TMS320C54x DSP/BIOS User's Guide

Literature Number: SPRU326A
September 1999

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserves the right to make changes to their products or to discontinue any product or service without notice, and advises customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current and complete. All products are sold subject to the terms and conditions of sale 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.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

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.

Before you read this manual, you should follow the tutorials in the *TMS320C54x Code Composer Studio Tutorial* (literature number SPRU327a) 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 'C54x.
- 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 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 'C54x generation of devices.

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

TMS320C54x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the 'C54x. 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 'C54x 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 'C54x. 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 'C54x 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 'C54x. 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 'C54x 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 user's 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

Contents

Chapter 1

About DSP/BIOS

DSP/BIOS gives developers of applications for DSP chips the ability to develop and analyze embedded real-time software. DSP/BIOS includes a small firmware real-time library, the DSP/BIOS API for using real-time library services, and easy-to-use tools for configuration and for real-time tracing and analysis.

Topic **Page**

1.1 DSP/BIOS Features and Benefits

The DSP/BIOS API and host tools are designed to minimize the memory and CPU requirements on the target. This design goal was accomplished in the following ways:

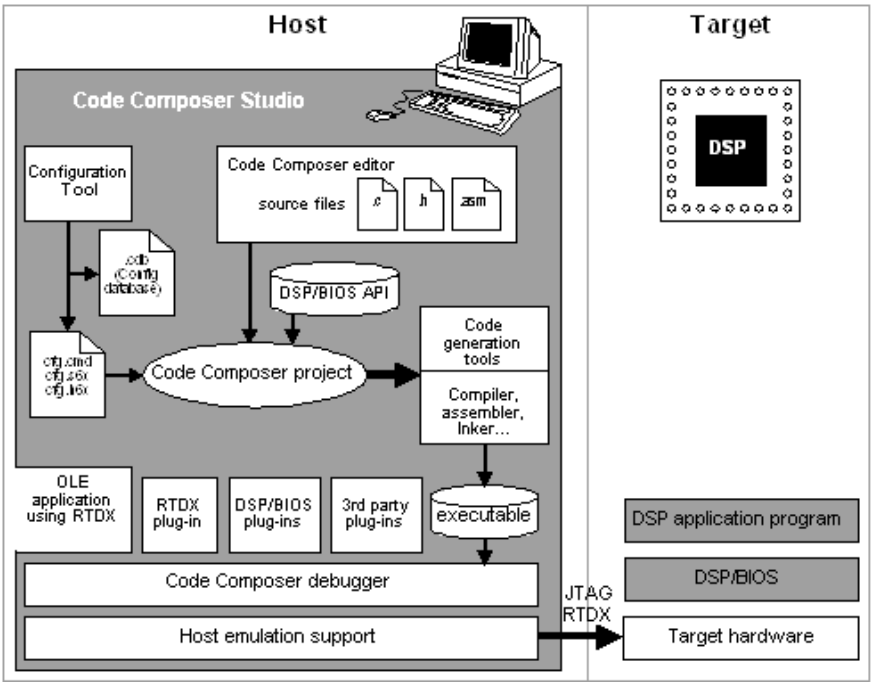
- ❏ All DSP/BIOS objects are created in the Configuration Tool and bound into an executable program image. This reduces code size and optimizes internal data structures.
- ❏ All formatting of instrumentation data (such as logs and traces) is done on the host.
- ❏ The API is modularized, so that only the parts of the API that are used by the program need to be bound into the executable program.
- ❏ The API is optimized to require the smallest possible number of instruction cycles.
- ❏ Communication between the target and the DSP/BIOS plugins is performed within the background idle loop. This ensures that the DSP/BIOS plugins do not interfere with the program's tasks. If the target CPU is too busy to perform background tasks, the DSP/BIOS plugins stop receiving information from the target until the CPU is available.

The DSP/BIOS API standardizes DSP programming for a number of TI chips and provides easy-to-use, powerful program development and testing tools. The goal is to reduce the time required to create DSP programs. This goal was accomplished in the following ways:

- ❏ The Configuration Tool generates code required to declare objects used within the program.
- ❏ The Configuration Tool detects errors earlier by validating object properties before program execution.
- ❏ The DSP/BIOS plugins allow real-time monitoring of program behavior.
- ❏ The DSP/BIOS API is a standard API. This allows DSP algorithm developers to provide code that can be more easily integrated with other program functions.

1.2 DSP/BIOS Components

This figure shows the components of DSP/BIOS within the program generation and debugging environment of Code Composer:



On the host PC, you write programs that use the DSP/BIOS API (in C or assembly). The Configuration Tool lets you define objects to be used in your program. You then compile or assemble and link the program. The DSP/BIOS plugins let you test the program on the target chip from Code Composer while monitoring CPU load, timing, logs, thread execution, and more. (Threads is a general term used to refer to any thread of execution, e.g., a hardware ISR, a software interrupt, an idle function, or a periodic function.)

The following sections give a brief overview of the DSP/BIOS components.

1.2.1 DSP/BIOS Real-Time Library and API

The small, firmware DSP/BIOS real-time library provides basic run-time services to embedded programs that run on the target hardware. It includes operations for capturing information generated by the application in real time, I/O modules, a software interrupt manager, a clock manager, and more.

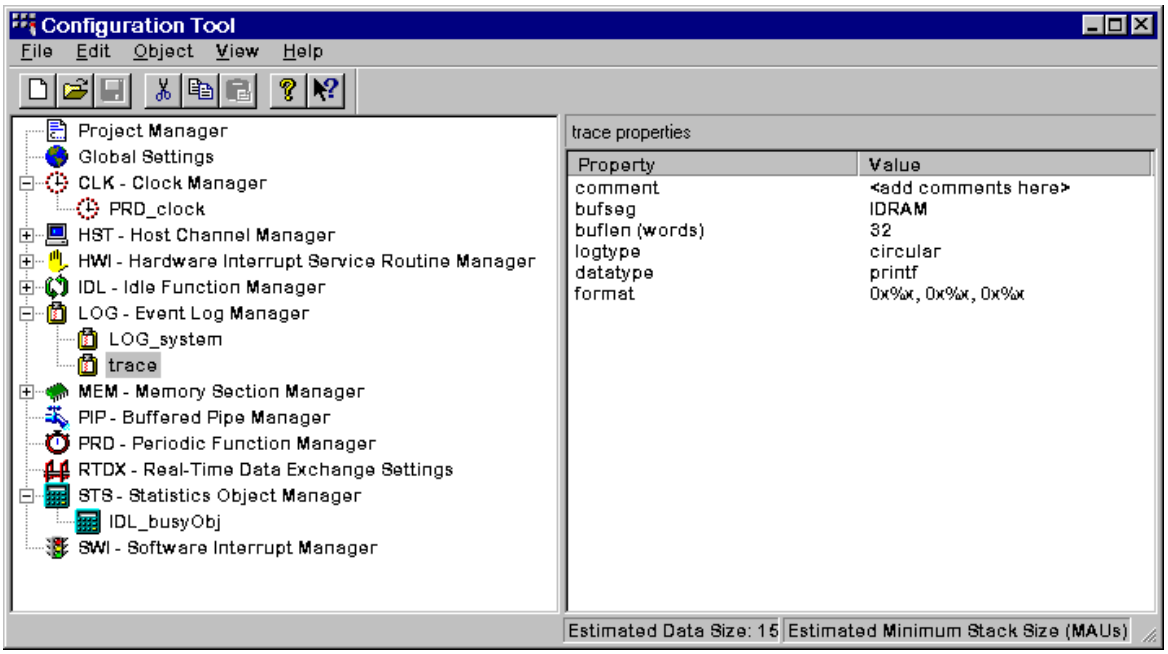
The DSP/BIOS API is divided into modules. Depending on what modules are configured and used by the application, the size of DSP/BIOS ranges from 200 to 2000 words of code.

Application programs use DSP/BIOS by making calls to the API. For C programs, header files define the API. For applications that need assembly language optimization, an optimized set of macros is provided. Using C with DSP/BIOS is optional, as the real-time library itself is written in assembler to minimize time and space.

1.2.2 The DSP/BIOS Configuration Tool

Using an interface similar to the Windows Explorer, the Configuration Tool has two roles:

- ❑ It lets you set a wide range of parameters used by the DSP/BIOS real-time library at run time.
- ❑ It serves as a visual editor for creating run-time objects that are used by the target application's DSP/BIOS API calls. These objects include software interrupts, I/O streams, and event logs. You also use this visual editor to set properties for these objects.



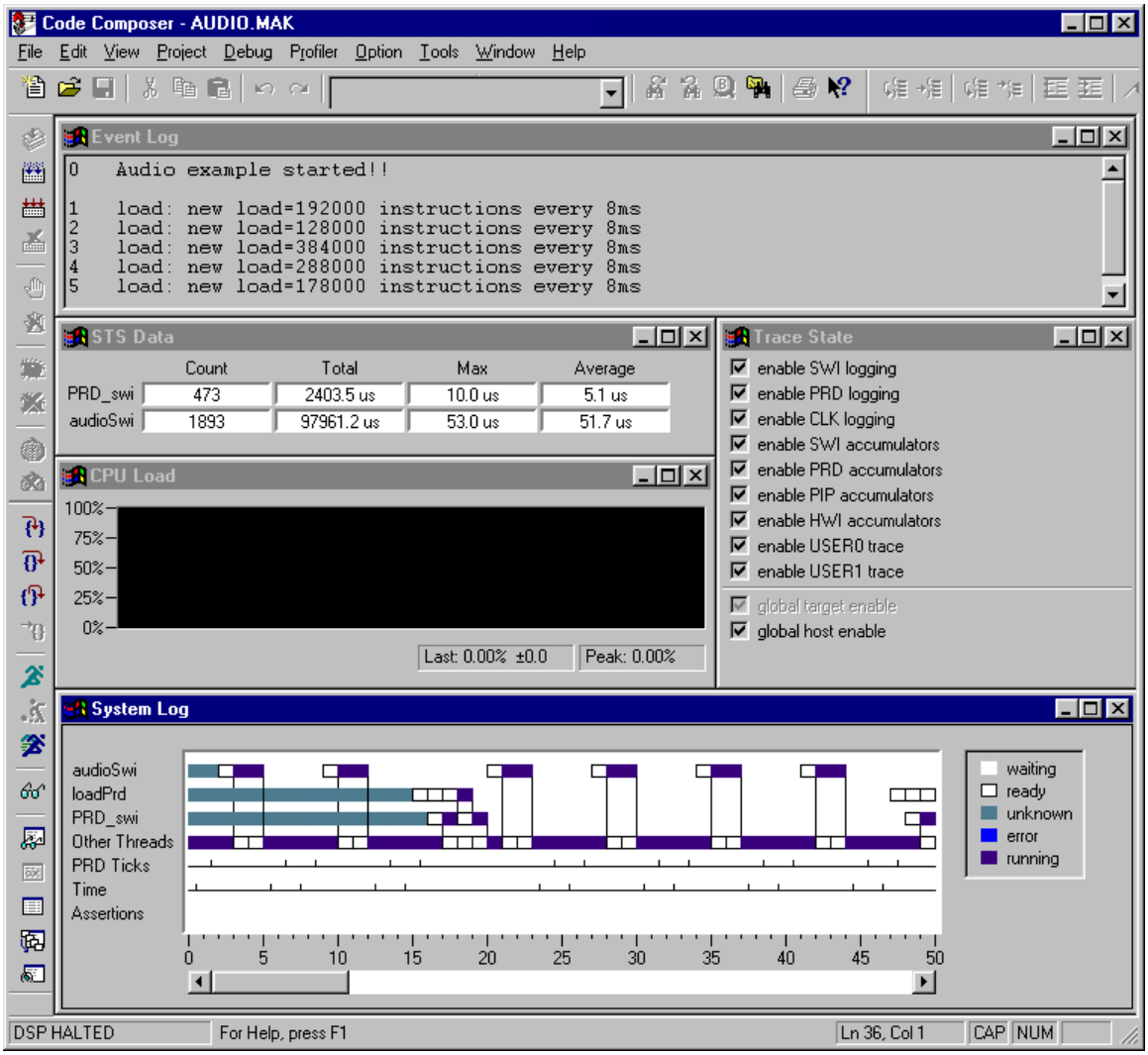
Unlike systems that create objects at run time with API calls that require extra target overhead (especially code space), all DSP/BIOS objects are pre-configured and bound into an executable program image. In addition to minimizing the target memory footprint by eliminating run-time code and optimizing internal data structures, this static configuration strategy detects errors earlier by validating object properties before program execution.

The Configuration Tool generates several files that are linked with the code you write. See section 2.2.5, *Files Generated by the Configuration Tool*, page 2-4, for details.

1.2.3 The DSP/BIOS plugins

The DSP/BIOS plugins in Code Composer complement the program debugging utilities by enabling real-time program analysis of a DSP/BIOS application. You can visually probe, trace, and monitor a DSP application as it runs with minimal impact on the application's real-time performance.

Unlike traditional debugging, which is external to the executing program, program analysis requires that the target program contain real-time instrumentation services. By using DSP/BIOS APIs and objects, developers have automatically instrumented the target for capturing and uploading real-time information to the host through the DSP/BIOS plugins in Code Composer.



Several broad real-time program analysis capabilities are provided:

- **Program tracing**. Displaying events written to target logs, reflecting dynamic control flow during program execution
- **Performance monitoring**. Tracking summary statistics that reflect use of target resources, such as processor load and timing
- **File streaming**. Binding target-resident I/O objects to host files

When used in tandem with the other debugging capabilities in Code Composer, the DSP/BIOS real-time analysis tools provide critical views into target program behavior in the area where traditional debugging techniques that stop the target offer little or no insight—during program execution. Even after the debugger halts the program, information already captured by the host with the DSP/BIOS plugins can provide insights into the sequence of events that led up to the current point of execution.

Later in the software development cycle, when regular debugging techniques become ineffective for attacking problems arising from time-dependent interactions, the DSP/BIOS plugins have an expanded role as the software counterpart of the hardware logic analyzer.

1.3 Naming Conventions

Each DSP/BIOS module has a unique 3- or 4-letter name that is used as a prefix for operations (functions), header files, and objects for the module.

All identifiers beginning with upper-case letters followed by an underscore (XXX_*) should be treated as reserved words. Identifiers beginning with an underscore are also reserved for internal system names.

1.3.1 Module Header Names

Each DSP/BIOS module has two header files containing declarations of all constants, types, and functions made available through that module's interface.

- **module.h**. DSP/BIOS API header files for C programs. Your C source files should include std.h and the header files for any modules the C functions use.
- **module.h54**. DSP/BIOS API header files for assembly programs. Assembly source files should include the *.h54 header file for any module the assembly source uses. This file contains macro definitions specific to this chip. Data structure definitions shared for all supported chips are stored in the module.hti files, which are included by the *.h54 header files.

Your program must include the corresponding header for each module used in a particular program source file. In addition, C source files must include std.h before any module header files (see section 1.3.4, *Data Type Names*, page 1-10, for more information). The std.h file contains definitions for standard types and constants. Other than including std.h first, you may include the other header files in any sequence. For example:

```
#include <std.h>
#include <pip.h>
#include <prd.h>
#include <swi.h>
```

DSP/BIOS includes a number of modules whose functions are for internal use. These modules are consequently undocumented and subject to change at any time. Header files for these internal modules are distributed as part of DSP/BIOS and must be present on your system when compiling and linking DSP/BIOS programs.

1.3.2 Object Names

System objects that are included in the configuration by default typically have names beginning with a 3- or 4-letter code for the module that defines or uses the object. For example, the default configuration includes a LOG object called LOG_system.

Objects you create with the Configuration Tool should use a common naming convention of your choosing. You might want to use the module name as a suffix in object names. For example, a SWI object that encodes data might be called encoderSwi.

1.3.3 Operation Names

The format for a DSP/BIOS API operation name is MOD_action where MOD is the letter code for the module that contains the operation, and action is the action performed by the operation. For example, the SWI_post function is defined by the SWI module; it posts a software interrupt.

This implementation of the DSP/BIOS API also includes several built-in functions that are run by various built-in objects. Here are some examples:

- **CLK_F_isr**. Run by the HWI_TINT object to provide the low-resolution CLK tick
- **PRD_F_tick**. Run by the PRD_clock object to provide the system tick
- **IDL_F_busy**. Run by the IDL_cpuLoad object to compute the current CPU load
- **RTA_F_dispatch**. Run by the RTA_dispatcher object to gather real-time analysis data
- **LNK_F_dataPump**. Run by the LNK_dataPump object to transfer real-time analysis and HST channel data to the host
- **HWI_unused**. Not actually a function name. This string is used in the Configuration Tool to mark unused HWI objects.

Note: Your program code should not call any built-in functions whose names begin with MOD_F_. These functions are intended to be called only as function parameters specified within the Configuration Tool.

Operation names beginning with MOD_ and MOD_F_ (where MOD is any letter code for a DSP/BIOS module) are reserved for internal use.

1.3.4 Data Type Names

The DSP/BIOS API does not explicitly use the fundamental types of C such as int or char. Instead, to ensure portability to other processors that support the DSP/BIOS API, DSP/BIOS defines its own standard data types. In most cases, the standard DSP/BIOS types are simply capitalized versions of the corresponding C types.

The following types are defined in the std.h header file:

Type	Description
Arg	Type capable of holding both Ptr and Int arguments
Bool	Boolean value
Char	Character value
Int	Signed integer value
LgInt	Large signed integer value
LgUns	Large unsigned integer value
Ptr	Generic pointer value
String	Zero-terminated (\0) sequence (array) of characters
Uns	Unsigned integer value
Void	Empty type

Additional data types are defined in std.h, but are not used by the DSP/BIOS API.

In addition, the standard constant NULL (0) is used by DSP/BIOS to signify an empty pointer value. The constants TRUE (1) and FALSE (0) are used for values of type Bool.

Object structures used by the DSP/BIOS API modules use a naming convention of MOD_Obj, where MOD is the letter code for the object's module. If your program uses any such objects, it should include an extern declaration for the object. For example:

```
extern LOG_Obj trace;
```

1.3.5 Memory Segment Names

The memory segment names used by DSP/BIOS are described in the following table.

Memory Segment	Description
IDATA	Internal (on-chip) data memory
EDATA	Primary block of external data memory
EDATA1	Secondary block of external data memory (not contiguous with EDATA)
IPROG	Internal (on-chip) program memory
EPROG	Primary block of external program memory
EPROG1	Secondary block of external program memory (not contiguous with EPROG)
USERREGS	Page 0 user memory (28 words)
BIOSREGS	Page 0 reserved registers (4 words)
VECT	Interrupt vector segment

You can change the origin, size, and name of the default memory segments described above using the Configuration Tool.

The Configuration Tool defines standard memory sections and their default allocations as follows:

Memory Segment	Description
IDATA	Application Stack Memory
EDATA	Application Argument Memory
EDATA	Application Constants Memory
IPROG	BIOS Program Memory
EDATA	BIOS Data Memory
IDATA	BIOS Heap Memory
EPROG	BIOS Startup Code Memory

You can change these default allocations by using the MEM manager in the Configuration Tool. See *MEM Module*, page 6-49, for more details.

1.4 For More Information

For more information about the components of DSP/BIOS and the modules in the DSP/BIOS API, see the *TMS320C54x Code Composer Studio Tutorial*.

Chapter 2

Program Generation

This chapter describes the process of generating programs with DSP/BIOS. It also explains which files are generated by DSP/BIOS components and how they are used.

Topic **Page**

2.1 Development Cycle

DSP/BIOS supports iterative program development cycles. You can create the basic framework for an application and test it with a simulated processing load before the DSP algorithms are in place. You can easily change the priorities and types of program components that perform various functions.

A sample DSP/BIOS development cycle would include the following steps, though iteration could occur for any step or group of steps:

1) Write a framework for your program. You can use C or assembly code.
2) Use the Configuration Tool to create objects for your program to use.
3) Save the configuration file, which generates files to be included when you compile and link your program.
4) Compile and link the program using a makefile or a Code Composer project.
5) Test program behavior using a simulator or initial hardware and the DSP/BIOS plugins. You can monitor logs and traces, statistics objects, timing, software interrupts, and more.
6) Repeat steps 2-5 until the program runs correctly. You can add functionality and make changes to the basic program structure.
7) When production hardware is ready, modify the configuration file to support the production board and test your program on the board.

2.2 Using the Configuration Tool

The Configuration Tool is a visual editor with an interface similar to Windows Explorer. It allows you to initialize data structures and set various parameters. When you save a file, the Configuration Tool creates assembly and header files and a linker command file to match your settings. When you build your application, these files are linked with your application programs.

2.2.1 Creating a New Configuration

1) Open the Configuration Tool. From Code Composer, open the Configuration Tool by selecting File→New→DSP/BIOS Config.
2) Select the appropriate template and click OK. Alternatively, you can open the Configuration Tool outside of Code Composer from the Start menu.
3) From the File menu, select New.
4) Double-click on the configuration template for the board you are using.

2.2.2 Creating a Custom Template

You can add a custom template by creating a configuration file and storing it in your include folder. This saves time by allowing you to define configuration settings for your hardware once and then reuse the file as a template.

For example, to build DSP/BIOS programs for the ’C54x fixed point DSP, you may use settings as provided (for the ’C54x). Or you may instruct the Configuration Tool to create a new custom template file for projects that should take advantage of the fixed point run-time library.

To create a custom template, perform the following steps:

1) Invoke the Configuration Tool from outside Code Composer via Start→Programs→Code Composer Studio ’C54x→Configuration Tool.
2) From the File menu, choose New.
3) In the New window select evm54.cdb and click OK.
4) Right-click on Global Settings and select Properties.
5) Set Target Board Name to evm54.
6) Set DSP Type to 54 and click OK.
7) Select File→Save As. In the Save As dialog box navigate to ti\c5400\bios\include.
8) In the File Name box type evm54.cdb.
9) In the Save as type box select Seed files (*.cdb) and click Save.
10) In the Set Description Dialog type a description and click OK.
11) From the Configuration Tool's main menu select File→Exit.

2.2.3 Setting Global Properties for a Module

1) When you select a module (by clicking on it), the right side of the window shows the current properties for the module. (If you see a list of priorities instead of a property list, right-click on the module and select Property/value view. If the right side of the window is gray, this module has no global properties.)

   For help about a module, click  and then click on the module.

2) Right-click the icon next to the module and select Properties from the pop-up menu. This opens the property sheet.

3) Change properties as needed. For help on the module's properties, click Help in the property sheet.

2.2.4 Creating an Object and Specifying its Properties

1) Right-click on a module and select Insert MOD, where MOD is the name of the module. This adds a new object for this module. (You cannot create an object for the GBL, HWI, or RTDX modules.)

2) Rename the object. Right-click on the name and choose Rename from the pop-up menu.

3) Right-click the icon next to the object and select Properties to open the property sheet.

> **Note**: When specifying C functions to be run by various objects, add an underscore before the C function name. For example, type _myfunc to run a C function called myfunc(). The underscore prefix is necessary because the Configuration Tool creates assembly source, and C calling conventions require an underscore before C functions called from assembly.

4) Change property settings and click OK. For help on specific properties, click Help in any property sheet.

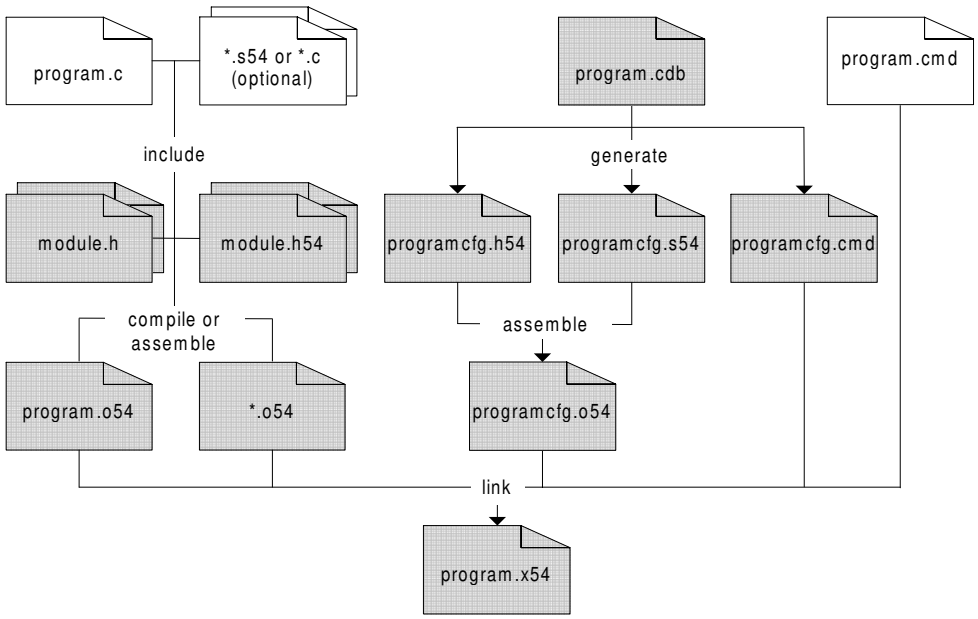
2.2.5 Files Generated by the Configuration Tool

When you save a configuration file for your program with the Configuration Tool, the following files are created. These files are described in section 2.3, *Files Used to Create DSP/BIOS Programs*, page 2-5.

- ❏ program.cdb
- ❏ programcfg.h54
- ❏ programcfg.s54
- ❏ programcfg.cmd

2.3 Files Used to Create DSP/BIOS Programs

This diagram shows the files used to create DSP/BIOS programs. Files you write are represented with a white background; generated files are represented with a gray background.



A 54 in the file extension shown above indicates that the chip number abbreviation is used here. (For ’C54x chips, the abbreviation is 54.)

- **program.c**. C program source file containing the main() function. You may also have additional .c source files.
- ***.asm**. Optional assembly source file(s). One of these files can contain an assembly language function called _main as an alternative to using a C function called main().
- **module.h**. DSP/BIOS API header files for C programs. Your C source files should include std.h and the header files for any modules the C program uses.
- **module.h54**. DSP/BIOS API header files for assembly programs. Assembly source files should include the *.h54 header file for any module the assembly source uses.
- **program.obj**. Object file(s) compiled or assembled from your source file(s)
- ***.obj**. Object files for optional assembly source file(s)

- **program.cdb**. Configuration file, which stores configuration settings. This file is created by the Configuration Tool and used by both the Configuration Tool and the DSP/BIOS plugins.
- **programcfg.h54**. Header file generated by the Configuration Tool. This header file is included by the programcfg.s54 file.
- **programcfg.s54**. Assembly source generated by the Configuration Tool
- **programcfg.cmd**. Linker command file created by the Configuration Tool and used when linking the executable file. This file defines DSP/BIOS-specific link options and object names, and generic data sections for DSP programs (such as .text, .bss, .data, etc.).
- **programcfg.obj**. Object file created from source file generated by the Configuration Tool.
- ***.cmd**. Optional linker command file(s) that contains additional sections for your program not defined by the Configuration Tool.
- **program.out**. An executable program for the 'C54x target (fully compiled, assembled, and linked). You can load and run this program with Code Composer.

2.3.1 Files Used by the DSP/BIOS Plugins

The following files are used by the DSP/BIOS plugins:

- **program.cdb**. The DSP/BIOS plugins use the configuration file to get object names and other program information.
- **program.out**. The DSP/BIOS plugins use the executable file to get symbol addresses and other program information.

2.4 Compiling and Linking Programs

You can build your DSP/BIOS executables using a Code Composer project or using your own makefile. Code Composer includes gmake.exe, GNU's make utility, and sample makefiles for gmake to build the tutorial examples.

2.4.1 Building with a Code Composer Project

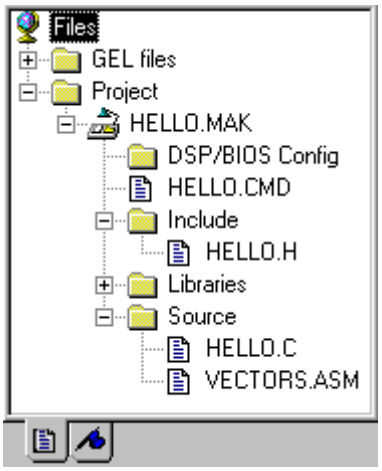
When building a DSP/BIOS application using a Code Composer project, you must add the following files to the project in addition to your own source code files:

- program.cdb (the configuration file)
- programcfg.cmd (the linker command file)

Code Composer adds programcfg.s54, the configuration source file, automatically.

Note that in a DSP/BIOS application, programcfg.cmd is your project's linker command file. programcfg.cmd already includes directives for the linker to search the appropriate libraries (e.g., bios.a54, rtdx.lib, rts5401.lib), so you do not need to add any of these library files to your project.

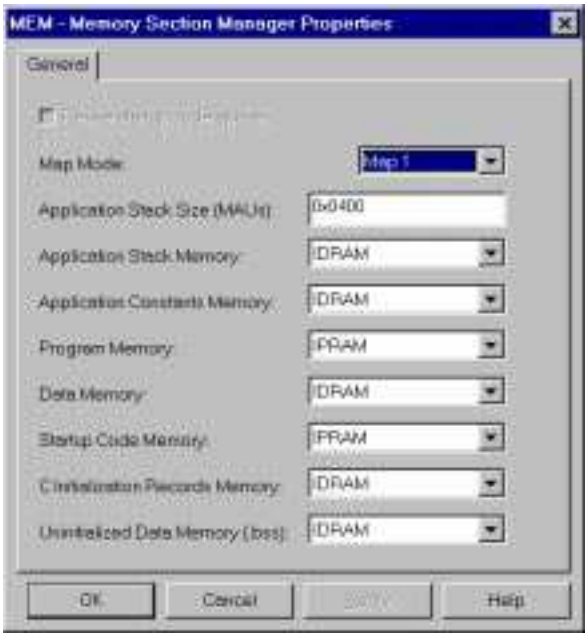
Code Composer automatically scans all dependencies in your project files, adding the necessary DSP/BIOS and RTDX header files for your configuration to your project's include folder.



For details on how to create a Code Composer project and build an executable from it, refer to the *Code Composer Studio User's Guide* or the *TMS320C54x Code Composer Studio Tutorial*.

2.4.1.1 Building with Multiple Linker Command Files

For most DSP/BIOS applications the generated linker command file, programcfg.cmd, suffices to describe all memory sections and allocations. All DSP/BIOS memory sections and objects are handled by this linker command file. In addition, most commonly used sections (such as .text, .bss, .data, etc.) are already included in programcfg.cmd. Their locations (and sizes, when appropriate) can be controlled from the MEM manager in the Configuration Tool.



In some cases the application may require an additional linker command file (app.cmd) describing application-specific sections that are not described in the linker command file generated by the Configuration Tool (programcfg.cmd).

Note: Code Composer allows only one linker command file per project. When both programcfg.cmd and app.cmd are required by the application, the project should use app.cmd (rather than programcfg.cmd) as the project's linker command file. To include programcfg.cmd in the linking process, you must add the following line to the beginning of app.cmd:

```
-lprogramcfg.cmd
```

Note that it is important that this line appear at the beginning, so that programcfg.cmd is the first linker command file used by the linker.

2.4.2 Makefiles

As an alternative to building your program as a Code Composer project, you can use a makefile.

In the following example, the C source code file is volume.c, additional assembly source is in load.asm, and the configuration file is volume.cdb. This makefile is for use with gmake, which is included with Code Composer. You can find documentation for gmake on the product CD in PDF format. Adobe Acrobat Reader is included. This makefile and the source and configuration files mentioned are located in the volume2 subdirectory of the tutorial directory of Code Composer.

A typical makefile for compiling and linking a DSP/BIOS program looks like the following.

```
# 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
#

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

# Every BIOS program must be linked with:
#    $(PROG)cfg.o54 - object resulting from assembling
#                     $(PROG)cfg.s54
#    $(PROG)cfg.cmd - linker command file generated by
#                     the Configuration Tool. If additional
#                     linker command files exist,
#                     $(PROG)cfg.cmd must appear first.
#
PROG    = volume
OBJS    = $(PROG)cfg.obj load.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:
    @ echo Error: $@ must be manually regenerated:
    @ echo Open and save $(PROG).cdb within the BIOS
      Configuration Tool.
    @ check $@

.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
```

You can copy an example makefile to your program folder and modify the makefile as necessary.

Unlike the Code Composer project, makefiles allow for multiple linker command files. If the application requires additional linker command files you can easily add them to the CMDS variable in the example makefile shown above. However, they must always appear after the programcfg.cmd linker command file generated by the Configuration Tool.

2.5 DSP/BIOS Startup Sequence

When a DSP/BIOS application starts up, the startup sequence is determined by the instructions in the boot.s54 file. A compiled version of this file is provided with the bios.a54 library. You should not need to make any changes to this file; nevertheless, the source file for boot.s54 is provided and presented here to illustrate the DSP/BIOS startup sequence.

```
;
;  ======== boot.s54 ========
;
   .include bios.h54
   .c_mode
   .mmregs
CONST_COPY   .set 0
*****************************************************************************
*                                                                           *
*   This module contains the following definitions :                        *
*                                                                           *
*         __stack    - Stack memory area                                    *
*         _c_int00   - Boot function                                        *
*         _var_init  - Function which processes initialization tables       *
*                                                                           *
*****************************************************************************
    .ref    cinit, pinit
   .global  _c_int00
   .global  _main, __STACK_SIZE
   ; alternate label for c_int00 -- referenced by HWI_RESET
   ; to pull in BIOS boot code instead of rts.lib
   .global BIOS_reset
*****************************************************************************
* Declare the stack.  Size is determined by the linker option -stack.  The *
* default value is 1K words.                                                *
*****************************************************************************
__stack:  .usect  ".stack",0
args      .sect   ".args"
*****************************************************************************
* FUNCTION DEF : _c_int00                                                   *
*                                                                           *
*   1) Set up stack                                                         *
*   2) Set up proper status                                                 *
*   3) If "cinit" is not -1, init global variables                          *
*   4) call users' program                                                  *
*                                                                           *
*****************************************************************************
      .sect ".sysinit"
BIOS_reset:
_c_int00:
   xc   1, unc
     ssbx   intm         ; set interrupt mask bit
   stm   #0, imr         ; disable all interrupts
```

```
*****************************************************************************
*  INIT STACK POINTER.  REMEMBER STACK GROWS FROM HIGH TO LOW ADDRESSES.    *
*****************************************************************************
   STM    #__stack,SP             ; set to beginning of stack memory
   ADDM   #(__STACK_SIZE-1),*(SP) ; add size to get to top
   ANDM   #0fffeh,*(SP)           ; make sure it is an even address

   SSBX   SXM                     ; turn on SXM for LD #cinit,A
*****************************************************************************
* SET UP REQUIRED VALUES IN STATUS REGISTER                                  *
*****************************************************************************
   SSBX   CPL                     ; turn on compiler mode bit
   RSBX   OVM                ; clear overflow mode bit
*****************************************************************************
* SETTING THESE STATUS BITS TO RESET VALUES.  IF YOU RUN _c_int00 FROM      *
* RESET, YOU CAN REMOVE THIS CODE                                           *
*****************************************************************************
   LD     #0,ARP
   RSBX   C16
   RSBX   CMPT
   RSBX   FRCT
*****************************************************************************
*  SETUP PMST -   GBL_PMST is defined by DSP/BIOS Configuration Tool
*
*  The PMST must be initialized before the .pinit section is processed since
*  the RTDX initialization triggers an interrupt (IVTP is in PMST).  If
*  the Interrupt Vector Table Pointer is not correct, the processor will
*  exececute an undefined interrupt vector.
*
*****************************************************************************
   .ref    GBL_PMST
   stm     #GBL_PMST, pmst
*****************************************************************************
*  IF cinit IS NOT -1, PROCESS INITIALIZATION TABLES                        *
*  TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT:                    *
*                                                                           *
*       .word  <length of init data in words>                               *
*       .word  <address of variable to initialize>                          *
*       .word  <init data>                                                  *
*       .word  ...                                                          *
*                                                                           *
*  The cinit table is terminated with a zero length                         *
*                                                                           *
*****************************************************************************
   .if   __far_mode
   LDX   #cinit,16,A
   OR    #cinit,A,A
   .else
   LD    #cinit,A                 ; Get pointer to init tables
   .endif
   ADD   #1,A,B
   BC    DONE_CINIT,BEQ           ; if (cinit == -1) no init tables
```

```
*************************************************************************
*  PROCESS INITIALIZATION TABLES.  TABLES ARE IN PROGRAM MEMORY IN THE     *
*  FOLLOWING FORMAT:                                                      *
*                                                                         *
*       .word  <length of init data in words>                             *
*       .word  <address of variable to initialize>                        *
*       .word  <init data>                                                *
*       .word  ...                                                        *
*                                                                         *
*  The init table is terminated with a zero length                        *
*                                                                         *
*************************************************************************
   BD     START_CINIT      ; start processing
   RSBX   SXM              ; do address arithmetic unsignedly
   nop
LOOP_CINIT:
   READA   *(AR2)           ; AR2 = address
   ADD   #1,A               ; A += 1
   RPT   *(AR1)             ; repeat length+1 times
   READA   *AR2+            ; copy from table to memory
   ADD   *(AR1),A           ; A += length (READA doesn't change A)
   ADD   #1,A               ; A += 1
START_CINIT:
   READA   *(AR1)           ; AR1 = length
   ADD   #1,A               ; A += 1
   BANZ   LOOP_CINIT,*AR1-  ; if (length-- != 0) continue
DONE_CINIT:
```

```
*************************************************************************
*  IF pinit IS NOT -1, PROCESS INITIALIZATION TABLES                    *
*  TABLES ARE IN PROGRAM MEMORY IN THE FOLLOWING FORMAT:                *
*                                                                       *
*   .if __far_mode                                                      *
*   .long <32-bit address of initialization routine to call>            *
*   .else                                                               *
*   .word <16-bit address of initialization routine to call>            *
*   .endif                                                              *
*   ...                                                                 *
*                                                                       *
*  The pinit table is terminated with a NULL pointer                    *
*                                                                       *
*************************************************************************
   SSBX   SXM
   FRAME   -4
;   NOP
   .if    __far_mode
   LDX    #pinit,16,A
   OR     #pinit,A,A          ; A = &pinit table
   .else
   LD     #pinit,A            ; A = &pinit table
   .endif
   ADD    #1,A,B              ; B = A + 1
   BC     DONE_PINIT,BEQ      ; if (pinit == -1) no pinit tables
   BD     START_PINIT
   DST    A, @2               ; save 32-bit PGM pointer on stack
   NOP
LOOP_PINIT:
   .if    __far_mode
   FCALA  B                   ; call function
   .else
   CALA   B                   ; call function
   .endif
   DLD    @2, A               ; put 32-bit PINIT pointer in A
START_PINIT:
   READA  @0                 ; "push" (MSB) address of function
   .if    __far_mode
   ADD    #1, A
   READA  @1                 ; "push" LSB address of function
   .endif
   .if    __far_mode
   ADD    #1, A
   DST    A, @2
   DLD    @0, B
   BC     LOOP_PINIT,BNEQ
   .else
   LD     @0, B              ; "pop" address of function
   BCD    LOOP_PINIT,BNEQ    ; if not NULL, loop.
   ADDM   #1,@3              ; move PINIT pointer (in stack)
   .endif

DONE_PINIT:
   RSBX   SXM
   FRAME   4
```

```
*************************************************************************
*  COPY .CONST SECTION
*************************************************************************
   .if CONST_COPY
   .if __far_mode                  ; Use far calls for C548 in far mode
   FCALL   _const_init             ; move .const section to DATA mem
      .else
      CALL   _const_init
   .endif
   .endif
*************************************************************************
*  CALL BIOS_init
*************************************************************************
   call    BIOS_init      ; initialize the BIOS
                          ; cpl = 1 when return from BIOS_init
*************************************************************************
*  Set up C environment before calling main(argc, argv, envp).        *
*************************************************************************
     ld   #args,b
   stlm   b,ar1
   nop                  ; these 2 nops are necessary for
   nop                  ; the latency of "stlm b, ar1"
   ld   *ar1+,a         ; a = argc
   frame   -2
   ld   *ar1+,b
   stl   b,*sp(0)       ; sp(0) = argv
   ld   *ar1,b
   stl   b,*sp(1)       ; sp(1) = envp
*************************************************************************
*  CALL main()
*************************************************************************
   .if __far_mode       ; Use far calls for C548 in far mode
   FCALL    _main       ; far call to the user's entry point
   .else
   CALL    _main
   .endif
   frame 2
*************************************************************************
*  CALL BIOS_start()
*************************************************************************
    call    BIOS_start  ; 'start' the BIOS
*************************************************************************
*  DROP INTO IDL LOOP
*************************************************************************
    rsbx   cpl         ; cpl = 0 is precond of IDL_loop
   IDL_loop            ; fall into BIOS "idle task", never
                       ; return
   .if CONST_COPY
```

```
*****************************************************************************
* FUNCTION DEF : __const_init                                               *
*                                                                           *
*  COPY .CONST SECTION FROM PROGRAM TO DATA MEMORY                          *
*                                                                           *
*   The function depends on the following variables                         *
*   defined in the linker command file                                      *
*                                                                           *
*   __c_load         ; global var containing start                          *
*                      of .const in program memory                          *
*   __const_run      ; global var containing run                            *
*                      address in data memory                               *
*   __const_length   ; global var length of .const                         *
*                      section                                              *
*                                                                           *
*****************************************************************************
        .global __const_length,__c_load
        .global __const_run
_const_init:

        .sect ".c_mark"             ; establish LOAD adress of
        .label __c_load             ; .const section

   .sect ".sysinit"
*****************************************************
* C54x VERSION                                      *
*****************************************************

        LD      #__const_length, A
        BC      __end_const,AEQ
        STM     #__const_run,AR2     ; Load RUN address of .const

        RPT     #__const_length-1
        MVPD    #__c_load,*AR2+      ; Copy .const from program to data

*****************************************************
*  AT END OF .CONST SECTION RETURN TO CALLER         *
*****************************************************
__end_const:
   .if    __far_mode
   .if __no_fret
   FB   _freti549
   .else
   FRET
   .endif
   .else
        RET
   .endif
        .endif
   .end
```

The steps followed in the startup sequence are:

1) **Initialize the DSP**. A DSP/BIOS program starts at the C environment entry point c_int00. The reset interrupt vector is set up to branch to c_int00 after reset. At the beginning of c_int00, the software stack pointer (SP) is set up to point to the end of .stack. Status registers such as st0 and st1 are also initialized. Once the SP is set up, the initialization routine is called to initialize the variables from the .cinit records.

2) **Call BIOS_init to initialize the DSP/BIOS modules**. BIOS_init is generated by the Configuration Tool and is located in the programcfg.s54 file. BIOS_init is responsible for basic module initialization. BIOS_init invokes the MOD_init macro for each DSP/BIOS module.

   - HWI_init clears the IFR. See Chapter 6, *API Functions*, for more information.
   - HST_init initializes the host I/O channel interface. The specifics of this routine depend on the particular implementation used for the host to target link.
   - If the Auto calculate idle loop instruction count box was selected in the Idle Function Manager in the Configuration Tool, IDL_init calculates the idle loop instruction count at this point in the startup sequence. The idle loop instruction count is used to calibrate the CPU load displayed by the CPU Load Graph (see also section 3.5.2, *The CPU Load*, page 3-15).

3) **Call your program's main routine**. After all DSP/BIOS modules have completed their initialization procedures, your main routine is called. This routine can be written in assembly or C. Because the C compiler adds an underscore prefix to function names, this can be a C function called main or an assembly function called _main. The boot routine passes three parameters to main: argc, argv, and envp, which correspond to the C command line argument count, command line arguments array, and environment variables array.

   Since neither hardware or software interrupts are enabled yet, you can take care of initialization procedures for your own application (such as calling your own hardware initialization routines) from the main routine.

4) **Call BIOS_start to start DSP/BIOS**. Like BIOS_init, BIOS_start is also generated by the Configuration Tool and is located in the programcfg.s54 file. BIOS_start is called after the return from your main routine. BIOS_start is responsible for enabling the DSP/BIOS modules and invoking the MOD_startup macro for each DSP/BIOS module. For example:

   - CLK_startup sets up the PRD register, enables the bit in the IMR for the timer selected in the CLK manager, and finally starts the timer. (This macro is only expanded if you enable the CLK manager in the Configuration Tool.)
   - PIP_startup calls the notifyWriter function for each created pipe object.
   - SWI_startup enables software interrupts.
   - HWI_startup enables hardware interrupts by clearing the INTM bit in the st1 register.

5) **Drop into the idle loop**. By calling IDL_loop the boot routine falls into the DSP/BIOS idle loop forever. At this point hardware and software interrupts can occur and preempt idle execution. Since the idle loop manages communication with the host, data transfer between the host and the target can now take place.

Chapter 3

Instrumentation

DSP/BIOS provides both explicit and implicit ways to perform real-time program analysis. These mechanisms are designed to have minimal impact on the application's real-time performance.

Topic **Page**

3.1 Real-Time Analysis

Real-time analysis is the analysis of data acquired during real-time operation of a system. The intent is to easily determine whether the system is operating within its design constraints, is meeting its performance targets, and has room for further development.

The traditional debugging method for sequential software is to execute the program until an error occurs. You then stop the execution, examine the program state, insert breakpoints, and reexecute the program to collect information. This kind of cyclic debugging is effective for non-real-time sequential software. However, cyclic debugging is rarely as effective in real-time systems because real-time systems require continuous operation, nondeterministic execution, and stringent timing constraints.

The DSP/BIOS instrumentation APIs and the DSP/BIOS plugins are designed to complement cyclic debugging tools to enable you to monitor real-time systems as they run. This real-time monitoring data lets you view the real-time system operation so that you can effectively debug and performance-tune the system.

3.2 Software vs. Hardware Instrumentation

Software monitoring consists of instrumentation code that is part of the target application. This code is executed at run time, and data about the events of interest is stored in the target system's memory. Thus, the instrumentation code uses both the computing power and memory of the target system.

The advantage of software instrumentation is that it is flexible and that no additional hardware is required. Unfortunately, because the instrumentation is part of the target application, performance and program behavior can be affected. Without using a hardware monitor, you face the problem of finding a balance between program perturbation and recording sufficient information. Limited instrumentation provides inadequate detail, but excessive instrumentation perturbs the measured system to an unacceptable degree.

DSP/BIOS provides a variety of mechanisms that allow you to precisely control the balance between intrusion and information gathered. In addition, the DSP/BIOS instrumentation operations all have fixed, short execution times. Since the overhead time is fixed, the effects of instrumentation are known in advance and can be factored out of measurements.

3.3 Instrumentation Performance Issues

When all implicit instrumentation is enabled, the CPU load increases less than 1 percent in a typical application. Several techniques have been used to minimize the impact of instrumentation on application performance:

- Instrumentation communication between the target and the host is performed in the background (IDL) thread, which has the lowest priority, so communicating instrumentation data does not affect the real-time behavior of the application.
- From the host you can control the rate at which the host polls the target. You can stop all host interaction with the target if you want to eliminate all unnecessary external interaction with the target.
- The target does not store Execution Graph or implicit statistics information unless tracing is enabled. You also have the ability to enable or disable the explicit instrumentation of the application by using the TRC module and one of the reserved trace masks (TRC_USER0 and TRC_USER1).
- Log and statistics data are always formatted on the host. The average value for an STS object and the CPU load are computed on the host. Computations needed to display the Execution Graph are performed on the host.
- LOG, STS, and TRC module operations are very fast and execute in constant time, as shown in the following list:
  - LOG_printf and LOG_event: approximately 30 instructions
  - STS_add: approximately 30 instructions
  - STS_delta: approximately 40 instructions
  - TRC_enable and TRC_disable: approximately four instructions
- Each STS object uses only eight words of data memory. This means that the host transfers only eight words to upload data from a statistics object.
- 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.
- You can specify the buffer size for LOG objects. The buffer size affects the program's data size and the time required to upload log data.
- For performance reasons, implicit hardware interrupt monitoring is disabled by default. When disabled, there is no effect on performance. When enabled, updating the data in statistics objects consumes between 20 and 30 instructions per interrupt for each interrupt monitored.

3.4 Instrumentation APIs

Effective instrumentation requires both operations that gather data and operations that control the gathering of data in response to program events. DSP/BIOS provides the following three API modules for data gathering:

- **LOG (Message Log Manager)**. Log objects capture information about events in real time. System events are captured in the system log. You can create additional logs using the Configuration Tool. Your program can add messages to any log.
- **STS (Statistics Manager)**. Statistics objects capture count, maximum, and total values for any variables in real time. Statistics about SWI (software interrupt), PRD (period), HWI (hardware interrupt), and PIP (pipe) objects can be captured automatically. In addition, your program can create statistics objects to capture other statistics.
- **HST (Host Input/Output Manager)**. The host channel objects described in Chapter 5, *Input/Output*, allow a program to send raw data streams to the host for analysis.

LOG and STS provide an efficient way to capture subsets of a real-time sequence of events that occur at high frequencies or a statistical summary of data values that vary rapidly. The rate at which these events occur or values change may be so high that it is either not possible to transfer the entire sequence to the host (due to bandwidth limitations) or the overhead of transferring this sequence to the host would interfere with program operation. Therefore, DSP/BIOS also provides an API module for controlling the data gathering mechanisms provided by the other modules:

- **TRC (Trace Manager)**. Controls which events and statistics are captured either in real time by the target program or interactively through the DSP/BIOS plugins.

Controlling data gathering is important because it allows you to limit the effects of instrumentation on program behavior, ensure that LOG and STS objects contain the necessary information, and start or stop recording of events and data values at run time.

3.4.1 Explicit vs. Implicit Instrumentation

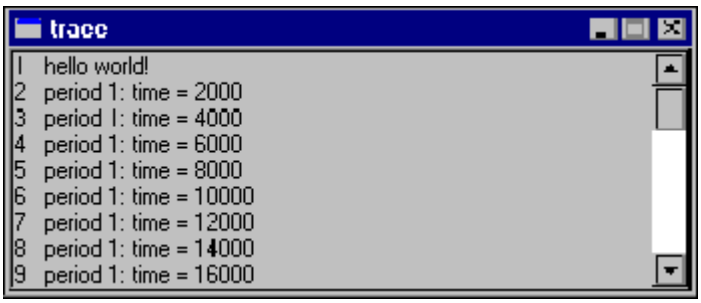
The instrumentation API operations are designed to be called explicitly by the application. The LOG module operations allow you to explicitly write messages to any log. The STS module operations allow you to store statistics about data variables or system performance. The TRC module allows you to enable or disable log and statistics tracing in response to a program event.

The LOG and STS APIs are also used internally by DSP/BIOS to collect information about program execution. These internal calls in DSP/BIOS routines provide implicit instrumentation support. As a result, even applications that do not contain any explicit calls to the DSP/BIOS instrumentation APIs can be monitored and analyzed using the DSP/BIOS plugins. For example, the execution of a software interrupt is recorded in a LOG object called LOG_system. In addition, worst-case ready-to-completion times for software interrupts and overall CPU load are accumulated in STS objects. The occurrence of a system tick can also be recorded in the Execution Graph. See section 3.4.4.2, *Control of Implicit Instrumentation*, page 3-12, for more information about what implicit instrumentation can be collected.

3.4.2 Message Log Manager (LOG Module)

This module manages LOG objects, which capture events in real time while the target program executes. You can use the Execution Graph, or create user-defined logs with the Configuration Tool.

User-defined logs contain any information your program stores in them using the LOG_event and LOG_printf operations. You can view messages in these logs in real time with the Message Log.



The Execution Graph can also be viewed as a graph of the activity for each program component.

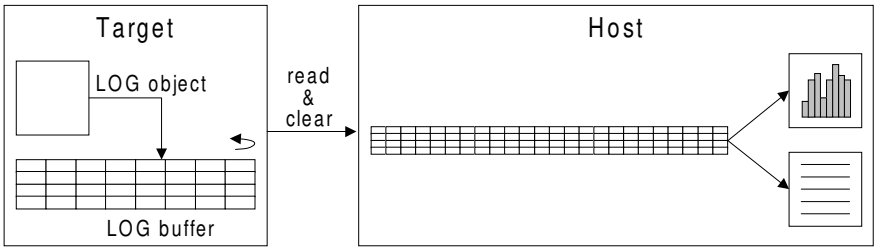
A log can be either fixed or circular. This distinction is valuable in applications that enable and disable logging programmatically (using the TRC module operations as described in section 3.4.4, *Trace Control Manager (TRC Module)*, page 3-11).

- **Fixed**. The log stores the first messages it receives and stops accepting messages when its message buffer is full. As a result, a fixed log stores the first events that occur since the log was enabled.
- **Circular**. The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.

You create LOG objects using the Configuration Tool, in which you assign properties such as the length and location of the message buffer.

You can specify the length of each message buffer in words. Individual messages use four words of storage in the log's buffer. The first word holds a sequence number. The remaining three words of the message structure hold event-dependent codes and data values supplied as parameters to operations such as LOG_event, which appends new events to a LOG object.

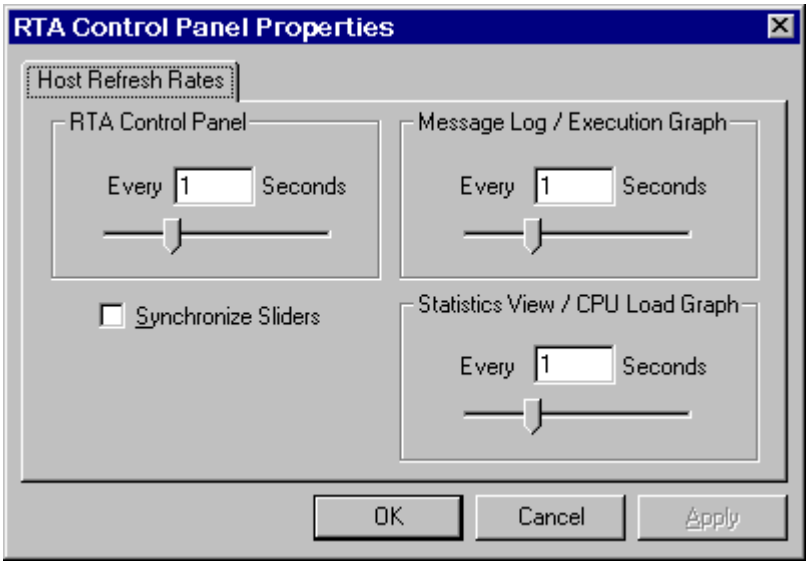
As shown in the following figure, LOG buffers are read from the target and stored in a much larger buffer on the host. Records are marked empty as they are copied up to the host.



LOG_printf uses the fourth word of the message structure for the offset or address of the format string (e.g., %d, %d). The host uses this format string and the two remaining words to format the data for display. This minimizes both the time and code space used on the target since the actual printf operation (and the code to perform the operation) are handled on the host.

LOG_event and LOG_printf both operate on logs atomically. This allows ISRs and other threads of different priorities to write to the same log without having to worry about synchronization.

Using the RTA Control Panel Property Page for each message log, you can control how frequently the host polls the target for information on a particular log. 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 for log information unless you right-click on a log window and choose Refresh Window from the pop-up menu. You can also use the pop-up menu to pause and resume polling for log information.



Log messages shown in a message log window are numbered (in the left column of the trace window) to indicate the order in which the events occurred. These numbers are an increasing sequence starting at 0. If your log never fills up, you can use a smaller log size. If a circular log is not long enough or you do not poll the log often enough, you may miss some log entries that are overwritten before they are polled. In this case, you see gaps in the log message numbers. You may want to add an additional sequence number to the log messages to make it clear whether log entries are being missed.

The online help in the Configuration Tool describes LOG objects and their parameters. See *LOG Module*, page 6–36, for reference information on the LOG module API calls.

3.4.3 Statistics Accumulator Manager (STS Module)

This module manages objects called statistics accumulators, which store key statistics while a program runs.

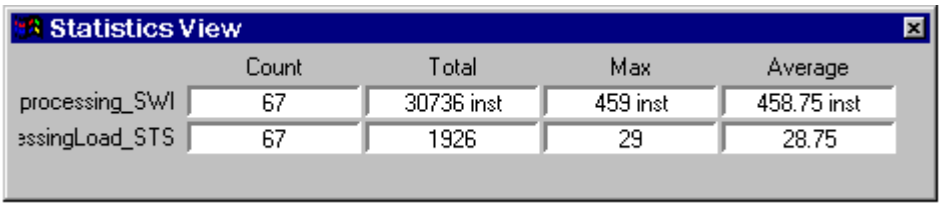
You create individual statistics accumulators using the Configuration Tool. Each STS object accumulates the following statistical information about an arbitrary 32-bit wide data series:

- **Count**. The number of values in an application-supplied data series
- **Total**. The arithmetic sum of the individual data values in this series
- **Maximum**. The largest value already encountered in this series
- **Average**. Using the count and total, the Statistics View plugin also calculates the average

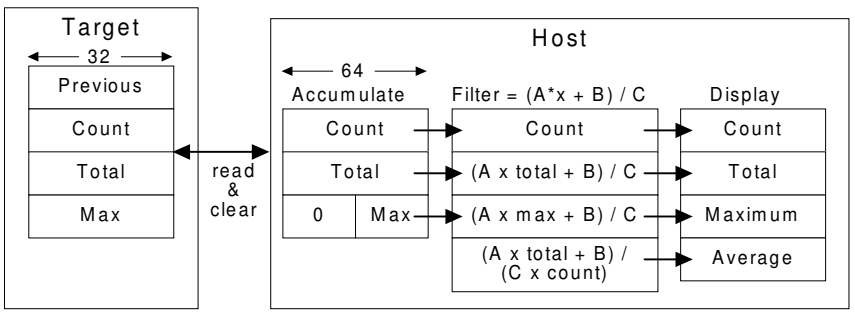
Calling the STS_add operation updates the statistics accumulator of the data series being studied. For example, you might study the pitch and gain in a software interrupt analysis algorithm or the expected and actual error in a closed-loop control algorithm.

DSP/BIOS statistics accumulators are also useful for tracking absolute CPU use of various routines during execution. By bracketing appropriate sections of the program with the STS_set and STS_delta operations, you can gather real-time performance statistics about different portions of the application.

You can view these statistics in real time with the Statistics View.



Although statistics are accumulated in 32-bit variables on the target, they are accumulated 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. The Statistics View may optionally filter the data arithmetically before displaying it.



By clearing the values on the target, the host allows the values displayed to be much larger without risking lost data due to values on the target wrapping around to 0. If polling of STS data is disabled or very infrequent there is a possibility that the STS data wraps around, resulting in incorrect information.

While the host clears the values on the target automatically, you can clear the 64-bit accumulators stored on the host by right-clicking on the STS Data window and choosing Clear from the shortcut menu.

The host read and clear operations are performed atomically to allow any thread to update any STS object reliably. For example, an HWI function can call STS_add on an STS object and no data is missing from any STS fields.

This instrumentation process provides minimal intrusion into the target program. A call to STS_add requires approximately 20 instructions and an STS object uses only eight words of data memory. Data filtering, formatting, and computation of the average is done on the host.

You can control the polling rate for statistics information with the Statistics View Property Page. If you set the polling rate to 0, the host does not poll the target for information about the STS objects unless you right-click on the Statistics View window and choose Refresh Window from the pop-up menu.

3.4.3.1 Statistics About Varying Values

STS objects can be used to accumulate statistical information about a time series of 32-bit data values.

For example, let P_i be the pitch detected by an algorithm on the i^{th} frame of audio data. An STS object can store summary information about the time series $\{P_i\}$. The following code fragment includes the current pitch value in the series of values tracked by the STS object:

```
pitch = `do pitch detection`
STS_add(&stsObj, pitch);
```

The Statistics View displays the number of values in the series, the maximum value, the total of all values in the series, and the average value.

3.4.3.2 Statistics About Time Periods

In any real-time system, there are important time periods. Since a period is the difference between successive time values, STS provides explicit support for these measurements.

For example, let T_i be the time taken by an algorithm to process the i^{th} frame of data. An STS object can store summary information about the time series $\{T_i\}$. The following code fragment illustrates the use of CLK_gethtime (high-resolution time), STS_set, and STS_delta to track statistical information about the time required to perform an algorithm:

```
STS_set(&stsObj, CLK_gethtime());
`do algorithm`
STS_delta(&stsObj, CLK_gethtime());
```

STS_set saves the value of CLK_gethtime as the previous value in the STS object. STS_delta subtracts this saved value from the value it is passed. The result is the difference between the time recorded before the algorithm started and after it was completed; i.e., the time it took to execute the algorithm (T_i). STS_delta then invokes STS_add and passes this result as the new value to be tracked.

The host can display the count of times the algorithm was performed, the maximum time to perform the algorithm, the total time performing the algorithm, and the average time.

The previous field is the fourth component of an STS object. It is provided to support statistical analysis of a data series that consist of value differences, rather than absolute values.

3.4.3.3 Statistics About Value Differences

Both STS_set and STS_delta update the previous field in an STS object. Depending on the call sequence, you can measure specific value differences or the value difference since the last STS update. The following example gathers information about a difference between specific values.

```
STS_set(&sts, targetValue);
   "processing"
STS_delta(&sts, currentValue);
   "processing"
STS_delta(&sts, currentValue);
   "processing"
STS_delta(&sts, currentValue);
```

The next 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);
```

The online help in the Configuration Tool describes statistics accumulators and their parameters. See *STS Module*, page 6–88, for reference information on the STS module API calls.

3.4.4 Trace Control Manager (TRC Module)

The TRC module allows an application to enable and disable the acquisition of analysis data in real time. For example, the target can use the TRC module to stop or start the acquisition of data when it discovers an anomaly in the application's behavior.

Control of data gathering is important because it allows you to limit the effects of instrumentation on program behavior, ensure that LOG and STS objects contain the necessary information, and start or stop recording of events and data values at run time.

For example, by enabling instrumentation when an event occurs, you can use a fixed log to store the first n events after you enable the log. By disabling tracing when an event occurs, you can use a circular log to store the last n events before you disable the log.

3.4.4.1 Control of Explicit Instrumentation

You can use the TRC module to control explicit instrumentation as shown in this code fragment:

```
if (TRC_query(TRC_USER0) == 0) {
`LOG or STS operation`
}
```

Note: TRC_query returns 0 if all trace types in the mask passed to it are enabled, and is not 0 if any trace types in the mask are disabled.

The overhead of this code fragment is just a few instruction cycles if the tested bit is not set. If an application can afford the extra program size required for the test and associated instrumentation calls, it is very practical to keep this code in the production application simplifying the development process and enabling field diagnostics. This is, in fact, the model used within DSP/BIOS itself.

3.4.4.2 Control of Implicit Instrumentation

The TRC module manages a set of trace bits that control the real-time capture of implicit instrumentation data through logs and statistics accumulators. For greater efficiency, the target does not store log or statistics information unless tracing is enabled. (You do not need to enable tracing for messages explicitly written with LOG_printf or LOG_event and statistics added with STS_add or STS_delta.)

The trace bits allow the target application to control when to start and stop gathering system information. This can be important when trying to capture information about a specific event or combination of events.

DSP/BIOS defines the following constants for referencing specific trace bits:

Constant	Tracing Enabled/Disabled	Default
TRC_LOGCLK	Logs low-resolution clock interrupts	off
TRC_LOGPRD	Logs system ticks and start of periodic functions	off
TRC_LOGSWI	Logs posting, start, and completion of software interrupt functions	off
TRC_STSHWI	Gathers statistics on monitored register values within HWIs	off
TRC_STSPIP	Counts the number of frames read from or written to data pipe	off
TRC_STSPRD	Gathers statistics on the number of ticks elapsed during execution of periodic functions	off
TRC_STSSWI	Gathers statistics on number of instruction cycles or time elapsed from post to completion of software interrupt	off
TRC_USER0 and TRC_USER1	Enables or disables 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	Simultaneously starts or stops gathering all enabled types of tracing. This bit must be set in order for any implicit instrumentation to be performed. 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 with the RTA Control Panel.	off
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

You can enable and disable these trace bits in the following ways:

- From the host, use the RTA Control Panel. This window allows you to adjust the balance between information gathering and time intrusion at run time. By disabling various implicit instrumentation types, you lose information but reduce overhead of processing.

  You can control the refresh rate for trace state information by right-clicking on the Property Page of the RTA Control Panel. If you set the refresh rate to 0, the host does not poll the target for trace state information unless you right-click on the RTA Control Panel and choose Refresh Window from the pop-up menu.

- From the target code, enable and disable trace bits using the TRC_enable and TRC_disable operations, respectively. For example, the following C code would disable tracing of log information for software interrupts and periodic functions:

  ```
  TRC_disable(TRC_LOGSWI | TRC_LOGPRD);
  ```

  For example, in an overnight run you might be looking for a specific circumstance. When it occurs, your program can perform the following statement to turn off all tracing so that the current instrumentation information is preserved:

  ```
  TRC_disable(TRC_GBLTARG);
  ```

Any changes made by the target program to the trace bits are reflected in the RTA Control Panel. For example, you could cause the target program to disable the tracing of information when an event occurs. On the host, you can simply wait for the global target enable check box to be cleared and then examine the log.

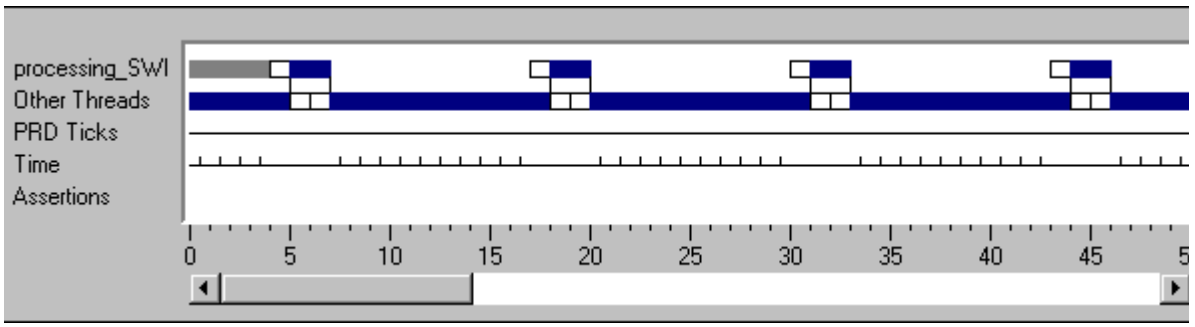
3.5 Implicit DSP/BIOS Instrumentation

The instrumentation needed to allow the DSP/BIOS plugins to display the Execution Graph, system statistics, and CPU load are all automatically built into a DSP/BIOS program to provide implicit instrumentation. You can enable different components of DSP/BIOS implicit instrumentation by using the RTA Control Panel plugin in Code Composer, as described in section 3.4.4.2, *Control of Implicit Instrumentation*, page 3-12.

DSP/BIOS instrumentation is efficient—when all implicit instrumentation is enabled, the CPU load increases less than one percent for a typical application. See section 3.3, *Instrumentation Performance Issues*, page 3-3, for details about instrumentation performance.

3.5.1 The Execution Graph

The Execution Graph is a special log used to store information about SWI, PRD, and CLK processing. You can enable or disable logging for each of these object types at run time using the TRC module API or the RTA Control Panel in the host. The Execution Graph window in the host shows the Execution Graph information as a graph of the activity of each object.



CLK and PRD events are shown to provide a measure of time intervals within the Execution Graph. Rather than timestamping each log event, which is expensive (because of the time required to get the timestamp and the extra log space required), the Execution Graph simply records CLK events along with other system events.

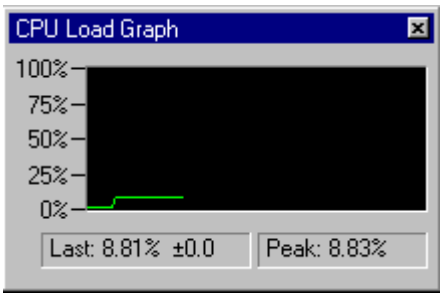
In addition to SWI, PRD, and CLK events, the Execution Graph shows additional information in the graphical display. Errors are indications that either a real-time deadline has been missed or an invalid state has been detected (either because the system log has been corrupted or the target has performed an illegal operation).

See section 4.9, *Using the Execution Graph to View Program Execution*, page 4-26, for details on how to interpret the Execution Graph information in relation to DSP/BIOS program execution.

3.5.2 The CPU Load

The CPU load is defined as the percentage of instruction cycles that the CPU spends doing application work; i.e., the percentage of the total time that the CPU is:

- ❏ Running ISRs, software interrupts, or periodic functions
- ❏ Performing I/O with the host
- ❏ Running any user routine



When the CPU is not doing any of these, it is considered idle, even if the CPU is not in a power-save or hardware-idle mode.

All CPU activity is divided into work time and idle time. To measure the CPU load over a time interval T, you need to know how much time during that interval was spent doing application work (t_w) and how much of it was idle time (t_i). From this you can calculate the CPU load as follows:

$$CPUload = \frac{t_w}{T} \times 100$$

Since the CPU is always either doing work or in idle it is represented as follows:

$$T = t_w + t_i$$

You can rewrite this equation:

$$CPUload = \frac{t_w}{t_w + t_i} \times 100$$

You can also express CPU load using instruction cycles rather than time intervals:

$$CPUload = \frac{c_w}{c_w + c_i} \times 100$$

In a DSP/BIOS application, the CPU is doing work when hardware interrupts are serviced, software interrupts and periodic functions are run, user functions are executed from the idle loop, HST channels are transferring data to the host, or real-time analysis information is being uploaded to the DSP/BIOS plugins. When the CPU is not performing any of those activities, it is going through the idle loop, executing the IDL_cpuLoad function, and

calling the other DSP/BIOS IDL objects. In other words, the CPU idle time in a DSP/BIOS application is the time that the CPU spends doing the following routine.

To measure the CPU load in a DSP/BIOS application over a time interval T, it is sufficient to know how much time was spent going through the loop, shown in the IDL_loop example below, and how much time was spent doing application work; i.e., doing any other activity not included in the example:

```
Idle_loop:
    Perform IDL_cpuLoad
   Perform all other IDL functions (these return without doing
        work)
    Goto IDL_loop
```

Over a period of time T, a CPU with M MIPS (million instructions per second) executes M x T instruction cycles. Of those instruction cycles, c_w are spent doing application work. The rest are spent executing the idle loop shown above. If the number of instruction cycles required to execute this loop once is I_1, the total number of instruction cycles spent executing the loop is N x I_1 where N is the number of times the loop is repeated over the period T. Hence you have total instruction cycles equals work instruction cycles plus idle instruction cycles.

$$MT = c_w + NI_1$$

From this expression you can rewrite c_w as:

$$c_w = MT - NI_1$$

Using previous equations, you can calculate the CPU load in a DSP/BIOS application as:

$$CPUload = \frac{c_w}{MT} \times 100 = \frac{MT - NI_1}{MT} \times 100 = \left(1 - \frac{NI_1}{MT}\right) \times 100$$

To calculate the CPU load you need to know I_1 and the value of N for a chosen time interval T, over which the CPU load is being measured.

The IDL_cpuLoad object in the DSP/BIOS idle loop updates an STS object, IDL_busyObj, that keeps track of the number of times the IDL_loop runs, and the time as kept by the DSP/BIOS low-resolution clock (see section 4.7, *Clock Manager (CLK Module)*, page 4-22). This information is used by the host to calculate the CPU load according to the equation above.

The host uploads the STS objects from the target at time intervals that you determine (the time interval between updates of the IDL_cpuLoad STS object, set in its Property Page). The information contained in IDL_busyObj is used to calculate the CPU load over the period of time between two uploads of IDL_busyObj. The IDL_busyObj count provides a measure of N (the

number of times the idle loop ran). The IDL_busyObj maximum is not used in CPU load calculation. The IDL_busyObj total provides the value T in units of the low-resolution clock.

To calculate the CPU load you still need to know I_1 (the number of instruction cycles spent in the idle loop). When the Auto calculate idle loop instruction count box is checked in the Idle Function Manager in the Configuration Tool, DSP/BIOS calculates I_1 at initialization from BIOS_init.

The host uses the values described for N, T, I_1, and the CPU MIPS to calculate the CPU load as follows:

$$\text{CPUload} = \left[1 - \frac{NI_1}{MT}\right]100$$

Since the CPU load is calculated over the STS polling rate period, the value displayed is the average CPU load during that time. As the polling period increases, it becomes more likely that short spikes in the CPU load are not shown on the graph.

Considering the definition of idle time and work time used to calculate the CPU load, it follows that a DSP/BIOS application that performs only the loop shown in the previous IDL_loop example displays 0% CPU load. Since each IDL function runs once in every idle loop cycle, adding IDL objects to such an application dramatically increases the measure of CPU load. For example, adding an IDL function that consumes as many cycles as the rest of the components in the IDL_loop example results in a CPU load display of 50%. This increase in the CPU load is real, since the time spent executing the new IDL user function is, by definition, work time. However, this increase in CPU load does not affect the availability of the CPU to higher priority threads such as software or hardware interrupts.

In some cases you may want to consider one or more user IDL routines as CPU idle time, rather than CPU work time. This changes the CPU idle time to the time the CPU spends doing the following routine:

```
Idle_loop:
    Perform IDL_cpuLoad
    Perform user IDL function(s)
    Perform all other IDL functions
    Goto IDL_loop
```

The CPU load can now be calculated in the same way as previously described. However, what is considered idle time has now changed, and you need a new instruction cycle count for the idle loop described above. This new value must be provided to the host so that it can calculate the CPU load. To do this, enter the new cycle count in the Idle Function Manager Properties in the Configuration Tool. The IDL_loop cycle count can be calculated using the Code Composer Profiler to benchmark one pass through the IDL_loop when there is no host I/O or real-time analysis information transfer to the host,

and the only routines executed are the IDL_cpuLoad function and any other user functions you want to include as idle time. (See the *TMS320C54x Code Composer Studio Tutorial* manual for a description on how to use the Profiler.)

3.5.3 CPU Load Accuracy

The accuracy of the CPU load value measured as described above is affected by the accuracy in the measurements of T, N, and I_1.

- To measure T you use the low resolution clock, so you can only know T with the accuracy of the resolution of this clock. If the measured time is T ticks, but the real time elapsed is ticks + ε, with $0 < \varepsilon < 1$, the error introduced is as follows:

  $$\begin{aligned}\text{Error} &= |\text{ actual load} - \text{measured load }| \\ &= \left(1 - \frac{NI_1}{M(T+\varepsilon)}\right) - \left(1 - \frac{NI_1}{MT}\right) \\ &= \frac{NI_1}{M}\left(\frac{1}{T} - \frac{1}{T+\varepsilon}\right) \\ &= \frac{NI_1}{MT} \times \frac{\varepsilon}{T+\varepsilon}\end{aligned}$$

  Since $(N \times I_1)/(M \times T)$ can be 1 at the most (when the CPU load is 0), the error is bounded:

  $$\begin{aligned}\text{Error} &\le \frac{\varepsilon}{T+\varepsilon} \qquad 0 < \varepsilon < 1 \\ &< \frac{1}{T}\end{aligned}$$

  In a typical application, the CPU load is measured at time intervals on the order of 1 second. The timer interrupt, which gives the resolution of the tick used to measure T, is triggered at time intervals on the order of 1 millisecond. Hence T is 1000 in low-resolution ticks. This results in a bounded error of less than 0.1% for the CPU load.

- To obtain a measurement of N, the host uses the integer value provided by the accumulated count in the IDL_busyObj STS object. However, the value of N could be overestimated or underestimated by as much as 1. The error introduced by this is:

$$\begin{aligned}
\text{Error} &= |\ \text{actual load} - \text{measured load}\ | \\
&= \left(1 - \frac{NI_1}{MT}\right) - \left(1 - \frac{(N+\varepsilon)I_1}{MT}\right) \\
&= (N + \varepsilon - N)\left(\frac{I_1}{MT}\right) \\
&= \frac{\varepsilon \times I_1}{MT} \qquad 0 < \varepsilon < 1 \\
&< \frac{I_1}{MT}
\end{aligned}$$

  For a measurement period on the order of 1 second and a typical idle loop cycle count on the order of 200 instruction cycles, the additional error due to the approximation of N is far below the 0.1% error due to the resolution in the measure of T.

- Finally, there may also be an error in the calculation of I_1, the idle cycle instruction count, that affects the CPU load accuracy. This error depends on how I_1 is measured. When I_1 is autocalculated, BIOS_init uses the on-chip timer with CLKSRC = CPU/4 and the timer counter register value to estimate the idle loop instruction cycles count. Since the timer counter register increases at a rate of CPU/4 (one increase for every four CPU cycles), the resolution that can be achieved when measuring instruction cycles by reading the timer counter is worse than a single instruction cycle. This causes an uncertainty in the value estimated for I_1 that introduces a corresponding error in the value of the CPU load. This error is:

$$\text{Error} = \Delta I_1\left(\frac{N}{MT}\right) \qquad \Delta I_1 = \text{error in the measured value of } I_1$$

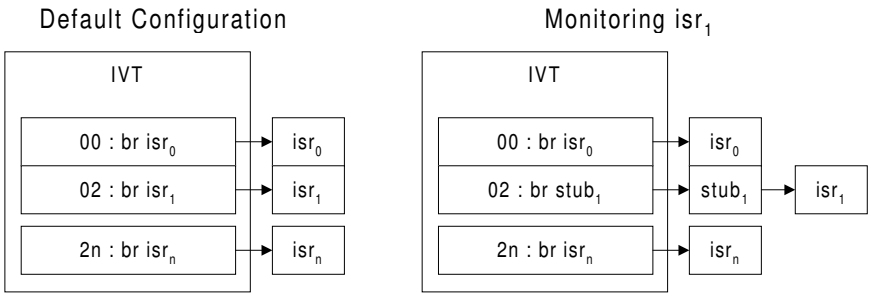
This error is the greatest when N is large; i.e., for CPU loads close to 0%. For this value, the error equals $\Delta I_1/I_1$; i.e., the maximum error in the CPU load calculation equals the percentage that ΔI_1 represents in total idle cycle count I_1. ΔI_1 is six instruction cycles when BIOS_init auto-calculates I_1 using the on-chip timer counter. Hence, the maximum CPU load error for a typical application with I_1 = 200 instruction cycles is 2.8% for a CPU load of 0.1%, and it decreases to less than 0.1% error for a CPU load of 99%, as shown in the following table.

CPU Load	CPU Load Error due to ΔI1
99%	<0.1%
80%	0.6%
75%	0.7%
50%	1.4%
25%	2.1%
10%	2.5%
1%	2.8%
0.1%	2.8%

The host calculates all the error components for each CPU load reported and displays the total accuracy for the CPU load value currently reported in the CPU load display. However, when you enter a value for I_1 manually, the last component of the error (due to I_1) is not added to the total error displayed.

3.5.4 Hardware Interrupt Count and Maximum Stack Depth

You can track the number of times an individual HWI function has been triggered by using the Configuration Tool to set the monitor parameter for an HWI object to monitor the stack pointer. An STS object is automatically created for each hardware ISR that is monitored.



For hardware interrupts that are not monitored, there is no overhead—control passes directly to the HWI function. For interrupts that are monitored, control first passes to a stub function generated by the Configuration Tool. This function reads the selected data location, passes the value to the selected STS operation, and finally branches to the HWI function.

The enable HWI accumulators check box in the RTA Control Panel must be selected in order for HWI function monitoring to take place. If this type of tracing is not enabled, the stub function branches to the HWI function without updating the STS object.

The number of times an interrupt is triggered is recorded in the Count field of the STS object. When the stack pointer is monitored, the maximum value reflects the maximum position of the top of the application stack when the interrupt occurs; this can be useful for determining the application stack size needed by an application. To determine the maximum depth of the stack, follow these steps:

1) Using the Configuration Tool right-click on the HWI object and select Properties, and change the monitor field to Stack Pointer. Leave the default setting of unsigned for true and STS_add(-*addr) for operation.

2) Link your program and use the nmti program described in Chapter 7, *Utility Programs*, page 7–3, to find the address of the base of the application stack. Or, you can find the address of the base of the application stack in Code Composer by using a Memory window or the map file to find the address referenced by the GBL_stackbeg symbol. (The GBL_stackend symbol references the top of the stack.)

3) Run your program and view the STS object that monitors the stack pointer for this HWI function in the Statistics View window.

4) Subtract the address of the base of the application stack from the maximum value of the stack pointer to find the maximum depth of the stack.Subtract the minimum value of the stack pointer (maximum field in the STS object) from the end of the application stack to find the maximum depth of the stack.

3.5.5 Monitoring Variables

In addition to counting hardware interrupt occurrences and monitoring the stack pointer, you can monitor any register or data value each time a hardware interrupt is triggered.

This implicit instrumentation can be enabled for any HWI object. Such monitoring is not enabled by default; the performance of your interrupt processing is not affected unless you enable this type of instrumentation in the Configuration Tool. The statistics object is updated each time hardware interrupt processing begins. Updating such a statistics object consumes between 20 and 30 instructions per interrupt for each interrupt monitored.

To enable implicit HWI instrumentation:

1) Open the properties window for any HWI object and choose a register to monitor in the monitor field.

   You can monitor any of the following values. When you choose a register or data value to monitor, the Configuration Tool automatically creates an STS object which stores statistics for any one of these values:

Data Value (type in addr field)	ag	ar3	bg	ifr	st0
	ah	ar4	bh	imr	st1
	al	ar5	bk	pmst	treg
Stack Pointer	ar0	ar6	bl	rea	tim
Top of SW Stack	ar1	ar7	brc	rsa	trn
	ar2				

2) Set the operation parameter to the STS operation you want to perform on this value.

You can perform one of the following operations on the value stored in the data value or register you select. For all these operations, the count stores a count of the number of times this hardware interrupt has been executed. The max and total values are stored in the STS object on the target. The average is computed on the host.

STS Operation	Result
STS_add(*addr)	Stores maximum and total for the data value or register value
STS_delta(*addr)	Compares the data value or register value to the prev property of the STS object (or a value set consistently with STS_set) and stores the maximum and total differences.
STS_add(-*addr)	Negates the data value or register value and stores the maximum and total. As a result, the value stored as the maximum is the negated minimum value. The total and average are the negated total and average values.
STS_delta(-*addr)	Negates the data value or register value and compares the data value or register value to the prev property of the STS object (or a value set programmatically with STS_set). Stores the maximum and total differences. As a result, the value stored as the maximum is the negated minimum difference.
STS_add(|*addr|)	Takes the absolute value of the data value or register value and stores the maximum and total. As a result, the value stored as the maximum is the largest negative or positive value. The average is the average absolute value.
STS_delta(|*addr|)	Compares the absolute value of the register or data value to the prev property of the STS object (or a value set programmatically with STS_set). Stores the maximum and total differences. As a result, the value stored as the maximum is the largest negative or positive difference and the average is the average variation from the specified value.

3) You may also set the properties of the corresponding STS object to filter the values of this STS object on the host.

For example, you might want to watch the top of the software stack to see whether the application is exceeding the allocated stack size. The top of the software stack is initialized to 0xBEEF when the program is loaded. If this value ever changes, the application has either exceeded the allocated stack or some error has caused the application to overwrite the application's stack.

One way to watch for this condition is to follow these steps:

1) In the Configuration Tool, enable implicit instrumentation on any regularly occurring HWI function. Right-click on the HWI object, select Properties, and change the monitor field to Top of SW Stack with STS_delta(*addr) as the operation.

2) Set the prev property of the corresponding STS object to 0xBEEF.

3) Load your program in Code Composer and use the Statistics View to view the STS object that monitors the stack pointer for this HWI function.

4) Run your program. Any change to the value at the top of the stack is seen as a non-zero total (or maximum) in the corresponding STS object.

3.5.6 Interrupt Latency

Interrupt latency is the maximum time between the triggering of an interrupt and when the first instruction of the ISR executes. You can measure interrupt latency for the timer interrupt by following these steps:

1) Configure the HWI_TINT object to monitor the tim register.

2) Set the operation parameter to STS_add(-*addr).

3) Set the host operation parameter of the HWI_TINT_STS object to A*x + B. Set A to 1 and B to the value of the PRD Register (shown in the global CLK properties list).

The STS object HWI_TINT_STS then displays the maximum time (in instruction cycles) between when the timer interrupt was triggered and when the Timer Counter Register was able to be read. This is the interrupt latency experienced by the timer interrupt. The interrupt latency in the system is at least as large as this value. You can follow the same steps with a different HWI object to measure interrupt latency for the corresponding interrupt.

3.6 Instrumentation for Field Testing

The embedded DSP/BIOS run-time library and DSP/BIOS plugins support a new generation of testing and diagnostic tools that interact with programs running on production systems. Since DSP/BIOS instrumentation is so efficient, your production program can retain explicit instrumentation for use with manufacturing test and field diagnostic tools, which can be designed to interact with both implicit and explicit instrumentation.

3.7 Real-Time Data Exchange

Real-Time Data Exchange (RTDX) provides real-time, continuous visibility into the way DSP applications operate in the real world. RTDX allows system developers to transfer data between a host computer and DSP devices without interfering with the target application. The data can be analyzed and visualized on the host using any OLE automation client. This shortens development time by giving you a realistic representation of the way your system actually operates.

RTDX consists of both target and host components. A small RTDX software library runs on the target DSP. The DSP application makes function calls to this library's API in order to pass data to or from it. This library makes use of a scan-based emulator to move data to or from the host platform via a JTAG interface. Data transfer to the host occurs in real time while the DSP application is running.

On the host platform, an RTDX host library operates in conjunction with Code Composer Studio. Displays and analysis tools communicate with RTDX via an easy-to-use COM API to obtain the target data and/or to send data to the DSP application. Designers may use their choice of standard software display packages, including:

- ❏ National Instruments' LabVIEW
- ❏ Quinn-Curtis' Real-Time Graphics Tools
- ❏ Microsoft Excel

Alternatively, you can develop your own Visual Basic or Visual C++ applications. Instead of focusing on obtaining the data, you can concentrate on designing the display to visualize the data in the most meaningful way.

3.7.1 RTDX Applications

RTDX is well suited for a variety of control, servo, and audio applications. For example, wireless telecommunications manufacturers can capture the outputs of their vocoder algorithms to check the implementations of speech applications.

Embedded control systems also benefit from RTDX. Hard disk drive designers can test their applications without crashing the drive with improper signals to the servo-motor. Engine control designers can analyze changing factors (like heat and environmental conditions) while the control application is running.

For all of these applications, you can select visualization tools that display information in a way that is most meaningful to you. Future TI DSPs will enable RTDX bandwidth increases, providing greater system visibility to an even larger number of applications.

3.7.2 RTDX Usage

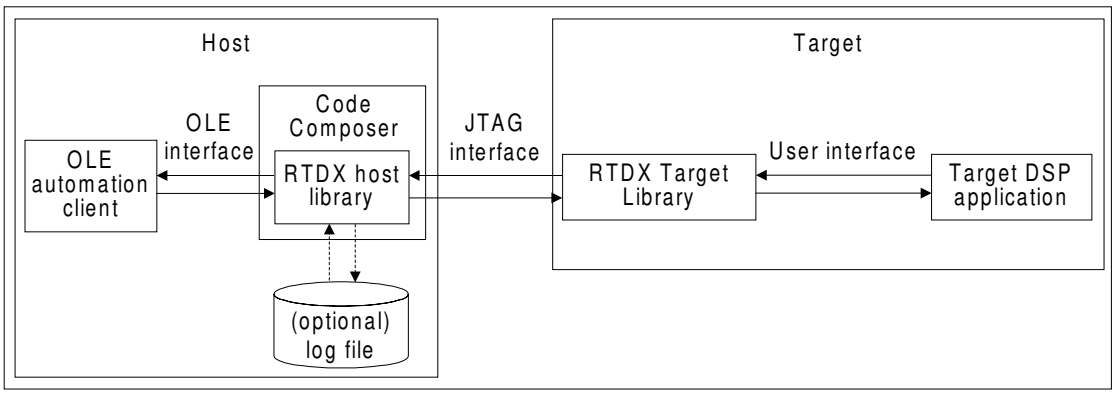
RTDX can be used within DSP/BIOS or, alternatively, without DSP/BIOS. The examples presented throughout the online help are written without DSP/BIOS.

RTDX is available with the PC-hosted Code Composer running Windows 95, Windows 98, or Windows NT version 4.0. RTDX is not currently available with the Code Composer Simulator.

This document assumes that the reader is familiar with C, Visual Basic or Visual C++, and OLE/ActiveX programming.

3.7.3 RTDX Flow of Data

Code Composer controls the flow of data between the host (PC) and the target (TI processor).



3.7.3.1 Target to Host Data Flow

To record data on the target, you must declare an output channel and write data to it using routines defined in the user interface. This data is immediately recorded into an RTDX target buffer defined in the RTDX target library. The data in the buffer is then sent to the host via the JTAG interface.

The RTDX host library receives this data from the JTAG interface and records it. The host records the data into either a memory buffer or to an RTDX log file (depending on the RTDX host recording mode specified).

The recorded data can be retrieved by any host application that is an OLE automation client. Some typical examples of OLE-capable host applications are:

- ❏ Visual Basic applications
- ❏ Visual C++ applications
- ❏ Lab View
- ❏ Microsoft Excel

Typically, an RTDX OLE automation client is a display that allows you to visualize the data in a meaningful way.

3.7.3.2 Host to Target Data Flow

For the target to receive data from the host, you must first declare an input channel and request data from it using routines defined in the user interface. The request for data is recorded into the RTDX target buffer and sent to the host via the JTAG interface.

An OLE automation client can send data to the target using the OLE Interface. All data to be sent to the target is written to a memory buffer within the RTDX host library. When the RTDX host library receives a read request from the target application, the data in the host buffer is sent to the target via the JTAG interface. The data is written to the requested location on the target in real time. The host notifies the RTDX target library when the operation is complete.

3.7.3.3 RTDX Target Library User Interface

The user interface provides the safest method of exchanging data between a target application and the RTDX host library.

The data types and functions defined in the user interface:

- ❏ Enable a target application to send data to the RTDX host library
- ❏ Enable a target application to request data from the RTDX host library
- ❏ Provide data buffering on the target. A copy of your data is stored in a target buffer prior to being sent to the host. This action helps ensure the integrity of the data and minimizes real-time interference.
- ❏ Provide interrupt safety. You can call the routines defined in the user interface from within interrupt handlers.
- ❏ Ensures correct utilization of the communication mechanism. It is a requirement that only one datum at a time can be exchanged between the host and target using the JTAG interface. The routines defined in the user interface handle the timing of calls into the lower-level interfaces.

3.7.3.4 RTDX Host OLE Interface

The OLE interface describes the methods that enable an OLE automation client to communicate with the RTDX host library.

The functions defined in the OLE interface:

- ❏ Enable an OLE automation client to access the data that was recorded in an RTDX log file or is being buffered by the RTDX Host Library
- ❏ Enable an OLE automation client to send data to the target via the RTDX host library

3.7.4 RTDX Modes

The RTDX host library provides the following modes of receiving data from a target application.

- ❏ **Noncontinuous**. In noncontinuous mode, data is written to a log file on the host.

  Noncontinuous mode should be used when you want to capture a finite amount of data and record it in a log file.
- ❏ **Continuous**. In continuous mode, the data is simply buffered by the RTDX host library; it is not written to a log file.

  Continuous mode should be used when you want to continuously obtain and display the data from a DSP application, and you don't need to store the data in a log file.

Note: To drain the buffer(s) and allow data to continuously flow up from the target, the OLE automation client must read from each target output channel on a continual basis. Failure to comply with this constraint may cause data flow from the target to cease, thus reducing the data rate, and possibly resulting in channels being unable to obtain data. In addition, the OLE automation client should open all target output channels on startup to avoid data loss to any of the channels.

3.7.5 Special Considerations When Writing Assembly Code

The RTDX functionality in the user library interface can be accessed by a target application written in assembly code.

See the Texas Instruments C compiler documentation for information about the C calling conventions, run-time environment, and runtime-support functions.

3.7.6 Target Buffer Size

The RTDX target buffer is used to temporarily store data that is waiting to be transferred to the host. You may want to reduce the size of the buffer if you are transferring only a small amount of data or you may need to increase the size of the buffer if you are transferring blocks of data larger than the default buffer size.

Using the Configuration Tool you can change the RTDX buffer size by right-clicking on the RTDX module and selecting Properties.

3.7.7 Sending Data From Target to Host or Host to Target

The user library interface provides the data types and functions for:

- Sending data from the target to the host
- Sending data from the host to the target

The following data types and functions are defined in the header file rtdx.h. They are available via DSP/BIOS or standalone.

- Declaration Macros
  - RTDX_CreateInputChannel
  - RTDX_CreateOutputChannel
- Functions
  - RTDX_channelBusy
  - RTDX_disableInput
  - RTDX_disableOutput
  - RTDX_enableOutput
  - RTDX_enableInput
  - RTDX_read
  - RTDX_readNB
  - RTDX_sizeofInput
  - RTDX_write
- Macros
  - RTDX_isInputEnabled
  - RTDX_isOutputEnabled

  See Chapter 6, *API Functions*, for detailed descriptions of all RTDX functions.

Chapter 4

Program Execution

This chapter describes the types of functions that make up a DSP/BIOS application and their behavior and priorities during program execution.

Topic **Page**

4.1 Program Components

There are three major types of threads in a DSP/BIOS program:

- **Background thread**. Has the lowest priority in a DSP/BIOS application and executes the idle loop (IDL). After main() returns, a DSP/BIOS application calls the startup routine for each DSP/BIOS module and then falls into the idle loop. The idle loop is a continuous loop that calls all functions for the objects in the IDL module. Each function must wait for all others to finish executing before it is called again. The idle loop runs continuously except when it is preempted by higher-priority threads. Only functions that do not have hard deadlines should be executed in the idle loop.
- **Software interrupts (SWIs)**. Patterned after hardware ISRs. While ISRs are triggered by a hardware interrupt, software interrupts are triggered by calling SWI functions from the program. Software interrupts provide additional priority levels between hardware interrupts and the background thread. SWIs handle tasks subject to time constraints that preclude them from being run from the idle loop, but whose deadlines are not as severe as those of hardware ISRs. Software interrupts should be used to schedule events with deadlines of 100 microseconds or more. SWIs allow HWIs to defer less critical processing to a lower-priority thread, minimizing the time the CPU spends inside an ISR, where other HWIs may be disabled.
- **Hardware interrupts (HWIs)**. Triggered in response to external asynchronous events that occur in the DSP environment. An HWI function (also called an interrupt service routine or ISR) is executed after a hardware interrupt is triggered, to perform a critical task that is subject to a hard deadline. HWI functions are the threads with the highest priority in a DSP/BIOS application. HWIs should be used for application tasks that may need to run at frequencies approaching 200 kHz, and that need to be completed within deadlines of 2 to 100 microseconds.

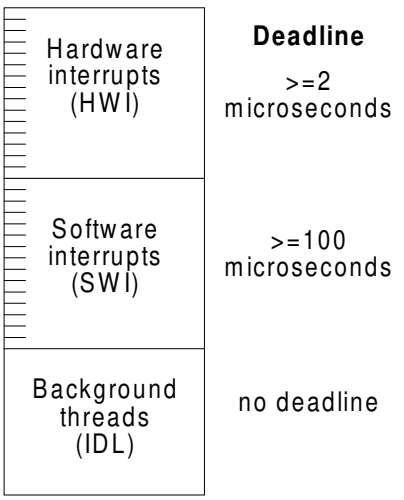
There are several other kinds of functions that can be performed in a DSP/BIOS program:

- **Clock (CLK) functions**. Triggered regularly at the rate of the on-chip timer interrupt. By default, these functions are triggered by the HWI_TINT hardware interrupt and are performed as HWI functions.
- **Periodic (PRD) functions**. Performed based on a multiple of either the on-chip timer interrupt or some other regular occurrence. Periodic functions are a special type of software interrupt.
- **Data notification functions**. Performed when you use pipes (PIP) or host channels (HST) to transfer data. The functions are triggered when a frame of data is read or written to notify the writer or reader that a frame is free or data is available. These functions are performed as part of the context of the function that called PIP_alloc, PIP_get, PIP_free, or PIP_put.

4.2 Choosing Which Types of Threads to Use

The type of thread and the priority level that you choose for each routine in an application program has an impact on whether the program tasks are scheduled on time and executed correctly. The Configuration Tool makes it easy to change a routine from one thread type to another.

Here are some rules for deciding which type of object to use for each task to be performed by a program:



- **HWI**. Perform only critical processing within a hardware interrupt service routine. Your HWI function should post a software interrupt to perform any lower-priority processing. If this lower-priority processing is still high-priority, give the software interrupt a high priority relative to other software interrupts. This allows other hardware interrupts to occur. HWIs can run at frequencies approaching 200 kHz.
- **SWI**. Use software interrupts to schedule events with deadlines of 100 microseconds or more. Using an SWI thread instead of an HWI thread in this case minimizes the length of time interrupts are disabled (interrupt latency). HWIs may defer less-critical processing to an SWI.
- **IDL**. Create background functions to perform noncritical housekeeping tasks when no other processing is necessary. IDL functions do not typically have hard deadlines; instead they run whenever the system has unused processor time.
- **CLK**. Use CLK functions when you want a function to be triggered directly by a timer interrupt. These functions are run as HWI functions and should take minimal processing time. The default CLK object, PRD_clock, causes a tick for the periodic functions. You can add additional CLK objects to be run at the same rate. However, you should minimize the time required to perform all CLK functions because they all run as HWI functions.
- **PRD**. Use PRD functions when you want a function to run at a rate based on a multiple of either the on-chip timer's low-resolution rate or by some event (such as an external interrupt). These functions run as SWI functions.

- **PRD vs. SWI**. All PRD functions are run at the same SWI priority, so one PRD function cannot preempt another. However, PRD functions can post lower-priority software interrupts to take care of lengthy processing routines. This ensures that the PRD_swi software interrupt can preempt those routines when the next system tick occurs and PRD_swi is posted again.

All DSP/BIOS threads run to completion. While some threads may be preempted, they cannot be suspended to wait for another event. Therefore, all input needed by a thread's function should be ready when the program posts the thread. The flexible mailbox structure used by SWI functions provides an efficient way to determine when a software interrupt is ready to run.

4.3 The Idle Loop

The idle loop is the background thread of DSP/BIOS, which runs continuously when no hardware interrupt service routines or software interrupts are running. Any hardware or software interrupt can preempt the idle loop at any point.

The IDL manager in the Configuration Tool allows you to insert functions that execute within the idle loop. The idle loop runs the IDL functions that you configured with the Configuration Tool. IDL_loop() calls the functions associated with each one of the IDL objects one at a time, and then starts over again in a continuous loop. The functions are called in the same order in which they were created in the Configuration Tool. Therefore, an IDL function must run to completion before the next IDL function can start running. When the last idle function has completed, the idle loop starts the first IDL function again. Idle loop functions are often used to poll non-real-time devices that do not (or cannot) generate interrupts, monitor system status, or perform other background activities.

The idle loop is the thread with lowest priority in a DSP/BIOS application. The idle loop functions run only when no other hardware or software interrupts need to run. Communication between the target and the DSP/BIOS plugins is performed within the background idle loop. This ensures that the DSP/BIOS plugins do not interfere with the program's tasks. If the target CPU is too busy to perform background tasks, the DSP/BIOS plugins stop receiving information from the target until the CPU is available.

By default, there are three DSP/BIOS IDL objects in the idle loop:

- LNK_dataPump manages transfer of real-time analysis data (e.g., LOG and STS data), and HST channel data between the target DSP and the host. Different variants of LNK_dataPump support different target/host links; e.g., JTAG (via RTDX), shared memory, etc.
- RTA_dispatcher is a real-time analysis server on the target that accepts commands from DSP/BIOS plugins, gathers instrumentation information from the target, and uploads it at run time. RTA_dispatcher sits at the end of two dedicated HST channels; its commands/responses are routed from/to the host via LNK_dataPump.
- IDL_cpuLoad uses an STS object (IDL_busyObj) to calculate the target load. The contents of this object are uploaded to the DSP/BIOS plugins through RTA_dispatcher to display the CPU load.

4.4 Software Interrupts

Software interrupts are patterned after hardware interrupt service routines. Software interrupts are triggered programmatically, through a call to a DSP/BIOS API such as SWI_post. Software interrupts provide a range of threads that have intermediate priority between HWI functions and the background idle loop.

These threads are suitable to handle application tasks that recur with slower rates or are subject to less severe real-time deadlines than those of hardware interrupts.

The DSP/BIOS APIs that can trigger or post a software interrupt are:

- SWI_andn
- SWI_dec
- SWI_inc
- SWI_or
- SWI_post

The SWI manager controls the execution of all software interrupts. When one of the APIs above is called by the application code, the SWI manager schedules the function corresponding to the software interrupt for execution. To handle all software interrupts in an application, the SWI manager uses SWI objects. To add a new software interrupt to an application, create a new SWI object from the SWI manager in the Configuration Tool. From the Property Page of each SWI object, you can set the function associated with each software interrupt that runs when the corresponding SWI object is triggered by the application. The Configuration Tool also allows you to enter two arguments for each SWI function. From the Property Page of the SWI manager, you can determine from which memory segment SWI objects are allocated. SWI objects are accessed by the SWI manager when software interrupts are posted and scheduled for execution.

The online help in the Configuration Tool describes SWI objects and their parameters. See *SWI Module*, page 6–99, for reference information on the SWI module API calls.

4.4.1 Setting Software Interrupt Priorities in the Configuration Tool

There are different priority levels among software interrupts. You can create as many software interrupts as your memory constraints allow for each priority level. You can choose a higher priority for a software interrupt that handles a thread with a shorter real-time deadline, and a lower priority for a software interrupt that handles a thread with a less critical execution deadline.

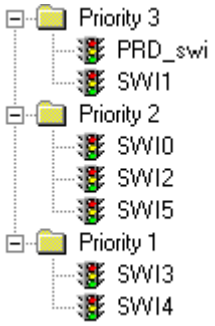
Notes

- ❏ You can create up to 15 priority levels. See section 4.6.3, *Software Interrupt Priorities and Application Stack Size*, page 4-20, for stack size restrictions.
- ❏ You cannot sort software interrupts within a single priority level.
- ❏ Priority levels automatically disappear if you drag all the software interrupts out of a priority level.
- ❏ The Property window for an SWI object shows its numeric priority level (from 1 to 15; 15 is the highest level). You cannot set a numeric priority in this window.

To set software interrupt priorities with the Configuration Tool, follow these steps:

1) In the Configuration Tool, highlight the Software Interrupt Manager.

   Notice the SWI objects in the right half of the window. If you have not added priority levels, all SWI objects have the same priority level. (If you do not see a list of SWI objects in the right half of the window, choose View→Ordered collection view.)



2) To add a priority level, drag a software interrupt to the bottom or top of the list or between two existing priority levels.
3) Drag the highest priority software interrupts up to the highest numbered level. Drag any lower priority software interrupts down to a lower numbered level.
4) Continue adding levels and sorting software interrupts.

4.4.2 Execution of Software Interrupts

Software interrupts can be scheduled for execution with a call to SWI_andn, SWI_dec, SWI_inc, SWI_or, and SWI_post. These calls can be used virtually anywhere in the program—interrupt service routines, periodic functions, idle functions, or other software interrupt functions.

When an SWI object is posted, the SWI manager adds it to a list of posted software interrupts that are pending execution. Then the SWI manager checks whether software interrupts are currently enabled. If they are not, as is the case inside an HWI function, the SWI manager returns control to the current thread.

If software interrupts are enabled, the SWI manager checks the priority of the posted SWI object against the priority of the thread that is currently running. If the thread currently running is the background idle loop or a lower priority SWI, the SWI manager removes the SWI from the list of posted SWI objects and switches the CPU control from the current thread to start execution of the posted SWI function.

If the thread currently running is an SWI of the same or higher priority, the SWI manager returns control to the current thread, and the posted SWI function runs after all other SWIs of higher priority or the same priority that were previously posted finish execution.

Note: When an SWI starts executing it must run to completion without blocking.

SWI functions can be preempted by threads of higher priority (such as an HWI or an SWI of higher priority). However, SWI functions cannot block. You cannot suspend a software interrupt while it waits for something—like a device—to be ready.

If an SWI is posted multiple times before the SWI manager has removed it from the posted SWI list, its SWI function executes only once, much like an ISR is executed only once if the hardware interrupt is triggered multiple times before the CPU clears the corresponding interrupt flag bit in the interrupt flag register. (See section 4.4.3, *Using an SWI Object's Mailbox*, page 4-8, for more information on how to handle SWIs that are posted multiple times before they are scheduled for execution.)

4.4.3 Using an SWI Object's Mailbox

Each SWI object has a 32-bit mailbox, which is used either to determine whether to post the software interrupt or as a value that can be evaluated within the SWI function.

SWI_post, SWI_or, and SWI_inc post an SWI object unconditionally:

- ❏ SWI_post does not modify the value of the SWI object mailbox when it is used to post a software interrupt.
- ❏ SWI_or sets the bits in the mailbox determined by a mask that is passed as a parameter, and then posts the software interrupt.
- ❏ SWI_inc increases the SWI's mailbox value by one before posting the SWI object.

SWI_andn and SWI_dec post the SWI object only if the value of its mailbox becomes 0:

- SWI_andn clears the bits in the mailbox determined by a mask passed as a parameter.
- SWI_dec decreases the value of the mailbox by one.

The following table summarizes the differences between these functions:

	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	

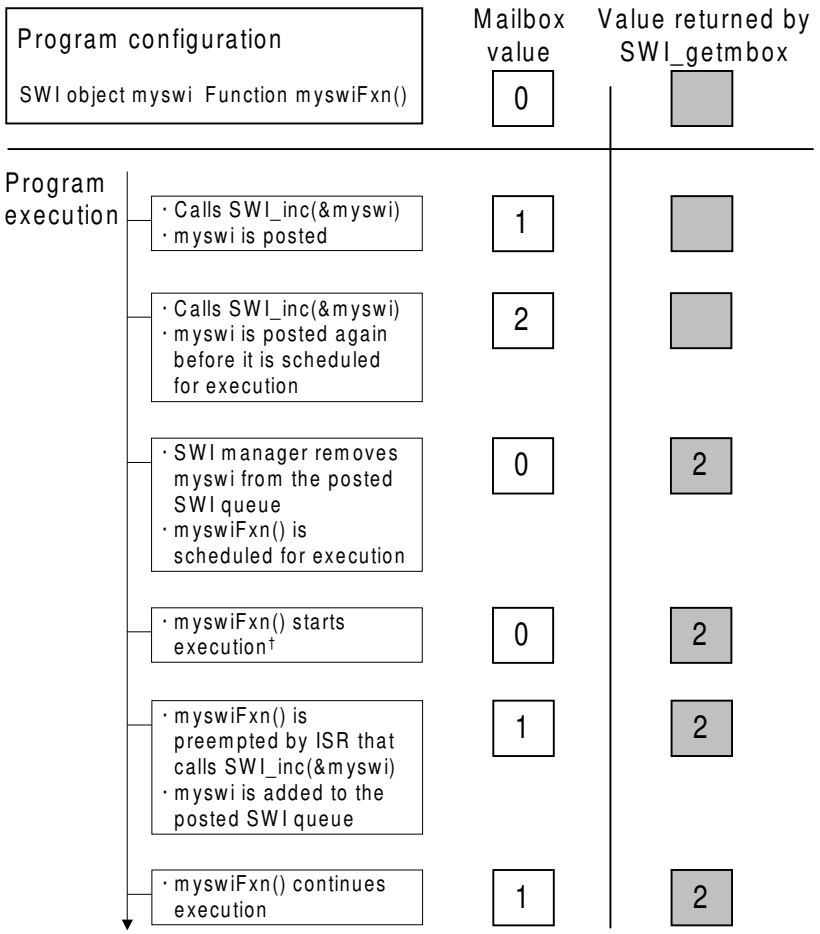
The SWI mailbox allows you to have a tighter control over the conditions that should cause an SWI function to be posted or the number of times the SWI function should be executed once the software interrupt is posted and scheduled for execution.

To access the value of its mailbox, an SWI function can call SWI_getmbox. SWI_getmbox can be called only from the SWI's object function. The value returned by SWI_getmbox is the value of the mailbox before the SWI object was removed from the posted SWI queue and the SWI function was scheduled for execution. When the SWI manager removes a pending SWI object from the posted objects queue, its mailbox is reset to its initial value. The initial value of the mailbox is set from the Property Page when the SWI object is created with the Configuration Tool. If while the SWI function is executing it is posted again, its mailbox is updated accordingly. However, this does not affect the value returned by SWI_getmbox while the SWI functions execute. That is, the mailbox value that SWI_getmbox returns is the latched mailbox value when the software interrupt was removed from the list of pending SWIs. The SWI's mailbox however, is immediately reset after the SWI is removed from the list of pending SWIs and scheduled for execution. This gives the application the ability to keep updating the value of the SWI mailbox if a new posting occurs, even if the SWI function has not finished its execution.

For example, if an SWI object is posted multiple times before it is removed from the queue of posted SWIs, the SWI manager schedules its function to execute only once. However, if an SWI function must always run multiple times when the SWI object is posted multiple times, SWI_inc should be used to post the SWI. When an SWI has been posted using SWI_inc, once the SWI

manager calls the corresponding SWI function for execution, the SWI function can access the SWI object mailbox to know how many times it was posted before it was scheduled to run, and proceed to execute the same routine as many times as the value of the mailbox.

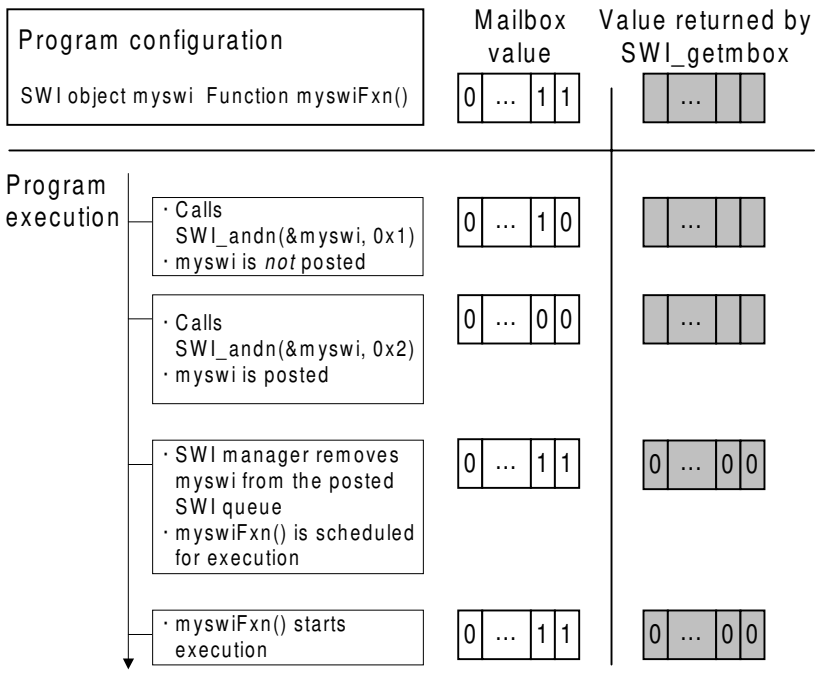
Figure 4–1 Using SWI_inc to Post an SWI




```
† myswiFxn()
 {    . . .
      repetitions = SWI_getmbox();
      while (repetitions --){
           'run SWI routine'
      }
 . . .
 }
```

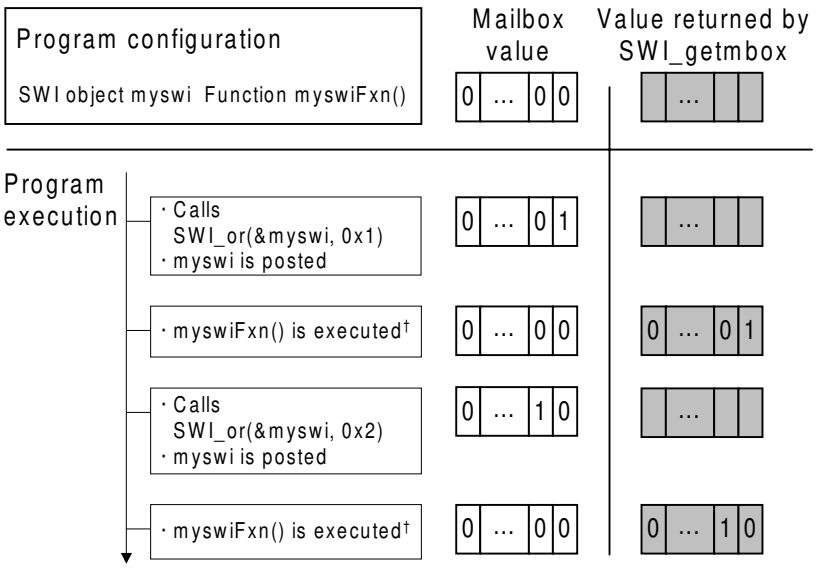
If more than one event must always happen for a given software interrupt to be triggered, SWI_andn should be used to post the corresponding SWI object. For example, if a software interrupt must wait for input data from two different devices before it can proceed, its mailbox should have two set bits when the SWI object was created with the Configuration Tool. When both routines that provide input data have completed their tasks, they should both call SWI_andn with complementary bitmasks that clear each of the bits set in the SWI mailbox default value. Hence, the software interrupt is posted only when data from both processes is ready.

Figure 4–2 Using SWI_andn to Post an SWI



In some situations the SWI function may call different routines depending on the event that posted it. In that case the program can use SWI_or to post the SWI object unconditionally when an event happens. The value of the bitmask used by SWI_or encodes the event type that triggered the post operation, and can be used by the SWI function as a flag that identifies the event and serves to choose the routine to execute

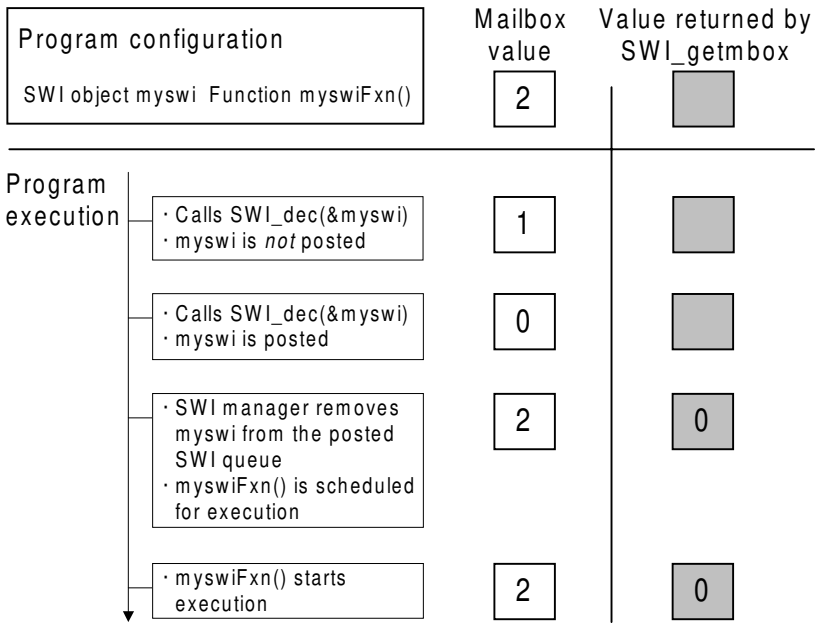
Figure 4–3 Using SWI_or to Post an SWI.




```
† myswiFxn()
 {
     ...
     eventType = SWI_getmbox();
     switch (eventType) {
         case '0x1':
             'run processing algorithm 1'
         case '0x2':
             'run processing algorithm 2'
         case '0x4':
             'run processing algorithm 3'
         ...
     }
     ...
 }
```

If the program execution requires that multiple occurrences of the same event must take place before an SWI is posted, SWI_dec should be used to post the SWI. By configuring the SWI mailbox to be equal to the number of occurrences of the event before the SWI should be posted and calling SWI_dec every time the event occurs, the SWI is posted only after its mailbox reaches 0; i.e., after the event has occurred a number of times equal to the mailbox value.

Figure 4–4 Using SWI_dec to Post an SWI



4.5 Hardware Interrupts

Hardware interrupts handle critical processing that the application must perform in response to external asynchronous events. The DSP/BIOS HWI module is used to manage hardware interrupts. Using the HWI manager in the Configuration Tool, you can configure the ISR for each hardware interrupt in the DSP. The HWI manager contains an HWI object for each hardware interrupt in your DSP. All HWI objects are listed in the Configuration Tool in order of priority, from the highest to the lowest priority interrupt.

You need to enter only the name of the ISR that is called in response to a hardware interrupt in the Property Page of the corresponding HWI object in the Configuration Tool. DSP/BIOS takes care of setting up the interrupt vector table so that each hardware interrupt is handled by the appropriate ISR. The Configuration Tool also allows you to select the memory segment where the interrupt vector table is located.

The online help in the Configuration Tool describes HWI objects and their parameters. See *HWI Module*, page 6–22, for reference information on the HWI module API calls.

4.5.1 Writing an HWI Routine

When a hardware interrupt preempts the function that is currently executing, the HWI function must save and restore any registers it uses or modifies. This gives the function that resumes running when the HWI function completes the same context it had when it was preempted.

HWI functions are usually written in assembly language for efficiency. DSP/BIOS provides two assembly macros to be used as preamble and postamble to an ISR: HWI_enter and HWI_exit. HWI_enter saves the register context for a DSP/BIOS ISR. HWI_exit restores the register context for a DSP/BIOS ISR. Both macros use the input parameter MASK, which specifies the set of registers that must be saved and restored.

If you want to write your ISR in C, you need to write a minimal assembly routine to call HWI_enter and HWI_exit to save the required registers around the call to your C function. You must save all registers that might be used in C before calling a C function from assembly. You can use HWI_enter with the C54_ABTEMPS mask to save these registers. This mask and others are defined in the c54.h54 file, which is in the bios\include folder provided with Code Composer.

```
;
;  ======== _DSS_isr ========
;
; Calls the C ISR code after setting cpl
; and saving C54_CNOTPRESERVED
;
_DSS_isr:
    HWI_enter    C54_CNOTPRESERVED, 0fff7h
    ; cpl = 0
    ; dp = GBL_A_SYSPAGE
    ; We need to set cpl bit when going to C
    ssbx    cpl
    nop                ; cpl latency
    nop                ; cpl latency
    call    _DSS_cisr
    rsbx    cpl            ; HWI_exit precondition
    nop                ; cpl latency
    nop                ; cpl latency
    ld    #GBL_A_SYSPAGE, dp
    HWI_exit    C54_CNOTPRESERVED, 0fff7h
```

The HWI_enter and HWI_exit macros also ensure that no software interrupts preempt the ISR, even if they are posted from the HWI function. Hence, within an HWI function, the HWI_enter macro must be called previous to any DSP/BIOS API call that could post or affect a software interrupt. HWI_enter guarantees that any software interrupt posted during the execution of an ISR does not run until the HWI function calls the HWI_exit macro. 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
- PRD_tick

If your ISR is written in C and calls any of the functions listed above, you must write a minimal assembly routine to call HWI_enter and HWI_exit around the calls to your C function.

Note that if an HWI function calls any of the PIP APIs—PIP_alloc, PIP_free, PIP_get, PIP_put—the pipe's notifyWriter or notifyReader functions run as part of the HWI context. Any HWI function must use HWI_enter if it indirectly runs a function containing any of the SWI or PRD calls listed above. Also, any registers that the notification functions might change should also be saved and restored with HWI_enter and HWI_exit.

HWI_enter and HWI_exit macros must coordinate the disabling and enabling of the SWI manager to ensure that software interrupts do not run until the HWI function has completed execution.

4.5.2 Nesting Interrupts

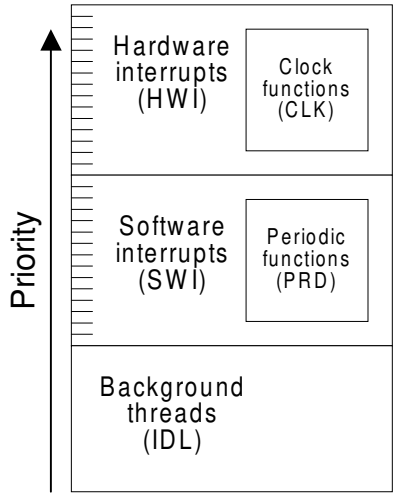
When an interrupt is triggered, the processor disables interrupts globally (by setting the INTM bit in the status register ST1) and then jumps to the ISR set up in the interrupt vector table. The HWI_enter macro reenables interrupts by clearing the INTM bit in the ST1 register. Before doing so, HWI_enter selectively disables some interrupts by clearing the appropriate bits in the interrupt mask register (IMR). The bits that are cleared in the IMR are determined by the IMRDISABLEMASK input parameter passed to the HWI_enter macro. Hence, HWI_enter gives you control to select what interrupts can and cannot preempt the current HWI function.

When HWI_exit is called, you can also provide an IMRRESTOREMASK parameter. The bit pattern in the IMRRESTOREMASK determines what interrupts are restored by HWI_exit, by setting the corresponding bits in the IMR. Of the interrupts in IMRRESTOREMASK, HWI_exit restores only those that were disabled with HWI_enter. If upon exiting the ISR you do not wish to restore one of the interrupts that was disabled with HWI_enter, do not set that interrupt bit in IMRRESTOREMASK in HWI_exit. HWI_exit does not affect the status of interrupt bits that are not in IMRRESTOREMASK.

4.6 Preemption and Yielding

Within DSP/BIOS, hardware interrupts have the highest priority. Software interrupts have lower priority than hardware interrupts. The background idle loop is the thread with the lowest priority of all.

Figure 4–5 Thread Priorities



This figure shows what happens when one type of thread is running (left column) and another thread is posted (top row). When a software interrupt or background thread is running, the results depend on whether or not hardware interrupts or software interrupts have been disabled. (The action indicated in the boxes is that of the posted thread.)

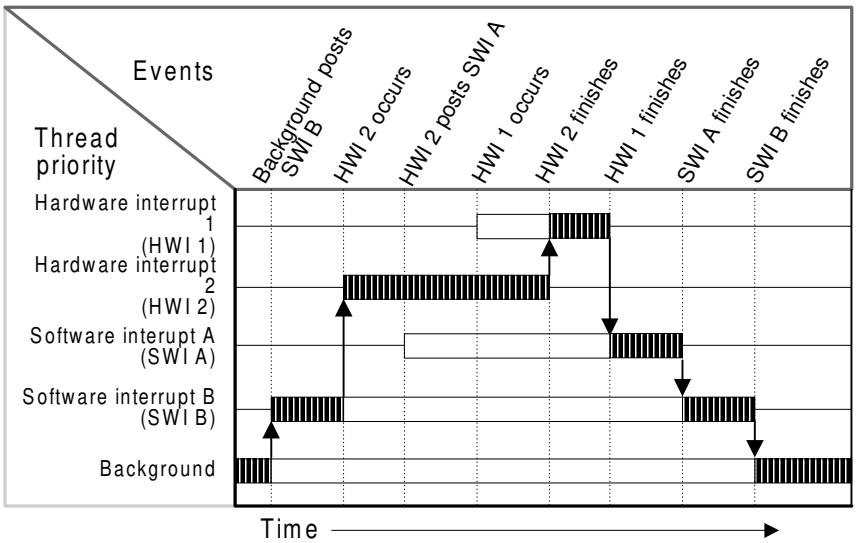
Figure 4–6 Thread Preemption

Thread running \ Thread posted		HWI (any priority)	SWI (higher priority)	SWI (lower priority)
Hardware interrupt		W	--	W
Software interrupt	-- HWIs enabled	P	P	W
	-- HWIs disabled	W	P	W
	-- SWIs enabled	P	P	W
	-- SWIs disabled	P	W	W
Background	-- HWIs enabled	P	P	--
	-- HWIs disabled	W	P	--
	-- SWIs enabled	P	P	--
	-- SWIs disabled	P	W	--

P = Preempts
W = Waits
-- = No such object of this priority

The figure below shows the thread of execution for a scenario in which SWIs and HWIs are enabled (the default), and a hardware interrupt routine posts a software interrupt whose priority is higher than that of the software interrupt running when the interrupt occurs. Also, a higher priority hardware interrupt occurs while the first ISR is running. The second ISR is held off because the first ISR masks off (i.e., disables) the higher priority interrupt during the first ISR.

Figure 4–7 Preemption Scenario



The low priority software interrupt is asynchronously preempted by the hardware interrupts. The first ISR posts a higher priority software interrupt, which is executed after both hardware interrupt routines finish executing.

4.6.1 Preventing Preemption by a Higher-Priority Thread

Within an idle loop function or a software interrupt function, you can temporarily prevent preemption by a higher priority software interrupt by calling SWI_disable, which disables all SWI preemption. To reenable SWI preemption you must call SWI_enable.

Calls to SWI_disable can be nested. When a series of SWI_disable calls occur contiguously, the same number of SWI_enable calls must occur before SWI preemption is enabled again.

You can also protect any thread from being preempted by a hardware interrupt. By calling HWI_disable, interrupts are globally disabled in your application. HWI_disable sets the INTM bit in the ST1 register, preventing the CPU from taking any maskable hardware interrupt. To reenable interrupts, call HWI_enable. HWI_enable clears the INTM bit in the ST1 register.

4.6.2 Saving Registers During Software Interrupt Preemption

When a software interrupt preempts another software interrupt or the background idle loop, DSP/BIOS preserves the context of the preempted thread by automatically saving all of the following CPU registers onto the application stack:

ar0	ag	pmst
ar1	ah	rea
ar2	al	rsa
ar3	bg	sp
ar4	bh	st0
ar5	bl	st1
ar6	bk	t
ar7	brc	trn

Your SWI function does not need to save and restore all these registers, even if the SWI function is written in assembly.

However, SWI functions that are written in assembly must follow C register usage conventions: they must save and restore registers ar1, ar6, and ar7. (See the *TMS320C54x Optimizing C Compiler User's Guide* for more details on C register conventions.)

The context is not saved automatically within an HWI function. You must use the HWI_enter and HWI_exit macros to preserve the interrupted context when an HWI function is triggered.

4.6.3 Software Interrupt Priorities and Application Stack Size

All threads in DSP/BIOS, including hardware interrupts, software interrupts, and the functions of the background idle loop, are executed using the same software stack (the application stack).

The application stack stores the register context when a software interrupt preempts another thread. To allow the maximum number of preemptions that may occur at run time, the required stack size grows each time you add a software interrupt priority level. Thus, giving software interrupts the same priority level is more efficient in terms of stack size than giving each software interrupt a separate priority.

The default application stack size for the MEM module is 256 words. You can change the sizes in the Configuration Tool. The estimated sizes required are shown in the status bar at the bottom of the Configuration Tool.

You can create up to 15 software interrupt priority levels, but each level requires a larger application stack. If you see a pop-up message that says “the application stack size is too small to support a new software interrupt priority level,” increase the Application Stack Size property of the Memory Section Manager.

Creating the first PRD object creates a new SWI object called PRD_swi (see section 4.8, *Periodic Function Manager (PRD) and the System Clock*, page 4-24, for more information on PRD). If no SWI objects have been created before the first PRD object is added, adding PRD_swi creates the first priority level, producing a corresponding increase in the required application stack.

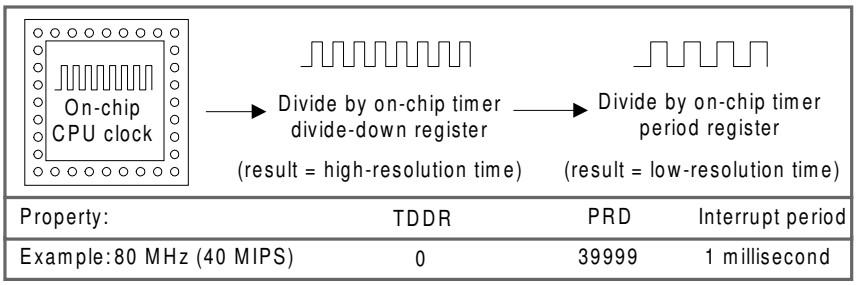
4.7 Clock Manager (CLK Module)

DSPs typically have one or more on-chip timers that generate a hardware interrupt at periodic intervals. DSP/BIOS normally uses one of the available on-chip timers as the source for its own real-time clocks.

The CLK module provides two 32-bit real-time clocks with different resolutions: the high-resolution and low-resolution clocks. These clocks can be used to measure the passage of time in conjunction with STS accumulator objects, as well as to add time stamp messages to message logs. Using the on-chip timer hardware present on most TMS320 DSPs, the CLK module supports time resolutions close to the single instruction cycle.

The following figure shows the relationship between the on-chip timer, configuration properties, and timer interrupt rates

Figure 4–8 CLK Module Properties



Property:	TDDR	PRD	Interrupt period
Example:80 MHz (40 MIPS)	0	39999	1 millisecond

The CLK manager also allows you to create an arbitrary number of clock functions. Clock functions are executed by the CLK manager each time a timer interrupt occurs.

4.7.1 High- and Low-Resolution Clocks

Using the CLK manager in the Configuration Tool, you can disable or enable DSP/BIOS to use an on-chip timer to drive high- and low-resolution times. The TMS320C54x has one general-purpose timer. The Configuration Tool allows you to enter the period at which the timer interrupt is triggered. See *CLK Module*, page 6–7, for more details about these properties. By entering the period of the timer interrupt, DSP/BIOS automatically sets up the appropriate value for the period register.

If the CLK manager is enabled in the Configuration Tool, the timer counter register is decremented every instruction cycle. 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 time is equal to the number of timer interrupts that have occurred. To obtain the low-resolution time, you can call CLK_getltime from your application code.

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 invoke only 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 from CLK_F_isr before and after CLK functions are called.)

The high-resolution clock ticks at the same rate the timer counter register is decremented. Hence, the high-resolution time is the number of times the timer counter register has been decremented. Given the high CPU clock rate, the 16-bit timer counter register may reach 0 very quickly. The 32-bit high-resolution time is actually calculated by multiplying the low-resolution time (i.e., the interrupt count) by the value of the period register and adding the difference between the period register value and the value of the timer counter register. To obtain the value of the high-resolution time you can call CLK_gethtime from your application code.

The value of the clock restarts at the value in the period register when 0 is reached.

Other CLK module APIs are CLK_getprd, which returns the value set for the period register in the Configuration Tool; and CLK_countspms, which returns the number of timer counter register decrements per millisecond.

Modify the properties of the CLK manager with the Configuration Tool to configure the low-resolution clock. For example, to make the low-resolution clock tick every millisecond (.001 sec), type 1000 in the CLK manager's Microseconds/Int field. The Configuration Tool automatically calculates the correct value for the period register.

You can directly specify the period register value if you put a checkmark in the Directly configure on-chip timer registers box. To generate a 1 millisecond (.001 sec) system clock period on a 40 MIPS processor using the CPU to drive the clock, the period register value is:

```
Period = 0.001 sec * 40,000,000 cycles per second = 40,000
```

4.8 Periodic Function Manager (PRD) and the System Clock

Many applications need to schedule functions based on I/O availability or some other programmed event. Other applications can schedule functions based on a real-time clock.

The PRD manager allows you to create objects that schedule periodic execution of program functions. To drive the PRD module, DSP/BIOS provides a system clock. The system clock is a 32-bit counter that ticks every time PRD_tick is called. You can use the timer interrupt or some other periodic event to call PRD_tick and drive the system clock.

There can be several PRD objects but all are driven by the same system clock. The period of each PRD object determines the frequency at which its function is called. The period of each PRD object is specified in terms of the system clock time; i.e., in system clock ticks.

To schedule functions based on certain events, use the following procedures:

- **Based on a real-time clock**. Put a check mark in the Use CLK Manager to Drive PRD box by right-clicking on the PRD manager and selecting Properties in the Configuration Tool. By doing this you are setting the timer interrupt used by the CLK manager to drive the system clock. Note that when you do this a CLK object called PRD_clock is added to the CLK module. This object calls PRD_tick every time the timer interrupt goes off, advancing the system clock by one tick.

Note: When the CLK manager is used to drive PRD, the system clock that drives PRD functions ticks at the same rate as the low-resolution clock. The low-resolution and system time coincide.

- **Based on I/O availability or some other event**. Remove the check mark from the Use the CLK Manager to Drive PRD box for the PRD manager. Your program should call PRD_tick to increment the system clock. In this case the resolution of the system clock equals the frequency of the interrupt from which PRD_tick is called.

4.8.1 Invoking Functions for PRD Objects

When PRD_tick is called two things occur:

- PRD_D_tick, the system clock counter, increases by one; i.e., the system clock ticks.
- An SWI called PRD_swi is posted.

Note that when a PRD object is created with the Configuration Tool, a new SWI object is automatically added called PRD_swi.

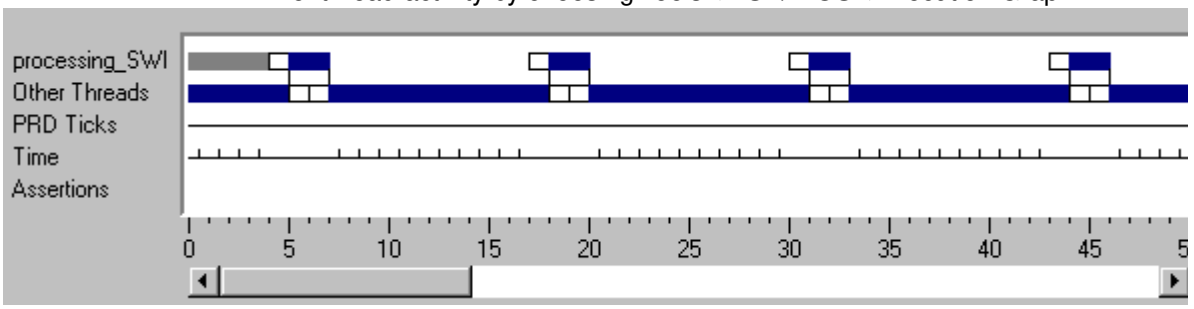
When PRD_swi runs, its function executes the following type of loop:

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

Hence, the execution of periodic functions is deferred to the context of PRD_swi, rather than being executed in the context of the ISR where PRD_tick was called. As a result, there may be a delay between the time the system tick occurs and the execution of the periodic objects whose periods have expired with the tick. If these functions run immediately after the tick, you should configure PRD_swi to have a high priority with respect to other threads in your application.

4.9 Using the Execution Graph to View Program Execution

You can use the Execution Graph in Code Composer to see a visual display of thread activity by choosing Tools→DSP/BIOS→Execution Graph.



4.9.1 States in the Execution Graph Window

This window examines the information in the system log (LOG_system in the Configuration Tool) and shows the thread states in relation to the timer interrupt (Time) and system clock ticks (PRD Ticks).

White boxes indicate that a thread has been posted and is ready to run. Blue-green boxes indicate that the host had not yet received any information about the state of this thread at that point in the log. Dark blue boxes indicate that a thread is running.

Bright blue boxes in the Errors row indicate that an error has occurred. For example, an error is shown when the Execution Graph detects that a thread did not meet its real-time deadline. It also shows invalid log records, which may be caused by the program writing over the system log. Double-click on an error to see the details.

4.9.2 Threads in the Execution Graph Window

The SWI and PRD functions listed in the left column are listed from highest to lowest priority. However, for performance reasons, there is no information in the Execution Graph about interrupt and background threads (aside from the CLK ticks, which are normally performed by an interrupt). Time not spent within an SWI or PRD thread must be within an HWI or IDL thread, so this time is shown in the Other Threads row.

Functions run by PIP (notify functions) run as part of the thread that called the PIP API. The LNK_dataPump object runs a function that manages the host's end of an HST (Host Channel manager) object. This object and other IDL objects run from the IDL background thread, and are included in the Other Threads row.

Note: The Time marks and the PRD Ticks are not equally spaced. This graph shows a square for each event. If many events occur between two Time interrupts or PRD Ticks, the distance between the marks is wider than for intervals during which fewer events occurred.

4.9.3 Sequence Numbers in the Execution Graph Window

The numbers below the bottom scroll bar show the sequence numbers in the Execution Graph for the events.

Note: Circular logs (the default for the Execution Graph) contain only the most recent n events. Normally, there are events that are not listed in the log because they occur after the host polls the log and are overwritten before the next time the host polls the log. The Execution Graph shows a red vertical line and a break in the log sequence numbers at the end of each group of log events it polls.

You can view more log events by increasing the size of the log to hold the full sequence of events you want to examine. You can also set the RTA Control Panel to log only the events you want to examine.

4.9.4 RTA Control Panel Settings for Use with the Execution Graph

The TRC module allows you to control what events are recorded in the Execution Graph at any given time during the application execution. The recording of SWI, PRD, and CLK events in the Execution Graph can be controlled from the host (using the RTA Control Panel; Tools→DSP/BIOS→ RTA Control Panel in Code Composer) or from the target code (through the TRC_enable and TRC_disable APIs). See section 3.4.4.2, *Control of Implicit Instrumentation*, page 3-12, for details on how to control implicit instrumentation.

When using the Execution Graph, turning off automatic polling stops the log from scrolling frequently and gives you time to examine the graph. You can use either of these methods to turn off automatic polling:

- Right-click on the Execution Graph and choose Pause from the shortcut menu.
- Right-click on the RTA Control Panel and choose Property Page. Set the Message Log/Execution Graph refresh rate to 0. Click OK.

You can poll log data from the target whenever you want to update the graph:

- Right-click on the Execution Graph and choose Refresh Window from the shortcut menu.

  You can choose Refresh Window several times to see additional data. The shortcut menu you see when you right-click on the graph also allows you to clear the previous data shown on the graph.

If you plan to use the Execution Graph and your program has a complex execution sequence, you can increase the size of the Execution Graph in the Configuration Tool. Right-click on the LOG_system LOG object and select Properties to increase the buflen property. Each log message uses four words, so the buflen should be at least the number of events you want to store multiplied by four.

4.10 SWI and PRD Accumulators: Real-Time Deadline Headroom

Many tasks in a real-time system are periodic; that is, they execute continuously and at regular intervals. It is important that such tasks finish executing before it is time for them to run again. A failure to complete in this time represents a missed real-time deadline. While internal data buffering can be used to recover from occasional missed deadlines, repeated missed deadlines eventually result in an unrecoverable failure.

The implicit statistics gathered for SWI functions measure the time from when a software interrupt is ready to run and the time it completes. This timing is critical because the processor is actually executing numerous hardware and software interrupts. If a software interrupt is ready to execute but must wait too long for other software interrupts to complete, the real-time deadline is missed. Additionally, if a task starts executing, but is interrupted too many times for too long a period of time, the real-time deadline is also missed.

The maximum ready-to-complete time is a good measure of how close the system is to potential failure. The closer a software interrupt's maximum ready-to-complete time is to its period, the more likely it is that the system cannot survive occasional bursts of activity or temporary data-dependent increases in computational requirements. The maximum ready-to-complete time is also an indication of how much headroom exists for future product enhancements (which invariably require more MIPS).

Note: DSP/BIOS does not implicitly measure the amount of time each software interrupt takes to execute. This measurement can be determined by running the software interrupt in isolation using either the simulator or the emulator to count the precise number of execution cycles required.

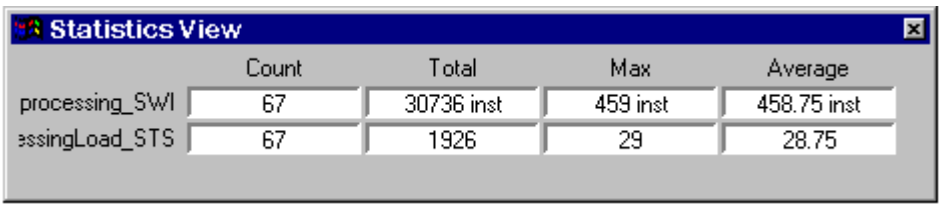
It is important to realize even when the sum of the MIPS requirements of all routines in a system is well below the MIPS rating of the DSP, the system may not meet its real-time deadlines. It is not uncommon for a system with a CPU load of less than 70% to miss its real-time deadlines due to prioritization problems. The maximum ready-to-complete times monitored by DSP/BIOS, however, provide an immediate indication when these situations exist.

When statistics accumulators for software interrupts and periodic objects are enabled, the host automatically gathers the count, total, maximum, and average for the following types of statistics:

- **SWI**. Statistics about the period elapsed from the time the software interrupt was posted to its completion.
- **PRD**. The number of periodic system ticks elapsed from the time the periodic function is ready to run until its completion. By definition, the i x period execution of a periodic function is ready to run when i x period ticks have occurred, where period is the period parameter for this periodic object.

You can set the units for the SWI completion period measurement by setting global SWI and CLK parameters. This period is measured in instruction cycles if the CLK module's Use high resolution time for internal timings parameter is set to True (the default) and the SWI module's Statistics Units parameter is set to Raw (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 for the Statistics Units parameter.

For example, if the maximum value for a PRD object increases continuously, the object is probably not meeting its real-time deadline. In fact, the maximum value for a PRD object should be less than or equal to the period (in system ticks) property of this PRD object. If the maximum value is greater than the period, the periodic function has missed its real-time deadline.



Chapter 5

Input/Output

This chapter discusses data transfer methods.

5.1 Objects Used for I/O

DSP/BIOS provides the following modules for data transfer:

- **PIP**. 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 threads and between the DSP chip and all kinds of real-time peripheral devices.
- **HST**. For simplified I/O between the target and the host, DSP/BIOS provides host channel objects. Pipes are used internally to implement and interface with host channels. The Host Channel Control in Code Composer simplifies the process by managing one end of the pipe.

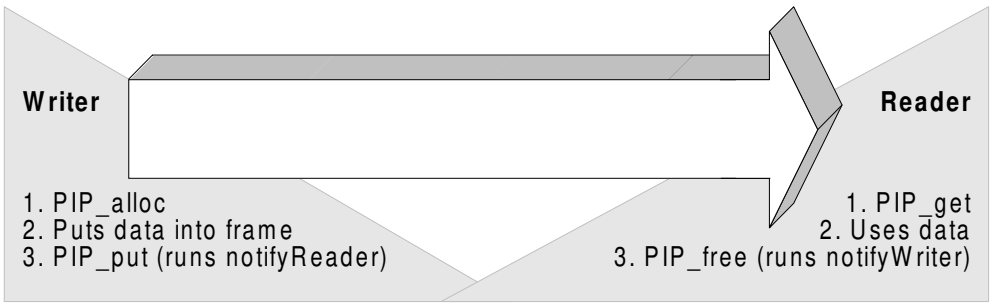
During early development—especially when testing processing algorithms—programs can use host channels to input canned data sets and to output the results to the host for analysis.

Once the algorithm appears sound, you can replace host channel objects with I/O drivers for production hardware built around DSP/BIOS pipe objects.

5.2 Data Pipe Manager (PIP Module)

Pipes are designed to manage block I/O (also called stream-based or asynchronous I/O). 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. The writer end is where the program writes frames of data. The reader end is where the program reads frames of data.



Data notification functions (notifyReader and notifyWriter) are performed to synchronize data transfer. These functions are triggered when a frame of data is read or written to notify the program that a frame is free or data is available. These functions are performed in the context of the function that calls PIP_free or PIP_put. They may also be called from the thread that calls PIP_get or PIP_alloc. When PIP_get is called, DSP/BIOS checks whether there are more full frames in the pipe. If so, the notifyReader function is executed. When PIP_alloc is called, DSP/BIOS checks whether there are more empty frames in the pipe. If so, the notifyWriter function is executed.

A pipe should have a single reader and a single writer. Often, one end of a pipe is controlled by a hardware ISR and the other end is controlled by a software interrupt function. Pipes can also be used to transfer data within the program between two application threads.

During program startup (which is described in detail in section 2.5, *DSP/BIOS Startup Sequence*, page 2-11), the BIOS_start function enables the DSP/BIOS modules. As part of this step, the PIP_startup function calls the notifyWriter function for each pipe object, since at startup all pipes have available empty frames.

There are no special format or data type requirements for the data to be transferred by a pipe.

The online help in the Configuration Tool describes data pipe objects and their parameters. See *PIP Module*, page 6–51, for reference information on the PIP module API.

5.2.1 Writing Data to a Pipe

The steps that a program should perform to write data to a pipe are as follows:

1) A function should first check the number of empty frames available to be filled with data. To do this, the program must check the return value of PIP_getWriterNumFrames. This function call returns the number of empty frames in a pipe object.

2) If the number of empty frames is greater than 0, the function then calls PIP_alloc to get an empty frame from the pipe.

3) Before returning from the PIP_alloc call, DSP/BIOS checks whether there are additional empty frames available in the pipe. If so, the notifyWriter function is called at this time.

4) Once PIP_alloc returns, the empty frame can be used by the application code to store data. To do this the function needs to know the frame's start address and its size. The API function PIP_getWriterAddr returns the address of the beginning of the allocated frame. The API function PIP_getWriterSize returns the number of words that can be written to the frame. (The default value for an empty frame is the configured frame size.)

5) When the frame is full of data, it can be returned to the pipe. If the number of words written to the frame is less than the frame size, the function can specify this by calling the PIP_setWriterSize function. Afterwards, call PIP_put to return the data to the pipe.

6) Calling PIP_put causes the notifyReader function to run. This enables the writer thread to notify the reader thread that there is data available in the pipe to be read.

The following code fragment demonstrates the previous steps:

```
extern far PIP_Obj writerPipe;   /* pipe object created with
                                    the Configuration Tool */
writer()
{
    Uns size;
    Uns newsize;
    Ptr addr;

    if (PIP_getWriterNumFrames(&writerPipe) > 0) {
        PIP_alloc(&writerPipe);  /* allocate an empty frame */
    }
    else {
        return;      /* There are no available empty frames */
    }

    addr = PIP_getWriterAddr(&writerPipe);
    size = PIP_getWriterSize(&writerPipe);

    ' fill up the frame '

    /* optional */
    newsize = 'number of words written to the frame';
    PIP_setWriterSize(&writerPipe, newsize);

    /* release the full frame back to the pipe */
    PIP_put(&writerPipe);
}
```

5.2.2 Reading Data from a Pipe

To read a full frame from a pipe, a program should perform the following steps:

1) The function should first check the number of full frames available to be read from the pipe. To do this, the program must check the return value of PIP_getReaderNumFrames. This function call returns the number of full frames in a pipe object.

2) If the number of full frames is greater than 0, the function then calls PIP_get to get a full frame from the pipe.

3) Before returning from the PIP_get call, DSP/BIOS checks whether there are additional full frames available in the pipe. If so, the notifyReader function is called at this time.

4) Once PIP_get returns, the data in the full frame can be read by the application. To do this the function needs to know the frame's start address and its size. The API function PIP_getReaderAddr returns the address of the beginning of the full frame. The API function PIP_getReaderSize returns the number of valid data words in the frame.

5) When the application has finished reading all the data, the frame can be returned to the pipe by calling PIP_free.

6) Calling PIP_free causes the notifyWriter function to run. This enables the reader thread to notify the writer thread that there is a new empty frame available in the pipe.

The following code fragment demonstrates the previous steps:

```
extern far PIP_Obj readerPipe;  /* created with the
                                   Configuration Tool */
reader()
{
    Uns size;
    Ptr addr;

    if (PIP_getReaderNumFrames(&readerPipe) > 0) {
        PIP_get(&readerPipe);    /* get a full frame */
    }
    else {
        return;       /* There are no available full frames */
    }

    addr = PIP_getReaderAddr(&readerPipe);
    size = PIP_getReaderSize(&readerPipe);

    ' read the data from the frame '

    /* release the empty frame back to the pipe */
    PIP_free(&readerPipe);
}
```

5.2.3 Using a Pipe's Notify Functions

The reader or writer threads of a pipe can operate in a polled mode and directly test the number of full or empty frames available before retrieving the next full or empty frame. The example code in section 5.2.1, *Writing Data to a Pipe*, page 5-4, and section 5.2.2, *Reading Data from a Pipe*, page 5-5, demonstrates this type of polled read and write operation.

When used to buffer real-time I/O streams written (read) by a hardware peripheral, pipe objects often serve as a data channel between the HWI routine triggered by the peripheral itself and the program function that ultimately reads (writes) the data. In such situations, the application can effectively synchronize itself with the underlying I/O stream by configuring the pipe's notifyReader (notifyWriter) function to automatically post a software interrupt that runs the reader (writer) function. When the HWI routine finishes filling up (reading) a frame and calls PIP_put (PIP_free), the pipe's notify function can be used to automatically post a software interrupt. In this case, rather than polling the pipe for frame availability, the reader (writer) function runs only when the software interrupt is triggered; i.e., when frames are available to be read (written).

Such a function would not need to check for the availability of frames in the pipe, since it is called only when data is ready. As a precaution, the function may still check whether frames are ready, and if not, cause an error condition, as in the following example code:

```
if (PIP_getReaderNumFrames(&readerPipe) = 0) {
   error();   /* writer function should not have been posted! */
}
```

Hence, the notify function of pipe objects can serve as a flow-control mechanism to manage I/O to other threads and hardware devices.

5.2.4 Calling Order for PIP APIs

Each pipe object internally maintains a list of empty frames and a counter with the number of empty frames on the writer side of the pipe, and a list of full frames and a counter with the number of full frames on the reader side of the pipe. The pipe object also contains a descriptor of the current writer frame (i.e., the last frame allocated and currently being filled by the application) and the current reader frame (i.e., the last full frame that the application got and that is currently reading).

When PIP_alloc is called, the writer counter is decreased by 1. An empty frame is removed from the writer list and the writer frame descriptor is updated with the information from this frame. When the application calls PIP_put after filling the frame, the reader counter is increased by one, and the writer frame descriptor is used by DSP/BIOS to add the new full frame to the pipe's reader list.

Note: Every call to PIP_alloc must be followed by a call to PIP_put before PIP_alloc can be called again: the pipe I/O mechanism does not allow consecutive PIP_alloc calls. Doing so would overwrite previous descriptor information and would produce undetermined results.

```
/* correct */          /* error! */
PIP_alloc();           PIP_alloc();
...                    ...
PIP_put();             PIP_alloc();
...                    ...
PIP_alloc();           PIP_put();
...                    ...
PIP_put();             PIP_put();
```

Similarly when PIP_get is called, the reader counter is decreased by 1. A full frame is removed from the reader list and the reader frame descriptor is updated with the information from this frame. When the application calls PIP_free after reading the frame, the writer counter is increased by 1, and the reader frame descriptor is used by DSP/BIOS to add the new empty frame to the pipe's writer list. Hence, every call to PIP_get must be followed by a call to PIP_free before PIP_get can be called again: the pipe I/O mechanism does not allow consecutive PIP_get calls. Doing so would overwrite previous descriptor information and would produce undetermined results.

```
/* correct */          /* error! */
PIP_get();             PIP_get();
...                    ...
PIP_free();            PIP_get();
...                    ...
PIP_get();             PIP_free();
...                    ...
PIP_free();            PIP_free();
```

5.2.4.1 Avoiding Recursion Problems

Care should be applied when a pipe's notify functions call PIP APIs for the same pipe.

Consider the following example: A pipe's notifyReader function calls PIP_get for the same pipe. The pipe's reader is an HWI routine. The pipe's writer is an SWI routine. Hence the reader has higher priority than the writer. (Calling PIP_get from the notifyReader in this situation may make sense because this allows the application to get the next full buffer ready to be used by the reader—the HWI routine—as soon as it is available and before the hardware interrupt is triggered again.)

The pipe's reader function, the HWI routine, calls PIP_get to read data from the pipe. The pipe's writer function, the SWI routine, calls PIP_put. Since the call to the notifyReader happens within PIP_put in the context of the current routine, a call to PIP_get also happens from the SWI writer routine.

Hence, in the example described two threads with different priorities call PIP_get for the same pipe. This could have catastrophic consequences if one thread preempts the other and as a result, PIP_get is called twice before calling PIP_free, or PIP_get is preempted and called again for the same pipe from a different thread.

Note: As a general rule to avoid recursion, you should avoid calling PIP functions as part of notifyReader and notifyWriter. If necessary for application efficiency, such calls should be protected to prevent reentrancy for the same pipe object and the wrong calling sequence for the PIP APIs.

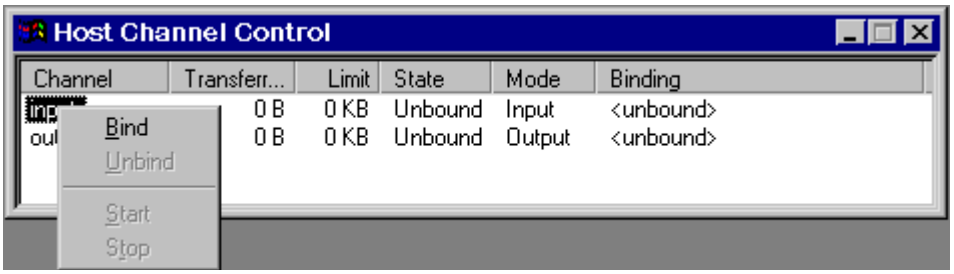
5.3 Host Input/Output Manager (HST Module)

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

Note:

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

You dynamically bind channels to files on the PC host by right-clicking on the Host Channel Control in Code Composer. Then you start the data transfer for each channel.



Each host channel is internally implemented using a pipe 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).

The code for reading data might look like the following:

```
extern far HST_Obj input;

readFromHost()
{
    PIP_Obj *pipe;
    Uns size;
    Ptr addr;

    pipe = HST_getpipe(&input)    /* get a pointer to the host
                                     channel's pipe object */
    PIP_get(pipe);                /* get a full frame from the
                                     host */
    size = PIP_getReaderSize(pipe);
    addr = PIP_getReaderAddr(pipe);

    ' read data from frame '

    PIP_free(pipe);       /* release empty frame to the host */
}
```

Each host channel can specify a data notification function to be performed when a frame of data for an input channel (or free space for an output channel) is available. This function is triggered when the host writes or reads a frame of data.

HST channels treat files as 16-bit words of raw data. The format of the data is application-specific, and you should verify that the host and the target agree on the data format and ordering. For example, if you are reading 32-bit integers from the host, you need to make sure the host file contains the data in the correct byte order. Other than correct byte order, there are no special format or data type requirements for data to be transferred between the host and the target.

While you are developing a program, you may want to use HST objects to simulate data flow and to test changes made to canned data by program algorithms. During early development, especially when testing signal processing algorithms, the program would explicitly use input channels to access data sets from a file for input for the algorithm and would use output channels to record algorithm output. The data saved to a file with the output host channel can be compared with expected results to detect algorithm errors. Later in the program development cycle, when the algorithm appears sound, you can change the HST objects to PIP objects communicating with other threads or I/O drivers for production hardware.

5.3.1 Transfer of HST Data to the Host

While the amount of usable bandwidth for real-time transfer of data streams to the host ultimately depends on the choice of physical data link, the HST Channel interface remains independent of the physical link. The HST manager in the Configuration Tool allows you to choose among the physical connections available.

The actual data transfer to the host occurs during the idle loop, from the LNK_dataPump idle function.

5.4 I/O Performance Issues

If you are using an HST object, the host PC reads or writes data using the function specified by the LNK_dataPump object. This is a built-in IDL object that runs its function as part of the background thread. Since background threads have the lowest priority, software interrupts and hardware interrupts preempt data transfer.

Note that the polling rates you set in the LOG, STS, and TRC controls do not control the data transfer rate for HST objects. (Faster polling rates actually slow the data transfer rate somewhat because LOG, STS, and TRC data also need to be transferred.)

Chapter 6

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**

6.1 DSP/BIOS Modules

These are the DSP/BIOS modules:

Module	Description
CLK	System clock manager
GBL	Global setting manager
HST	Host input/output manager
HWI	Hardware interrupt manager
IDL	Idle function and processing loop manager
LOG	Message Log manager
MEM	Memory manager
PIP	Data pipe manager
PRD	Periodic function manager
RTDX	Real-Time Data Exchange manager
STS	Statistics accumulator manager
SWI	Software interrupt manager
TRC	Trace manager

6.2 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.

6.3 List of Operations

This is a list of the DSP/BIOS operations.

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

Function	Operation
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_put	Put a full frame into a pipe
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
RTDX_channelBusy	Return status indicating whether a channel is busy
RTDX_CreateInputChannel	Declare an input channel
RTDX_CreateOutputChannel	Declare an output channel
RTDX_disableInput	Disable an input channel
RTDX_disableOutput	Disable an output channel
RTDX_enableInput	Enable an input channel
RTDX_enableOutput	Enable an output channel
RTDX_isInputEnabled	Return true if the input channel is enabled
RTDX_isOutputEnabled	Return true if the output channel is enabled
RTDX_read	Read from an input channel
RTDX_readNB	Read from an input channel without blocking
RTDX_sizeofInput	Return the number of sizeof() units read from an input channel

Function	Operation
RTDX_write	Write to an output channel
STS_add	Add a value to a statistics accumulator
STS_delta	Add computed value of an interval to accumulator
STS_reset	Reset the values stored in an STS object
STS_set	Store initial value of an interval to accumulator
SWI_andn	Clear bits from SWI's mailbox and post if becomes 0
SWI_dec	Decrement SWI's mailbox and post if becomes 0
SWI_disable	Disable software interrupts
SWI_enable	Enable software interrupts
SWI_getmbox	Return SWI's mailbox value
SWI_getpri	Return a SWI's priority mask
SWI_inc	Increment SWI's mailbox and post
SWI_or	Set or mask in SWI's mailbox and post
SWI_post	Post a software interrupt
SWI_raisepri	Raise a SWI's priority
SWI_restorepri	Restore a SWI's priority
SWI_self	Return address of currently executing SWI object
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

6.4 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 IMRDISABLE
Preconditions	intm = 1
Postconditions	dp = GBL_A_SYSPAGE cpl = ovm = c16 = frct = cmpt = 0
Modifies	c, cpl, dp, sp

Assembly functions can call C functions. Remember that the C compiler adds an underscore prefix to function names, so when calling a C function from assembly, add an underscore to the beginning of the C function name. For example, call _myfunction instead of myfunction. See the *TMS320C54x Optimizing 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.

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 timer counter register 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.

The TMS320C54x has one general-purpose timer. 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 parameters can be set for the CLK module:

- **Object Memory**. The memory segment 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 will not be 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:

- **CPU Interrupt**. Shows which HWI interrupt is used to drive the timer services.
- **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 fields 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 - DSP/BIOS Plug-ins 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_gethtime(Void);

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.

The timer counter is incremented every four CPU cycles. The high-resolution time is the number of times the timer counter has been incremented (number of instruction cycles divided by 4).

The 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.

In contrast, CLK_getltime returns a value that is also affected by the period register value. 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(Void);
```

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 a value that is not affected by the period register value. 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 (40 MIPS), 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(Void);

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

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.
- **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.
- **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 ’C54x chip.
- **Function Call Model**. This setting controls which libraries are used to link the application. If you change this setting, you must set the compiler and linker options to correspond. Use the far option only with ’C54x chips that support extended addressing (e.g., 5402, 549, 5410).
- **C Autoinitialization Model**. Select the run-time initialization model

HST Module

Host input/output 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 start 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 parameters can be set for the HST module:

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

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.

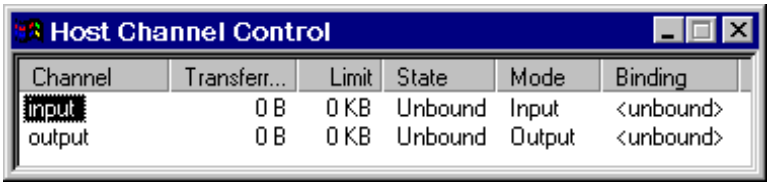
The following fields can be set for a host file object:

- **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 segment from which the buffer is allocated; all frames are allocated from a single contiguous buffer (of size framesize x numframes).
- **bufalign**. The alignment (in words) of the buffer allocated within the specified memory segment
- **framesize**. The length of each frame (in words)
- **numframes**. The number of frames
- **statistics**. Check this box if you want to monitor this channel with an STS object. You can display the STS object for this channel to see a count of the number of frames transferred with the Statistics View 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.

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 PIP_Obj *HST_getpipe(HST_Obj *hst);

Parameters HST_Obj *host /* host object */

Return Value PIP_Obj *pipe /* corresponding pipe */

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 is automatically configured to run a macro that increments the low-resolution time. See the CLK module for more details.

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

Note: Do not call SWI_disable or SWI_enable within an HWI function.

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

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

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

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 parameter can be set for the HWI module:

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

HWI Object Properties

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

- **comment**. A comment is provided to identify each HWI object
- **interrupt source**. Select the pin, DMA channel, timer, or other source of the interrupt
- **function**. The function to execute. Interrupt routines must be written 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.

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_SINT4-15	20-31	These interrupts can be used by various ’C54x peripherals.

HWI - DSP/BIOS Plug-ins 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(Void);

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.

Constraints and Calling Context

- HWI_disable cannot be called from an ISR context.

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 Void HWI_enable(Void);

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.

Constraints and Calling Context

- ❏ HWI_enable cannot be called from an ISR context.

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 an ISR before any DSP/BIOS API calls which 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. 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
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 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. Basically, this API must be called at the end of a function used to process a hardware interrupt. Such functions must be written in assembly language.
- ❏ The MASK parameter must match the corresponding parameter 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 Void 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

- 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

Idle function and processing loop manager

Functions

- **IDL_run**. Make one pass through idle functions

Description

The IDL module manages the lowest-level task in the application. This task executes functions that communicate with the host.

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

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 parameters can be set for the IDL module:

- **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. The instruction count takes into account all IDL functions, not just LNK_dataPump, RTA_dispatcher, and IDL_cpuLoad. If this box is checked, it is important to make sure that the IDL functions do not block on this first pass, otherwise your program never gets to main.
- **Object Memory**. The memory segment that contains the IDL objects

The following informational field is also displayed for the IDL module:

- **Idle Loop Instruction Count**. The number of instruction cycles required to perform the IDL loop and the default IDL functions (LNK_dataPump 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 fields 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.)

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 Void IDL_run(Void)

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 user 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 */
```

LOG Module

Message Log manager

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 Message Log manager 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 6-120, 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 Message 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 parameter can be set for the LOG module:

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

LOG Object Properties

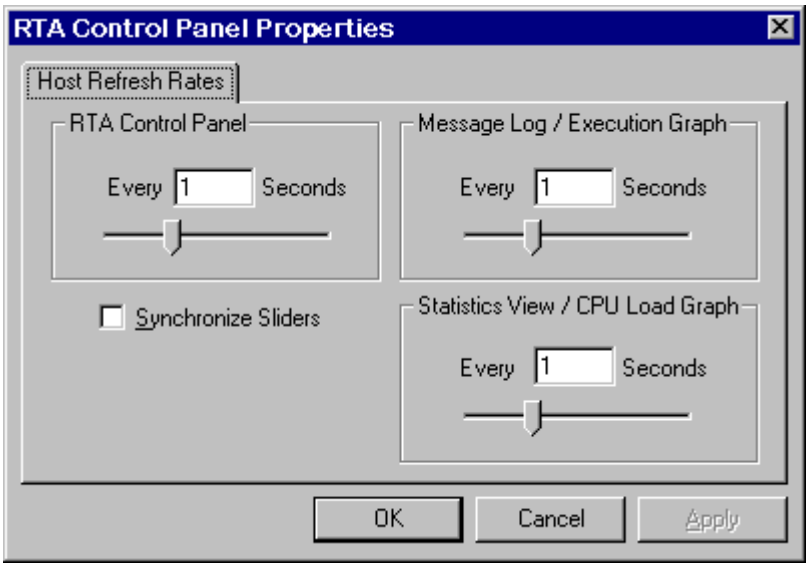
The following fields can be set for a log object:

- **comment**. Type a comment to identify this LOG object
- **bufseg**. The name of a memory segment to contain the log buffer
- **buflen**. The length of the log buffer (in words)
- **logtype**. The type of the log: circular or fixed. Events added to a full circular log overwrite the oldest event in the buffer, whereas events added to a full fixed log are dropped.
  - **Fixed**. The log stores the first messages it receives and stops accepting messages when its message buffer is full
  - **Circular**. The log automatically overwrites earlier messages when its buffer is full. As a result, a circular log stores the last events that occur.
- **datatype**. Choose printf if you use LOG_printf to write to this log and provide a format string.
  Choose raw data if you want to use LOG_event to write to this log and have the Message 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 6-45, and *LOG_event*, page 6-43, for information about the structure of a log record.

LOG - DSP/BIOS Plug-ins Interface

You can view log messages in real time while your program is running with the Message Log. To see the system log as a graph, choose Tools→DSP/BIOS→Execution Graph. To see a user log, choose Tools→DSP/BIOS→Message 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

Void LOG_disable(LOG_Obj *log);

Parameters

LOG_Obj *log /* log to be disabled */

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 Void LOG_enable(LOG_Obj *log);

Parameters LOG_Obj *log /* log to be enabled */

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, LOG_message

Write a message to the system log

C Interface

Syntax

```
Void LOG_error(String format, Arg arg0);
Void LOG_message(String format, Arg 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]; LOG_message format [section]

Preconditions

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.

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_error and LOG_message may contain any of the conversion characters supported for LOG_printf. See *LOG_printf*, page 6-45, for details.

The LOG_error and LOG_message assembly macros take an optional section argument. If you do not specify a section argument, assembly code following the macros 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 macros.

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_printf
TRC_disable
TRC_enable

LOG_event

Append an unformatted message to a message log

C Interface

Syntax

Void LOG_event(LOG_Obj *log, Arg arg0, Arg arg1, Arg arg2);

Parameters

```
LOG_Obj    *log;   /* log handle */
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

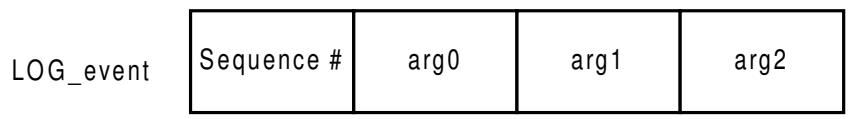
ag, ah, al, ar0, ar2, ar3, c, tc

Reentrant

yes

Description

LOG_event copies a sequence number and three arguments to the specified log's 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 Message 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 Data type 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

```
Void LOG_printf(LOG_Obj *log, String format);
   or
Void LOG_printf(LOG_Obj *log, String format, Int arg0);
   or
Void LOG_printf(LOG_Obj *log, String format, Int arg0, Int arg1);
```

Parameters

```
LOG_Obj    *log;     /* log handle */
String     format;   /* printf format stringb */
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's 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 Message 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

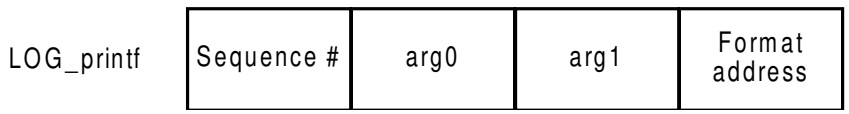
%s	Character string
	This character can only be used with constant string pointers. That is, the string must appear in the source and be passed to LOG_printf. For example, the following is supported:
	`char *msg = "Hello world!";`
	`LOG_printf(&trace, "%s", msg);`
	However, the following example is not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

If you want the Message 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 Data type 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 Void LOG_reset(LOG_Obj *log);

Parameters LOG_Obj *log /* log to be reset */

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

MEM Module

Memory section manager

Functions

None

Description

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

MEM Manager Properties

The DSP/BIOS memory section manager allows you to specify the memory segments required to locate the various code and data sections of a DSP/BIOS application.

The following global parameters 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
- **Stack Size (MAUs)**. The size of the software stack in MAUs. This value is shown in hex.
  The Configuration Tool status bar shows the estimated minimum stack size required for this application (as a decimal number).
- **Stack Section (.stack)**. The memory segment containing the software stack
- **Constant Section (.const)**. The memory segment containing the .const section generated by the C compiler to hold program constants such as string constants; if the C compiler is not used, this parameter is unused.
- **Text Section (.text)**. The memory segment containing the application code
- **BIOS Code Section (.bios)**. The memory segment containing the DSP/BIOS code
- **Data Sections (.data, .switch, .cio, .sysmem)**. These data sections contain program data, C switch statements, C standard I/O buffers, and the memory heap used by malloc and free.
- **Startup Code Section (.sysinit)**. The memory segment containing DSP/BIOS startup initialization code; this memory may be reused after main() starts executing
- **C Initialization Section (.cinit)**. The memory segment containing the .cinit section, to hold initialization records for C run-time autoinitialization
- **Uninitialized Sections (.bss, .far)**. The memory segment containing the .bss, .far, and .sysdata sections

MEM Object Properties

A memory segment represents a contiguous length of code or data memory in the address space of the processor. A MEM object has the following fields. The values in these fields cannot be changed; they are set automatically to match the board you choose for the Global Settings.

- **comment**. Type a comment to identify this MEM object
- **base**. The address at which this memory segment begins. This value is shown in hex.
- **len**. The length of this memory segment in words. This value is shown in hex.
- **space**. Type of memory segment. This is set to code for memory segments that store programs, and data for memory segments that store program data.

The following memory segments are predefined:

Name	Memory Segment Type
VECT	Interrupt vector memory
EPROG0	External program memory
EPROG1	External program memory
IPROG	Internal program memory
USERREGS	Application on-chip page 0 memory mapped registers; this segment cannot be moved or resized.
BIOSREGS	DSP/BIOS on-chip page 0 memory mapped registers; this segment cannot be moved or resized.
BIOSDATA	DSP/BIOS data page
IDATA	On-chip data memory
EDATA	External data memory

PIP Module

Data 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_put**. Put a full frame into the pipe
- **PIP_setWriterSize**. Set the number of valid words written to a pipe frame

PIP_Obj Structure Members

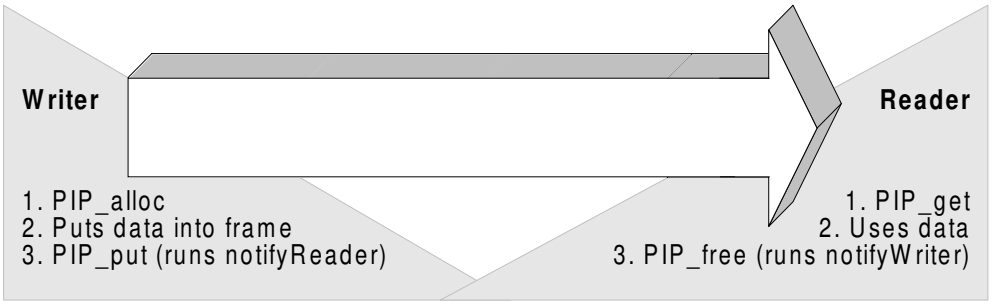
- **Ptr readerAddr**. Pointer to the address to begin reading from after calling PIP_get
- **Uns readerSize**. Number of words of data in the frame read with PIP_get
- **Uns readerNumFrames**. Number of frames available to be read
- **Ptr writerAddr**. Pointer to the address to begin writing to after 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 parameter can be set for the PIP module:

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

PIP Object Properties

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

The following fields can be set for a pipe object:

- **comment**. Type a comment to identify this PIP object
- **bufseg**. The memory segment that the buffer is allocated within; all frames are allocated from a single contiguous buffer (of size framesize x numframes)
- **bufalign**. The alignment (in words) of the buffer allocated within the specified memory segment
- **framesize**. The length of each frame (in words)
- **numframes**. The number of frames
- **monitor**. The end of the pipe to be monitored by a hidden STS object. Can be set to reader, writer, or nothing. In the Statistics View 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 - DSP/BIOS Plug-ins 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 Void PIP_alloc(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* pipe to be allocated */

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 6-20, 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

Void PIP_free(PIP_Obj *pipe);

Parameters

PIP_Obj *pipe /* pipe to be freed */

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 6-55. The example for *HST_getpipe*, page 6-20, 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 Void PIP_get(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* pipe giving a frame */

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 6-55. The example for HST_getpipe, page 6-20, 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 Ptr PIP_getReaderAddr(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* address of the PIP object */

Return Value Ptr ra

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

```
Uns PIP_getReaderNumFrames(PIP_Obj *pipe);
```

Parameters

```
PIP_Obj    *pipe    /* address of the PIP object */
```

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 6-59.

PIP_getReaderSize *Get the number of words of data in a pipe frame*

C Interface

Syntax Uns PIP_getReaderSize(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* address of the PIP object */

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 6-59.

PIP_getWriterAddr

Get the value of the writerAddr pointer of the pipe

C Interface

Syntax Ptr PIP_getWriterAddr(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* address of the PIP object */

Return Value Ptr wa

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 6-59.

PIP_getWriterNumFrames

Get the number of pipe frames available to be written to

C Interface

Syntax Uns PIP_getWriterNumFrames(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* address of the PIP object */

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 6-59.

PIP_getWriterSize

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

C Interface

Syntax Uns PIP_getWriterSize(PIP_Obj *pipe);

Parameters PIP_Obj *pipe /* address of the PIP object */

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_put

Put a full frame into the pipe

C Interface

Syntax

```
Void PIP_put(PIP_Obj *pipe);
```

Parameters

```
PIP_Obj    *pipe       /* pipe accepting a frame */
```

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 6-55. The example for *HST_getpipe*, page 6-20, also uses a pipe with host channel objects.

See Also

PIP_alloc
PIP_free
PIP_get
HST_getpipe

PIP_setWriterSize

Set the number of valid words written to a pipe frame

C Interface

Syntax

```
Void PIP_setWriterSize(PIP_Obj *pipe, Uns size);
```

Parameters

```
PIP_Obj    *pipe       /* relevant pipe */
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 6-59.

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 PRD_swi SWI. 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 parameters can be set for the PRD module:

- **Object Memory**. The memory segment 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 instance fields 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 field is also displayed for each PRD object:

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

PRD - DSP/BIOS Plug-ins 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 PIP 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

LgUns PRD_getticks(Void);

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 6-7, 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 Void PRD_start(PRD_Obj *period);

Parameters PRD_Obj *prd /* periodic object *

Return Value Void

Assembly Interface

Syntax PRD_start

Preconditions ar2 = address of the PRD object

Postconditions none

Modifies c

Reentrant no

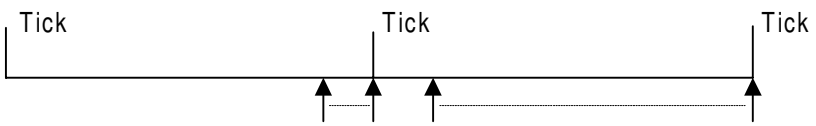
Description

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 that 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 Void PRD_stop(PRD_Obj *period);

Parameters PRD_Obj *prd /* periodic object */

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 Void PRD_tick(Void);

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

RTDX Module

Real-Time Data Exchange manager

Syntax

#include <rtdx.h>

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 segment used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this segment.
- **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 segment

For comprehensive information about RTDX, you can choose Help→Tools→RTDXclick .

RTDX_CreateInputChannel, RTDX_CreateOutputChannel

Declare channel structure

C Interface

Syntax

```
RTDX_CreateInputChannel( name );
RTDX_CreateOutputChannel( name );
```

Parameters

```
name    /* Label of the channel. */
```

Return Value

none

Description

These macros declare and initialize RTDX data channels for input and output, respectively.

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.

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 *TMS320C54x 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, RTDX_disableOutput, RTDX_enableInput, RTDX_enableOutput

Enable or disable a data channel

C Interface

Syntax

```
void RTDX_disableInput( RTDX_inputChannel *ichan );
void RTDX_disableOutput( RTDX_outputChannel *ochan );
void RTDX_enableInput( RTDX_inputChannel *ichan );
void RTDX_enableOutput( RTDX_outputChannel *ochan );
```

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 *TMS320C54x Optimizing C Compiler User's Guide* for more information.

Description

A call to an enable function causes the specified data channel to be enabled. Likewise, a call to a disable function causes the specified channel to be disabled.

See Also

RTDX_read
RTDX_write

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 the
                       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 *TMS320C54x 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_ERROR   /* Channel 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 *TMS320C54x 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 *TMS320C54x 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 *TMS320C54x 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

RTDX_isInputEnabled, RTDX_isOutputEnabled

Return status of the data channel

C Interface

Syntax

```
#include<rtdx.h>
RTDX_isInputEnabled( c );
RTDX_isOutputEnabled( c );
```

Parameter

```
c            /* Identifier for an input/output channel. */
```

Return Value

```
0            /* Not enabled. */
non-zero     /* Enabled. */
```

Description

The RTDX_isInputEnabled and RTDX_isOutputEnabled macros return the enabled status of a data channel.

STS Module

Statistics accumulator manager

Syntax

```
#include <sts.h>
```

Functions

```
struct STS_Obj {
   LgInt   num;    /* count */
   LgInt   acc;    /* total value */
   LgInt   max;    /* maximum value */
}
```

- **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

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 right-clicking on them. 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.

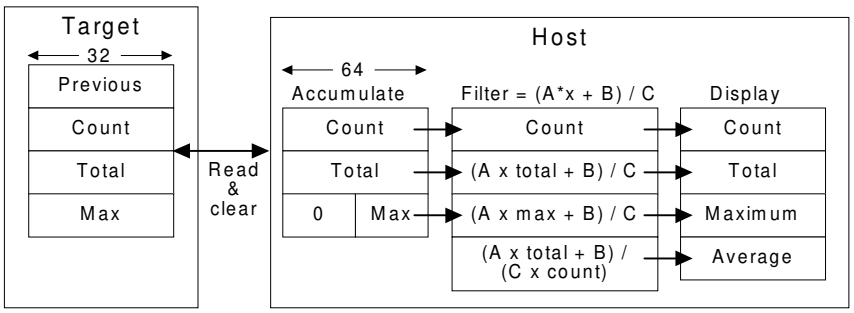
You can further customize the statistics data by setting the STS object properties to apply a printf format to the Total, Max, and Average fields in the Statistics View window and choosing a formula to apply to the data values on the host.

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 2^{32} - 1 samples), and the 32-bit sum of all samples.

These statistics are accumulated on the target in real time until the host reads and clears these values on the target. The host, however, continues to accumulate the values read from the target in a host buffer which is displayed by the Statistics View real-time analysis tool. Provided that the host reads and clears the target statistics objects faster than the target can overflow the 32-bit wide values being accumulated, no information loss occurs.

Using the Configuration Tool, you can select a Host Operation for an STS object. The statistics are filtered on the host using the operation and variables you specify. This figure shows the effects of the (A x X + B) / C operation.



STS Manager Properties

The following global parameter can be set for the STS module:

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

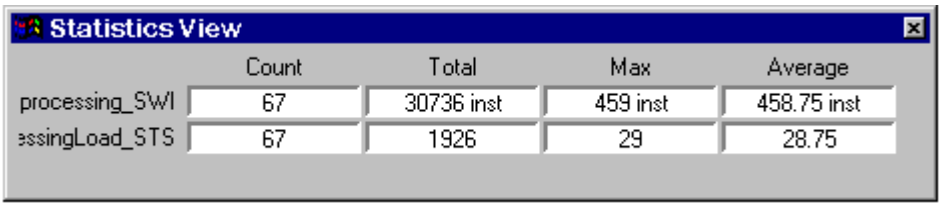
STS Object Properties

The following fields 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

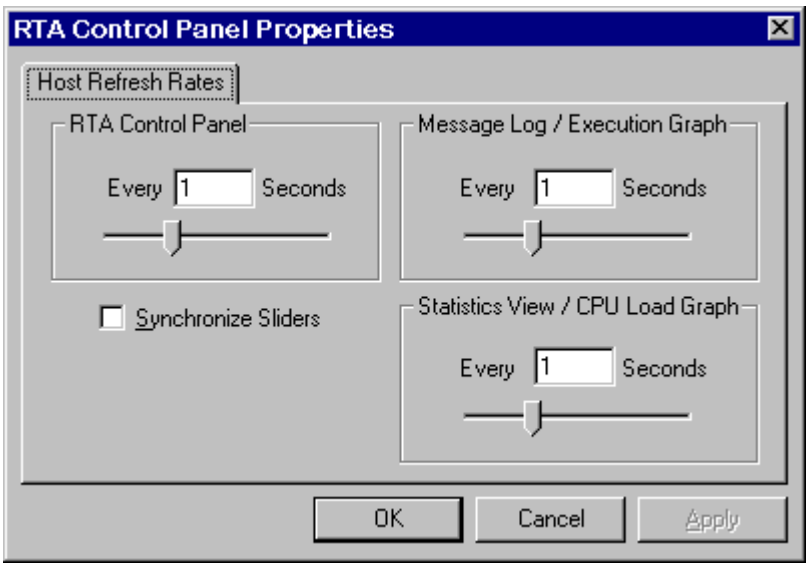
STS - Statistics View Interface

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

```
Void STS_add(STS_Obj *sts, LgInt value);
```

Parameters

```
STS_Obj  *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

Void STS_delta(STS_Obj *sts, LgInt value);

Parameters

STS_Obj *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.

```
STS_set(&sts, CLK_gethtime());
  "processing to be benchmarked"
STS_delta(&sts, CLK_gethtime());
```

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_getltime());
  "processing to be benchmarked"
STS_delta(&sts, CLK_getltime());
```

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 Void STS_reset(STS_Obj *sts);

Parameters STS_Obj *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

Void STS_set(STS_Obj *sts, LgInt value);

Parameters

```
STS_Obj  *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_dec**. Decrement SWI's mailbox value; post if becomes 0
- **SWI_disable**. Disable software interrupts
- **SWI_enable**. Enable software interrupts
- **SWI_getmbox**. Return a SWI's mailbox value
- **SWI_getpri**. Return a 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 interrupts
- **SWI_raisepri**. Raise a SWI's priority
- **SWI_restorepri**. Restore a SWI's priority
- **SWI_self**. Return address of currently executing SWI object

Description

The SWI module manages software interrupt service routines, which are patterned after HWI hardware interrupt service routines.

DSP/BIOS manages three distinct levels of execution threads: background idle functions, hardware interrupt service routines, and software interrupts. A software interrupt is an object that encapsulates a function to be executed and a priority. Software interrupts are prioritized, preempt background 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 of one priority preempts any lower priority software interrupt currently executing.

A target program uses an API call to post a SWI object. This causes the SWI module to schedule execution of the software interrupt's function. When a 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.

A software interrupt preempts any currently running software interrupt with a lower priority. Software interrupts can have up to 15 priority levels. 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.

Threads—including hardware interrupts, software interrupts, and background threads—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.

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 parameters can be set for the SWI module:

- **Object Memory**. The memory segment 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 fields can be set for a SWI object:

- **comment**. Type a comment to identify this SWI object
- **priority**. This field shows the numeric priority level for this SWI object. Priority levels range from 1 to 15, with 15 being the highest priority. Instead of typing a number in this field, you change the relative priority levels of SWI objects.
- **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 - DSP/BIOS Plug-ins Interface

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

Void SWI_andn(SWI_Obj *swi, Uns mask);

Parameters

```
SWI_Obj    *swi        /* SWI object */
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

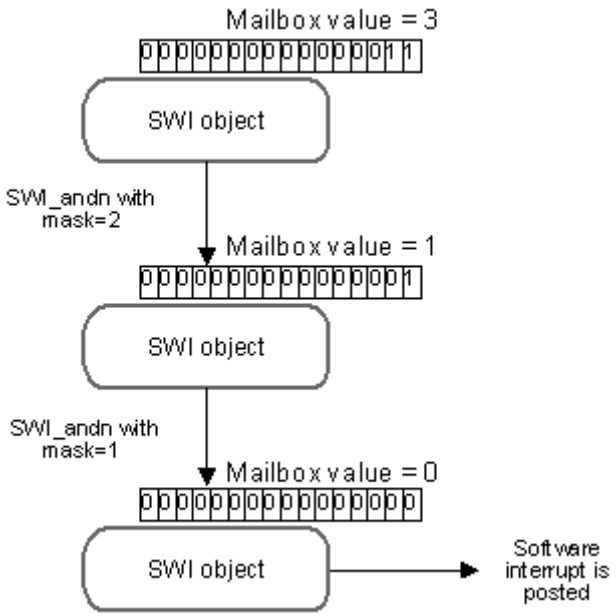
SWI_andn is used to conditionally post a software interrupt. SWI_andn clears the bits specified by a mask from SWI's internal mailbox. If SWI's mailbox becomes 0, SWI_andn posts the software interrupt. The bitwise logical operation performed is:

```
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_dec

Decrement SWI's mailbox value and post if mailbox becomes 0

C Interface

Syntax Void SWI_dec(SWI_Obj *swi);

Parameters SWI_Obj *swi /* SWI object */

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_andn
SWI_getmbox
SWI_inc
SWI_or
SWI_post
SWI_self

SWI_disable

Disable software interrupts

C Interface

Syntax Void SWI_disable(Void);

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 Void SWI_enable(Void);

Parameters Void

Return Value Void

Assembly Interface

Syntax SWI_enable

Preconditions can only be called if SWI_disable was called before
cpl = ovm = c16 = frct = cmpt = 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_getmbox

Return a SWI's mailbox value

C Interface

Syntax

Uns SWI_getmbox(Void);

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

al = 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 a 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_Obj *swi);
```

Parameters

```
SWI_Obj    *swi    /* SWI object */
```

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

```
Void SWI_inc(SWI_Obj *swi);
```

Parameters

```
SWI_Obj    *swi        /* SWI object */
```

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

```
Void SWI_or(SWI_Obj *swi, Uns mask);
```

Parameters

```
SWI_Obj   *swi      /* SWI object */
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

Void SWI_post(SWI_Obj *swi);

Parameters

SWI_Handle swi; /* software interrupt 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 a 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(Uns 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 will keep the SWI's priority for the rest of the processing at the raised level. A SWI_post of the SWI will post the SWI at its original priority level.

SWI_raisepri will never lower 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 Void SWI_restorepri(Uns 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

SWI_Obj *SWI_self(Void);

Parameters

Void

Return Value

SWI_Obj *swi /* currently executing SWI */

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

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_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_USER0 and TRC_USER1	Your program can use these bits to enable or disable sets of explicit instrumentation actions. You can use TRC_query to check the settings of these bits and either perform or omit instrumentation calls based on the result. DSP/BIOS does not use or set these bits.	off
TRC_GBLHOST	This bit must be set in order for any implicit instrumentation to be performed. Simultaneously starts or stops gathering of all enabled types of tracing. This can be important if you are trying to correlate events of different types. This bit is usually set at run time on the host in the RTA Control Panel.	off
TRC_GBLTARG	This bit must also be set 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

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.

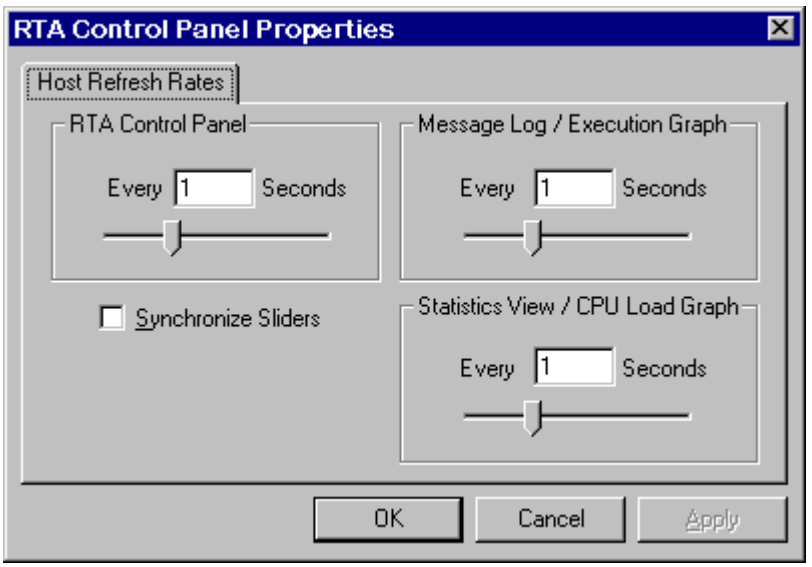
Your program can call the TRC_enable and TRC_disable operations to explicitly start and stop event logging or statistics accumulation in response to conditions encountered during real-time execution. This enables you to preserve the specific log or statistics information you need to see.

TRC - DSP/BIOS Plug-ins Interface

You can choose Tools→DSP/BIOS→RTA Control Panel to open a window that allows you to control run-time tracing.

Once you have enabled tracing, you can use Tools→DSP/BIOS→Execution Graph and Tools→DSP/BIOS→Message 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

Void TRC_disable(Uns 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 Void TRC_enable(Uns 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(Uns 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

Chapter 7

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.

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
```

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.

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 plugins. If you omit the -a flag, sectti displays only the program sections that are loaded on the target.

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.

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y