not match vector or filter type.
SYS_ETIMEOUT	9	"(SYS_ETIMEOUT): timeout error" Used by device drivers to indicate that reclaim timed out.
SYS_EEOF	10	"(SYS_EEOF): end-of-file error" Used by device drivers to indicate the end of a file.
SYS_EDEAD	11	"(SYS_EDEAD): previously deleted object" An attempt was made to use an object that has been deleted.
SYS_EBADOBJ	12	"(SYS_EBADOBJ): invalid object" An attempt was made to use an object that does not exist.
SYS_EUSER	>=256	"(SYS EUSER): <user-defined string>" User-defined error.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W