TI SYS/BIOS v6.35 Real-time Operating System

User's Guide

Contents

List of Figures

List of Tables

Read This First

About This Manual

This manual describes the TI SYS/BIOS Real-time Operating System, which is also called "SYS/BIOS". The latest version number as of the publication of this manual is SYS/BIOS 6.35. Versions of SYS/BIOS prior to 6.30 were called DSP/BIOS. The new name reflects that this operating system can also be use on processors other than DSPs.

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

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface. Examples use a bold version of the special typeface for emphasis.

  Here is a sample program listing:

  ```
  #include <xdc/runtime/System.h>
  int main(){
      System_printf("Hello World!\n");
      return (0);
  }
  ```

- 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

See the detailed list and links in Section 1.6.

Related Documentation

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

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

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

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

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

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

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

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, Code Composer, Code Composer Studio, DSP/BIOS, SPOX, TMS320, TMS320C54x, TMS320C55x, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C28x, TMS320C5000, TMS320C6000 and TMS320C2000.

Windows is a registered trademark of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

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

February 7, 2013

About SYS/BIOS

This chapter provides an overview of SYS/BIOS and describes its relationship to XDCtools.

1.1 What is SYS/BIOS?

SYS/BIOS is a scalable real-time kernel. It is designed to be used by applications that require real-time scheduling and synchronization or real-time instrumentation. SYS/BIOS provides preemptive multi-threading, hardware abstraction, real-time analysis, and configuration tools. SYS/BIOS is designed to minimize memory and CPU requirements on the target. See the video introducing SYS/BIOS for an overview.

SYS/BIOS provides the following benefits:

- All SYS/BIOS objects can be configured statically or dynamically.

- To minimize memory size, the APIs are modularized so that only those APIs that are used by the program need to be bound into the executable program. In addition, statically-configured objects reduce code size by eliminating the need to include object creation calls.

- Error checking and debug instrumentation is configurable and can be completely removed from production code versions to maximize performance and minimize memory size.

- Almost all system calls provide deterministic performance to enable applications to reliably meet real-time deadlines.

- To improve performance, instrumentation data (such as logs and traces) is formatted on the host.

- The threading model provides thread types for a variety of situations. Hardware interrupts, software interrupts, tasks, idle functions, and periodic functions are all supported. You can control the priorities and blocking characteristics of threads through your choice of thread types.

- Structures to support communication and synchronization between threads are provided. These include semaphores, mailboxes, events, gates, and variable-length messaging.

- Dynamic memory management services offering both variable-sized and fixed-sized block allocation.

- An interrupt dispatcher handles low-level context save/restore operations and enables interrupt service routines to be written entirely in C.

- System services support the enabling/disabling of interrupts and the plugging of interrupt vectors, including multiplexing interrupt vectors onto multiple sources.

1.2 How is SYS/BIOS Different from DSP/BIOS?

This book describes SYS/BIOS 6.x, a release that introduced significant changes from DSP/BIOS 5.x. If you have used previous versions, you will encounter these major changes to basic functionality:

- The name DSP/BIOS has been changed to SYS/BIOS to reflect the fact that it can be used on processors other than DSPs.

- SYS/BIOS uses the configuration technology in XDCtools. For more information, see Section 1.3 of this book.

- The APIs have changed. A compatibility layer ensures that DSP/BIOS 5.4x or earlier applications will work unmodified. However, note that the PIP module is no longer supported. For details, see the *Migrating a DSP/BIOS 5 Application to SYS/BIOS 6* (SPRAAS7A) application note.

In addition, significant enhancements have been made in the areas that include the following:

- Up to 32 priority levels are available for both tasks and software interrupt (Swi) threads.

- A new timer module is provided that enables applications to configure and use timers directly rather than have time-driven events limited to using the system tick.

- All kernel objects may be created statically or dynamically.

- An additional heap manager, called HeapMultiBuf, enables fast, deterministic variable-sized memory allocation performance that does not degrade regardless of memory fragmentation.

- A more flexible memory manager supports the use of multiple, concurrent heaps and enables developers to easily add custom heaps.

- A new Event object enables tasks to pend on multiple events, including semaphores, mailboxes, message queues, and user-defined events.

- An additional Gate object supports priority inheritance.

- Hook functions are supported for hardware and software interrupt objects as well as tasks.

- An option is provided to build the operating system with parameter checking APIs that assert if invalid parameter values are passed to a system call.

- A standardized method allows SYS/BIOS APIs to handle errors, based on an error block approach. This enables errors to be handled efficiently without requiring the application to catch return codes. In addition, you can easily have the application halted whenever a SYS/BIOS error occurs, because all errors now pass through a single handler.

- The instrumentation tools support both dynamically and statically-created tasks.

- More powerful logging functions include a timestamp, up to 6 words per log entry, and the ability for logging events to span more than one log if additional storage is required.

- Per-task CPU load is now supported in addition to total CPU load.

1.3 How are SYS/BIOS and XDCtools Related?

XDCtools is a separate software component provided by Texas Instruments that provides the underlying tooling needed by SYS/BIOS. You must have both XDCtools and SYS/BIOS installed in order to use SYS/BIOS. The SYS/BIOS release notes in the top-level SYS/BIOS installation directory provide information about the versions of XDCtools that are compatible with your version of SYS/BIOS. Typically, when you install a new version of SYS/BIOS, you will also need to install a new version of XDCtools.

XDCtools is important to SYS/BIOS users because:

- XDCtools provides the technology that users use to configure the SYS/BIOS and XDCtools modules used by the application. See Section 1.3.2.

- XDCtools provides the tools used to build the configuration file. This build step generates source code files that are then compiled and linked with your application code. See Section 1.3.2.

- XDCtools provides a number of modules and runtime APIs that SYS/BIOS leverages for memory allocation, logging, system control, and more. See Section 1.3.3.

XDCtools is sometimes referred to as "RTSC" (pronounced "rit-see"—Real Time Software Components), which is the name for the open-source project within the Eclipse.org ecosystem for providing reusable software components (called "packages") for use in embedded systems. For documentation about XDCtools modules, see the online help within CCS. For information about packaging of reusable software components and details about the tooling portion of XDCtools, see the RTSC-pedia web site.

1.3.1 *SYS/BIOS as a Set of Packages*

SYS/BIOS and XDCtools are sets of "packages," each of which delivers a subset of the product's functionality. XDCtools uses a naming convention for packages to aid readability and to ensure that packages delivered from different sources don't have namespace collisions that will pose problems for the system integrator. If you are familiar with the Java package naming convention, you will find it to be quite similar.

SYS/BIOS packages conform to this convention with names that consist of a hierarchical naming pattern; each level is separated by a period ("."). Usually, the highest level of the name is the vendor ("ti"), followed by the product ("sysbios"), and then followed by the module and submodule names (for example, "knl").

These names have the added benefit of reflecting the physical layout of the package within the file system where SYS/BIOS has been installed. For example, the ti.sysbios.knl package files can be found in the following folder:

```
BIOS_INSTALL_DIR\bios_6_3#_##\packages\ti\sysbios\knl
```

See Section 1.4 for a partial list of the packages provided by SYS/BIOS and Section 1.3.3 for a partial list of the modules provided by XDCtools.

You can picture the architecture of the tools used to create applications as shown in the following figure. The xdc.runtime package provided by XDCtools contains modules and APIs your application can use along with the modules and APIs in SYS/BIOS.
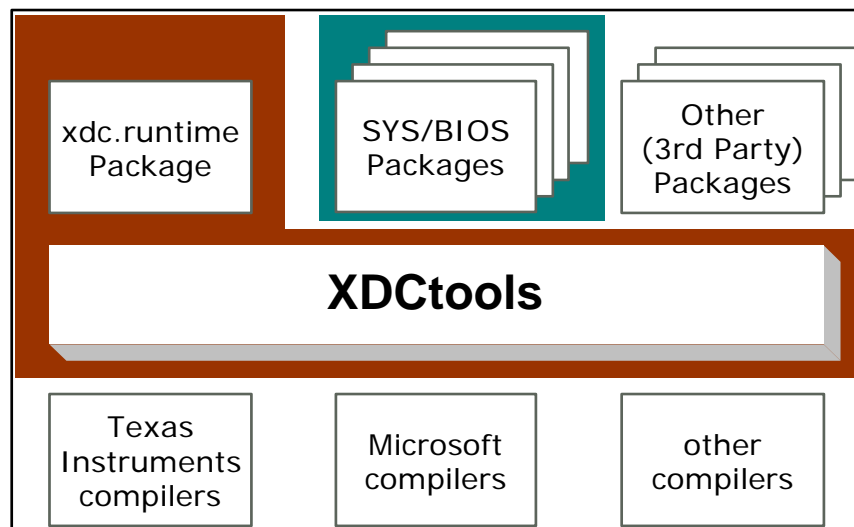
1.3.2 *Configuring SYS/BIOS Using XDCtools*

Configuration is an essential part of using SYS/BIOS and is used for the following purposes:

- It specifies the modules and packages that will be used by the application.

- It can statically create objects for the modules that are used by the application.

- It validates the set of modules used explicitly and implicitly to make sure they are compatible.

- It statically sets parameters for the system, modules, and objects to change their runtime behavior.

An application's configuration is stored in one or more script files with a file extension of *.cfg. These are parsed by XDCtools to generate corresponding C source code, C header, and linker command files that are then compiled and linked into the end application. The following diagram depicts a build flow for a typical SYS/BIOS application.
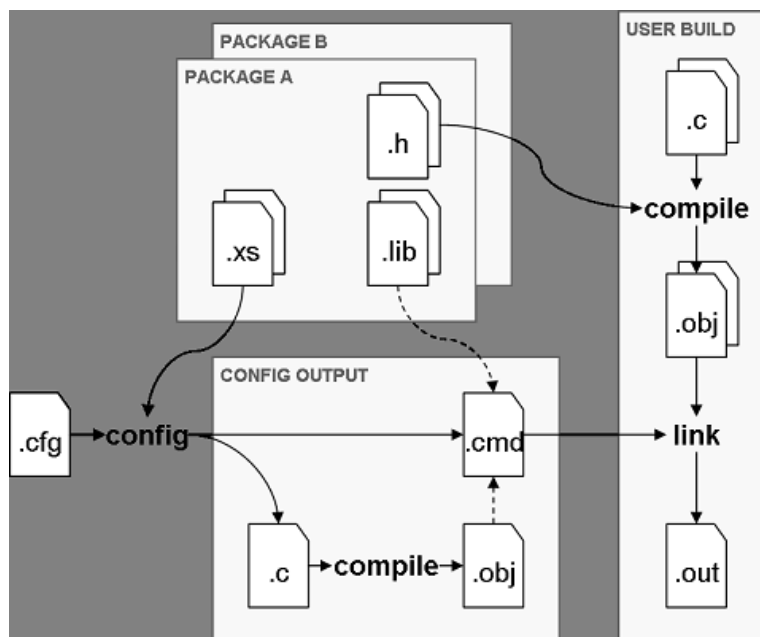
The configuration .cfg file uses simple JavaScript syntax to set properties and call methods provided by objects. The combination of JavaScript and the script objects provided by XDCtools is referred to as an XDCscript.

You can create and modify a configuration file in two different ways:

- Writing the textual .cfg file directly with a text editor or the XDCscript Editor in CCS.

- Using the visual configuration tool (XGCONF) embedded in CCS.

The following figure shows the XGCONF configuration tool in CCS being used to configure a static SYS/BIOS Task instance. You can see this configuration for yourself in the "Static Example" SYS/BIOS project template in CCS. See Section 2.1.2 for how to create a CCS project using a SYS/BIOS template.
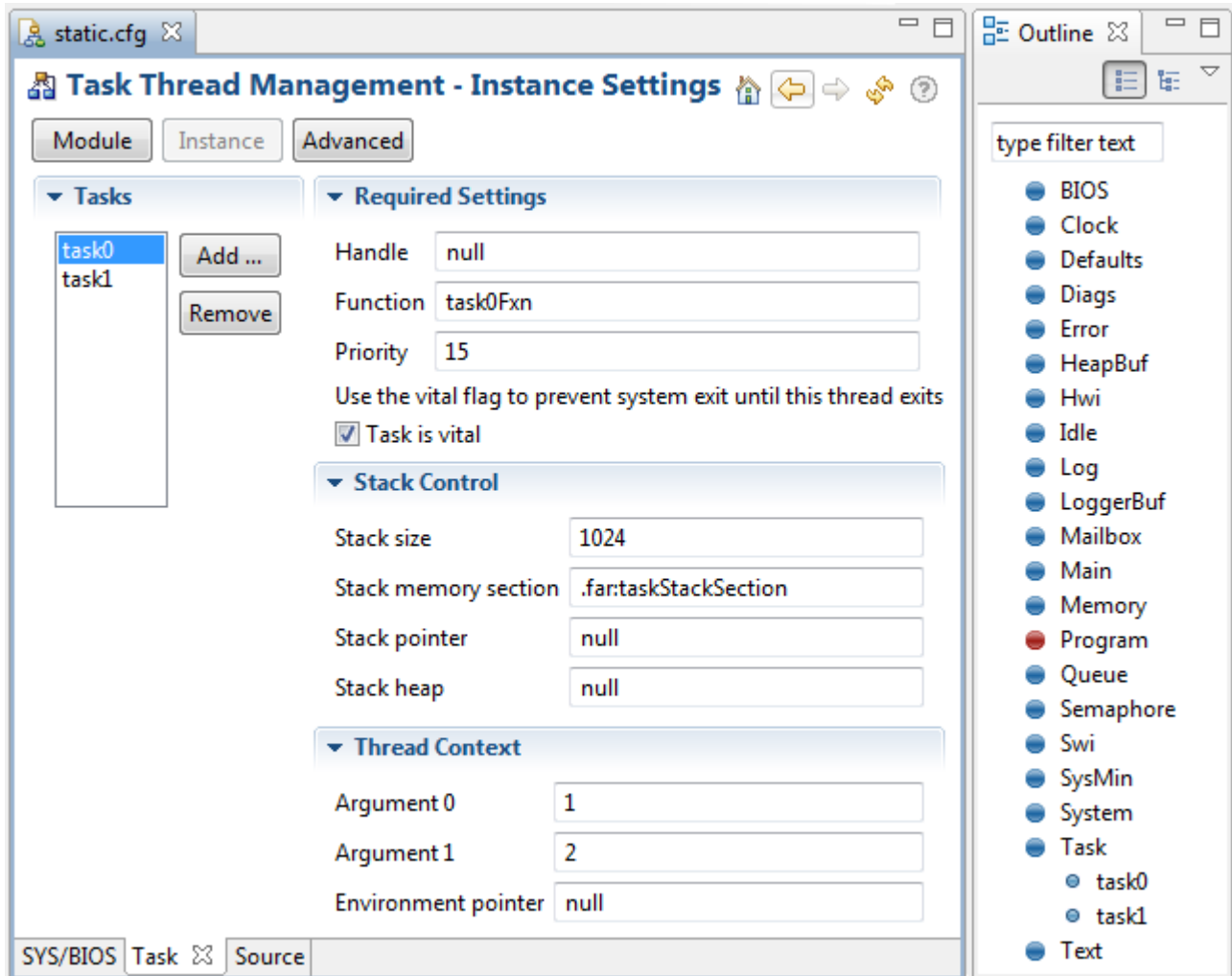
The Task instance named "task0" set up in the configuration tool corresponds to the following XDCscript code:

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
Task.numPriorities = 16;
Task.idleTaskStackSize = 1024;

var tskParams = new Task.Params;
tskParams.arg0 = 1;
tskParams.arg1 = 2;
tskParams.priority = 15;
tskParams.stack = null;
tskParams.stackSize = 1024;
var task0 = Task.create('&task0Fxn', tskParams);
```

1.3.3   XDCtools Modules and Runtime APIs

XDCtools contains several modules that provide basic system services your SYS/BIOS application will need to operate successfully. Most of these modules are located in the xdc.runtime package in XDCtools. By default, all SYS/BIOS applications automatically add the xdc.runtime package during build time.

The functionality provided by XDCtools for use in your C code and configuration file can be roughly divided into four categories. The modules listed in the following table are in the xdc.runtime package, unless otherwise noted.

Table 1–1. XDCtools Modules Using in C Code and Configuration

Category	Modules	Description
System Services	System	Basic low-level "system" services. For example, character output, printf-like output, and exit handling. See Section 8.3. Proxies that plug into this module include xdc.runtime.SysMin and xdc.runtime.SysStd. See Section D.4.
	Startup	Allows functions defined by different modules to be run before main(). See Section 3.1.
	Defaults	Sets event logging, assertion checking, and memory use options for all modules for which you do not explicitly set a value. See Section 6.7.1 and Section 8.5.1.1.
	Main	Sets event logging and assertion checking options that apply to your application code.
	Program	Sets options for runtime memory sizes, program build options, and memory sections and segments. This module is used as the "root" for the configuration object model. This module is in the xdc.cfg package. See Section 3.3.1 and Section 6.3.2.
Memory Management	Memory	Creates/frees memory heaps statically or dynamically. SeeSection 6.7.3.
Diagnostics	Log and Loggers	Allows events to be logged and then passes those events to a Log handler. Proxies that plug into this module include xdc.runtime.LoggerBuf and xdc.runtime.LoggerSys. See Section 8.2.1 and Section 3.5.4.
	Error	Allows raising, checking, and handling errors defined by any modules. See Section 8.3.
	Diags	Allows diagnostics to be enabled/disabled at either configuration- or runtime on a per-module basis. See Section 8.5.1.
	Timestamp and Providers	Provides time-stamping APIs that forward calls to a platform-specific time stamper (or one provided by CCS). See Section 5.4,
	Text	Provides string management services to minimize the string data required on the target. See Section D.2.5.
Synchronization	Gate	Protects against concurrent access to critical data structures. See Section 4.3.
	Sync	Provides basic synchronization between threads using wait() and signal() functions. See Section 9.5.

1.4 SYS/BIOS Packages

SYS/BIOS provides the following packages:

Table 1–2. Packages and Modules Provided by SYS/BIOS

Package	Description
ti.sysbios.benchmarks	Contains specifications for benchmark tests. Provides no modules, APIs, or configuration. See Appendix B.
ti.sysbios.family.*	Contains specifications for target/device-specific functions. See Section 7.5.
ti.sysbios.gates	Contains several implementations of the IGateProvider interface for use in various situations. These include GateHwi, GateSwi, GateTask, GateMutex, and GateMutexPri. See Section 4.3.
ti.sysbios.hal	Contains Hwi, Timer, and Cache modules. See Section 7.2, Section 7.3, and Section 7.4.
ti.sysbios.heaps	Provides several implementations of the XDCtools IHeap interface. These include HeapBuf (fixed-size buffers), HeapMem (variable-sized buffers), and HeapMultiBuf (multiple fixed-size buffers). See Chapter 6.
ti.sysbios.interfaces	Contains interfaces for modules to be implemented, for example, on a device or platform basis.
ti.sysbios.io	Contains modules for performing input/output actions and interacting with peripheral drivers. Chapter 9.
ti.sysbios.knl	Contains modules for the SYS/BIOS kernel, including Swi, Task, Idle, and Clock. See Chapter 3 and Chapter 5. Also contains modules related to inter-process communication: Event, Mailbox, and Semaphore. See Chapter 4.
ti.sysbios.utils	Contains the Load module, which provides global CPU load as well as thread-specific load. See Section 8.2.

1.5 Using C++ with SYS/BIOS

SYS/BIOS applications can be written in C or C++. An understanding of several issues regarding C++ and SYS/BIOS can help to make C++ application development proceed smoothly. These issues concern memory management, name mangling, calling class methods from configured properties, and special considerations for class constructors and destructors.

SYS/BIOS provides an example that is written in C++. The example code is in the bigtime.cpp file in the packages\ti\sysbios\examples\generic\bigtime directory of the SYS/BIOS installation.

1.5.1 Memory Management

The functions new and delete are the C++ operators for dynamic memory allocation and deallocation. For TI targets, these operators use malloc() and free(). SYS/BIOS provides reentrant versions of malloc() and free() that internally use the xdc.runtime.Memory module and (by default) the ti.sysbios.heaps.HeapMem module.

1.5.2   *Name Mangling*

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is referred to as name mangling.

Name mangling could potentially interfere with a SYS/BIOS application since you use function names within the configuration to refer to functions declared in your C++ source files. To prevent name mangling and thus to make your functions recognizable within the configuration, it is necessary to declare your functions in an extern C block as shown in the following code fragment from the bigtime.cpp example:

```
/*
 * Extern "C" block to prevent name mangling
 * of functions called within the Configuration Tool
 */
extern "C" {
    /* Wrapper functions to call Clock::tick() */
    void clockTask(Clock clock);
    void clockPrd(Clock clock);
    void clockIdle(void);
} // end extern "C"
```

This extern C block allows you to refer to the functions within the configuration file. For example, if you have a Task object that should run clockTask() every time the Task runs, you could configure a Task as follows:

```
var task0Params = new Task.Params();
task0Params.instance.name = "task0";
task0Params.arg0 = $externPtr("cl3");
Program.global.task0 = Task.create("&clockTask", task0Params);
```

Notice that in the configuration example above, the arg0 parameter of the Task is set to `$externPtr("cl3")`. The C++ code to create a global clock object for this argument is as follows:

```
/* Global clock objects */
Clock cl3(3);   /* task clock */
```

Functions declared within the extern C block are not subject to name mangling. Since function overloading is accomplished through name mangling, function overloading has limitations for functions that are called from the configuration. Only one version of an overloaded function can appear within the extern C block. The code in the following example would result in an error.

```
extern "C" {                              // Example causes ERROR
    Int addNums(Int x, Int y);
    Int addNums(Int x, Int y, Int z); // error, only one version
                                      // of addNums is allowed
}
```

While you can use name overloading in your SYS/BIOS C++ applications, only one version of the overloaded function can be called from the configuration.

Default parameters is a C++ feature that is not available for functions called from the configuration. C++ allows you to specify default values for formal parameters within the function declaration. However, a function called from the configuration must provide parameter values. If no values are specified, the actual parameter values are undefined.

1.5.3 Calling Class Methods from the Configuration

Often, the function that you want to reference within the configuration is the member function of a class object. It is not possible to call these member functions directly from the configuration, but it is possible to accomplish the same action through wrapper functions. By writing a wrapper function which accepts a class instance as a parameter, you can invoke the class member function from within the wrapper.

A wrapper function for a class method is shown in the following code fragment from the bigtime.cpp example:

```
/*
 * ======== clockPrd ========
 * Wrapper function for PRD objects calling
 * Clock::tick()
 */
void clockPrd(Clock clock)
{
    clock.tick();
    return;
}
```

Any additional parameters that the class method requires can be passed to the wrapper function.

1.5.4 Class Constructors and Destructors

Any time that a C++ class object is instantiated, the class constructor executes. Likewise, any time that a class object is deleted, the class destructor is called. Therefore, when writing constructors and destructors, you should consider the times at which the functions are expected to execute and tailor them accordingly. It is important to consider what type of thread will be running when the class constructor or destructor is invoked.

Various guidelines apply to which SYS/BIOS API functions can be called from different SYS/BIOS threads (tasks, software interrupts, and hardware interrupts). For example, memory allocation APIs such as Memory_alloc() and Memory_calloc() cannot be called from within the context of a software interrupt. Thus, if a particular class is instantiated by a software interrupt, its constructor must avoid performing memory allocation.

Similarly, it is important to keep in mind the time at which a class destructor is expected to run. Not only does a class destructor execute when an object is explicitly deleted, but also when a local object goes out of scope. You need to be aware of what type of thread is executing when the class destructor is called and make only those SYS/BIOS API calls that are appropriate for that thread. For further information on function callability, see the CDOC online documentation.

1.6 For More Information

You can read the following additional documents to learn more about SYS/BIOS, XDCtools, and Code Composer Studio:

- **SYS/BIOS**
  - *SYS/BIOS Release Notes.* Located in the top-level SYS/BIOS installation directory, or choose **Help > Help Contents** in CCS and expand the **SYS/BIOS** item.
  - *SYS/BIOS Getting Started Guide.* In *<bios_install_dir>*/docs/Bios_Getting_Started_Guide.pdf
  - *SYS/BIOS API Reference* (also called "CDOC"). Run *<bios_install_dir>*/docs/cdoc/index.html, or choose **Help > Help Contents** in CCS and expand the **SYS/BIOS** item. See Section 1.6.1.
  - *Migrating a DSP/BIOS 5 Application to SYS/BIOS 6* (SPRAAS7). In *<bios_install_dir>*/docs/ Bios_Legacy_App_Note.pdf, or choose **Help > Help Contents** in CCS and expand the **SYS/BIOS** item. Also available on the Texas Instruments website.
  - SYS/BIOS main page on TI Embedded Processors Wiki contains links to many SYS/BIOS resources.
  - BIOS forum on TI's E2E Community lets you submit your questions.
  - SYS/BIOS 6.x Product Folder on TI.com
  - Embedded Software Download Page
- **XDCtools**
  - *XDCtools API Reference* (also called "CDOC"). Run *<xdc_install_dir>*/docs/xdctools.chm, or choose **Help > Help Contents** in CCS and expand the **XDCtools** item. See Section 1.6.1.
  - RTSC-Pedia Wiki
  - BIOS forum on TI's E2E Community
  - Embedded Software Download Page
- **Code Composer Studio (CCS)**
  - *CCS online help.* Choose **Help > Help Contents** in CCS.
  - CCSv5 on TI Embedded Processors Wiki
  - Code Composer forum on TI's E2E Community

1.6.1 *Using the API Reference Help System*

The API Reference help for SYS/BIOS and XDCtools is called "CDOC".

You view the online help for SYS/BIOS and XDCtools from within the CCS online help system. Choose **Help > Help Contents** in CCS to open this system. Then, expand the **SYS/BIOS** or **XDCtools** item to see the documentation provided. Select the **API reference** item to go to the reference help.

You can also open the online help for SYS/BIOS and XDCtools on Microsoft Windows systems as follows:

- To open the online help for SYS/BIOS, you can choose **SYS/BIOS API Documentation** from the **Texas Instruments > SYS/BIOS** group in the Windows **Start** menu.

- To open online help for XDCtools, you can choose **XDCtools Documentation** from the **Texas Instruments > XDCtools** group in the Windows **Start** menu.

Click "+" next to a repository to expand its list of packages. Click "+" next to a package name to see the list of modules it provides. You can further expand the tree to see a list of the functions provided by a module. Double-click on a package or module to see its reference information.

The SYS/BIOS API documentation is within the "sysbios" package. To view API documentation on memory allocation, logs, timestamps, asserts, and system, expand the "xdc.runtime" and "xdc.runtime.knl" packages. The "bios" package contains only the compatibility modules for earlier versions of SYS/BIOS.

Each reference page is divided into two main sections:

- **C Reference.** This section has blue table borders. It begins with a table of the APIs you can call from your application's C code. It also lists C structures, typedefs, and constants that are defined by including this module's header file. A description of the module's use follows; often this includes a table of the calling contexts from which each API can be called. Detailed syntax for the functions, typedefs, structures, and constants follows.

- **XDCscript Reference.** This section has red table borders. You can jump to this section in any topic by clicking the **XDCscript usage** link near the top of the page. This section provides information you can use when you are configuring the application in the *.cfg file (either using XGCONF or editing the source code for the configuration file directly). This section lists the types, structures, and constants defined by the module. It also lists parameters you can configure on a module-wide basis, and parameters that apply to individual instances you create.
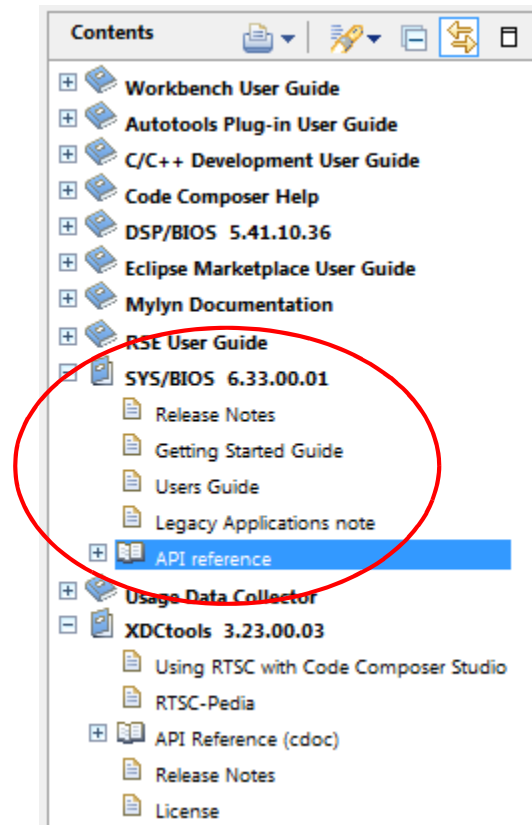
SYS/BIOS Configuration and Building

This chapter describes how to configure and build SYS/BIOS applications.

2.1 Creating a SYS/BIOS Project

You can use Code Composer Studio (CCS) to create example projects that use SYS/BIOS in two ways:

- **TI Resource Explorer.** Use this window, which opens when you start CCS, to create example projects with all the settings for your specific device. Follow the instructions in Section 2.1.1. This technique works with CCSv5.3 and higher. **It is not supported with CCSv5.2.**

- **CCS New Project Wizard.** Choose **File > New > CCS Project** from the menu bar and follow the instructions in Section 2.1.2. You will need to choose settings for your device and platform. This technique works with CCSv5.2 and higher.

Versions of CCS prior to v5.2 are not supported by SYS/BIOS 6.34 and higher.

2.1.1 Creating a SYS/BIOS Project with the TI Resource Explorer

Follow these steps to use the TI Resource Explorer in CCSv5.3 or higher to create a project that uses SYS/BIOS. (If you are using CCSv5.2, see Section 2.1.2.)

1. Open CCS.

2. If you do not see the TI Resource Explorer area, make sure you are in the CCS Edit perspective and choose **View > TI Resource Explorer** from the menus.

3. Expand the SYS/BIOS item in the tree to show **SYS/BIOS >** *family > board*, where *family > board* is your platform.
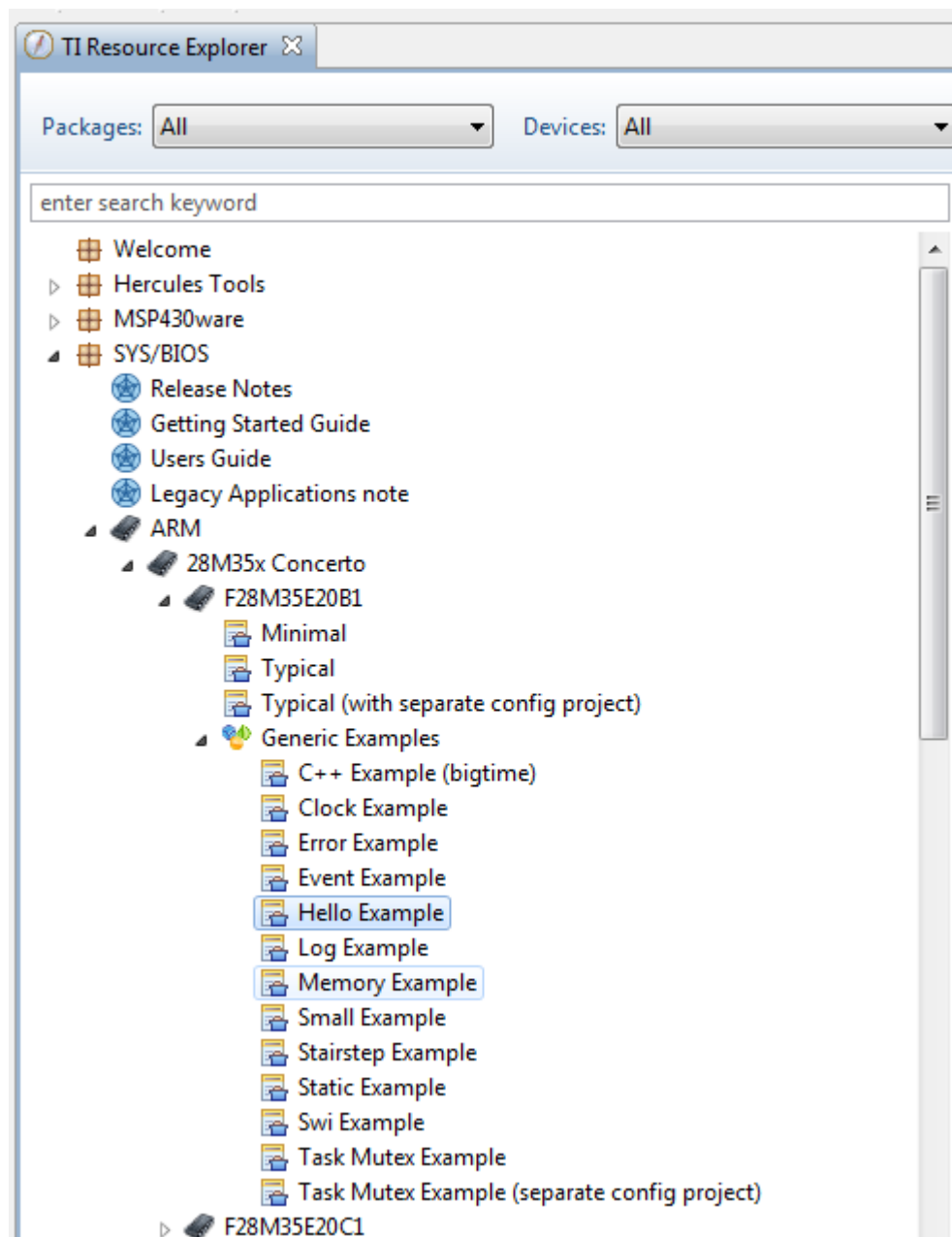
Select the example you want to create. A description of the selected example is shown at the top of the page to the right of the example list. To get started with SYS/BIOS, you can choose one of the Generic Examples, such as the **Log Example** or **Task Mutex Example**.

When you are ready to create your own application project, you might choose the "Minimal" or "Typical" example depending on how memory-limited your target is. For some device families, device-specific SYS/BIOS templates are also provided. (If you have other software components that use SYS/BIOS, such as IPC, you can choose a template provided for that component.)

4. Click the **Step 1** link in the right pane of the TI Resource Explorer to **Import the example project into CCS**. This adds a new project to your Project Explorer view.

Step 1:   Import the example project into CCS

5. The project created will have a name with the format *<example_name>_<board>*.You can expand the project to view or change the source code and configuration file.

6. The page shown when you select an example in the TI Resource Explorer provides additional links to perform common actions with that example.

7. Use the **Step 2** link when you are ready to build the project. If you want to change any build options, right click on the project and select **Properties** from the context menu. For example, you can change compiler, linker, and RTSC (XDCtools) options.

Step 2:   Build the imported project

8. Use the **Step 3** link to change the connection used to communicate with the board. The current setting is shown in the TI Resource Explorer page for the selected example. (If you want to use a simulator instead of a hardware connection, double-click the *.ccxml file in the project to open the Target Configuration File editor. Change the **Connection** as needed, and click **Save**.)

Step 3:   Debugger Configuration

9. Use the **Step 4** link to launch a debug session for the project and switch to the CCS Debug Perspective.

Step 4:   Debug the imported project

2.1.2 Creating a SYS/BIOS Project with the New Project Wizard

Follow these steps to use CCSv5.2 or higher to create a project that uses SYS/BIOS.

1. Open CCS and choose **File > New > CCS Project** from the menu bar.

2. In the New CCS Project dialog, type a **Project name**. For example, to begin creating a project using "hello world" example code provided with SYS/BIOS, you can type "hello". The default project location is a folder with the same name as the project in your current workspace.
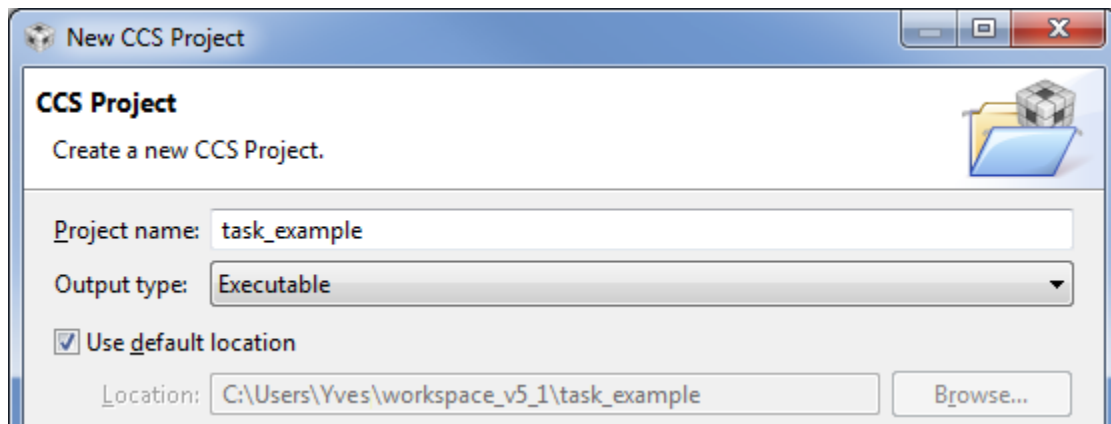
3. In the **Family** drop-down field, select your platform type. For example, you might select "C2000" or "C6000".

4. In the **Variant** row, select or type a filter on the left. This shortens the list of device variants in the right drop-down field. Then, select the actual device you are using. For example, you might select "Generic devices" in the filter field and "Generic C64x+ Device" in the second field.

5. In the **Connection** drop-down field, select how you connect to the device. The choices depend on the device you selected; typically you can choose the Data Snapshot Viewer, a simulator, or an emulator.
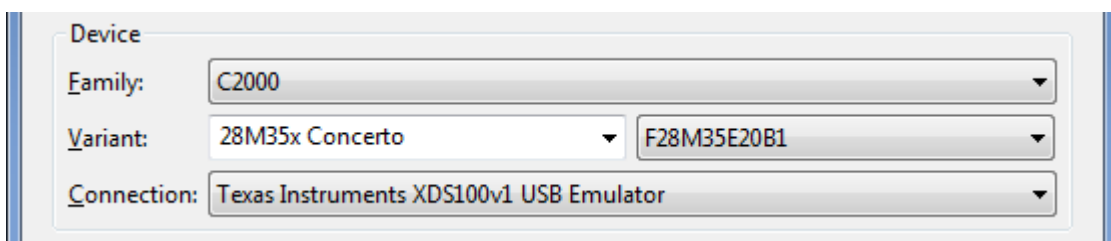
6. If you need to use a non-default setting for the device endianness, the TI Code Generation Tools version, the output format (COFF or ELF), or the Runtime support library, click the arrow next to the **Advanced settings** label to display fields for those settings. Typically, you will not need to do this.
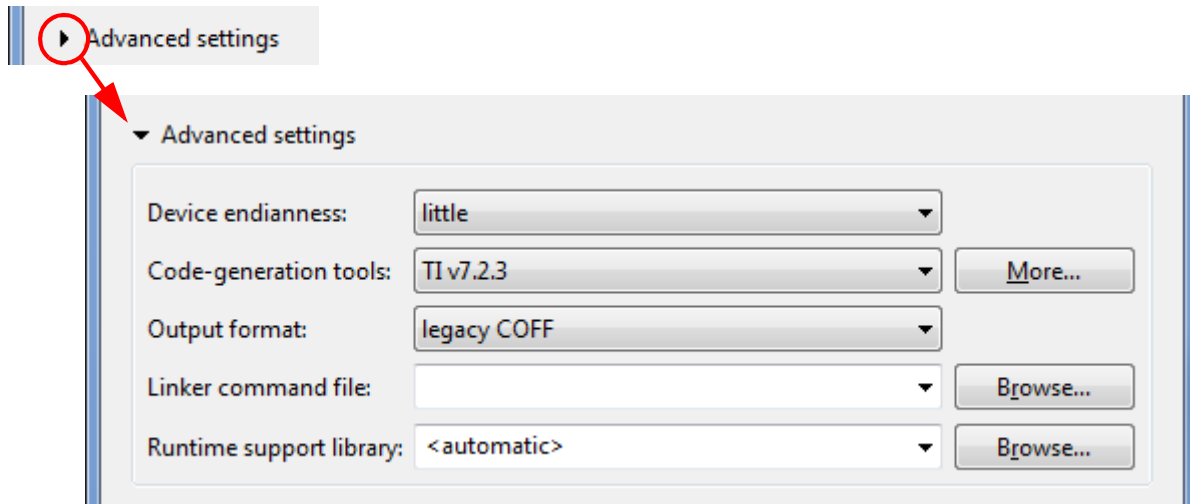
Note:	You should not specify your own linker command file when you are getting started using SYS/BIOS. A linker command file will be created and used automatically when you build the project. When you have learned more about SYS/BIOS, you can add your own linker command file, but must make sure it does not conflict with the one created by SYS/BIOS.

7. In the **Project templates** area, scroll down to the **SYS/BIOS** item and expand the list of templates. A description of the template you highlight is shown to the right of the template list.

    To get started with SYS/BIOS, you can choose one of the Generic Examples, such as the **Log Example** or **Task Mutex Example**.

    When you are ready to create your own application project, you might choose the "Minimal" or "Typical" example depending on how memory-limited your target is. For some device families, device-specific SYS/BIOS templates are also provided. (If you have other software components that use SYS/BIOS, such as IPC, you can choose a template provided for that component.)

8. Click **Next** to move to the RTSC Configuration Settings page. (RTSC is another term for the XDCtools component used by SYS/BIOS to build the configuration for the device platform you select.)
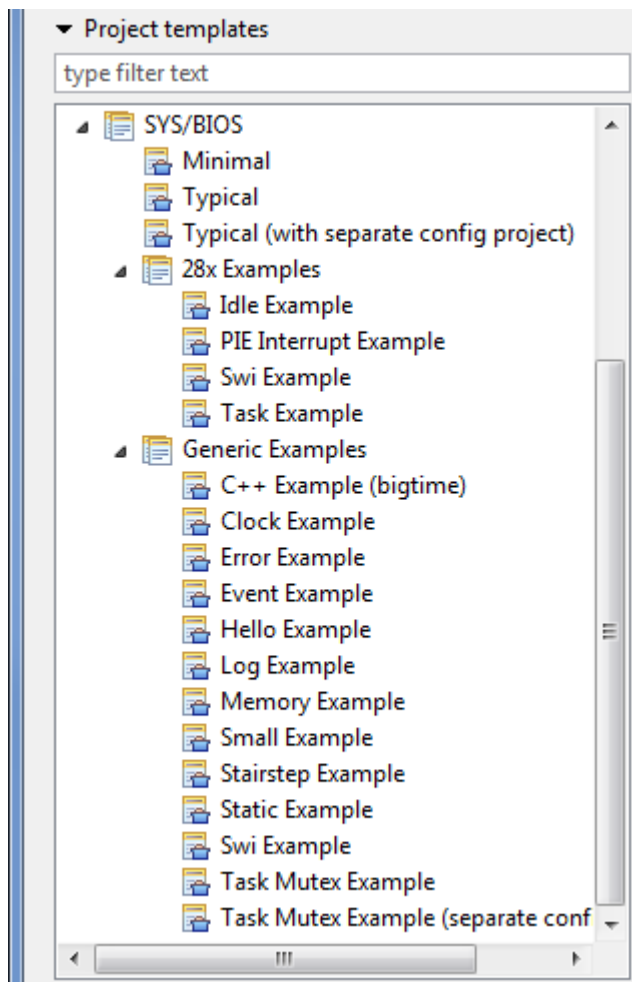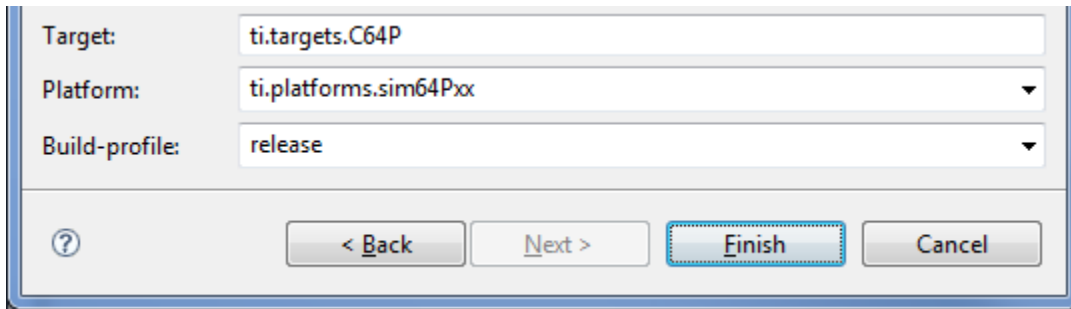
    Note: Do not click **Finish** at this point.

9. On the "RTSC Configuration Settings" page, make sure the versions of XDCtools, SYS/BIOS, and any other components you want to use are selected. By default, the most recent versions are selected.
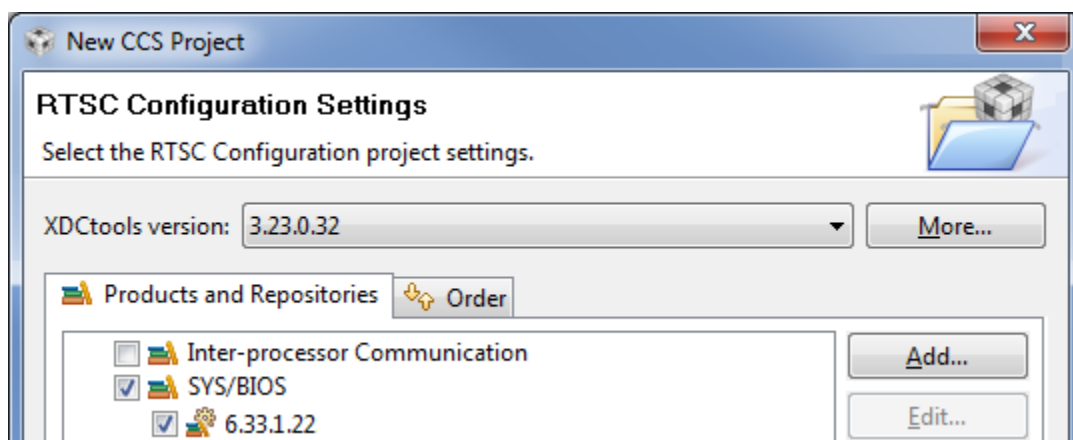
10. The **Target** setting is based on device settings you made on earlier pages, and should not need to be changed.

11. If the **Platform** has not been filled in automatically, click the drop-down arrow next to the field. CCS scans the available products for platforms that match your device settings. Click on the list and choose the platform you want to use.

12. The **Build-profile** field determines which libraries the application will link with. We recommend that you use the "release" setting even when you are creating and debugging an application.

Target:	ti.targets.C64P	
Platform:	ti.platforms.sim64Pxx	▼
Build-profile:	release	▼
⑦	< *B*ack *N*ext > **Finish** Cancel	

13. Click **Finish** to create a new project and add it to the C/C++ Projects list.

2.1.3   *Adding SYS/BIOS Support to a Project*

If you created a project using the TI Resource Explorer or the New Project wizard, a configuration file is automatically added to your project and SYS/BIOS support is automatically enabled.

Note: Applications that can use SYS/BIOS are referred to as having RTSC support enabled. RTSC is Real-Time Software Components, which is implemented by the XDCtools component. See Section 1.3 for details.

If you start with an empty CCS project template, you can add a configuration file for use with SYS/BIOS to your CCS project by choosing **File > New > RTSC Configuration File**. If the project does not have RTSC support enabled, you will be asked if you want to enable RTSC support for the current project. Click **Yes**.

2.1.4   *Creating a Separate Configuration Project*

If you want to save the configuration file in a separate project, for example so that multiple applications can use the same configuration, you can create two separate projects:

- C source code project
- Configuration project

The configuration project can be referenced by the main project, which causes it to be automatically built when you build the main project. To create two projects with this reference already set up, select the **SYS/BIOS > Typical (with separate config project)** template when creating a new CCS project.

2.2 Configuring SYS/BIOS Applications

You configure SYS/BIOS applications by modifying the *.cfg configuration file in the project. These files are written in the XDCscript language, which is a superset of JavaScript. While you can edit this file with a text editor, CCS provides a graphical configuration editor called XGCONF.

XGCONF is useful because it gives you an easy way to view the available options and your current configuration. Since modules and instances are activated behind-the-scenes when the configuration is processed, XGCONF is a useful tool for viewing the effects of these internal actions.

For example, the following figure shows the XGCONF configuration tool in Code Composer Studio used to configure a static SYS/BIOS Swi (software interrupt) instance.
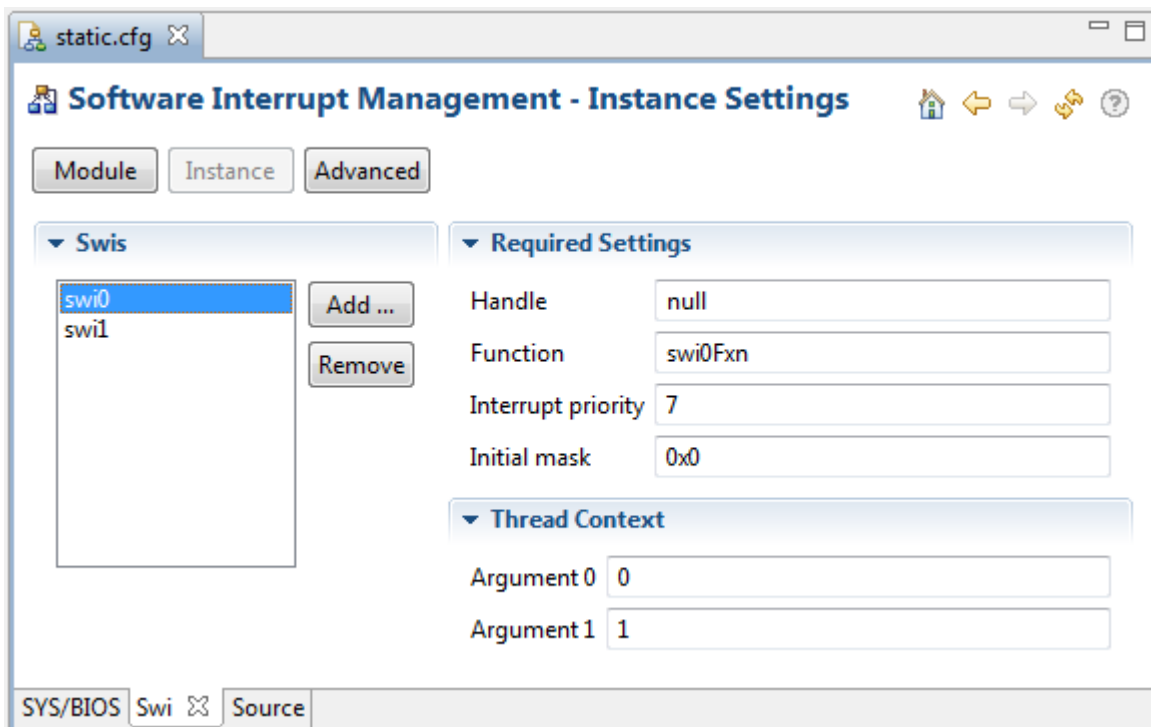
The Source tab shows that the code to create this Swi would be as follows:

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');

/* Create a Swi Instance and manipulate its instance parameters. */
var swiParams = new Swi.Params;
swiParams.arg0 = 0;
swiParams.arg1 = 1;
swiParams.priority = 7;
Program.global.swi0 = Swi.create('&swi0Fxn', swiParams);

/* Create another Swi Instance using the default instance parameters */
Program.global.swi1 = Swi.create('&swi1Fxn');
```

2.2.1 *Opening a Configuration File with XGCONF*

To open XGCONF, follow these steps:

1. Make sure you are in the **C/C++** perspective of CCS. If you are not in that perspective, click the C/C++ icon to switch back.

2. Double-click on the *.cfg configuration file in the **Project Explorer** tree. While XGCONF is opening, the CCS status bar shows that the configuration is being processed and validated. (If your project does not yet contain a *.cfg file, see Section 2.1.3.)

3. When XGCONF opens, you see the **Welcome** sheet for SYS/BIOS. This sheet provides links to SYS/BIOS documentation resources.

4. Click the **System Overview** button to see a handy overview of the main modules you can use in SYS/BIOS applications (see page 2–37).
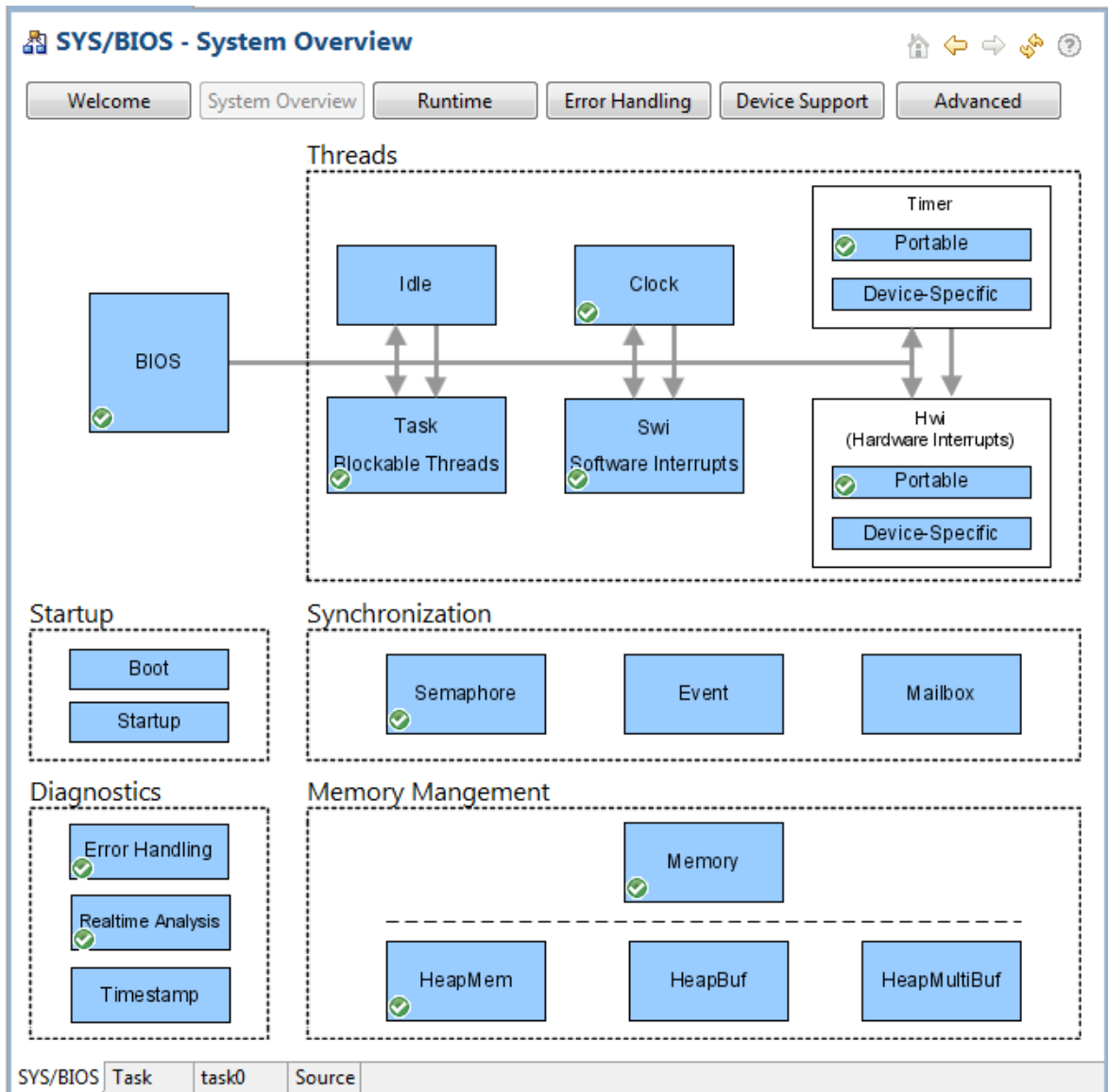
> **Note:** If the configuration is shown in a text editor instead of XGCONF, right-click on the .cfg file and choose **Open With > XGCONF**.

You can open multiple configuration files at the same time. However, using XGCONF with several configuration files is resource intensive and may slow down your system.

2.2.2 Performing Tasks with XGCONF

The following list shows the configuration tasks you can perform with XGCONF and provides links to explain how:

- Make more modules available. See page 2–35.
- Find a module. See page 2–34 and page 2–35.
- Add a module to the configuration. See page 2–34.
- Delete a module from the configuration. See page 2–35.
- Add an instance to the configuration. See page 2–35.
- Delete an instance from the configuration. See page 2–35.
- Change property values for a module. See page 2–36.
- Change property values for an instance. See page 2–36.
- Get help about a module. You can right-click in most places in XGCONF for and choose **Help** to get information about a specific module or property. See page 2–35 and page 2–36.
- Configuring the memory map and section placement. The configuration file allows you to specify which sections and heaps are used by various SYS/BIOS modules, but not their placement on the target. Memory mapping and section placement is described in Chapter 6.
- Save the configuration or revert to the last saved file. See page 2–32.
- Fix errors in the configuration. See page 2–40.

2.2.3 Saving the Configuration

If you have modified the configuration, you can press Ctrl+S to save the file. Or, choose **File > Save** from the CCS menu bar.

In the Source tab of a configuration, you can right-click and choose to **Revert File** to reload the last saved configuration file or **Save** to save the current configuration to a file.

See Section 2.2.9 for information about the validation checks performed when you save a configuration with XGCONF.

2.2.4   About the XGCONF views

The XGCONF tool is made up of several panes that are used together:
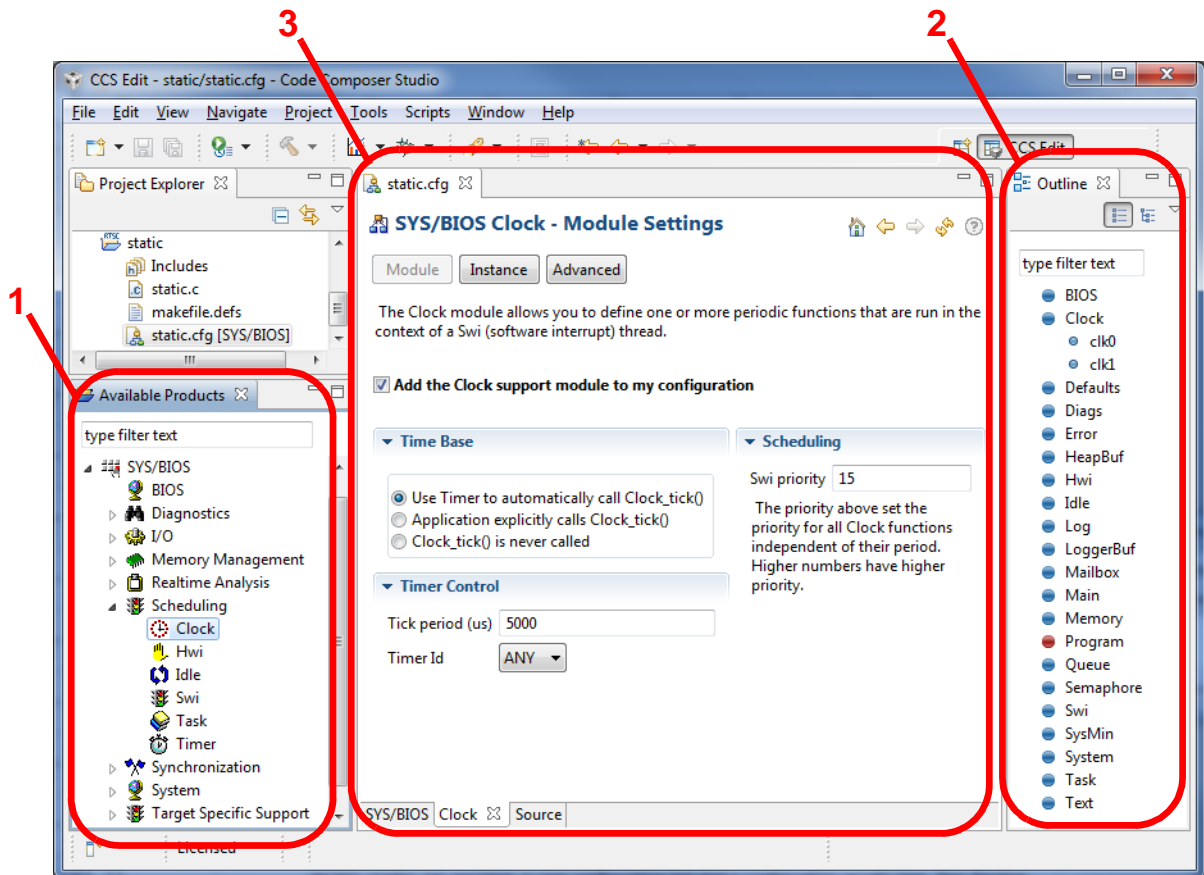
1. **Available Products view** lets you add modules to your configuration. By default, this view appears in the lower-left corner of the window. See Section 2.2.5.

2. **Outline view** shows modules used in your current configuration and lets you choose the module to display in the Properties view. By default, this view appears on the right side of the window. Two display modes are provided: a user configuration list and a configuration results tree. See Section 2.2.6.

3. **Property view** shows the property settings of the selected module or instance and lets you make changes. You can also use the Source tab in the Property view to modify the script directly. See Section 2.2.7.

4. **Problems view** appears if errors and warnings are detected in the configuration during validation. See Section 2.2.8.

2.2.5 *Using the Available Products View*

The Available Products view lists the packages and modules available for use in your configuration. It list both modules you are already using and modules you can add to your configuration. The list is organized first by the software component that contains the modules and then using functional categories.

Modules you can configure are listed in this tree. Modules that do not apply to your target or are only used internally are hidden in this tree.

Finding Modules

To find a particular module, you can expand the tree to see the modules. If you don't know where the module is located or there are several modules with similar names, type some text in the "type filter text" box. For example, you can type "gate" to find all the Gate implementations in XDCtools, SYS/BIOS, and any other repositories. You can use * and ? as wildcard characters.

If you want to look for a module using its full path within the repository, right-click and choose **Show Repositories**. After the category-based tree, you will see an **All Repositories** node. You can expand this node to find particular modules. For example, the full path to the SYS/BIOS Task module is ti.sysbios.knl.Task.

Note that if you turn on **Show Repositories**, all modules are listed. This includes modules that do not apply to your target family and some modules (often shown as red balls) that you cannot add to the configuration.

Adding Modules and Instances to the Configuration

To start using a module, right-click and choose **Use <module>**. For example, choosing **Use Swi** adds the ability to create and configure software interrupts to your application. You can also drag modules from the Available Products view to the Outline view to add them to the configuration.

When you select a module in the Available Products view, you see the properties you can set for that module in the Property view (whether you are using it yet or not). When you add use of a module to the configuration, that module is shown in the Outline view.

You can get help on a particular module by right-clicking on the module name and choosing **Help** from the pop-up menu.

Adding a module to the configuration causes an xdc.useModule() statement to be added to the configuration script.
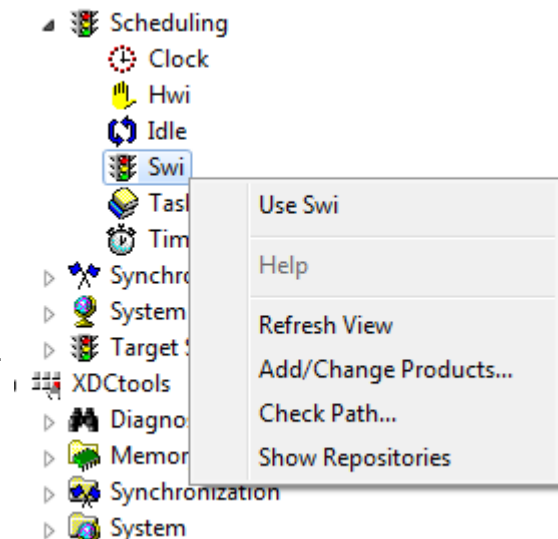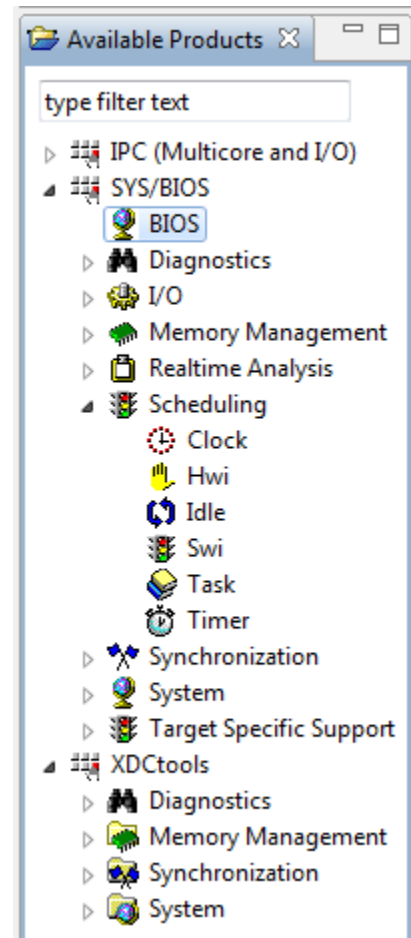
Managing the Available Products List

When you open a configuration file with XGCONF, the package repositories that your application is set to use in the Properties dialog are scanned for modules and the results are listed here.

You can add or remove products by right-clicking and choosing **Add/Change Products**. (This opens the dialog you see by choosing **Project > Properties** from the CCS menus, then choosing the **CCS General** category and the **RTSC** tab.) Check boxes next to versions of products you want to be able to use. If a product isn't listed, click the **Add** button to browse for a package repository in the file system. When you click **OK**, the Available Products list is refreshed.

You can open the Package Repository Path Browser by right-clicking and choosing **Check Path**. This tool lists all the repositories and packages on the package path, shows the effects of adding repositories or changing the order of locations in the path, and sorts the packages by various fields.

If there is a problem with the display or you have changed something in the file system that should affect the modules shown, you can right-click and choose **Refresh View**.

2.2.6 *Using the Outline View*
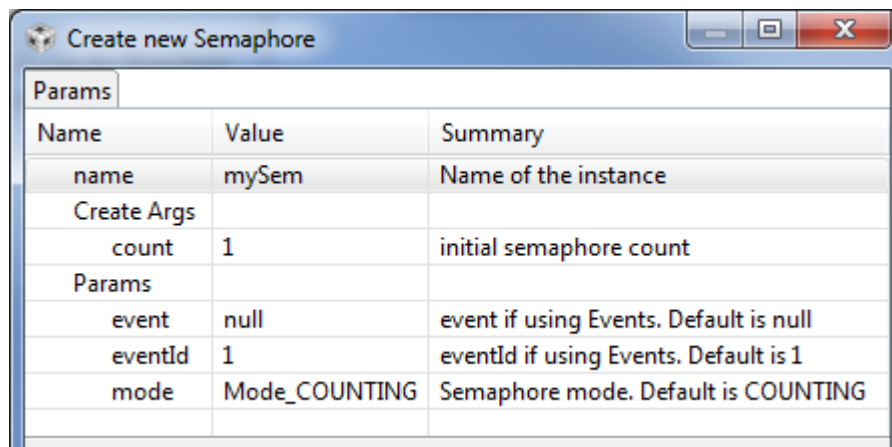
The Outline view shows modules and instances that are available for configuration in your *.cfg file. You can view the Outline in two ways:

- **Show User Configuration.** Select the ⬚ icon. This is the easier-to-use view. This view mode shows a flat list of only those modules directly referenced in the *.cfg file and instances created in the *.cfg file. You can use this view to add instances of modules and delete the use of a module use from the configuration.

- **Show Configuration Results.** Select the ⬚ icon. This is the more advanced view. This mode shows a tree view of all modules and instances that are used both implicitly (behind the scenes) and explicitly (because they are referenced directly in the *.cfg file). You can edit any module that does not have the "locked" icon. You can "unlock" some locked modules by choosing to add them to your configuration. Instances that are shown as "locked" are used internally, and you should not attempt to modify them.

As in the Available Products view, you can type filter text at the top of the Outline view to find modules and instances by name.

To create an instance, right-click on a module and choose **New <module>**. For example, **New Semaphore**. Notice that not all modules let you create instance objects. You see a dialog that lets you specify values you want to use for the properties of that instance. You can type or select values in the Value column.

If you want to delete an instance from the configuration, right-click on that instance in the Outline view, and select **Delete <name>** from the menu.

When you select a module or instance in the Outline view, you see the properties you can set for that module in the Property view.

You can get help on a particular module by right-clicking on the module name and choosing **Help** from the pop-up menu. Help on SYS/BIOS and XDCtools configuration is in the **red** (XDCscript) section of the online documentation. For each module, the configuration help follows the **blue** sections that document that module's C APIs.

Some modules have a red ball next to them, while others have a blue ball. The 🔵 blue ball indicates that this is a target module, which provides code or data that can be referenced at runtime on the embedded target. The 🔴 red ball indicates that this is a meta-only module, which exists in the configuration but does not directly exist on the target.

To stop using a module in the configuration, right-click on that module in the Outline view, and select **Stop Using <module>** from the menu. Deleting a module from the configuration removes the corresponding xdc.useModule() statement and any instances of the module from the configuration script. If deleting the module would result in an invalid script, an error message is displayed and the module is not deleted. This can happen if a different module or instance refers to it in the script, for example in an assignment to a proxy or a configuration parameter.

2.2.7   *Using the Property View*

If you select a module or instance in the Outline view or Available Products view, the Property view shows properties for the selected item. There are several ways to view the properties.

- **System Overview.** This sheet provides a block diagram overview of the modules in SYS/BIOS. See page 2–37.

- **Module, Instance, or Basic.** This layout organizes the properties visually. See page 2–38.

- **Advanced.** This layout provides a tabular list of property names and lets you set values in the table. See page 2–39.

- **Source.** The source editor lets you edit the configuration script using a text editor. See page 2–39.

All the property sheets you have viewed are accessible from the tabs at the bottom of the Property view.
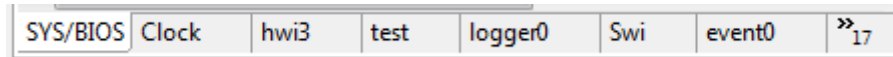
You can use the 🏠 ⇦ ⇨ arrow buttons in the upper-right of the Property view to move through sheets you have already viewed. The Home icon returns you to the BIOS module System Overview.

For numeric fields, you can right-click on a field and choose **Set Radix** to choose whether to display this field as a decimal or hex value.

Point to a field with your mouse for brief information about a property. Right-click on a field and choose **Help** to jump directly to the documentation for that property. Click the ⑦ **Help** icon to get documentation for the current module. Help on SYS/BIOS and XDCtools configuration is in the **red** (XDCscript) section of the online documentation. For each module, the configuration help follows the **blue** sections that document that module's C APIs.

System Overview Block Diagram

The System Overview shows all of the core modules in SYS/BIOS as blocks. A green checkmark shows the modules that are currently used by your configuration. You can add other modules in the diagram to your configuration by right-clicking on the block and choosing **Use**. You can configure any module by clicking on it.

To return to the System Overview from some other property sheet, select the BIOS module in the Outline View and click the **System Overview** button.
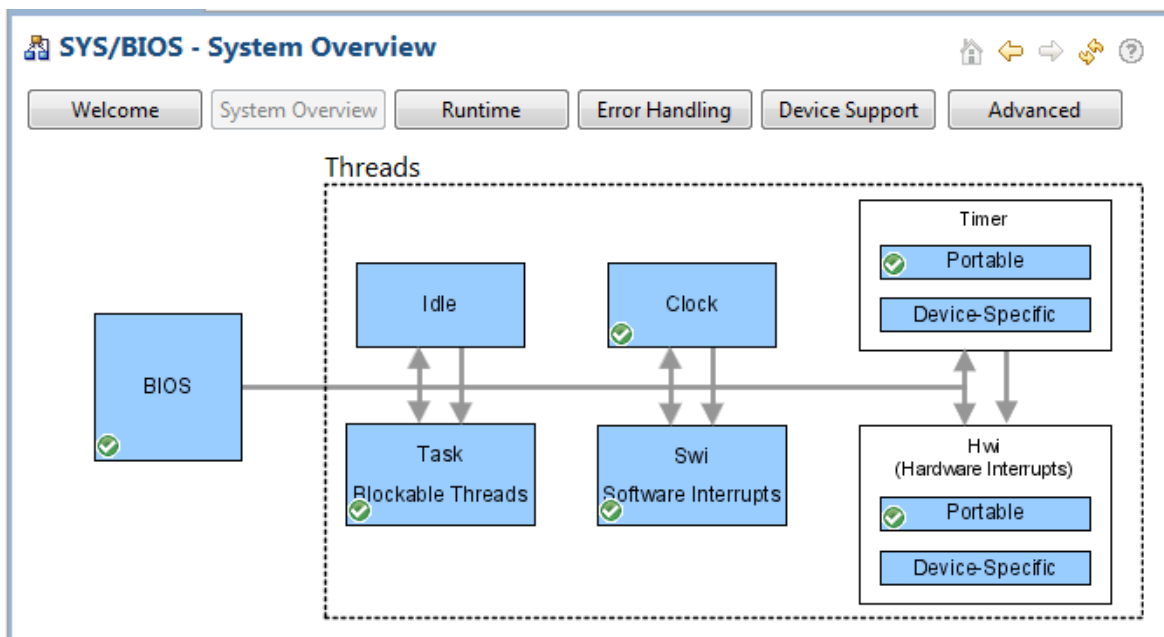
You can add object instances to the configuration by right-clicking on a module and choosing the **New** command.

Module and Instance Property Sheets

The **Module** and **Instance** property sheets organize properties into categories and provides brief descriptions of some properties. Checkboxes, selection fields, and text fields are provided depending on the type of values a property can have.
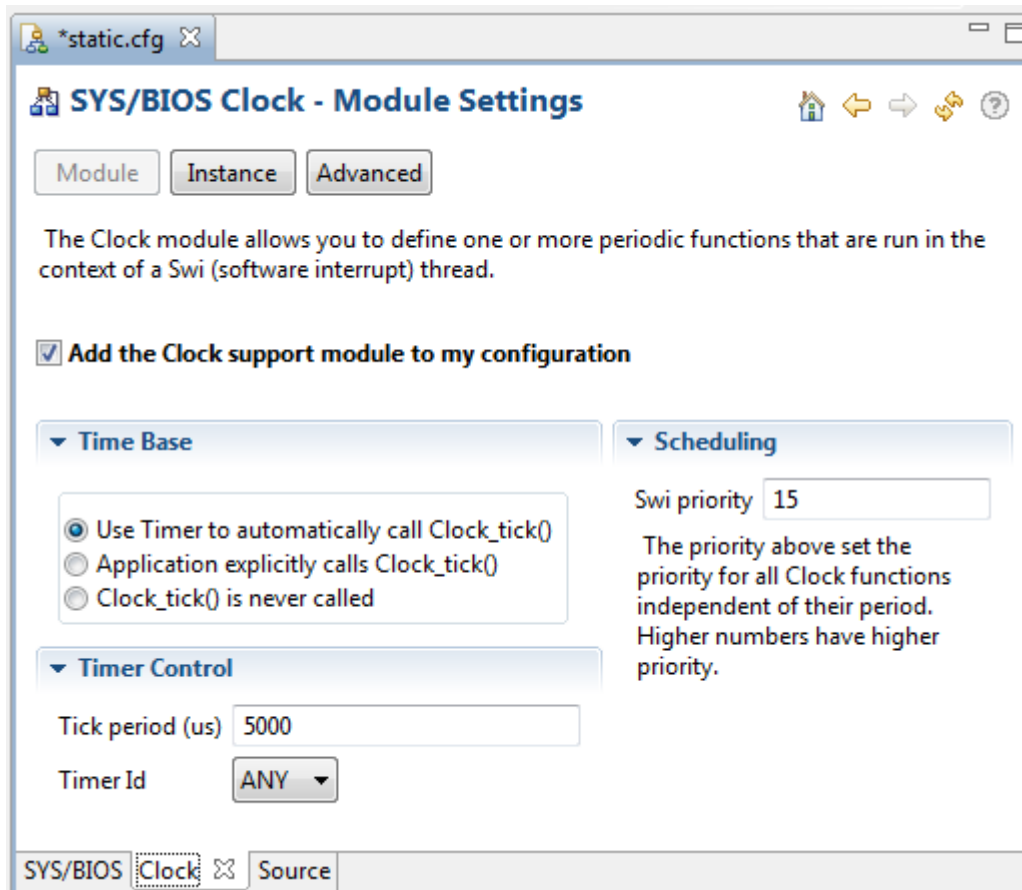
Click the **Module** button to see and modify global properties for the module. In the Module property sheet for optional modules, you can uncheck the **Add <module> to my configuration** box to stop using a module. If you aren't using a module, you can add it to your configuration by checking the box.

Click the **Instance** button to see properties for instances. A list on the left side of the page shows the instances that have been created and lets you add or remove instances.

Some advanced properties aren't shown in the **Module** and **Instance** sheets. For these, see the **Advanced** sheet (page 2–39).

As you change properties, they are applied to the configuration and validated. However, they are not saved until you save the configuration file.

You can fold up or reopen portions of the property sheet by clicking the arrow next to a section heading.

Advanced Properties Sheet

The **Advanced** layout provides a tabular list of property names and lets you set values in the table.
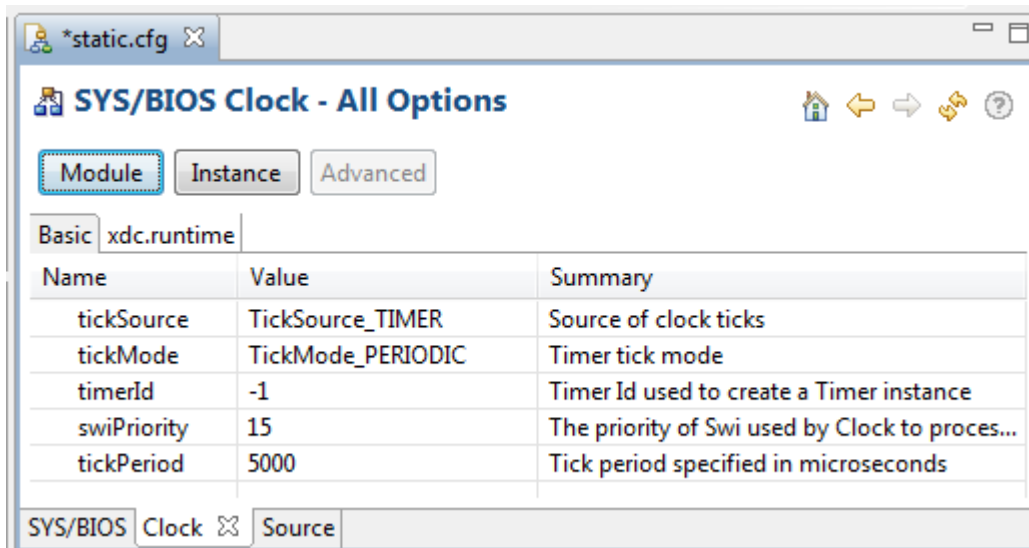
To modify the value of a property, click on a row in the Value column and type or select a new value.

When you type a value for a property, XGCONF checks to make sure the type of the value matches the type expected for the property. This is separate from the more extensive validation checks that are performed when you save a configuration.

For many modules, the Advanced layout has both a **Basic** tab and an **xdc.runtime** tab. The Basic tab shows the same properties as the Basic view, but in tabular form. The xdc.runtime tab shows <module>.common$ properties inherited by the module. For example, these include properties related to the memory used by the module and any instances, the diagnostics settings for the module, and the Gate object used by this module if any. In addition, events, asserts, and errors that can occur for this module are listed. See the online documentation for xdc.runtime.Types for more about the common$ properties.

Point to a field with your mouse for brief information about a property. Right-click on a field and choose **Help** to jump directly to the documentation for that property. Click the ⑦ **Help** icon to get documentation for the current module.

Source Editor

The source editor lets you edit the configuration script using a text editor by choosing the Source tab. Some advanced scripting features of XDCscript are available only by editing the script directly. For more information see links from the RTSC-pedia http://rtsc.eclipse.org/docs-tip/RTSC_Scripting_Primer page.

You can use Ctrl+S to save any changes you make. You can right-click on the file and choose **Undo Typing** to undo the most recent graphical editing operation, or **Revert File** to return to the most recently saved version of this file. When you save the file, it is validated and the other panes are refreshed to reflect your changes.

When you select a module or instance in the Outline view, the Source tab highlights all the lines of the configuration file that are related to that module or instance.

2.2.8 Using the Problems View

The **Problems** view lists all errors and warnings detected during a build or validation of the configuration script. This view is automatically displayed whenever new errors or warnings are detected.
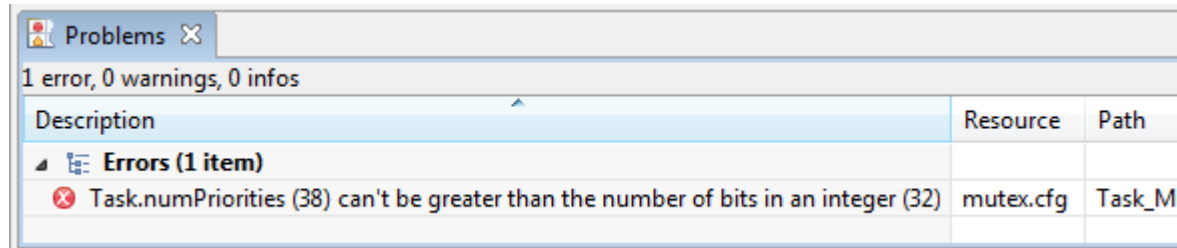
The Outline view shows an ⬛ error icon next to any modules or instances for which problems were detected. If you select such a module, the Properties view shows a red X icon next to the properties that are incorrectly set.

If you double-click on an item in the Problems view while the Source tab is displayed, the Source tab highlights the statement that caused the error or warning, and an error icon is shown in the left margin. Position indicators for any problems also appear to the right of the scrollbar.
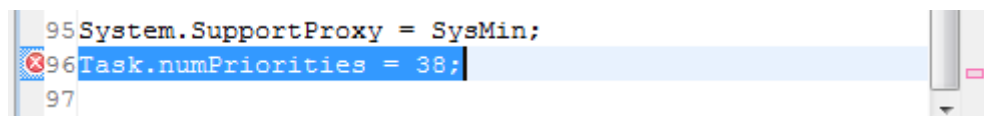
Depending on the type of problem, the validation may only report the first item found in the configuration. For example, a syntax error such as an unknown variable in the script may prevent later problems, such as invalid values being assigned to a property, from being reported. Such problems can be detected after you fix the syntax error and re-validate the configuration.

If there is not enough detail available to identify a specific configuration parameter, no problem icon will be shown in the Properties tab. A problem icon will still be shown on the module or instance in the Outline view.

You can sort the problem list by clicking on the headings. You can filter the problem list to display fewer items by clicking the ⇥ **Filter** icon to open the Filters dialog.

2.2.9 Finding and Fixing Errors

A configuration is validated if you perform any of the following actions:

- Add a module to be used in the configuration
- Delete a module from use by the configuration
- Add a instance to the configuration
- Delete an instance from the configuration
- Save the configuration

You can force the configuration to be validated by saving the configuration or by clicking the ⚙ Refresh icon.

Validation means that semantic checks are performed to make sure that, for example, objects that are referenced actually exist. These checks also make sure that values are within the valid ranges for a property. Some datatype checking is performed when you set a property in the Properties tab.

If you are editing the configuration code directly in the Source tab, the configuration is validated only if you save the configuration.

While a configuration is being validated (this takes a few seconds), you see the a progress icon in the lower-right corner of the window.
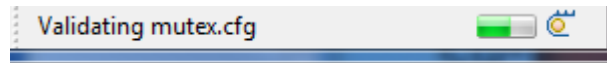
After the configuration has been validated, any errors that were detected in the configuration are shown in the Problems view. For example, if you delete the statement that creates an instance that is referenced by another configuration parameter, you will see an error.

See Section 2.2.8 for more about finding and fixing errors.

2.2.10   Accessing the Global Namespace

Many of the configuration examples in this document define variables in the `Program.global` namespace. For example:

```
Program.global.myTimer = Timer.create(1, "&myIsr", timerParams);
```

The Program module is the root of the configuration object model created by XDCtools; the Program module is implicitly used by configuration scripts; you do not need to add a useModule statement to make it available.

Variables defined in Program.global become global symbols that can be used to directly reference objects in C code. These objects are declared in a generated header file. In order to use these variables, your C code needs to include the generated header file as follows:

```
#include <xdc/cfg/global.h>
```

C code can then access these global symbols directly. For example:

```
Timer_reconfig(myTimer, tickFxn, &timerParams, &eb);
```

If you do not want to #include the generated global.h file, you can declare external handles explicitly. For example, adding the following declaration to your C code would allow you to use the statically configured myTimer object in the previous example:

```
#include <ti/sysbios/hal/Timer.h>

extern Timer_Handle myTimer;
```

For more about the Program module, see http://rtsc.eclipse.org/cdoc-tip/xdc/cfg/Program.html.

2.3   Building SYS/BIOS Applications

When you build an application project, the associated configuration file is rebuilt if the configuration has been changed. The folders listed in the "Includes" list of the CCS project tree (except for the compiler-related folder) are folders that are on the package path.

To build a project, follow these steps:

1. Choose **Project > Build Project**.

2. Examine the log in the **Console** view to see if errors occurred.

3. After you build the project, look at the C/C++ Projects view. You can expand the Debug folder to see the files that were generated by the build process.

For help with build errors, see the RTSC-pedia page at http://rtsc.eclipse.org/docs-tip/Trouble_Shooting.

2.3.1   Understanding the Build Flow

The build flow for SYS/BIOS applications begins with an extra step to process the configuration file (*.cfg) in the project. The configuration file is processed by XDCtools. If you look at the messages printed during the build, you will see a command line that runs the "xs" executable in the XDCtools component with the "xdc.tools.configuro" tool specified. For example:

```
'Invoking: XDCtools'

"<xdctools_install>/xs" --xdcpath="<sysbios_install>/packages;" xdc.tools.configuro
-o configPkg -t ti.targets.arm.elf.M3 -p ti.platforms.concertoM3:F28M35H52C1
-r release -c "C:/ccs/ccsv5/tools/compiler/tms470" "../example.cfg"
```

In CCS, you can control the command-line options used with XDCtools by choosing **Project > Properties** from the menus and selecting the **CCS Build > XDCtools** category.
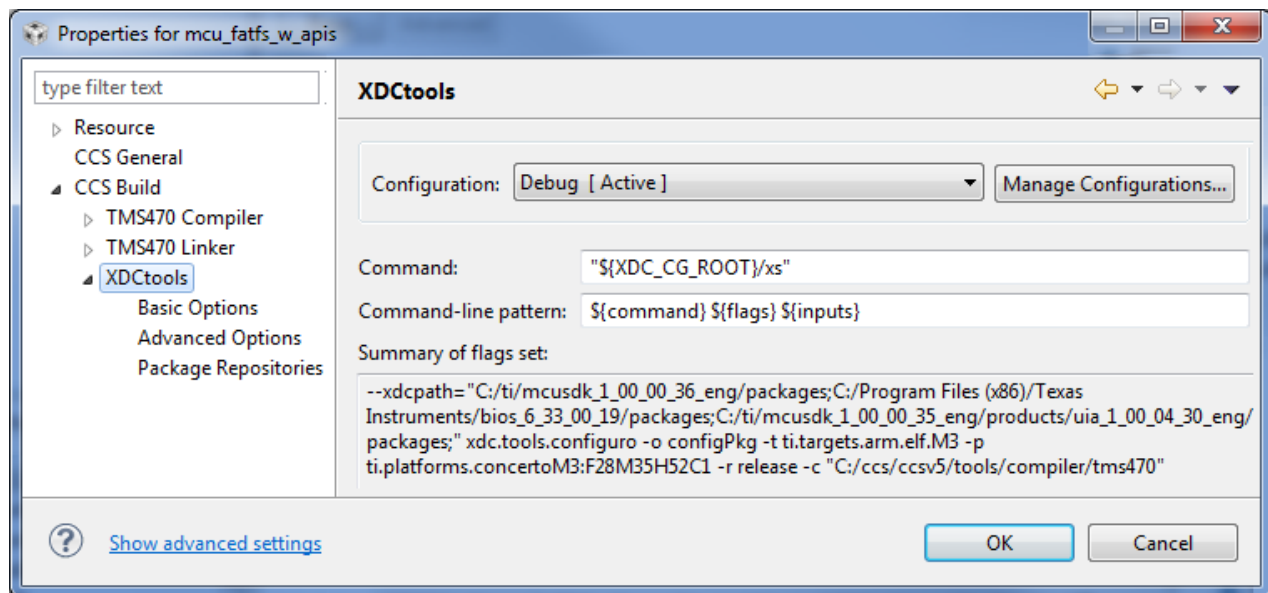
Target settings for processing your individual project are in the **RTSC** tab of the **CCS General** category. (RTSC is the name for the Eclipse specification implemented by XDCtools.)

When XDCtools processes your *.cfg file, code is placed in the *<project_dir>*/*<configuration>*/configPkg directory (where *<configuration>* is Debug or Release depending on your active CCS configuration. This code is compiled so that it can be linked with your final application. In addition, a compiler.opt file is created for use during program compilation, and a linker.cmd file is created for use in linking the application. You should not modify the files in the *<project_dir>*/*<configuration>*/configPkg directory after they are generated, since they will be overwritten the next time you build.

For command-line details about xdc.tools.configuro, see the RTSC-pedia reference topic. Configuro can also be used to build the configuration file with other build systems. For more information, see the RTSC-pedia page at http://rtsc.eclipse.org/docs-tip/Consuming_Configurable_Content.

2.3.2 Rules for Working with CCS Project Properties

After you have created a CCS project that contains a configuration file, you can change the properties of the project in CCS by right-clicking the project name and choosing **Properties**.

In the **CCS General** category of the Properties dialog, the **General** tab applies to compiler settings, and the **RTSC** tab applies to the "configuro" utility used to process the .cfg file.

If there is any platform-specific configuration in your .cfg file, you must change those settings in addition to any platform-related changes you make to the **CCS General > RTSC** settings.

If your configuration file is stored in a separate project from the project that contains your source code files, you should be careful about changing the CCS General settings for a configuration-only project. The build settings for the configuration project must match or be compatible with those of all application projects that reference the configuration project. So, if you change the CCS build settings for a configuration project, you should also change the build settings for the application projects that use that configuration.

2.3.3 Building an Application with GCC

The instructions in this section can be used to build SYS/BIOS applications on Windows or Linux. If you are using a Windows machine, you can use the regular DOS command shell provided with Windows. However, you may want to install a Unix-like shell, such as Cygwin.

For Windows users, the XDCtools top-level installation directory contains `gmake.exe`, which is used in the commands that follow to run the Makefile. The gmake utility is a Windows version of the standard GNU "make" utility provided with Linux.

If you are using Linux, change the "gmake" command to "make" in the commands that follow.

Requirements

- You must have SYS/BIOS and XDCtools installed on the system where you intend to build the application.

- You must have the GCC compiler system installed on the system where you intend to build the application. For example, you can use CodeSourcery's G++ Lite as a compiler for M3 targets. See the Getting Started Guide PDF link on the CodeSourcery page for installation instructions.

- Your application must have a configuration file (*.cfg) that configures the application's use of XDCtools and SYS/BIOS.

Limitations

Note:	XDCtools currently provides linker scripts for a few Stellaris devices only. If you are building for some other device, you will need to modify one of these script files.

Description

The command line tools can be invoked directly or (preferably) the build can be managed with makefiles.

When the SYS/BIOS configuration file is built, the output will include:

- `compiler.opt`, which contains a set of options for the C compiler to use when compiling user applications. The file contains is a set of #include options and pre-processor #defines. The compiler.opt file can be provided to GCC via the `@` option.

- `linker.cmd`, which contains a set of linker options for use when linking applications. The file contains a list of libraries and object files produced from the configuration. It also specifies memory placement for memory used by SYS/BIOS. This file should be passed to the linker using the `-Wl,-T,`*cfg-out-dir*`/linker.cmd` option.

Procedure

Follow these steps to build a SYS/BIOS application with the GCC compiler:

1. Download the sample package from the SYS/BIOS with GCC topic on the TI Embedded Processors Wiki. Uncompress it to the location where you will build your applications. The file contains:

   — `hello.c`, `clock.c`, and `task.c`: Three simple C code files are included—a simple "hello world" application, one that uses the SYS/BIOS Clock module, and one that uses the Task and Semaphore modules.

   — `app.cfg`: A simple shared configuration file for these applications.

   — `lm3s9b90.ld`: A linker script for a Stellaris LM3S9B90 device.

   — `Makefile`: A makefile that can be used to build the applications from the command line.

2. Open the `Makefile` with a text editor. Edit the first three lines of the sample `Makefile` to specify the locations on your system for M3TOOLS (the GCC compiler location), SYSBIOS, and XDCTOOLS. For example:

   ```
   M3TOOLS ?= /home/myusername/cs/arm-2011q3
   SYSBIOS ?= /home/myusername/bios_6_33_00_19
   XDCTOOLS ?= /home/myusername/xdctools_3_23_00_32
   ```

3. If you are building for a device other than a Stellaris LM3S9B90, first check the `XDCTOOLS/packages/ti/platforms/stellaris/include_gnu` directory to see if a linker script file is provided for your device.

4. If no linker script file is provided for your device, make a copy of the closest provided file and edit the "MEMORY" region specifications to make them correspond to the memory map for your device. You can specify your new linker script file as the LINKERCMD file in the `Makefile`.

5. If you are using a linker script file other than `lm3s9b90.ld`, edit the `Makefile` to change the LINKERCMD setting.

6. If you are using Windows and the gmake utility provided in the top-level directory of the XDCtools installation, you should add the *<xdc_install_dir>* to your PATH environment variable so that the gmake executable can be found.

7. Build the three sample applications by running the following command. (If you are running the build on Linux, change all "gmake" commands to "make".)

```
gmake
```

8. You can clean the build by running the following command:

```
gmake clean
```

9. Once built, applications can be loaded and debugged with CCS. Within CCS, you can use **Tools > RTOS Object View (ROV)** to browse SYS/BIOS kernel object states and various other tools for real-time analysis. Alternatively, you can use GDB or another debugger for basic debugging, but the ROV is not available outside of CCS.

For more information, such as the compiler and linker options needed by SYS/BIOS for the M3 and M4 devices, see the SYS/BIOS with GCC topic on the TI Embedded Processors Wiki. Additional information about building with various compilers may be found on the RTSC-pedia wiki.

2.3.4   *Running and Debugging an Application in CCS*

If you haven't already created a default target configuration, follow these steps:

1. Choose **File > New > Target Configuration File**.

2. Type a filename for the target configuration, which will be stored as part of the CCS project. For example, you might type **TCI6482sim.ccxml** if that is the target you want to use. Then, click **Finish**.

3. In the Connection field for your target configuration, choose the type of connection you have to the target. Then type part of the target name in the Device filter field. For example, you might choose the "TI Simulator" connection and filter by "64xp" to find a C64x+ simulator.

4. Choose **File > Save** or click the **Save** icon to save your target configuration.

5. You can right-click on a target configuration and choose **Set as Default Target** to set which target configuration is used for debugging.

To debug an application, follow these steps:

1. Choose **Target > Debug Active Project** or click the **Debug** icon. This loads the program and switches you to the "Debug" perspective.

2. You can set breakpoints in your code if desired. Press F8 to run.

3. Use various tools provided with XDCtools and SYS/BIOS to debug the application. See Section 8.4 for a more detailed comparison.

    —   **RTOS Object Viewer (ROV).** See section 6.5.3, *ROV for System Stacks and Task Stacks* and the RTSC-pedia page on ROV at http://rtsc.eclipse.org/docs-tip/RTSC_Object_Viewer.

    —   **System Analyzer.** See the System Analyzer User's Guide (SPRUH43) and the System Analyzer wiki page for details.

2.3.5 *Compiler and Linker Optimization*

You can optimize your application for better performance and code size or to give you more debugging information by selecting different ways of compiling and linking your application. For example, you can do this by linking with versions of the SYS/BIOS libraries that were compiled differently.

The choices you can make related to compiler and linker optimization are located in the following places:

- **Build-Profile.** You see this field when you are creating a new CCS project or modifying the CCS General settings. We recommend that you use the "release" setting. The "release" option is preferred even when you are creating and debugging an application; the "debug" option is mainly intended for internal use by Texas Instruments. The "release" option results in a somewhat smaller executable that can still be debugged. This build profile primarily affects how Codec Engine and some device drivers are built.

Note: The "whole_program" and "whole_program_debug" options for the Build-Profile have been deprecated, and are no longer recommended. The option that provides the most similar result is to set the BIOS.libType configuration parameter to BIOS.LibType_Custom.

- **Configuration.** The drop-down field at the top of the Properties dialog allows you to choose between and customize multiple build configurations. Each configuration can have the compiler and linker settings you choose. Debug and Release are the default configurations available.

- **BIOS.libType configuration parameter.** You can set this parameter in XGCONF or by editing the .cfg file in your project. This parameter lets you select one of two pre-compiled versions of the SYS/BIOS libraries or to have a custom version of the SYS/BIOS libraries compiled based on the needs of your application. See the table and discussion that follow for more information.

The options for the BIOS.libType configuration parameter are as follows:

BIOS.libType	Compile Time	Logging	Code Size	Runtime Performance
Instrumented (BIOS.LibType_Instrumented)	Fast	On	Good	Good
Non-Instrumented (BIOS.LibType_NonInstrumented)	Fast	Off	Better	Better
Custom (Optimized) (BIOS.LibType_Custom)	Fast (slow first time)	As configured	Best	Best
Custom (Debug) (BIOS.LibType_Debug)	Fast (slow first time)	As configured	Largest	No optimization

For all libType options, the executable that is created contains only the modules and APIs that your application needs to access. If you have not used a particular module in your .cfg file or your C code (and it is not required internally by a SYS/BIOS module that is used), that module is not linked with your application. Individual API functions that are not needed (either directly or indirectly) are also excluded during the linking phase of the build.

Note: If you disable SYS/BIOS Task or Swi scheduling, you must use the "Custom (optimized)" or "Custom (debug)" option in order to successfully link your application.

- **Instrumented.** (default) This option links with pre-built SYS/BIOS libraries that have instrumentation available. All Asserts and Diags settings are checked. Your configuration file can enable or disable various Diags and logging related settings. However, note that the checks to see if Diags are enabled before outputting a Log event are always performed, which has an impact on performance even if you use the ALWAYS_ON or ALWAYS_OFF setting. The resulting code size when using this option may be too large to fit on some targets, such as C28x and MSP430. This option is easy to use and debug and provides a fast build time.

- **Non-Instrumented.** This option links with pre-built SYS/BIOS libraries that have instrumentation turned off. No Assert or Diag settings are checked, and logging information is not available at runtime. The checking for Asserts and Diags is compiled out of the libraries, so runtime performance and code size are optimized. Checking of Error_Blocks and handling errors in ways other than logging an event are still supported. This option is easy to use and provides a fast build time.

- **Custom (Optimized).** This option builds custom versions of the SYS/BIOS libraries. This option is optimized to provide the best runtime performance and code size given the needs of your application. Instrumentation is available to whatever extent your application configures it.

  The first time you build a project with the BIOS.LibType_Custom libType, the build will be longer. The libraries are stored in the "src" directory of your project. Subsequent builds may be faster; libraries do not need to be rebuilt unless you change one of the few configuration parameters that affect the build settings or use an additional module that wasn't already used in the previous configuration.

  The BIOS.LibType_Custom option uses program optimization that removes many initialized constants and small code fragments (often "glue" code) from the final executable image. Such classic optimizations as constant folding and function inlining are used, including across module boundaries. This build preserves enough debug information to make it still possible to step through the optimized code in CCS and locate global variables.

- **Custom (Debug).** As with the Custom (optimized) libType, the Custom (debug) option builds custom versions of the SYS/BIOS libraries. However, no compiler or linker optimization is used. The resulting application is fully debuggable; you can step into the code performed by SYS/BIOS APIs. In addition, the CCS debugger will always be able to find the relevant source files as you debug the code. Since no optimization is performed, the code size is large and the runtime performance is slower than with the custom libType.

  The first time you build a project with the BIOS.LibType_Debug libType, the build will be longer. The libraries are stored in the "src" directory of your project. Subsequent builds may be faster; libraries do not need to be rebuilt unless you change one of the few configuration parameters that affect the build settings or use an additional module that wasn't already used in the previous configuration.

The following example statements set the BIOS.libType configuration parameter:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');

BIOS.libType = BIOS.LibType_Custom;
```

If you use the BIOS.LibType_Custom or BIOS.LibType_Debug option for the BIOS.libType, you can also set the BIOS.customCCOpts parameter to customize the C compiler command-line options used when compiling the SYS/BIOS libraries. If you want to change this parameter, it is important to first examine and understand the default command-line options used to compile the SYS/BIOS libraries for your target. You can see the default in XGCONF or by placing the following statement in your configuration script and building the project:

```
print("customCCOpts =", BIOS.customCCOpts);
```

You must be careful not to cause problems for the SYS/BIOS compilation when you modify the BIOS.customCCOpts parameter. For example, the --program_level_compile option is required. (Some --define and --include_path options are used on the compiler command line but are not listed in the customCCOpts definition; these also cannot be removed.)

For example, to create a debuggable custom library, you can remove the -o3 option from the BIOS.customCCOpts definition by specifying it with the following string for a C64x+ target:

```
BIOS.customCCOpts = "-mv64p --abi=eabi -q -mi10 -mo -pdr -pden -pds=238 -pds=880
-pds1110 --program_level_compile -g";
```

More information about configuring the BIOS.libType and BIOS.customCCOpts parameters is provided in the SYS/BIOS FAQs on the TI Embedded Processor wiki.

See Appendix A for information about how to rebuild SYS/BIOS manually.

Threading Modules

This chapter describes the types of threads a SYS/BIOS program can use.

3.1 SYS/BIOS Startup Sequence

The SYS/BIOS startup sequence is logically divided into two phases—those operations that occur prior to the application's "main()" function being called and those operations that are performed after the application's "main()" function is invoked. Control points are provided at various places in each of the two startup sequences for user startup functions to be inserted.

The "before main()" startup sequence is governed completely by the XDCtools runtime package. For more information about the boot sequence prior to main, refer to the "XDCtools Boot Sequence and Control Points" topic in the RTSC-pedia. The XDCtools runtime startup sequence is as follows:

1. Immediately after CPU reset, perform target/device-specific CPU initialization (beginning at c_int00).

2. Prior to cinit(), run the table of "reset functions" (the xdc.runtime.Reset module provides this hook).

3. Run cinit() to initialize C runtime environment.

4. Run the user-supplied "first functions" (the xdc.runtime.Startup module provides this hook).

5. Run all the module initialization functions.

6. Run the user-supplied "last functions" (the xdc.runtime.Startup module provides this hook).

7. Run pinit().

8. Run main().

The "after main()" startup sequence is governed by SYS/BIOS and is initiated by an explicit call to the BIOS_start() function at the end of the application's main() function. The SYS/BIOS startup sequence that run when BIOS_start() is called is as follows:

1. **Startup Functions.** Run the user-supplied "startup functions" (see BIOS.startupFxns).

2. **Enable Hardware Interrupts.**

3. **Timer Startup.** If the system supports Timers, then at this point all configured timers are initialized per their user-configuration. If a timer was configured to start "automatically", it is started here.

4. **Enable Software Interrupts.** If the system supports software interrupts (Swis) (see BIOS.swiEnabled), then the SYS/BIOS startup sequence enables Swis at this point.

5. **Task Startup.** If the system supports Tasks (see BIOS.taskEnabled), then task scheduling begins here. If there are no statically or dynamically created Tasks in the system, then execution proceeds directly to the idle loop.

The following configuration script excerpt installs a user-supplied startup function at every possible control point in the XDCtools and SYS/BIOS startup sequence. Configuration scripts have the filename extension ".cfg" and are written in the XDCscript language, which is used to configure modules.

```
/* get handle to xdc Reset module */
Reset = xdc.useModule('xdc.runtime.Reset');

/* install a "reset function" */
Reset.fxns[Reset.fxns.length++] = '&myReset';

/* get handle to xdc Startup module */
var Startup = xdc.useModule('xdc.runtime.Startup');

/* install a "first function" */
Startup.firstFxns[Startup.firstFxns.length++] = '&myFirst';

/* install a "last function" */
Startup.lastFxns[Startup.lastFxns.length++] = '&myLast';

/* get handle to BIOS module */
var BIOS = xdc.useModule('ti.sysbios.BIOS');

/* install a BIOS startup function */
BIOS.addUserStartupFunction('&myBiosStartup');
```

3.2   Overview of Threading Modules

Many real-time applications must perform a number of seemingly unrelated functions at the same time, often in response to external events such as the availability of data or the presence of a control signal. Both the functions performed and when they are performed are important.

These functions are called threads. Different systems define threads either narrowly or broadly. Within SYS/BIOS, the term is defined broadly to include any independent stream of instructions executed by the processor. A thread is a single point of control that can activate a function call or an interrupt service routine (ISR).

SYS/BIOS enables your applications to be structured as a collection of threads, each of which carries out a modularized function. Multithreaded programs run on a single processor by allowing higher-priority threads to preempt lower-priority threads and by allowing various types of interaction between threads, including blocking, communication, and synchronization.

Real-time application programs organized in such a modular fashion—as opposed to a single, centralized polling loop, for example—are easier to design, implement, and maintain.

SYS/BIOS provides support for several types of program threads with different priorities. Each thread type has different execution and preemption characteristics. The thread types (from highest to lowest priority) are:

- **Hardware interrupts (Hwi), which** includes Timer functions

- **Software interrupts (Swi), which** includes Clock functions

- **Tasks (Task)**

- **Background thread (Idle)**

These thread types are described briefly in the following section and discussed in more detail in the rest of this chapter.

3.2.1 Types of Threads

The four major types of threads in a SYS/BIOS program are:

- **Hardware interrupt (Hwi) threads**. Hwi threads (also called Interrupt Service Routines or ISRs) are the threads with the highest priority in a SYS/BIOS application. Hwi threads are used to perform time critical tasks that are subject to hard deadlines. They are triggered in response to external asynchronous events (interrupts) that occur in the real-time environment. Hwi threads always run to completion but can be preempted temporarily by Hwi threads triggered by other interrupts, if enabled. See Section 3.3, *Hardware Interrupts*, page 3-59, for details about hardware interrupts.

- **Software interrupt (Swi) threads**. Patterned after hardware interrupts (Hwi), software interrupt threads provide additional priority levels between Hwi threads and Task threads. Unlike Hwis, which are triggered by hardware interrupts, Swis are triggered programmatically by calling certain Swi module APIs. Swis handle threads subject to time constraints that preclude them from being run as tasks, but whose deadlines are not as severe as those of hardware ISRs. Like Hwi's, Swi's threads always run to completion. Swis allow Hwis to defer less critical processing to a lower-priority thread, minimizing the time the CPU spends inside an interrupt service routine, where other Hwis can be disabled. Swis require only enough space to save the context for each Swi interrupt priority level, while Tasks use a separate stack for each thread. See Section 3.4, *Software Interrupts*, page 3-68, for details about Swis.

- **Task (Task) threads.** Task threads have higher priority than the background (Idle) thread and lower priority than software interrupts. Tasks differ from software interrupts in that they can wait (block) during execution until necessary resources are available. Tasks require a separate stack for each thread. SYS/BIOS provides a number of mechanisms that can be used for inter-task communication and synchronization. These include Semaphores, Events, Message queues, and Mailboxes. See Section 3.5, *Tasks*, page 3-83, for details about tasks.

- **Idle Loop (Idle) thread**. Idle threads execute at the lowest priority in a SYS/BIOS application and are executed one after another in a continuous loop (the Idle Loop). After main returns, a SYS/BIOS application calls the startup routine for each SYS/BIOS module and then falls into the Idle Loop. Each thread 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. See Section 3.6, *The Idle Loop*, page 3-101, for details about the background thread.

Another type of thread, a Clock thread, is run within the context of a Swi thread that is triggered by a Hwi thread invoked by a repetitive timer peripheral interrupt. See Section 5.2 for details.

3.2.2 Choosing Which Types of Threads to Use

The type and priority level you choose for each thread in an application program has an impact on whether the threads are scheduled on time and executed correctly. SYS/BIOS static configuration makes it easy to change a thread from one type to another.

A program can use multiple types of threads. Here are some rules for deciding which type of object to use for each thread to be performed by a program.

- **Swi or Task versus Hwi**. Perform only critical processing within hardware interrupt service routines. Hwis should be considered for processing hardware interrupts (IRQs) with deadlines down to the 5-microsecond range, especially when data may be overwritten if the deadline is not met. Swis or Tasks should be considered for events with longer deadlines—around 100 microseconds or more. Your Hwi functions should post Swis or tasks to perform lower-priority processing. Using lower-priority threads minimizes the length of time interrupts are disabled (interrupt latency), allowing other hardware interrupts to occur.

- **Swi versus Task.** Use Swis if functions have relatively simple interdependencies and data sharing requirements. Use tasks if the requirements are more complex. While higher-priority threads can preempt lower priority threads, only tasks can wait for another event, such as resource availability. Tasks also have more options than Swis when using shared data. All input needed by a Swi's function should be ready when the program posts the Swi. The Swi object's trigger structure provides a way to determine when resources are available. Swis are more memory-efficient because they all run from a single stack.

- **Idle**. Create Idle threads to perform noncritical housekeeping tasks when no other processing is necessary. Idle threads typically have no hard deadlines. Instead, they run when the system has unused processor time. Idle threads run sequentially at the same priority. You may use Idle threads to reduce power needs when other processing is not being performed. In this case, you should not depend upon housekeeping tasks to occur during power reduction times.

- **Clock**. Use Clock functions when you want a function to run at a rate based on a multiple of the interrupt rate of the peripheral that is driving the Clock tick. Clock functions can be configured to execute either periodically or just once. These functions run as Swi functions.

- **Clock versus Swi**. All Clock functions run at the same Swi priority, so one Clock function cannot preempt another. However, Clock functions can post lower-priority Swi threads for lengthy processing. This ensures that the Clock Swi can preempt those functions when the next system tick occurs and when the Clock Swi is posted again.

- **Timer.** Timer threads are run within the context of a Hwi thread. As such, they inherit the priority of the corresponding Timer interrupt. They are invoked at the rate of the programmed Timer period. Timer threads should do the absolute minimum necessary to complete the task required. If more processing time is required, consider posting a Swi to do the work or posting a Semaphore for later processing by a task so that CPU time is efficiently managed.

3.2.3 *A Comparison of Thread Characteristics*

Table 3-1 provides a comparison of the thread types supported by SYS/BIOS.

Table 3-1. Comparison of Thread Characteristics

Characteristic	Hwi	Swi	Task	Idle
Priority	Highest	2nd highest	2nd lowest	Lowest
Number of priority levels	family/device-specific	Up to 32 (16 for MSP430 and C28x). Periodic functions run at the priority of the Clock Swi.	Up to 32 (16 for MSP430 and C28x). This includes 1 for the Idle Loop.	1
Can yield and pend	No, runs to completion except for preemption	No, runs to completion except for preemption	Yes	Should not pend. Pending would disable all registered Idle threads.

Characteristic	Hwi	Swi	Task	Idle
Execution states	Inactive, ready, running	Inactive, ready, running	Ready, running, blocked, terminated	Ready, running
Thread scheduler disabled by	Hwi_disable()	Swi_disable()	Task_disable()	Program exit
Posted or made ready to run by	Interrupt occurs	Swi_post(), Swi_andn(), Swi_dec(), Swi_inc(), Swi_or()	Task_create() and various task synchro-nization mechanisms (Event, Semaphore, Mailbox)	main() exits and no other thread is currently running
Stack used	System stack (1 per program)	System stack (1 per program)	Task stack (1 per task)	Task stack used by default (see Note 1)
Context saved when preempts other thread	Entire context minus saved-by-callee registers (as defined by the TI C compiler) are saved to system.	Certain registers saved to system.	Entire context saved to task stack	--Not applicable--
Context saved when blocked	--Not applicable--	--Not applicable--	Saves the saved-by-callee registers (see optimizing compiler user's guide for your platform).	--Not applicable--
Share data with thread via	Streams, lists, pipes, global variables	Streams, lists, pipes, global variables	Streams, lists, pipes, gates, mailboxes, message queues, global variables	Streams, lists, pipes, global variables
Synchronize with thread via	--Not applicable--	Swi trigger	Semaphores, events, mailboxes	-Not applicable--
Function hooks	Yes: register, create, begin, end, delete	Yes:register, create, ready, begin, end, delete	Yes: register, create, ready, switch, exit, delete	No
Static creation	Yes	Yes	Yes	Yes
Dynamic creation	Yes	Yes	Yes	No
Dynamically change priority	See Note 2	Yes	Yes	No
Implicit logging	Interrupt event	Post, begin, end	Switch, yield, ready, exit	None
Implicit statistics	None	None	None	None

Notes: 1) If you disable the Task manager, Idle threads use the system stack.
2) Some devices allow hardware interrupt priorities to by modified.

3.2.4   *Thread Priorities*

Within SYS/BIOS, hardware interrupts have the highest priority. The priorities among the set of Hwi objects are not maintained implicitly by SYS/BIOS. The Hwi priority only applies to the order in which multiple interrupts that are ready on a given CPU cycle are serviced by the CPU. Hardware interrupts are preempted by another interrupt unless interrupts are globally disabled or when specific interrupts are individually disabled.

Figure 3-1. Thread Priorities
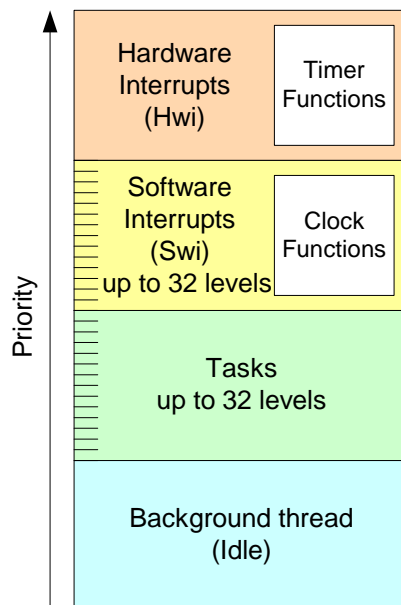
Swis have lower priority than Hwis. There are up to 32 priority levels available for Swis (16 by default) The maximum number of priority levels is 16 for MSP430 and C28x. Swis can be preempted by a higher-priority Swi or any Hwi. Swis cannot block.

Tasks have lower priority than Swis. There are up to 32 task priority levels (16 by default). The maximum number of priority levels is 16 for MSP430 and C28x. Tasks can be preempted by any higher-priority thread. Tasks can block while waiting for resource availability and lower-priority threads.

The background Idle Loop is the thread with the lowest priority of all. It runs in a loop when the CPU is not busy running another thread. When tasks are enabled, the Idle Loop is implemented as the only task running at priority 0. When tasks are disabled, the Idle Loop is fallen into after the application's "main()" function is called.

3.2.5 *Yielding and Preemption*

The SYS/BIOS thread schedulers run the highest-priority thread that is ready to run except in the following cases:

- The thread that is running disables some or all hardware interrupts temporarily with Hwi_disable() or Hwi_disableInterrupt(), preventing hardware ISRs from running.

- The thread that is running disables Swis temporarily with Swi_disable(). This prevents any higher-priority Swi from preempting the current thread. It does not prevent Hwis from preempting the current thread.

- The thread that is running disables task scheduling temporarily with Task_disable(). This prevents any higher-priority task from preempting the current task. It does not prevent Hwis and Swis from preempting the current task.

- If a lower priority task shares a gating resource with a higher task and changes its state to pending, the higher priority task may effectively have its priority set to that of the lower priority task. This is called Priority Inversion and is described in Section 4.3.3.

Both Hwis and Swis can interact with the SYS/BIOS task scheduler. When a task is blocked, it is often because the task is pending on a semaphore which is unavailable. Semaphores can be posted from Hwis and Swis as well as from other tasks. If a Hwi or Swi posts a semaphore to unblock a pending task, the processor switches to that task if that task has a higher priority than the currently running task (after the Hwi or Swi completes).

When running either a Hwi or Swi, SYS/BIOS uses a dedicated system interrupt stack, called the *system stack* (sometimes called the ISR stack). Each task uses its own private stack. Therefore, if there are no Tasks in the system, all threads share the same system stack. For performance reasons, sometimes it is advantageous to place the system stack in precious fast memory. See Section 3.4.3 for information about system stack size and Section 3.5.3 for information about task stack size.

Table 3-2 shows what happens when one type of thread is running (top row) and another thread becomes ready to run (left column). The action shown is that of the newly posted (ready to run) thread.

Table 3-2. Thread Preemption

Newly Posted Thread	Running Thread			
	Hwi	**Swi**	**Task**	**Idle**
Enabled Hwi	Preempts if enabled*	Preempts	Preempts	Preempts
Disabled Hwi	Waits for reenable	Waits for reenable	Waits for reenable	Waits for reenable
Enabled, higher-priority Swi	Waits	Preempts	Preempts	Preempts
Lower-priority Swi	Waits	Waits	Preempts	Preempts
Enabled, higher-priority Task	Waits	Waits	Preempts	Preempts
Low-priority Task	Waits	Waits	Waits	Preempts

* On some targets, hardware interrupts can be individually enabled and disabled. This is not true on all targets. Also, some targets have controllers that support hardware interrupt prioritization, in which case a Hwi can only be preempted by a higher-priority Hwi.

Note that Table 3-2 shows the results if the type of thread that is posted is enabled. If that thread type is disabled (for example, by Task_disable), a thread cannot run in any case until its thread type is reenabled.

Figure 3-2 shows the execution graph for a scenario in which Swis and Hwis are enabled (the default), and a Hwi posts a Swi whose priority is higher than that of the Swi running when the interrupt occurs. Also, a second Hwi occurs while the first ISR is running and preempts the first ISR.

Figure 3-2. Preemption Scenario
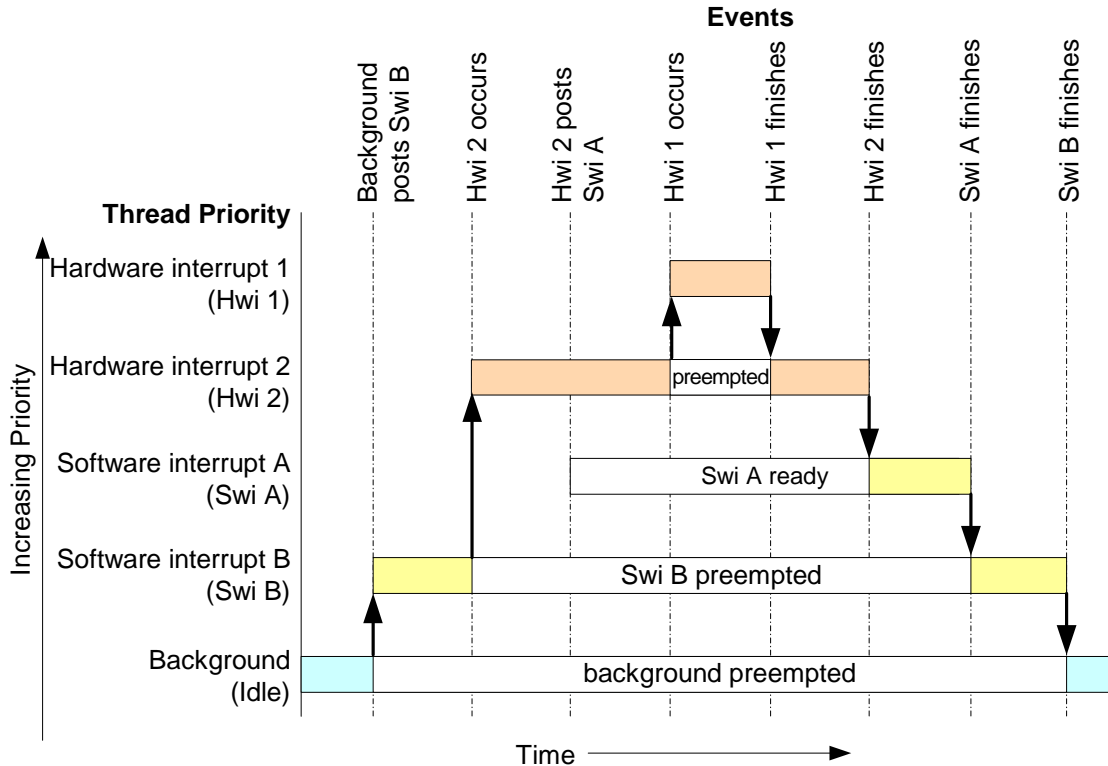
In Figure 3-2, the low-priority Swi is asynchronously preempted by the Hwis. The first Hwi posts a higher-priority Swi, which is executed after both Hwis finish executing.

Here is sample pseudo-code for the example depicted in Figure 3-2:

```
backgroundThread()
{
    Swi_post(Swi_B) /* priority = 5 */
}


Hwi_1 ()
{
    . . .
}
Hwi_2 ()
{
    Swi_post(Swi_A) /* priority = 7 */
}
```

3.2.6   Hooks

Hwi, Swi, and Task threads optionally provide points in a thread's life cycle to insert user code for instrumentation, monitoring, or statistics gathering purposes. Each of these code points is called a "hook" and the user function provided for the hook is called a "hook function".

The following hook functions can be set for the various thread types:

Table 3–3. Hook Functions by Thread Type

Thread Type	Hook Functions
Hwi	Register, Create, Begin, End, and Delete. See Section 3.3.3.
Swi	Register, Create, Ready, Begin, End, and Delete. See Section 3.4.8.
Task	Register, Create, Ready, Switch, Exit, and Delete. See Section 3.5.5.

Hooks are declared as a set of hook functions called "hook sets". You do not need to define all hook functions within a set, only those that are required by the application.

Hook functions can only be declared statically (in an XDCtools configuration script) so that they may be efficiently invoked when provided and result in *no runtime overhead* when a hook function is not provided.

Except for the Register hook, all hook functions are invoked with a handle to the object associated with that thread as its argument (that is, a Hwi object, a Swi object, or a Task object). Other arguments are provided for some thread-type-specific hook functions.

You can define as many hook sets as necessary for your application. When more than one hook set is defined, the individual hook functions within each set are invoked in hook ID order for a particular hook type. For example, during Task_create() the order that the Create hook within each Task hook set is invoked is the order in which the Task hook sets were originally defined.

The argument to a thread's Register hook (which is invoked only once) is an index (the "hook ID") indicating the hook set's relative order in the hook function calling sequence.

Each set of hook functions has a unique associated "hook context pointer". This general-purpose pointer can be used by itself to hold hook set specific information, or it can be initialized to point to a block of memory allocated by the Create hook function within a hook set if more space is required for a particular application.

An individual hook function obtains the value of its associated context pointer through the following thread-type-specific APIs: Hwi_getHookContext(), Swi_getHookContext(), and Task_getHookContext(). Corresponding APIs for initializing the context pointers are also provided: Hwi_setHookContext(), Swi_setHookContext(), and Task_setHookContext(). Each of these APIs take the hook ID as an argument.

The following diagram shows an application with three Hwi hook sets:

```
      Hwi Hook Set [0]              Hwi Hook Set [1]              Hwi Hook Set [2]

 ┌──────────────────────┐    ┌──────────────────────┐    ┌──────────────────────┐
 │ registerHookFunc0()  │    │ registerHookFunc1()  │    │ registerHookFunc2()  │
 │ createHookFunc0()    │    │ createHookFunc1()    │    │ createHookFunc2()    │
 │ beginHookFunc0()     │    │ beginHookFunc1()     │    │ beginHookFunc2()     │
 │ endHookFunc0()       │    │ endHookFunc1()       │    │ endHookFunc2()       │
 │ deleteHookFunc0()    │    │ deleteHookFunc1()    │    │ deleteHookFunc2()    │
 └──────────────────────┘    └──────────────────────┘    └──────────────────────┘
            │
            │                    Hwi_getHookContext(1)
            │
  Hwi_getHookContext(0)
                                                 Hwi_getHookContext(2)
            ▼
 ┌──────────────────────┐
 │ hookContextPtr[0]    │
 ├──────────────────────┤
 │ hookContextPtr[1]    │  ◄──
 ├──────────────────────┤
 │ hookContextPtr[2]    │  ◄──
 └──────────────────────┘
```

The hook context pointers are accessed using Hwi_getHookContext() using the index provided to the three Register hook functions.

Just prior to invoking your ISR functions, the Begin Hook functions are invoked in the following order:

1. beginHookFunc0();

2. beginHookFunc1();

3. beginHookFunc2();

Likewise, upon return from your ISR functions the End Hook functions are invoked in the following order:

1. endHookFunc0();

2. endHookFunc1();

3. endHookFunc2();

3.3   Hardware Interrupts

Hardware interrupts (Hwis) handle critical processing that the application must perform in response to external asynchronous events. The SYS/BIOS target/device specific Hwi modules are used to manage hardware interrupts. See the video introducing Hwis for an overview.

In a typical embedded system, hardware interrupts are triggered either by on-device peripherals or by devices external to the processor. In both cases, the interrupt causes the processor to vector to the ISR address.

Any interrupt processing that may invoke SYS/BIOS APIs that affect Swi and Task scheduling must be written in C or C++. The HWI_enter()/HWI_exit() macros provided in earlier versions of SYS/BIOS for calling assembly language ISRs are no longer provided.

Assembly language ISRs that do not interact with SYS/BIOS can be specified with Hwi_plug(). Such ISRs must do their own context preservation. They may use the "interrupt" keyword, C functions, or assembly language functions.

All hardware interrupts run to completion. If a Hwi is posted multiple times before its ISR has a chance to run, the ISR runs only one time. For this reason, you should minimize the amount of code performed by a Hwi function.

If interrupts are globally enabled—that is, by calling Hwi_enable()—an ISR can be preempted by any interrupt that has been enabled.

Hwis must not use the Chip Support Library (CSL) for the target. Instead, see Chapter 7 for a description of Hardware Abstraction Layer APIs.

Associating an ISR function with a particular interrupt is done by creating a Hwi object.

3.3.1 Creating Hwi Objects

The Hwi module maintains a table of pointers to Hwi objects that contain information about each Hwi managed by the dispatcher (or by generated interrupt stubs on platforms for which the Hwi dispatcher is not provided, such as the MSP430). To create a Hwi object dynamically, use calls similar to these:

```
Hwi_Handle hwi0;
Hwi_Params hwiParams;
Error_Block eb;

Error_init(&eb);
Hwi_Params_init(&hwiParams);

hwiParams.arg = 5;
hwi0 = Hwi_create(id, hwiFunc, &hwiParams, &eb);
if (hwi0 == NULL) {
    System_abort("Hwi create failed");
}
```

Here, hwi0 is a handle to the created Hwi object, id is the interrupt number being defined, hwiFunc is the name of the function associated with the Hwi, and hwiParams is a structure that contains Hwi instance parameters (enable/restore masks, the Hwi function argument, etc). Here, hwiParams.arg is set to 5. If NULL is passed instead of a pointer to an actual Hwi_Params struct, a default set of parameters is used. The "eb" is an error block that you can use to handle errors that may occur during Hwi object creation.

The corresponding static configuration Hwi object creation syntax is:

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params;

hwiParams.arg = 5;
Program.global.hwi0 = Hwi.create(id, '&hwiFunc', hwiParams);
```

Here, the "hwiParams = new Hwi.Params" statement does the equivalent of creating and initializing the hwiParams structure with default values. In the static configuration world, no error block (eb) is required for the "create" function. The "Program.global.hwi0" name becomes a a runtime-accessible handle (symbol name = "hwi0") to the statically-created Hwi object.

3.3.2   Hardware Interrupt Nesting and System Stack Size

When a Hwi runs, its function is invoked using the system stack. In the worst case, each Hwi can result in a nesting of the scheduling function (that is, the lowest priority Hwi is preempted by the next highest priority Hwi, which, in turn, is preempted by the next highest, …). This results in an increasing stack size requirement for each Hwi priority level actually used.

The default system stack size is 4096 bytes. You can set the system stack size by adding the following line to your config script:

```
Program.stack = yourStackSize;
```

The following table shows the amount of system stack required to absorb the worst-case Hwi interrupt nesting. This first number is the amount of system stack space required for the first priority level on a target. The second number shows the amount of stack space required for each subsequent priority level used in the application.

Table 3–4. System Stack Use for Hwi Nesting by Target Family

Target Family	Stack Consumed by First Hwi	Stack Consumed by Subsequent Nested Hwis	Units
M3	176	80	8-bit bytes
MSP430	36	26	8-bit bytes
MSP430X	38	46	8-bit bytes
MSP430X_small	36	26	8-bit bytes
C674	68	384	8-bit bytes
C64P	68	384	8-bit bytes
C64T	68	208	8-bit bytes
C28_float	65	60	16-bit words
C28_large	65	46	16-bit words
Arm9	136	80	8-bit bytes
A8F	136	144	8-bit bytes

See Section 3.4.3 for information about system stack use by software interrupts and Section 3.5.3 for information about task stack size.

3.3.3   Hwi Hooks

The Hwi module supports the following set of Hook functions:

- **Register.** A function called before any statically created Hwis are initialized at runtime. The register hook is called at boot time before main() and before interrupts are enabled.

- **Create.** A function called when a Hwi is created. This includes Hwis that are created statically and those created dynamically using Hwi_create().

- **Begin.** A function called just prior to running a Hwi ISR function.

- **End.** A function called just after a Hwi ISR function finishes.

- **Delete.** A function called when a Hwi is deleted at runtime with Hwi_delete().

The following HookSet structure type definition encapsulates the hook functions supported by the Hwi module:

```
typedef struct Hwi_HookSet {
    Void (*registerFxn)(Int);              /* Register Hook */
    Void (*createFxn)(Handle, Error.Block *); /* Create Hook */
    Void (*beginFxn)(Handle);              /* Begin Hook */
    Void (*endFxn)(Handle);                /* End Hook */
    Void (*deleteFxn)(Handle);             /* Delete Hook */
};
```

Hwi Hook functions can only be configured statically.

3.3.3.1 Register Function

The register function is provided to allow a hook set to store its corresponding hook ID. This ID can be passed to Hwi_setHookContext() and Hwi_getHookContext() to set or get hook-specific context. The Register function must be specified if the hook implementation needs to use Hwi_setHookContext() or Hwi_getHookContext().

The registerFxn hook function is called during system initialization before interrupts have been enabled.

The Register function has the following signature:

```
Void registerFxn(Int id);
```

3.3.3.2 Create and Delete Functions

The Create and Delete functions are called whenever a Hwi is created or deleted. The Create function is passed an Error_Block that is to be passed to Memory_alloc() for applications that require additional context storage space.

The createFxn and deleteFxn functions are called with interrupts enabled (unless called at boot time or from main()).

These functions have the following signatures:

```
Void createFxn(Hwi_Handle hwi, Error_Block *eb);
Void deleteFxn(Hwi_Handle hwi);
```

3.3.3.3 Begin and End Functions

The Begin and End hook functions are called with interrupts globally disabled. As a result, any hook processing function contributes to overall system interrupt response latency. In order to minimize this impact, carefully consider the processing time spent in a Hwi beginFxn or endFxn hook function.

The beginFxn is invoked just prior to calling the ISR function. The endFxn is invoked immediately after the return from the ISR function.

These functions have the following signatures:

```
Void beginFxn(Hwi_Handle hwi);
Void endFxn(Hwi_Handle hwi);
```

When more than one Hook Set is defined, the individual hook functions of a common type are invoked in hook ID order.

3.3.3.4   Hwi Hooks Example

The following example application uses two Hwi hook sets. The Hwi associated with a statically-created Timer is used to exercise the Hwi hook functions. This example demonstrates how to read and write the Hook Context Pointer associated with each hook set.

The XDCtools configuration script and program output are shown after the C code listing.

This is the C code for the example:

```
/* ======== HwiHookExample.c ========
 * This example demonstrates basic Hwi hook usage. */

#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Timestamp.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/hal/Timer.h>
#include <ti/sysbios/hal/Hwi.h>

extern Timer_Handle myTimer;
volatile Bool myEnd2Flag = FALSE;
Int myHookSetId1, myHookSetId2;
Error_Block eb;

Error_init(&eb);

/* HookSet 1 functions */

/* ======== myRegister1 ========
 * invoked during Hwi module startup before main()
 * for each HookSet */
Void myRegister1(Int hookSetId)
{
    System_printf("myRegister1: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId1 = hookSetId;
}
```

```
/* ======== myCreate1 ========
 * invoked during Hwi module startup before main()
 * for statically created Hwis */
Void myCreate1(Hwi_Handle hwi, Error_Block *eb)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId1);
    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreate1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId1, (Ptr)0xdead1);
}


/* ======== myBegin1 ========
 * invoked before Timer Hwi func */
Void myBegin1(Hwi_Handle hwi)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId1);
    System_printf("myBegin1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId1, (Ptr)0xbeef1);
}


/* ======== myEnd1 ========
 * invoked after Timer Hwi func */
Void myEnd1(Hwi_Handle hwi)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId1);
    System_printf("myEnd1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId1, (Ptr)0xc0de1);
}


/* HookSet 2 functions */
/* ======== myRegister2 ========
 * invoked during Hwi module startup before main
 * for each HookSet */
Void myRegister2(Int hookSetId)
{
    System_printf("myRegister2: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId2 = hookSetId;
}
```

```
/* ======== myCreate2 ========
 * invoked during Hwi module startup before main
 * for statically created Hwis */
Void myCreate2(Hwi_Handle hwi, Error_Block *eb)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId2);
    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreate2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId2, (Ptr)0xdead2);
}


/* ======== myBegin2 ========
 * invoked before Timer Hwi func */
Void myBegin2(Hwi_Handle hwi)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId2);
    System_printf("myBegin2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId2, (Ptr)0xbeef2);
}


/* ======== myEnd2 ========
 * invoked after Timer Hwi func */
Void myEnd2(Hwi_Handle hwi)
{
    Ptr pEnv;

    pEnv = Hwi_getHookContext(hwi, myHookSetId2);
    System_printf("myEnd2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId2, (Ptr)0xc0de2);
    myEnd2Flag = TRUE;
}


/* ======== myTimerFunc ========
 * Timer interrupt handler */
Void myTimerFunc(UArg arg)
{
    System_printf("Entering myTimerHwi\n");
}
```

```c
/* ======== myTaskFunc ======== */
Void myTaskFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering myTask.\n");

    Timer_start(myTimer);
    /* wait for timer interrupt and myEnd2 to complete */
    while (!myEnd2Flag) {
        ;
    }
    System_printf("myTask exiting ...\n");
}


/* ======== myIdleFunc ======== */
Void myIdleFunc()
{
    System_printf("Entering myIdleFunc().\n");
    System_exit(0);
}

/* ======== main ======== */
Int main(Int argc, Char* argv[])
{
    System_printf("Starting HwiHookExample...\n");
    BIOS_start();
    return (0);
}
```

This is the XDCtools configuration script for the example:

```
/* pull in Timestamp to print time in hook functions */
xdc.useModule('xdc.runtime.Timestamp');

/* Disable Clock so that ours is the only Timer allocated */
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.clockEnabled = false;

var Idle = xdc.useModule('ti.sysbios.knl.Idle');
Idle.addFunc('&myIdleFunc');


/* Create myTask with default task params */
var Task = xdc.useModule('ti.sysbios.knl.Task');
var taskParams = new Task.Params();
Program.global.myTask = Task.create('&myTaskFunc', taskParams);
```

```
/* Create myTimer as source of Hwi */
var Timer = xdc.useModule('ti.sysbios.hal.Timer');
var timerParams = new Timer.Params();
timerParams.startMode = Timer.StartMode_USER;
timerParams.runMode = Timer.RunMode_ONESHOT;
timerParams.period = 1000; // 1ms
Program.global.myTimer = Timer.create(Timer.ANY, "&myTimerFunc", timerParams);


/* Define and add two Hwi HookSets
 * Notice, no deleteFxn is provided.
 */
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');


/* Hook Set 1 */
Hwi.addHookSet({
      registerFxn: '&myRegister1',
      createFxn: '&myCreate1',
      beginFxn: '&myBegin1',
      endFxn: '&myEnd1',
});


/* Hook Set 2 */
Hwi.addHookSet({
      registerFxn: '&myRegister2',
      createFxn: '&myCreate2',
      beginFxn: '&myBegin2',
      endFxn: '&myEnd2',
});
```

The program output is as follows:

```
myRegister1: assigned hookSet Id = 0
myRegister2: assigned hookSet Id = 1
myCreate1: pEnv = 0x0, time = 0
myCreate2: pEnv = 0x0, time = 0
Starting HwiHookExample...
Entering myTask.
myBegin1: pEnv = 0xdead1, time = 75415
myBegin2: pEnv = 0xdead2, time = 75834
Entering myTimerHwi
myEnd1: pEnv = 0xbeef1, time = 76427
myEnd2: pEnv = 0xbeef2, time = 76830
myTask exiting ...
Entering myIdleFunc().
```

3.4 Software Interrupts

Software interrupts are patterned after hardware ISRs. The Swi module in SYS/BIOS provides a software interrupt capability. Software interrupts are triggered programmatically, through a call to a SYS/BIOS API such as Swi_post(). Software interrupts have priorities that are higher than tasks but lower than hardware interrupts. See the video introducing Swis for an overview.

Note:	The Swi module should not be confused with the SWI instruction that exists on many processors. The SYS/BIOS Swi module is independent from any target/device-specific software interrupt features.

Swi threads are suitable for handling application tasks that occur at slower rates or are subject to less severe real-time deadlines than those of Hwis.

The SYS/BIOS APIs that can trigger or post a Swi are:

- Swi_andn()
- Swi_dec()
- Swi_inc()
- Swi_or()
- Swi_post()

The Swi manager controls the execution of all Swi functions. When the application calls one of the APIs above, the Swi manager schedules the function corresponding to the specified Swi for execution. To handle Swi functions, the Swi manager uses Swi objects.

If a Swi is posted, it runs only after all pending Hwis have run. A Swi function in progress can be preempted at any time by a Hwi; the Hwi completes before the Swi function resumes. On the other hand, Swi functions always preempt tasks. All pending Swis run before even the highest priority task is allowed to run. In effect, a Swi is like a task with a priority higher than all ordinary tasks.

Note:	Two things to remember about Swi functions are:
	A Swi function runs to completion unless it is interrupted by a Hwi or preempted by a higher-priority Swi.
	Any hardware ISR that triggers or posts a Swi must have been invoked by the Hwi dispatcher (or by generated interrupt stubs on platforms for which the Hwi dispatcher is not provided, such as the MSP430). That is, the Swi must be triggered by a function called from a Hwi object.

3.4.1 Creating Swi Objects

As with many other SYS/BIOS objects, you can create Swi objects either dynamically—with a call to Swi_create()—or statically in the configuration. Swis you create dynamically can also be deleted during program execution.

To add a new Swi to the configuration, create a new Swi object in the configuration script. Set the function property for each Swi to run a function when the object is triggered by the application. You can also configure up to two arguments to be passed to each Swi function.

As with all modules with instances, you can determine from which memory segment Swi objects are allocated. Swi objects are accessed by the Swi manager when Swis are posted and scheduled for execution.

For complete reference information on the Swi API, configuration, and objects, see the Swi module in the "ti.sysbios.knl" package documentation in the online documentation. (For information on running online help, see Section 1.6.1, *Using the API Reference Help System*, page 1-22.)

To create a Swi object dynamically, use a call with this syntax:

```
Swi_Handle swi0;
Swi_Params swiParams;
Error_Block eb;

Error_init(&eb);
Swi_Params_init(&swiParams);

swi0 = Swi_create(swiFunc, &swiParams, &eb);
if (swi0 == NULL) {
    System_abort("Swi create failed");
}
```

Here, swi0 is a handle to the created Swi object, swiFunc is the name of the function associated with the Swi, and swiParams is a structure of type Swi_Params that contains the Swi instance parameters (priority, arg0, arg1, etc). If NULL is passed instead of a pointer to an actual Swi_Params struct, a default set of parameters is used. "eb" is an error block you can use to handle errors that may occur during Swi object creation.

Note: Swi_create() cannot be called from the context of a Hwi or another Swi thread. Applications that dynamically create Swi threads must do so from either the context of the main() function or a Task thread.

To create a Swi object in an XDCtools configuration file, use statements like these:

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var swiParams = new Swi.Params();
program.global.swi0 = Swi.create(swiParams);
```

3.4.2   *Setting Software Interrupt Priorities*

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

The number of Swi priorities supported within an application is configurable up to a maximum 32. The maximum number of priority levels is 16 for MSP430 and C28x. The default number of priority levels is 16. The lowest priority level is 0. Thus, by default, the highest priority level is 15.

You cannot sort Swis within a single priority level. They are serviced in the order in which they were posted.

3.4.3 *Software Interrupt Priorities and System Stack Size*

When a Swi is posted, its associated Swi function is invoked using the system stack. While you can have up to 32 Swi priority levels on some targets, keep in mind that in the worst case, each Swi priority level can result in a nesting of the Swi scheduling function (that is, the lowest priority Swi is preempted by the next highest priority Swi, which, in turn, is preempted by the next highest, …). This results in an increasing stack size requirement for each Swi priority level actually used. Thus, giving Swis the same priority level is more efficient in terms of stack size than giving each Swi a separate priority.

The default system stack size is 4096 bytes. You can set the system stack size by adding the following line to your config script:

```
Program.stack = yourStackSize;
```

> **Note:** The Clock module creates and uses a Swi with the maximum Swi priority (that is, if there are 16 Swi priorities, the Clock Swi has priority 15).

The following table shows the amount of system stack required to absorb the worst-case Swi interrupt nesting. This first number is the amount of system stack space required for the first priority level on a target. The second number shows the amount of stack space required for each subsequent priority level used in the application.

Table 3–5. System Stack Use for Swi Nesting by Target Family

Target Family	Stack Consumed by First Priority Level	Stack Consumed by Subsequent Priority Levels	Units
M3	104	88	8-bit bytes
MSP430	78	32	8-bit bytes
MSP430X	90	60	8-bit bytes
MSP430X_small	78	32	8-bit bytes
C674	108	120	8-bit bytes
C64P	108	120	8-bit bytes
C64T	108	120	8-bit bytes
C28_float	83	40	16-bit words
C28_large	81	34	16-bit words
Arm9	104	80	8-bit bytes
A8F	160	72	8-bit bytes

See Section 3.3.2 for information about system stack use by Hwis and Section 3.5.3 for information about task stack size.

3.4.4   Execution of Software Interrupts

Swis 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—Hwi functions, Clock functions, Idle functions, or other Swi functions.

When a Swi is posted, the Swi manager adds it to a list of posted Swis that are pending execution. The Swi manager checks whether Swis are currently enabled. If they are not, as is the case inside a Hwi function, the Swi manager returns control to the current thread.

If Swis 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, a Task, 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 a 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.

When multiple Swis of the same priority level have been posted, their respective Swi functions are executed in the order the Swis were posted.

There are two important things to remember about Swi:

- When a Swi starts executing, it must run to completion without blocking.

- When called from within a hardware ISR, the code calling any Swi function that can trigger or post a Swi must be invoked by the Hwi dispatcher (or by generated interrupt stubs on platforms for which the Hwi dispatcher is not provided, such as the MSP430). That is, the Swi must be triggered by a function called from a Hwi object.

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

If a 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 a Hwi is executed only once if the Hwi is triggered multiple times before the CPU clears the corresponding interrupt flag bit in the interrupt flag register. (See Section 3.4.5 for more information on how to handle Swis that are posted multiple times before they are scheduled for execution.)

Applications should not make any assumptions about the order in which Swi functions of equal priority are called. However, a Swi function can safely post itself (or be posted by another interrupt). If more than one is pending, all Swi functions are called before any tasks run.

3.4.5   Using a Swi Object's Trigger Variable

Each Swi object has an associated 32-bit trigger variable for C6x targets and a 16-bit trigger variable for C5x, C28x, and MSP430 targets. This is used either to determine whether to post the Swi or to provide values that can be evaluated within the Swi function.

Swi_post(), Swi_or(), and Swi_inc() post a Swi object unconditionally:

- Swi_post() does not modify the value of the Swi object trigger when it is used to post a Swi.

- Swi_or() sets the bits in the trigger determined by a mask that is passed as a parameter, and then posts the Swi.

- Swi_inc() increases the Swi's trigger value by one before posting the Swi object.

- Swi_andn() and Swi_dec() post a Swi object only if the value of its trigger becomes 0:

- Swi_andn() clears the bits in the trigger determined by a mask passed as a parameter.

- Swi_dec() decreases the value of the trigger by one.

Table 3-6 summarizes the differences between these functions.

Table 3-6. Swi Object Function Differences

Action	Treats Trigger as Bitmask	Treats Trigger as Counter	Does not Modify Trigger
Always post	Swi_or()	Swi_inc()	Swi_post()
Post if it becomes zero	Swi_andn()	Swi_dec()	—

The Swi trigger allows you to have tighter control over the conditions that should cause a Swi function to be posted, or the number of times the Swi function should be executed once the Swi is posted and scheduled for execution.

To access the value of its trigger, a Swi function can call Swi_getTrigger(). Swi_getTrigger() can be called only from the Swi object's function. The value returned by Swi_getTrigger() is the value of the trigger 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 object's queue, its trigger is reset to its initial value. The initial value of the trigger should be set in the application's configuration script. If while the Swi function is executing, the Swi is posted again, its trigger is updated accordingly. However, this does not affect the value returned by Swi_getTrigger() while the Swi function executes. That is, the trigger value that Swi_getTrigger() returns is the latched trigger value when the Swi was removed from the list of pending Swis. The Swi's trigger 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 trigger if a new posting occurs, even if the Swi function has not finished its execution.

For example, if a 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 a Swi function must always run multiple times when the Swi object is posted multiple times, Swi_inc() should be used to post the Swi as

shown in Figure 3-3.

When a 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 trigger to know how many times it was posted before it was scheduled to run, and proceed to execute the same function as many times as the value of the trigger.

Figure 3-3. Using Swi_inc() to Post a Swi


```
mySwiFxn()
{ . . .
  repetitions = SWI_getTrigger();
  while (repetitions --) {
    'run Swi function'
  }
...
}
```
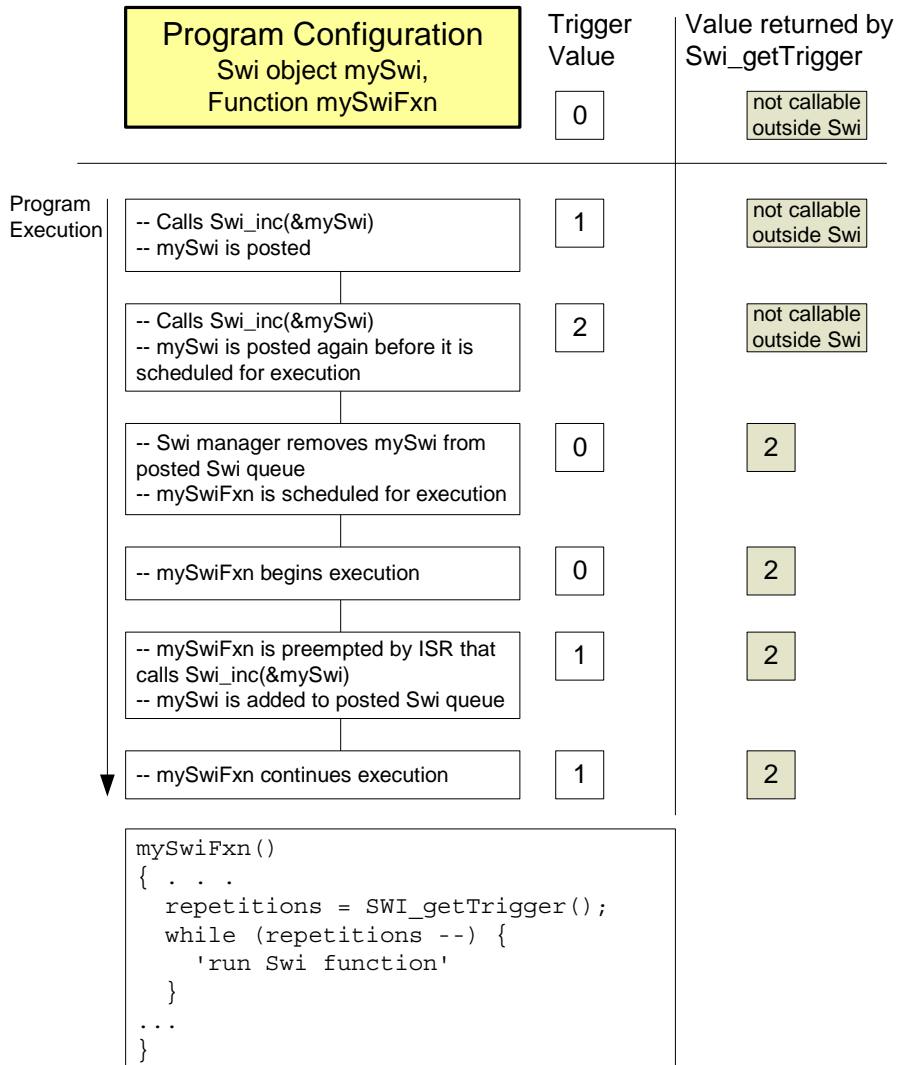
If more than one event must always happen for a given Swi to be triggered, Swi_andn() should be used to post the corresponding Swi object as shown in Figure 3-4. For example, if a Swi must wait for input data from two different devices before it can proceed, its trigger should have two set bits when the Swi object is configured. When both functions 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 trigger default value. Hence, the Swi is posted only when data from both processes is ready.

Figure 3-4. Using Swi_andn() to Post a Swi
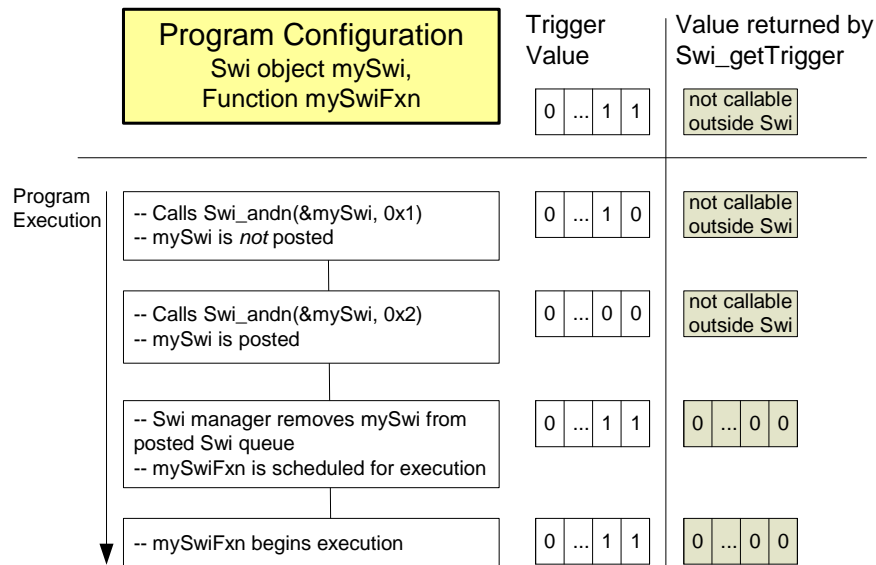
If the program execution requires that multiple occurrences of the same event must take place before a Swi is posted, Swi_dec() should be used to post the Swi as shown in Figure 3-5. By configuring the Swi trigger to be equal to the number of occurrences of the event before the Swi should be posted and calling Swi_dec() every time the event occurs, the Swi is posted only after its trigger reaches 0; that is, after the event has occurred a number of times equal to the trigger value.

Figure 3-5. Using Swi_dec() to Post a Swi
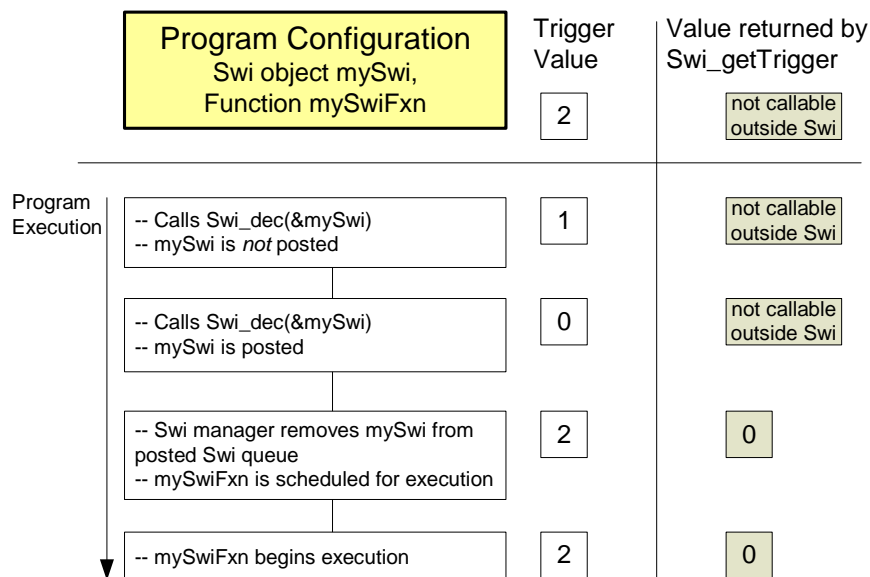
In some situations the Swi function can call different functions depending on the event that posted it. In that case the program can use Swi_or() to post the Swi object unconditionally when an event happens. This is shown in Figure 3-6. The value of the bitmask used by Swi_or() encodes the event type 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 function to execute.

Figure 3-6. Using Swi_or() to Post a Swi.

Program Configuration Swi object mySwi, Function mySwiFxn	Trigger Value	Value returned by Swi_getTrigger
	0 ... 0 0	not callable outside Swi
Program Execution -- Calls Swi_or(&mySwi, 0x1) -- mySwi is posted	0 ... 0 1	not callable outside Swi
-- mySwiFxn is executed	0 ... 0 0	0 ... 0 1
-- Calls Swi_or(&mySwi, 0x2) -- mySwi is posted	0 ... 1 0	not callable outside Swi
-- mySwiFxn is executed	0 ... 0 0	0 ... 1 0

```
mySwiFxn()
{
    ...
    eventType = Swi_getTrigger();
    switch (eventType) {
        case '0x1': 'run alg 1'
        case '0x2': 'run alg 2'
        case '0x3': 'run alg 3'
    }
    ...
}
```

3.4.6   Benefits and Tradeoffs

There are several benefits to using Swis instead of Hwis:

- By modifying shared data structures in a Swi function instead of a Hwi, you can get mutual exclusion by disabling Swis while a Task accesses the shared data structure (see page 3–76). This allows the system to respond to events in real-time using Hwis. In contrast, if a Hwi function modified a shared data structure directly, Tasks would need to disable Hwis to access data structures in a mutually exclusive way. Obviously, disabling Hwis may degrade the performance of a real-time system.

- It often makes sense to break long ISRs into two pieces. The Hwi takes care of the extremely time-critical operation and defers less critical processing to a Swi function by posting the Swi within the Hwi function.

Remember that a Swi function must complete before any blocked Task is allowed to run.

3.4.7 *Synchronizing Swi Functions*

Within an Idle, Task, or Swi function, you can temporarily prevent preemption by a higher-priority Swi by calling Swi_disable(), which disables all Swi preemption. To reenable Swi preemption, call Swi_restore(). Swis are enabled or disabled as a group. An individual Swi cannot be enabled or disabled on its own.

When SYS/BIOS finishes initialization and before the first task is called, Swis have been enabled. If an application wishes to disable Swis, it calls Swi_disable() as follows:

```
key = Swi_disable();
```

The corresponding enable function is Swi_restore() where key is a value used by the Swi module to determine if Swi_disable() has been called more than once.

```
Swi_restore(key);
```

This allows nesting of Swi_disable() / Swi_restore() calls, since only the outermost Swi_restore() call actually enables Swis. In other words, a task can disable and enable Swis without having to determine if Swi_disable() has already been called elsewhere.

When Swis are disabled, a posted Swi function does not run at that time. The interrupt is "latched" in software and runs when Swis are enabled and it is the highest-priority thread that is ready to run.

To delete a dynamically created Swi, use Swi_delete(). The memory associated with Swi is freed. Swi_delete() can only be called from the task level.

3.4.8 *Swi Hooks*

The Swi module supports the following set of Hook functions:

- **Register.** A function called before any statically created Swis are initialized at runtime. The register hook is called at boot time before main() and before interrupts are enabled.

- **Create.** A function called when a Swi is created. This includes Swis that are created statically and those created dynamically using Swi_create().

- **Ready.** A function called when any Swi becomes ready to run.

- **Begin.** A function called just prior to running a Swi function.

- **End.** A function called just after returning from a Swi function.

- **Delete.** A function called when a Swi is deleted at runtime with Swi_delete().

The following Swi_HookSet structure type definition encapsulates the hook functions supported by the Swi module:

```
typedef struct Swi_HookSet {
    Void (*registerFxn)(Int);               /* Register Hook */
    Void (*createFxn)(Handle, Error.Block *); /* Create Hook */
    Void (*readyFxn)(Handle);               /* Ready Hook */
    Void (*beginFxn)(Handle);               /* Begin Hook */
    Void (*endFxn)(Handle);                 /* End Hook */
    Void (*deleteFxn)(Handle);              /* Delete Hook */
};
```

Swi Hook functions can only be configured statically.

When more than one Hook Set is defined, the individual hook functions of a common type are invoked in hook ID order.

3.4.8.1 Register Function

The Register function is provided to allow a hook set to store its corresponding hook ID. This ID can be passed to Swi_setHookContext() and Swi_getHookContext() to set or get hook-specific context. The Register function must be specified if the hook implementation needs to use Swi_setHookContext() or Swi_getHookContext().

The registerFxn function is called during system initialization before interrupts have been enabled.

The Register functions has the following signature:

```
Void registerFxn(Int id);
```

3.4.8.2 Create and Delete Functions

The Create and Delete functions are called whenever a Swi is created or deleted. The Create function is passed an Error_Block that is to be passed to Memory_alloc() for applications that require additional context storage space.

The createFxn and deleteFxn functions are called with interrupts enabled (unless called at boot time or from main()).

These functions have the following signatures.

```
Void createFxn(Swi_Handle swi, Error_Block *eb);
Void deleteFxn(Swi_Handle swi);
```

3.4.8.3 Ready, Begin and End Functions

The Ready, Begin and End hook functions are called with interrupts enabled. The readyFxn function is called when a Swi is posted and made ready to run. The beginFxn function is called right before the function associated with the given Swi is run. The endFxn function is called right after returning from the Swi function.

Both readyFxn and beginFxn hooks are provided because a Swi may be posted and ready but still pending while a higher-priority thread completes.

These functions have the following signatures:

```
Void readyFxn(Swi_Handle swi);
Void beginFxn(Swi_Handle swi);
Void endFxn(Swi_Handle swi);
```

3.4.8.4 *Swi Hooks Example*

The following example application uses two Swi hook sets. This example demonstrates how to read and write the Hook Context Pointer associated with each hook set.

The XDCtools configuration script and program output are shown after the C code listing.

This is the C code for the example:

```c
/* ======== SwiHookExample.c ========
 * This example demonstrates basic Swi hook usage */

#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Timestamp.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/hal/Timer.h>
#include <ti/sysbios/knl/Swi.h>

Swi_Handle mySwi;
Int myHookSetId1, myHookSetId2;


/* HookSet 1 functions */

/* ======== myRegister1 ========
 * invoked during Swi module startup before main
 * for each HookSet */
Void myRegister1(Int hookSetId)
{
    System_printf("myRegister1: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId1 = hookSetId;
}


/* ======== myCreate1 ========
 * invoked during Swi_create for dynamically created Swis */
Void myCreate1(Swi_Handle swi, Error_Block *eb)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId1);

    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreate1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xdead1);
}
```

```
//* ======== myReady1 ========
 * invoked when Swi is posted */
Void myReady1(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId1);

    System_printf("myReady1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xbeef1);
}

/* ======== myBegin1 ========
 * invoked just before Swi func is run */
Void myBegin1(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId1);

    System_printf("myBegin1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xfeeb1);
}

/* ======== myEnd1 ========
 * invoked after Swi func returns */
Void myEnd1(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId1);

    System_printf("myEnd1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xc0de1);
}

/* ======== myDelete1 ========
 * invoked upon Swi deletion */
Void myDelete1(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId1);

    System_printf("myDelete1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
}
```

```
/* HookSet 2 functions */

/* ======== myRegister2 ========
 * invoked during Swi module startup before main
 * for each HookSet */
Void myRegister2(Int hookSetId)
{
    System_printf("myRegister2: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId2 = hookSetId;
}

/* ======== myCreate2 ========
 * invoked during Swi_create for dynamically created Swis */
Void myCreate2(Swi_Handle swi, Error_Block *eb)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId2);

    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreate2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xdead2);
}


/* ======== myReady2 ========
 * invoked when Swi is posted */
Void myReady2(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId2);

    System_printf("myReady2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xbeef2);
}

/* ======== myBegin2 ========
 * invoked just before Swi func is run */
Void myBegin2(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId2);

    System_printf("myBegin2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xfeeb2);
}
```

```
/* ======== myEnd2 ======== 
 * invoked after Swi func returns */
Void myEnd2(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId2);

    System_printf("myEnd2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xc0de2);
}

/* ======== myDelete2 ======== 
 * invoked upon Swi deletion */
Void myDelete2(Swi_Handle swi)
{
    Ptr pEnv;

    pEnv = Swi_getHookContext(swi, myHookSetId2);

    System_printf("myDelete2: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
}

/* ======== mySwiFunc ======== */
Void mySwiFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering mySwi.\n");
}

/* ======== myTaskFunc ======== */
Void myTaskFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering myTask.\n");

    System_printf("Posting mySwi.\n");
    Swi_post(mySwi);

    System_printf("Deleting mySwi.\n");
    Swi_delete(&mySwi);

    System_printf("myTask exiting ...\n");
}


/* ======== myIdleFunc ======== */
Void myIdleFunc()
{
    System_printf("Entering myIdleFunc().\n");
    System_exit(0);
}
```

```
/* ======== main ======== */
Int main(Int argc, Char* argv[])
{
    Error_Block eb;

    Error_init(&eb);

    System_printf("Starting SwiHookExample...\n");

    /* Create mySwi with default params
     * to exercise Swi Hook Functions     */
    mySwi = Swi_create(mySwiFunc, NULL, &eb);
    if (mySwi == NULL) {
        System_abort("Swi create failed");
    }

    BIOS_start();
    return (0);
}
```

This is the XDCtools configuration script for the example:

```
/* pull in Timestamp to print time in hook functions */
xdc.useModule('xdc.runtime.Timestamp');

/* Disable Clock so that ours is the only Swi in the application */
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.clockEnabled = false;

var Idle = xdc.useModule('ti.sysbios.knl.Idle');
Idle.addFunc('&myIdleFunc');

/* Create myTask with default task params */
var Task = xdc.useModule('ti.sysbios.knl.Task');
var taskParams = new Task.Params();
Program.global.myTask = Task.create('&myTaskFunc', taskParams);

/* Define and add two Swi Hook Sets */
var Swi = xdc.useModule("ti.sysbios.knl.Swi");

/* Hook Set 1 */
Swi.addHookSet({
    registerFxn: '&myRegister1',
    createFxn: '&myCreate1',
    readyFxn: '&myReady1',
    beginFxn: '&myBegin1',
    endFxn: '&myEnd1',
    deleteFxn: '&myDelete1'
});
```

```
/* Hook Set 2 */
Swi.addHookSet({
        registerFxn: '&myRegister2',
        createFxn: '&myCreate2',
        readyFxn: '&myReady2',
        beginFxn: '&myBegin2',
        endFxn: '&myEnd2',
        deleteFxn: '&myDelete2'
});
```

This is the output for the application:

```
myRegister1: assigned hookSet Id = 0
myRegister2: assigned hookSet Id = 1
Starting SwiHookExample...
myCreate1: pEnv = 0x0, time = 315
myCreate2: pEnv = 0x0, time = 650
Entering myTask.
Posting mySwi.
myReady1: pEnv = 0xdead1, time = 1275
myReady2: pEnv = 0xdead2, time = 1678
myBegin1: pEnv = 0xbeef1, time = 2093
myBegin2: pEnv = 0xbeef2, time = 2496
Entering mySwi.
myEnd1: pEnv = 0xfeeb1, time = 3033
myEnd2: pEnv = 0xfeeb2, time = 3421
Deleting mySwi.
myDelete1: pEnv = 0xc0de1, time = 3957
myDelete2: pEnv = 0xc0de2, time = 4366
myTask exiting ...
Entering myIdleFunc().
```

3.5 Tasks

SYS/BIOS task objects are threads that are managed by the Task module. Tasks have higher priority than the Idle Loop and lower priority than hardware and software interrupts. See the video introducing Tasks for an overview.

The Task module dynamically schedules and preempts tasks based on the task's priority level and the task's current execution state. This ensures that the processor is always given to the highest priority thread that is ready to run. There are up to 32 priority levels available for tasks, with the default number of levels being 16. The maximum number of priority levels is 16 for MSP430 and C28x. The lowest priority level (0) is reserved for running the Idle Loop.

The Task module provides a set of functions that manipulate task objects. They access Task objects through handles of type Task_Handle.

The kernel maintains a copy of the processor registers for each task object. Each task has its own runtime stack for storing local variables as well as for further nesting of function calls. See Section 3.5.3 for information about task stack sizes.

All tasks executing within a single program share a common set of global variables, accessed according to the standard rules of scope defined for C functions.

3.5.1 Creating Tasks

You can create Task objects either dynamically with a call to Task_create() or statically in the configuration. Tasks that you create dynamically can also be deleted during program execution.

3.5.1.1 Creating and Deleting Tasks Dynamically

You can spawn SYS/BIOS tasks by calling the function Task_create(), whose parameters include the address of a C function in which the new task begins its execution. The value returned by Task_create() is a handle of type Task_Handle, which you can then pass as an argument to other Task functions.

This C example creates a task:

```
Task_Params taskParams;
Task_Handle task0;
Error_Block eb;


Error_init(&eb);


/* Create 1 task with priority 15 */
Task_Params_init(&taskParams);
taskParams.stackSize = 512;
taskParams.priority = 15;
task0 = Task_create((Task_FuncPtr)hiPriTask, &taskParams, &eb);
if (task0 == NULL) {
    System_abort("Task create failed");
}
```

If NULL is passed instead of a pointer to an actual Task_Params struct, a default set of parameters is used. The "eb" is an error block that you can use to handle errors that may occur during Task object creation. See Section 3.5.3 for information about task stack sizes.

A task becomes active when it is created and preempts the currently running task if it has a higher priority.

The memory used by Task objects and stacks can be reclaimed by calling Task_delete(). Task_delete() removes the task from all internal queues and frees the task object and stack.

Any Semaphores or other resources held by the task are *not* released. Deleting a task that holds such resources is often an application design error, although not necessarily so. In most cases, such resources should be released prior to deleting the task. It is only safe to delete a Task that is either in the Terminated or Inactive State.

```
Void Task_delete(Task_Handle *task);
```

3.5.1.2 Creating Tasks Statically

You can also create tasks statically within a configuration script. The configuration allows you to set a number of properties for each task and for the Task manager itself.

For a complete description of all Task properties, see the Task module in the "ti.sysbios.knl" package documentation in the online documentation. (For information on running online help, see Section 1.6.1, *Using the API Reference Help System*, page 1-22.)

While it is running, a task that was created statically behaves exactly the same as a task created with Task_create(). You cannot use the Task_delete() function to delete statically-created tasks.

The Task module automatically creates the Task_idle task and gives it the lowest task priority (0). It runs the functions defined for the Idle objects when no higher-priority Hwi, Swi, or Task is running.

When you configure tasks to have equal priority, they are scheduled in the order in which they are created in the configuration script. Tasks can have up to 32 priority levels with 16 being the default. The maximum number of priority levels is 16 for MSP430 and C28x. The highest level is the number of priorities defined minus 1, and the lowest is 0. The priority level of 0 is reserved for the system idle task. You cannot sort tasks within a single priority level by setting the order property.

If you want a task to be initially inactive, set its priority to -1. Such tasks are not scheduled to run until their priority is raised at runtime.

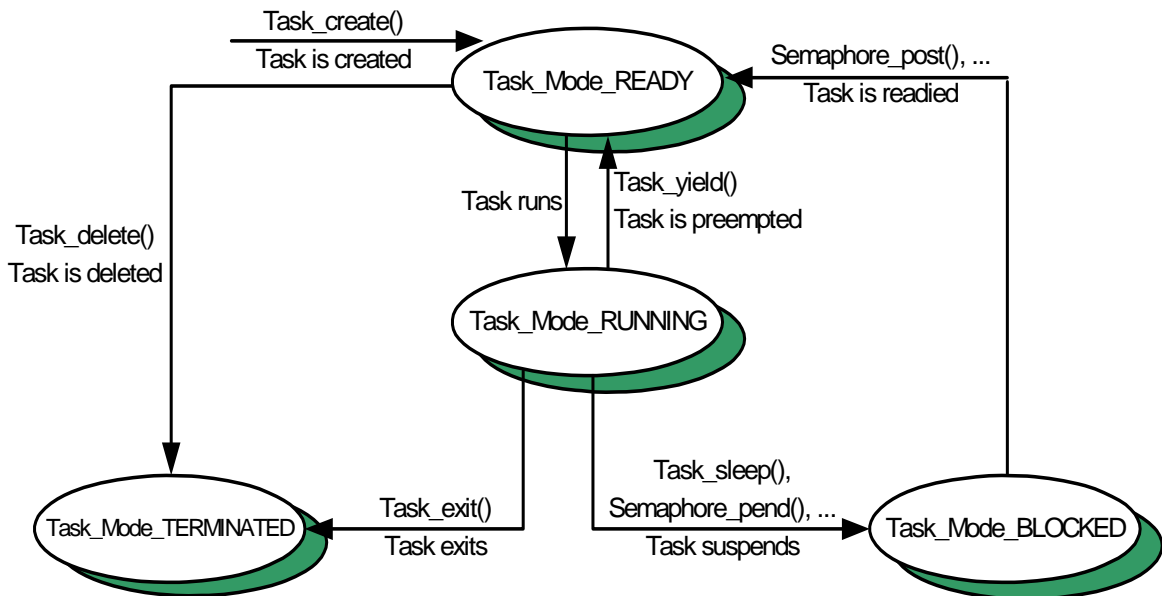### 3.5.2   *Task Execution States and Scheduling*

Each Task object is always in one of four possible states of execution:

- **Task_Mode_RUNNING**, which means the task is the one actually executing on the system's processor.

- **Task_Mode_READY,** which means the task is scheduled for execution subject to processor availability.

- **Task_Mode_BLOCKED,** which means the task cannot execute until a particular event occurs within the system.

- **Task_Mode_TERMINATED,** which means the task is "terminated" and does not execute again.

- **Task_Mode_INACTIVE,** which means the task has a priority equal to -1 and is in a pre-Ready state. This priority can be set when the task is created or by calling the Task_setPri() API at runtime.

Tasks are scheduled for execution according to a priority level assigned by the application. There can be no more than one running task. As a rule, no ready task has a priority level greater than that of the currently running task, since Task preempts the running task in favor of the higher-priority ready task. Unlike many time-sharing operating systems that give each task its "fair share" of the processor, SYS/BIOS *immediately* preempts the current task whenever a task of higher priority becomes ready to run.

The maximum priority level is Task_numPriorities-1 (default=15; maximum=31). The minimum priority is 1. If the priority is less than 0, the task is barred from further execution until its priority is raised at a later time by another task. If the priority equals Task_numPriorities-1, the task cannot be preempted by another task. A highest-priority task can still call Semaphore_pend(), Task_sleep(), or some other blocking call to allow tasks of lower priority to run. A Task's priority can be changed at runtime with a call to Task_setPr().

During the course of a program, each task's mode of execution can change for a number of reasons. Figure 3-7 shows how execution modes change.

Figure 3-7. Execution Mode Variations

Functions in the Task, Semaphore, Event, and Mailbox modules alter the execution state of task objects: blocking or terminating the currently running task, readying a previously suspended task, re-scheduling the current task, and so forth.

There is *one* task whose execution mode is Task_Mode_RUNNING. If all program tasks are blocked and no Hwi or Swi is running, Task executes the Task_idle task, whose priority is lower than all other tasks in the system. When a task is preempted by a Hwi or Swi, the task execution mode returned for that task by Task_stat() is still Task_Mode_RUNNING because the task will run when the preemption ends.

Notes:	Do not make blocking calls, such as Semaphore_pend() or Task_sleep(), from within an Idle function. Doing so causes the application to terminate.

When the Task_Mode_RUNNING task transitions to any of the other three states, control switches to the highest-priority task that is ready to run (that is, whose mode is Task_Mode_READY). A Task_Mode_RUNNING task transitions to one of the other modes in the following ways:

- The running task becomes Task_Mode_TERMINATED by calling Task_exit(), which is automatically called if and when a task returns from its top-level function. After all tasks have returned, the Task manager terminates program execution by calling System_exit() with a status code of 0.

- The running task becomes Task_Mode_BLOCKED when it calls a function (for example, Semaphore_pend() or Task_sleep()) that causes the current task to suspend its execution; tasks can move into this state when they are performing certain I/O operations, awaiting availability of some shared resource, or idling.

- The running task becomes Task_Mode_READY and is preempted whenever some other, higher-priority task becomes ready to run. Task_setPri() can cause this type of transition if the priority of the current task is no longer the highest in the system. A task can also use Task_yield() to yield to other tasks with the same priority. A task that yields becomes ready to run.

A task that is currently Task_Mode_BLOCKED transitions to the ready state in response to a particular event: completion of an I/O operation, availability of a shared resource, the elapse of a specified period of time, and so forth. By virtue of becoming Task_Mode_READY, this task is scheduled for execution according to its priority level; and, of course, this task immediately transitions to the running state if its priority is higher than the currently executing task. Task schedules tasks of equal priority on a first-come, first-served basis.

3.5.3 Task Stacks

The kernel maintains a copy of the processor registers for each Task object. Each Task has its own runtime stack for storing local variables as well as for further nesting of function calls.

You can specify the stack size separately for each Task object when you create the Task object statically or dynamically.

Each task stack must be large enough to handle both its normal function calls and two full interrupting Hwi contexts.

The following table shows the amount of task stack required to absorb the worst-case interrupt nesting. These numbers represent two full Hwi interrupt contexts plus space used by the task scheduler for its local variables. Additional nested interrupt contexts are pushed onto the common system stack.

Table 3–7. Task Stack Use by Target Family

Target Family	Stack Consumed	Units
M3	100	8-bit bytes
MSP430	52	8-bit bytes
MSP430X	96	8-bit bytes
MSP430X_small	52	8-bit bytes
C674	676	8-bit bytes
C64P	676	8-bit bytes
C64T	428	8-bit bytes
C28_float	123	16-bit words
C28_large	97	16-bit words
Arm9	184	8-bit bytes
A8F	280	8-bit bytes

When a Task is preempted, a task stack may be required to contain either two interrupting Hwi contexts (if the Task is preempted by a Hwi) or one interrupting Hwi context and one Task preemption context (if the Task is preempted by a higher-priority Task). Since the Hwi context is larger than the Task context, the numbers given are for two Hwi contexts. If a Task blocks, only those registers that a C function must save are saved to the task stack.

Another way to find the correct stack size is to make the stack size large and then use Code Composer Studio software to find the stack size actually used.

See Section 3.3.2 for information about system stack use by Hwis and Section 3.4.3 for information about system stack size.

3.5.4 Testing for Stack Overflow

When a task uses more memory than its stack has been allocated, it can write into an area of memory used by another task or data. This results in unpredictable and potentially fatal consequences. Therefore, a means of checking for stack overflow is useful.

By default, the Task module checks to see whether a Task stack has overflowed at each Task switch. To improve Task switching latency, you can disable this feature the Task.checkStackFlag property to false.

The function Task_stat() can be used to watch stack size. The structure returned by Task_stat() contains both the size of its stack and the maximum number of MAUs ever used on its stack, so this code segment could be used to warn of a nearly full stack:

```
Task_Stat statbuf;                  /* declare buffer */


Task_stat(Task_self(), &statbuf); /* call func to get status */
if (statbuf.used > (statbuf.stackSize * 9 / 10)) {
    Log_printf(&trace, "Over 90% of task's stack is in use.\n")
}
```

See the Task_stat() information in the "ti.sysbios.knl" package documentation in the online documentation.

You can use the RTOS Object Viewer (ROV) to examine runtime Task stack usage. For information, see Section 6.5.3.

3.5.5 Task Hooks

The Task module supports the following set of Hook functions:

- **Register.** A function called before any statically created Tasks are initialized at runtime. The register hook is called at boot time before main() and before interrupts are enabled.

- **Create.** A function called when a Task is created. This includes Tasks that are created statically and those created dynamically using Task_create() or Task_construct(). The Create hook is called outside of a Task_disable/enable block and before the task has been added to the ready list.

- **Ready.** A function called when a Task becomes ready to run. The ready hook is called from within a Task_disable/enable block with interrupts enabled.

- **Switch.** A function called just before a task switch occurs. The 'prev' and 'next' task handles are passed to the Switch hook. 'prev' is set to NULL for the initial task switch that occurs during SYS/BIOS startup. The Switch hook is called from within a Task_disable/enable block with interrupts enabled.

- **Exit.** A function called when a task exits using Task_exit(). The exit hook is passed the handle of the exiting task. The exit hook is called outside of a Task_disable/enable block and before the task has been removed from the kernel lists.

- **Delete.** A function called when a task is deleted at runtime with Task_delete().

The following HookSet structure type definition encapsulates the hook functions supported by the Task module:

```
typedef struct Task_HookSet {
    Void (*registerFxn)(Int);                 /* Register Hook */
    Void (*createFxn)(Handle, Error.Block *); /* Create Hook */
    Void (*readyFxn)(Handle);                 /* Ready Hook */
    Void (*switchFxn)(Handle, Handle);        /* Switch Hook */
    Void (*exitFxn)(Handle);                  /* Exit Hook */
    Void (*deleteFxn)(Handle);                /* Delete Hook */
};
```

When more than one hook set is defined, the individual hook functions of a common type are invoked in hook ID order.

Task hook functions can only be configured statically.

3.5.5.1 Register Function

The Register function is provided to allow a hook set to store its corresponding hook ID. This ID can be passed to Task_setHookContext() and Task_getHookContext() to set or get hook-specific context. The Register function must be specified if the hook implementation needs to use Task_setHookContext() or Task_getHookContext().

The registerFxn function is called during system initialization before interrupts have been enabled.

The Register function has the following signature:

```
Void registerFxn(Int id);
```

3.5.5.2 Create and Delete Functions

The Create and Delete functions are called whenever a Task is created or deleted. The Create function is passed an Error_Block that is to be passed to Memory_alloc() for applications that require additional context storage space.

The createFxn and deleteFxn functions are called with interrupts enabled (unless called at boot time or from main()).

These functions have the following signatures.

```
Void createFxn(Task_Handle task, Error_Block *eb);
Void deleteFxn(Task_Handle task);
```

3.5.5.3 Switch Function

If a Switch function is specified, it is invoked just before the new task is switched to. The switch function is called with interrupts enabled.

This function can be used for purposes such as saving/restoring additional task context (for example, external hardware registers), checking for task stack overflow, and monitoring the time used by each task.

The switchFxn has the following signature:

```
Void switchFxn(Task_Handle prev, Task_Handle next);
```

3.5.5.4 Ready Function

If a Ready Function is specified, it is invoked whenever a task is made ready to run. The Ready Function is called with interrupts enabled (unless called at boot time or from main()).

The readyFxn has the following signature:

```
Void readyFxn(Task_Handle task);
```

3.5.5.5 Exit Function

If an Exit Function is specified, it is invoked when a task exits (via call to Task_exit() or when a task returns from its' main function). The exitFxn is called with interrupts enabled.

The exitFxn has the following signature:

```
Void exitFxn(Task_Handle task);
```

3.5.5.6 Task Hooks Example

The following example application uses a single Task hook set. This example demonstrates how to read and write the Hook Context Pointer associated with each hook set.

The XDCtools configuration script and program output are shown after the C code listing.

This is the C code for the example:

```
/* ======== TaskHookExample.c ========
 * This example demonstrates basic task hook processing
 * operation for dynamically created tasks. */


#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Types.h>


#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>


Task_Handle myTsk0, myTsk1, myTsk2;
Int myHookSetId, myHookSetId2;


/* HookSet functions */

```

```
/* ======== myRegister ======== 
 * invoked during Swi module startup before main() 
 * for each HookSet */
Void myRegister(Int hookSetId)
{
    System_printf("myRegister: assigned HookSet Id = %d\n", hookSetId);
    myHookSetId = hookSetId;
}


/* ======== myCreate ======== 
 * invoked during Task_create for dynamically 
 * created Tasks */
Void myCreate(Task_Handle task, Error_Block *eb)
{
    String name;
    Ptr pEnv;


    name = Task_Handle_name(task);
    pEnv = Task_getHookContext(task, myHookSetId);


    System_printf("myCreate: task name = '%s', pEnv = 0x%x\n", name, pEnv);
    Task_setHookContext(task, myHookSetId, (Ptr)0xdead);
}


/* ======== myReady ======== 
 * invoked when Task is made ready to run */
Void myReady(Task_Handle task)
{
    String name;
    Ptr pEnv;


    name = Task_Handle_name(task);
    pEnv = Task_getHookContext(task, myHookSetId);


    System_printf("myReady: task name = '%s', pEnv = 0x%x\n", name, pEnv);
    Task_setHookContext(task, myHookSetId, (Ptr)0xc0de);
}


/* ======== mySwitch ======== 
 * invoked whenever a Task switch occurs/is made ready to run */
Void mySwitch(Task_Handle prev, Task_Handle next)
{
    String prevName;
    String nextName;
    Ptr pPrevEnv;
    Ptr pNextEnv;
```

```
        if (prev == NULL) {
            System_printf("mySwitch: ignoring dummy 1st prev Task\n");
        }
        else {
            prevName = Task_Handle_name(prev);
            pPrevEnv = Task_getHookContext(prev, myHookSetId);

            System_printf("mySwitch: prev name = '%s', pPrevEnv = 0x%x\n",
                    prevName, pPrevEnv);
            Task_setHookContext(prev, myHookSetId, (Ptr)0xcafec0de);
        }
        nextName = Task_Handle_name(next);
        pNextEnv = Task_getHookContext(next, myHookSetId);

        System_printf("          next name = '%s', pNextEnv = 0x%x\n",
                    nextName, pNextEnv);
        Task_setHookContext(next, myHookSetId, (Ptr)0xc001c0de);
}


/* ======== myExit ========
 * invoked whenever a Task calls Task_exit() or falls through
 * the bottom of its task function. */
Void myExit(Task_Handle task)
{
    Task_Handle curTask = task;
    String name;
    Ptr pEnv;

    name = Task_Handle_name(curTask);
    pEnv = Task_getHookContext(curTask, myHookSetId);

    System_printf("myExit: curTask name = '%s', pEnv = 0x%x\n", name, pEnv);
    Task_setHookContext(curTask, myHookSetId, (Ptr)0xdeadbeef);
}


/* ======== myDelete ========
 * invoked upon Task deletion */
Void myDelete(Task_Handle task)
{
    String name;
    Ptr pEnv;

    name = Task_Handle_name(task);
    pEnv = Task_getHookContext(task, myHookSetId);

    System_printf("myDelete: task name = '%s', pEnv = 0x%x\n", name, pEnv);
}
```

```
/* Define 3 identical tasks */
Void myTsk0Func(UArg arg0, UArg arg1)
{
    System_printf("myTsk0 Entering\n");
    System_printf("myTsk0 Calling Task_yield\n");
    Task_yield();
    System_printf("myTsk0 Exiting\n");
}


Void myTsk1Func(UArg arg0, UArg arg1)
{
    System_printf("myTsk1 Entering\n");
    System_printf("myTsk1 Calling Task_yield\n");
    Task_yield();
    System_printf("myTsk1 Exiting\n");
}


Void myTsk2Func(UArg arg0, UArg arg1)
{
    System_printf("myTsk2 Entering\n");
    System_printf("myTsk2 Calling Task_yield\n");
    Task_yield();
    System_printf("myTsk2 Exiting\n");
}


/* ======== main ======== */
Int main(Int argc, Char* argv[])
{
    Task_Params params;
    Error_Block eb;

    Error_init(&eb);
    Task_Params_init(&params);

    params.instance->name = "myTsk0";
    myTsk0 = Task_create(myTsk0Func, &params, &eb);
    if (myTsk0 == NULL) {
        System_abort("myTsk0 create failed");
    }

    params.instance->name = "myTsk1";
    myTsk1 = Task_create(myTsk1Func, &params, &eb);
    if (myTsk1 == NULL) {
        System_abort("myTsk1 create failed");
    }
```

```
    params.instance->name = "myTsk2";
    myTsk2 = Task_create(myTsk2Func, &params, &eb);
    if (myTsk2 == NULL) {
        System_abort("myTsk2 create failed");
    }


    BIOS_start();
    return (0);
}


/* ======== myIdleFunc ======== */
Void myIdleFunc()
{
    System_printf("Entering idleFunc().\n");


    Task_delete(&myTsk0);
    Task_delete(&myTsk1);
    Task_delete(&myTsk2);


    System_exit(0);
}
```

The XDCtools configuration script is as follows:

```
/* Lots of System_printf() output requires a bigger bufSize */
SysMin = xdc.useModule('xdc.runtime.SysMin');
SysMin.bufSize = 4096;


var Idle = xdc.useModule('ti.sysbios.knl.Idle');
Idle.addFunc('&myIdleFunc');


var Task = xdc.useModule('ti.sysbios.knl.Task');


/* Enable instance names */
Task.common$.namedInstance = true;


/* Define and add one Task Hook Set */
Task.addHookSet({
    registerFxn: '&myRegister',
    createFxn: '&myCreate',
    readyFxn: '&myReady',
    switchFxn: '&mySwitch',
    exitFxn: '&myExit',
    deleteFxn: '&myDelete',
});
```

The program output is as follows:

```
myRegister: assigned HookSet Id = 0
myCreate: task name = 'ti.sysbios.knl.Task.IdleTask', pEnv = 0x0
myReady: task name = 'ti.sysbios.knl.Task.IdleTask', pEnv = 0xdead
myCreate: task name = 'myTsk0', pEnv = 0x0
myReady: task name = 'myTsk0', pEnv = 0xdead
myCreate: task name = 'myTsk1', pEnv = 0x0
myReady: task name = 'myTsk1', pEnv = 0xdead
myCreate: task name = 'myTsk2', pEnv = 0x0
myReady: task name = 'myTsk2', pEnv = 0xdead
mySwitch: ignoring dummy 1st prev Task
         next name = 'myTsk0', pNextEnv = 0xc0de
myTsk0 Entering
myTsk0 Calling Task_yield
mySwitch: prev name = 'myTsk0', pPrevEnv = 0xc001c0de
         next name = 'myTsk1', pNextEnv = 0xc0de

myTsk1 Entering
myTsk1 Calling Task_yield
mySwitch: prev name = 'myTsk1', pPrevEnv = 0xc001c0de
         next name = 'myTsk2', pNextEnv = 0xc0de
myTsk2 Entering
myTsk2 Calling Task_yield
mySwitch: prev name = 'myTsk2', pPrevEnv = 0xc001c0de
         next name = 'myTsk0', pNextEnv = 0xcafec0de
myTsk0 Exiting
myExit: curTask name = 'myTsk0', pEnv = 0xc001c0de
mySwitch: prev name = 'myTsk0', pPrevEnv = 0xdeadbeef
         next name = 'myTsk1', pNextEnv = 0xcafec0de
myTsk1 Exiting
myExit: curTask name = 'myTsk1', pEnv = 0xc001c0de
mySwitch: prev name = 'myTsk1', pPrevEnv = 0xdeadbeef
         next name = 'myTsk2', pNextEnv = 0xcafec0de
myTsk2 Exiting
myExit: curTask name = 'myTsk2', pEnv = 0xc001c0de
mySwitch: prev name = 'myTsk2', pPrevEnv = 0xdeadbeef
         next name = 'ti.sysbios.knl.Task.IdleTask', pNextEnv = 0xc0de
Entering idleFunc().
myDelete: task name = 'myTsk0', pEnv = 0xcafec0de
myDelete: task name = 'myTsk1', pEnv = 0xcafec0de
myDelete: task name = 'myTsk2', pEnv = 0xcafec0de
```

3.5.6   Task Yielding for Time-Slice Scheduling

Example 3-1 demonstrates a time-slicing scheduling model that can be managed by a user. This model is preemptive and does not require any cooperation (that is, code) by the tasks. The tasks are programmed as if they were the only thread running. Although SYS/BIOS tasks of differing priorities can exist in any given application, the time-slicing model only applies to tasks of equal priority.

In this example, a periodic Clock object is configured to run a simple function that calls the Task_yield() function every 4 clock ticks. Another periodic Clock object is to run a simple function that calls the Semaphore_post() function every 16 milliseconds.

The output of the example code is shown after the code.

Example 3-1 Time-Slice Scheduling

```c
/*
 * ======== slice.c ========
 * This example utilizes time-slice scheduling among three
 * tasks of equal priority. A fourth task of higher
 * priority periodically preempts execution.
 *
 * A periodic Clock object drives the time-slice scheduling.
 * Every 4 milliseconds, the Clock object calls Task_yield()
 * which forces the current task to relinquish access to
 * to the CPU.
 *
 * Because a task is always ready to run, this program
 * does not spend time in the idle loop. Calls to Idle_run()
 * are added to give time to the Idle loop functions
 * occasionally. The call to Idle_run() is within a
 * Task_disable(), Task_restore() block because the call
 * to Idle_run() is not reentrant.
 */


#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Error.h>


#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Idle.h>
#include <ti/sysbios/knl/Task.h>


#include <xdc/cfg/global.h>


Void hiPriTask(UArg arg0, UArg arg1);
Void task(UArg arg0, UArg arg1);
Void clockHandler1(UArg arg);
Void clockHandler2(UArg arg);
Semaphore_Handle sem;
```

```
/* ======== main ======== */
Void main()
{
    Task_Params taskParams;
    Task_Handle myTsk0, myTski;
    Clock_Params clockParams;
    Clock_Handle myClk0, myClk1;
    Error_Block eb;
    UInt i;


    System_printf("Slice example started!\n");


    Error_init(&eb);


    /* Create 1 task with priority 15 */
    Task_Params_init(&taskParams);
    taskParams.stackSize = 512;
                    // Note: Larger stack needed for some targets, including 'C6748
    taskParams.priority = 15;
    myTsk0 = Task_create((Task_FuncPtr)hiPriTask, &taskParams, &eb);
    if (myTsk0 == NULL) {
        System_abort("hiPriTask create failed");
    }


    /* Create 3 tasks with priority 1 */
    /* re-uses taskParams */
    taskParams.priority = 1;
    for (i = 0; i < 3; i++) {
        taskParams.arg0 = i;
        myTski = Task_create((Task_FuncPtr)task, &taskParams, &eb);
        if (myTski == NULL) {
            System_abort("LoPri Task %d create failed", i);
        }
    }


    /*
     * Create clock that calls Task_yield() every 4 Clock ticks
     */
    Clock_Params_init(&clockParams);
    clockParams.period = 4;/* every 4 Clock ticks */
    clockParams.startFlag = TRUE;/* start immediately */
    myClk0 = Clock_create((Clock_FuncPtr)clockHandler1, 4, &clockParams, &eb);
    if (myClk0 == NULL) {
        System_abort("Clock0 create failed");
    }
```

```
    /*
     * Create clock that calls Semaphore_post() every
     * 16 Clock ticks
     */
    clockParams.period = 16;/* every 16 Clock ticks */
    clockParams.startFlag = TRUE;/* start immediately */
    myClk1 = Clock_create((Clock_FuncPtr)clockHandler2, 16, &clockParams, &eb);
    if (myClk1 == NULL) {
        System_abort("Clock1 create failed");
    }


    /*
     * Create semaphore with initial count = 0 and default params
     */
    sem = Semaphore_create(0, NULL, &eb);
    if (sem == NULL) {
        System_abort("Semaphore create failed");
    }


    /* Start SYS/BIOS */
    BIOS_start();
}


/* ======== clockHandler1 ======== */
Void clockHandler1(UArg arg)
{
    /* Call Task_yield every 4 ms */
    Task_yield();
}


/* ======== clockHandler2 ======== */
Void clockHandler2(UArg arg)
{
    /* Call Semaphore_post every 16 ms */
    Semaphore_post(sem);
}


/* ======== task ======== */
Void task(UArg arg0, UArg arg1)
{
    Int time;
    Int prevtime = -1;
    UInt taskKey;


    /* While loop simulates work load of time-sharing tasks */
    while (1) {
        time = Clock_getTicks();
```

```
            /* print time once per clock tick */
            if (time >= prevtime + 1) {
                prevtime = time;
                System_printf("Task %d: time is %d\n", (Int)arg0, time);
            }


            /* check for rollover */
            if (prevtime > time) {
                prevtime = time;
            }


            /* Process the Idle Loop functions */
            taskKey = Task_disable();
            Idle_run();
            Task_restore(taskKey);
        }
}


/* ======== hiPriTask ======== */
Void hiPriTask(UArg arg0, UArg arg1)
{
    static Int numTimes = 0;


    while (1) {
        System_printf("hiPriTask here\n");
        if (++numTimes < 3) {
            Semaphore_pend(sem, BIOS_WAIT_FOREVER);
        }
        else {
            System_printf("Slice example ending.\n");
            System_exit(0);
        }
    }
}
```

The System_printf() output for this example is as follows:

```
Slice example started!
hiPriTask here
Task 0: time is 0
Task 0: time is 1
Task 0: time is 2
Task 0: time is 3
Task 1: time is 4
Task 1: time is 5
Task 1: time is 6
Task 1: time is 7
Task 2: time is 8
Task 2: time is 9
Task 2: time is 10
Task 2: time is 11
Task 0: time is 12
Task 0: time is 13
Task 0: time is 14
Task 0: time is 15
hiPriTask here
Task 1: time is 16
Task 1: time is 17
Task 1: time is 18
Task 1: time is 19
Task 2: time is 20
Task 2: time is 21
Task 2: time is 22
Task 2: time is 23
Task 0: time is 24
Task 0: time is 25
Task 0: time is 26
Task 0: time is 27
Task 1: time is 28
Task 1: time is 29
Task 1: time is 30
Task 1: time is 31
hiPriTask here
Slice example ending.
```

3.6   The Idle Loop

The Idle Loop is the background thread of SYS/BIOS, which runs continuously when no Hwi, Swi, or Task is running. Any other thread can preempt the Idle Loop at any point.

The Idle manager allows you to insert functions that execute within the Idle Loop. The Idle Loop runs the Idle functions you configured. Idle_loop calls the functions associated with each one of the Idle objects one at a time, and then starts over again in a continuous loop.

Idle threads all run at the same priority, sequentially. The functions are called in the same order in which they were created. An Idle function must run to completion before the next Idle function can start running. When the last idle function has completed, the Idle Loop starts the first Idle 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 SYS/BIOS application. The Idle Loop functions run only when no Hwis, Swis, or Tasks need to run.

The CPU load and thread load are computed in an Idle loop function. (Data transfer for between the target and the host is handled by a low-priority task.)

If you configure Task.enableIdleTask to be false, no Idle task is created and the Idle functions are not run. If you want a function to run when there are no other threads ready to run, you can specify such a function using Task.allBlockedFunc.

If you want the Idle Loop to run without creating a dedicated Idle task, you can disable Task.enableIdleTask and configure Task.allBlockedFunc as follows. These statements cause the Idle functions to be run using the stack of the last Task to pend.

```
Task.enableIdleTask = false;
Task.allBlockedFunc = Idle.run;
```

3.7 Example Using Hwi, Swi, and Task Threads

This example depicts a stylized version of the SYS/BIOS Clock module design. It uses a combination of Hwi, Swi, and Task threads.

A periodic timer interrupt posts a Swi that processes the Clock object list. Each entry in the Clock object list has its own period and Clock function. When an object's period has expired, the Clock function is invoked and the period restarted.

Since there is no limit to the number of Clock objects that can be placed in the list and no way to determine the overhead of each Clock function, the length of time spent servicing all the Clock objects is non-deterministic. As such, servicing the Clock objects in the timer's Hwi thread is impractical. Using a Swi for this function is a relatively (as compared with using a Task) lightweight solution to this problem.

The XDCtools configuration script and program output are shown after the C code listing. This is the C code for the example:

```c
/*
 * ======== HwiSwiTaskExample.c ========
 */

#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Error.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/hal/Timer.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/knl/Swi.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Queue.h>

#include <xdc/cfg/global.h>

typedef struct {
    Queue_Elem elem;
    UInt32 timeout;
    UInt32 period;
    Void (*fxn)(UArg);
    UArg arg;
} Clock_Object;

Clock_Object clk1, clk2;
Timer_Handle timer;
Semaphore_Handle sem;
Swi_Handle swi;
Task_Handle task;
Queue_Handle clockQueue;

/* Here on Timer interrupt */
Void hwiFxn(UArg arg)
{
    Swi_post(swi);
}
```

```c
/* Swi thread to handle Timer interrupt */
Void swiFxn(UArg arg1, UArg arg2)
{
    Queue_Elem *elem;
    Clock_Object *obj;

    /* point to first clock object in the clockQueue */
    elem = Queue_next((Queue_Elem *)clockQueue);

    /* service all the Clock Objects in the clockQueue */
    while (elem != (Queue_Elem *)clockQueue) {
        obj = (Clock_Object *)elem;

        /* decrement the timeout counter */
        obj->timeout -= 1;

        /* if period has expired, refresh the timeout
         * value and invoke the clock func */
        if (obj->timeout == 0) {
            obj->timeout = obj->period;
            (obj->fxn)(obj->arg);
        }

        /* advance to next clock object in clockQueue */
        elem = Queue_next(elem);
    }
}

/* Task thread pends on Semaphore posted by Clock thread */
Void taskFxn(UArg arg1, UArg arg2)
{
    System_printf("In taskFxn pending on Sempahore.\n");
    Semaphore_pend(sem, BIOS_WAIT_FOREVER);
    System_printf("In taskFxn returned from Sempahore.\n");
    System_exit(0);
}

/* First Clock function, invoked every 5 timer interrupts */
Void clk1Fxn(UArg arg)
{
    System_printf("In clk1Fxn, arg = %d.\n", arg);
    clk1.arg++;
}

/* Second Clock function, invoked every 20 timer interrupts */
Void clk2Fxn(UArg sem)
{
    System_printf("In clk2Fxn, posting Semaphore.\n");
    Semaphore_post((Semaphore_Object *)sem);
}

/* main() */
Int main(Int argc, char* argv[])
{
    Timer_Params timerParams;
    Task_Params taskParams;
    Error_Block eb;
```

```
System_printf("Starting HwiSwiTask example.\n");

Error_init(&eb);
Timer_Params_init(&timerParams);
Task_Params_init(&taskParams);

/* Create a Swi with default priority (15).
 * Swi handler is 'swiFxn' which runs as a Swi thread. */
swi = Swi_create(swiFxn, NULL, &eb);
if (swi == NULL) {
    System_abort("Swi create failed");
}

/* Create a Task with priority 3.
 * Task function is 'taskFxn' which runs as a Task thread. */
taskParams.priority = 3;
task = Task_create(taskFxn, &taskParams, &eb);
if (task == NULL) {
    System_abort("Task create failed");
}

/* Create a binary Semaphore for example task to pend on */
sem = Semaphore_create(0, NULL, &eb);
if (sem == NULL) {
    System_abort("Semaphore create failed");
}

/* Create a Queue to hold the Clock Objects on */
clockQueue = Queue_create(NULL, &eb);
if (clockQueue == NULL) {
    System_abort("Queue create failed");
}

/* setup clk1 to go off every 5 timer interrupts. */
clk1.fxn = clk1Fxn;
clk1.period = 5;
clk1.timeout = 5;
clk1.arg = 1;
/* add the Clock object to the clockQueue */
Queue_put(clockQueue, &clk1.elem);

/* setup clk2 to go off every 20 timer interrupts. */
clk2.fxn = clk2Fxn;
clk2.period = 20;
clk2.timeout = 20;
clk2.arg = (UArg)sem;
/* add the Clock object to the clockQueue */
Queue_put(clockQueue, &clk2.elem);
```

```
    /* Configure a periodic interrupt using any available Timer
     * with a 1000 microsecond (1ms) interrupt period.
     *
     * The Timer interrupt will be handled by 'hwiFxn' which
     * will run as a Hwi thread.
     */
    timerParams.period = 1000;
    timer = Timer_create(Timer_ANY, hwiFxn, &timerParams, &eb);
    if (timer == NULL) {
        System_abort("Timer create failed");
    }

    BIOS_start();

    return(0);
}
```

The XDCtools configuration script is as follows:

```
/* ======== HwiSwiTaskExample.cfg ======== */

var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');
var Error = xdc.useModule('xdc.runtime.Error');
var Log = xdc.useModule('xdc.runtime.Log');
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
var Main = xdc.useModule('xdc.runtime.Main');
var Memory = xdc.useModule('xdc.runtime.Memory')
var SysMin = xdc.useModule('xdc.runtime.SysMin');
var System = xdc.useModule('xdc.runtime.System');
var Text = xdc.useModule('xdc.runtime.Text');


var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Clock = xdc.useModule('ti.sysbios.knl.Clock');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
var Queue = xdc.useModule('ti.sysbios.knl.Queue');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');


Program.argSize = 0x0;
System.maxAtexitHandlers = 4;
BIOS.heapSize = 0x2000;

/* System stack size (used by ISRs and Swis) */
Program.stack = 0x1000;

/* Circular buffer size for System_printf() */
SysMin.bufSize = 0x400;
```

```
/* Create and install logger for the whole system */
var loggerBufParams = new LoggerBuf.Params();
loggerBufParams.numEntries = 32;
var logger0 = LoggerBuf.create(loggerBufParams);
Defaults.common$.logger = logger0;
Main.common$.diags_INFO = Diags.ALWAYS_ON;


System.SupportProxy = SysMin;
BIOS.libType = BIOS.LibType_Custom;
```

The program output is as follows:

```
Starting HwiSwiTask example.
In taskFxn pending on Semaphore.
In clk1Fxn, arg = 1.
In clk1Fxn, arg = 2.
In clk1Fxn, arg = 3.
In clk1Fxn, arg = 4.
In clk2Fxn, posting Semaphore.
In taskFxn returned from Semaphore
```

Synchronization Modules

This chapter describes modules that can be used to synchronize access to shared resources.

4.1 Semaphores

SYS/BIOS provides a fundamental set of functions for inter-task synchronization and communication based upon *semaphores*. Semaphores are often used to coordinate access to a shared resource among a set of competing tasks. The Semaphore module provides functions that manipulate semaphore objects accessed through handles of type Semaphore_Handle. See the video introducing Semaphores for an overview.

Semaphore objects can be declared as either counting or binary semaphores. They can be used for task synchronization and mutual exclusion. The same APIs are used for both counting and binary semaphores.

Binary semaphores are either available or unavailable. Their value cannot be incremented beyond 1. Thus, they should be used for coordinating access to a shared resource by a maximum of two tasks. Binary semaphores provide better performance than counting semaphores.

Counting semaphores keep an internal count of the number of corresponding resources available. When count is greater than 0, tasks do not block when acquiring a semaphore. The maximum count value for a semaphores plus one is the number of tasks a counting semaphore can coordinate.

To configure the type of semaphore, use the following configuration parameter:

```
config Mode mode = Mode_COUNTING;
```

The functions Semaphore_create() and Semaphore_delete() are used to create and delete semaphore objects, respectively, as shown in Example 4-1. You can also create semaphore objects statically.

Example 4-1 Creating and Deleting a Semaphore

```
Semaphore_Handle Semaphore_create(
    Int                count,
    Semaphore_Params   *attrs
    Error_Block        *eb );

Void Semaphore_delete(Semaphore_Handle *sem);
```

The semaphore count is initialized to count when it is created. In general, count is set to the number of resources that the semaphore is synchronizing.

Semaphore_pend() waits for a semaphore. If the semaphore count is greater than 0, Semaphore_pend() simply decrements the count and returns. Otherwise, Semaphore_pend() waits for the semaphore to be posted by Semaphore_post().

The timeout parameter to Semaphore_pend(), as shown in Example 4-2, allows the task to wait until a timeout, to wait indefinitely (BIOS_WAIT_FOREVER), or to not wait at all (BIOS_NO_WAIT). Semaphore_pend()'s return value is used to indicate if the semaphore was acquired successfully.

Example 4-2 Setting a Timeout with Semaphore_pend()

```
Bool Semaphore_pend(
    Semaphore_Handle   sem,
    UInt               timeout);
```

Example 4-3 shows Semaphore_post(), which is used to signal a semaphore. If a task is waiting for the semaphore, Semaphore_post() removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, Semaphore_post() simply increments the semaphore count and returns.

Example 4-3 Signaling a Semaphore with Semaphore_post()

```
Void Semaphore_post(Semaphore_Handle sem);
```

4.1.1 Semaphore Example

Example 4-4 provides sample code for three writer tasks that create unique messages and place them on a list for one reader task. The writer tasks call Semaphore_post() to indicate that another message has been put on the list. The reader task calls Semaphore_pend() to wait for messages. Semaphore_pend() returns only when a message is available on the list. The reader task prints the message using the System_printf() function.

The three writer tasks, a reader task, a semaphore, and a queue in this example program were created statically as follows:

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');
var Error = xdc.useModule('xdc.runtime.Error');
var Log = xdc.useModule('xdc.runtime.Log');
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
var Main = xdc.useModule('xdc.runtime.Main');
var Memory = xdc.useModule('xdc.runtime.Memory')
var SysMin = xdc.useModule('xdc.runtime.SysMin');
var System = xdc.useModule('xdc.runtime.System');
var Text = xdc.useModule('xdc.runtime.Text');


var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Clock = xdc.useModule('ti.sysbios.knl.Clock');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');


/* set heap and stack sizes */
BIOS.heapSize = 0x2000;
Program.stack = 0x1000;
SysMin.bufSize = 0x400;


/* set library type */
BIOS.libType = BIOS.LibType_Custom;


/* Set logger for the whole system */
var loggerBufParams = new LoggerBuf.Params();
loggerBufParams.numEntries = 32;
var logger0 = LoggerBuf.create(loggerBufParams);
Defaults.common$.logger = logger0;
Main.common$.diags_INFO = Diags.ALWAYS_ON;
```

```
/* Use Semaphore, and Task modules and set global properties */
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
Program.global.sem = Semaphore.create(0);
var Task = xdc.useModule('ti.sysbios.knl.Task');
Task.idleTaskVitalTaskFlag = false;


/* Statically create reader and writer Tasks */
var reader = Task.create('&reader');
reader.priority = 5;

var writer0 = Task.create('&writer');
writer0.priority = 3;
writer0.arg0 = 0;

var writer1 = Task.create('&writer');
writer1.priority = 3;
writer1.arg0 = 1;

var writer2 = Task.create('&writer');
writer2.priority = 3;
writer2.arg0 = 2;


/* uses Queue module and create two instances statically */
var Queue = xdc.useModule('ti.sysbios.knl.Queue');
Program.global.msgQueue = Queue.create();
Program.global.freeQueue = Queue.create();
```

Since this program employs multiple tasks, a counting semaphore is used to synchronize access to the list. Figure 4-1 provides a view of the results from Example 4-3. Though the three writer tasks are scheduled first, the messages are read as soon as they have been put on the queue, because the reader's task priority is higher than that of the writer.

Example 4-4 Semaphore Example Using Three Writer Tasks

```
/* ======== semtest.c ======== */
#include <xdc/std.h>
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Error.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Queue.h>


#define NUMMSGS 3    /* number of messages */
#define NUMWRITERS 3 /* number of writer tasks created with */
```

```
/* Config Tool */
typedef struct MsgObj {
    Queue_Elem elem;      /* first field for Queue */
    Int id;               /* writer task id */
    Char val;             /* message value */
} MsgObj, *Msg;


Void reader();
Void writer();


/* The following objects are created statically. */
extern Semaphore_Handle sem;
extern Queue_Handle msgQueue;
extern Queue_Handle freeQueue;


/* ======== main ======== */
Int main(Int argc, Char* argv[])
{
    Int i;
    MsgObj *msg;
    Error_Block eb;


    Error_init(&eb);


    msg = (MsgObj *) Memory_alloc(NULL, NUMMSGS * sizeof(MsgObj), 0, &eb);
    if (msg == NULL) {
        System_abort("Memory allocation failed");
    }


    /* Put all messages on freeQueue */
    for (i = 0; i < NUMMSGS; msg++, i++) {
        Queue_put(freeQueue, (Queue_Elem *) msg);
    }
    BIOS_start();
    return(0);
}


/* ======== reader ======== */
Void reader()
{
    Msg msg;
    Int i;
    for (i = 0; i < NUMMSGS * NUMWRITERS; i++) {
        /* Wait for semaphore to be posted by writer(). */
        Semaphore_pend(sem, BIOS_WAIT_FOREVER);
```

```
        /* get message */
        msg = Queue_get(msgQueue);
        /* print value */
        System_printf("read '%c' from (%d).\n", msg->val, msg->id);
        /* free msg */
        Queue_put(freeQueue, (Queue_Elem *) msg);
    }
    System_printf("reader done.\n");
}


/* ======== writer ======== */
Void writer(Int id)
{
    Msg msg;
    Int i;

    for (i = 0; i < NUMMSGS; i++) {
        /* Get msg from the free list. Since reader is higher
         * priority and only blocks on sem, list is never
         * empty. */
        msg = Queue_get(freeQueue);

        /* fill in value */
        msg->id = id;
        msg->val = (i & 0xf) + 'a';
        System_printf("(%d) writing '%c' ...\n", id, msg->val);

        /* put message */
        Queue_put(msgQueue, (Queue_Elem *) msg);

        /* post semaphore */
        Semaphore_post(sem);
    }


    System_printf("writer (%d) done.\n", id);
}
```

Figure 4-1. Trace Window Results from Example 4-4

```
(0) writing 'a' ...
read 'a' from (0).
(0) writing 'b' ...
read 'b' from (0).
(0) writing 'c' ...
read 'c' from (0).
writer (0) done.
(1) writing 'a' ...
read 'a' from (1).
(1) writing 'b' ...
read 'b' from (1).
(1) writing 'c' ...
read 'c' from (1).
writer (1) done.
(2) writing 'a' ...
read 'a' from (2).
(2) writing 'b' ...
read 'b' from (2).
(2) writing 'c' ...
read 'c' from (2).
reader done.
writer (2) done.
```

4.2 Event Module

Events provide a means for communicating between and synchronizing threads. They are similar to Semaphores (see Section 4.1), except that they allow you to specify multiple conditions ("events") that must occur before the waiting thread returns.

An Event instance is used with calls to "pend" and "post", just as for a Semaphore. However, calls to Event_pend() additionally specify which events to wait for, and calls to Event_post() specify which events are being posted.

> **Note:**   Only a single Task can pend on an Event object at a time.

A single Event instance can manage up to 32 events, each represented by an event ID. Event IDs are simply bit masks that correspond to a unique event managed by the Event object.

Each Event behaves like a binary semaphore.

A call to Event_pend() takes an "andMask" and an "orMask". The andMask consists of the event IDs of all the events that must occur, and the orMask consists of the event IDs of any events of which only one must occur.

As with Semaphores, a call to Event_pend() takes a timeout value and returns 0 if the call times out. If a call to Event_pend() is successful, it returns a mask of the "consumed" events—that is, the events that occurred to satisfy the call to Event_pend(). The task is then responsible for handling ALL of the consumed events.

Only Tasks can call Event_pend(), whereas Hwis, Swis, and other Tasks can all call Event_post().

The Event_pend() prototype is as follows:

```
UInt Event_pend(Event_Handle event,
                UInt         andMask,
                UInt         orMask,
                UInt         timeout);
```

The Event_post() prototype is as follows:

```
Void Event_post(Event_Handle event,
                UInt         eventIds);
```

Configuration example: These XDCtools configuration statements create an event statically. The Event object has an Event_Handle named "myEvent".

```
var Event = xdc.useModule("ti.sysbios.knl.Event");
Program.global.myEvent = Event.create();
```

Runtime example: The following C code creates an Event object with an Event_Handle named "myEvent".

```
Event_Handle myEvent;
Error_Block eb;

Error_init(&eb);

/* Default instance configuration params */
myEvent = Event_create(NULL, &eb);
if (myEvent == NULL) {
    System_abort("Event create failed");
}
```

Runtime example: The following C code blocks on an event. It wakes the task only when both events 0 and 6 have occurred. It sets the andMask to enable both Event_Id_00 and Event_Id_06. It sets the orMask to Event_Id_NONE.

```
Event_pend(myEvent, (Event_Id_00 + Event_Id_06), Event_Id_NONE,
           BIOS_WAIT_FOREVER);
```

Runtime example: The following C code has a call to Event_post() to signal which events have occurred. The eventMask should contain the IDs of the events that are being posted.

```
Event_post(myEvent, Event_Id_00);
```

Runtime Example: The following C code example shows a task that provides the background processing required for three Interrupt Service Routines:
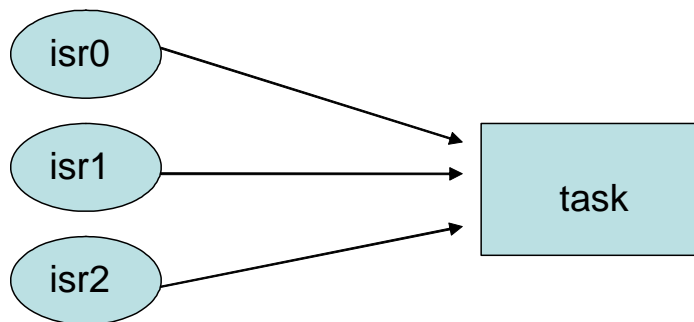

```
Event_Handle myEvent;


main()
{
    ...


    /* create an Event object. All events are binary */
    myEvent = Event_create(NULL, &eb);
    if (myEvent == NULL) {
        System_abort("Event create failed");
    }
}


isr0()
{
    ...
    Event_post(myEvent, Event_Id_00);
    ...
}


isr1()
{
    ...
    Event_post(myEvent, Event_Id_01);
    ...
}


isr2()
{
    ...
    Event_post(myEvent, Event_Id_02);
    ...
}
```

```
task()
{
    UInt events;

    while (TRUE) {
        /* Wait for ANY of the ISR events to be posted *
        events = Event_pend(myEvent, Event_Id_NONE,
            Event_Id_00 + Event_Id_01 + Event_Id_02,
            BIOS_WAIT_FOREVER);

        /* Process all the events that have occurred */
        if (events & Event_Id_00) {
            processISR0();
        }
        if (events & Event_Id_01) {
            processISR1();
        }
        if (events & Event_Id_02) {
            processISR2();
        }
    }
}
```

4.2.1 *Implicitly Posted Events*

In addition to supporting the explicit posting of events through the Event_post() API, some SYS/BIOS objects support implicit posting of events associated with their objects. For example, a Mailbox can be configured to post an associated event whenever a message is available (that is, whenever Mailbox_post() is called) thus allowing a task to block while waiting for a Mailbox message and/or some other event to occur.

Mailbox and Semaphore objects currently support the posting of events associated with their resources becoming available.

SYS/BIOS objects that support implicit event posting must be configured with an event object and event ID when created. You can decide which event ID to associate with the specific resource availability signal (that is, a message available in Mailbox, room available in Mailbox, or Semaphore available).

Note:	As mentioned earlier, only one Task can pend on an Event object at a time. Consequently, SYS/BIOS objects that are configured for implicit event posting should only be waited on by a single Task at a time.

When Event_pend() is used to acquire a resource from implicitly posting objects, the BIOS_NO_WAIT timeout parameter should be used to subsequently retrieve the resource from the object.

Runtime example: The following C code example shows a task processing the messages posted to a Mailbox message as well as performing an ISR's post-processing requirements.
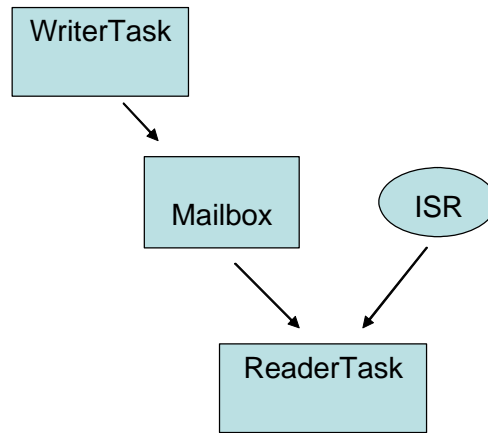

```
Event_Handle myEvent;
Mailbox_Handle mbox;

typedef struct msg {
   UInt id;
   Char buf[10];
}


main()
{
   Mailbox_Params mboxParams;
   Error_Block eb;

   Error_init(&eb);

   myEvent = Event_create(NULL, &eb);
   if (myEvent == NULL) {
      System_abort("Event create failed");
   }
   Mailbox_Params_init(&mboxParams);
   mboxParams.notEmptyEvent = myEvent;

   /* Assign Event_Id_00 to Mailbox "not empty" event */
   mboxParams.notEmptyEventId = Event_Id_00;
   mbox = Mailbox_create(sizeof(msg), 50, &mboxParams, &eb);
   if (mbox == NULL) {
      System_abort("Mailbox create failed");
   }

   /* Mailbox_create() sets Mailbox's notEmptyEvent to
    * counting mode and initial count = 50 */
}
```

```
writerTask()
{
   ...
   Mailbox_post(mbox, &msgA, BIOS_WAIT_FOREVER);
   /* implicitly posts Event_Id_00 to myEvent */
   ...
}


isr()
{
   Event_post(myEvent, Event_Id_01);
}


readerTask()
{
   while (TRUE) {/* Wait for either ISR or Mailbox message */
      events = Event_pend(myEvent,
                  Event_Id_NONE,            /* andMask = 0 */
                  Event_Id_00 + Event_Id_01, /* orMask */
                  BIOS_WAIT_FOREVER);        /* timeout */
      if (events & Event_Id_00) {
         /* Get the posted message.
          * Mailbox_pend() will not block since Event_pend()
          * has guaranteed that a message is available.
          * Notice that the special BIOS_NO_WAIT
          * parameter tells Mailbox that Event_pend()
          * was used to acquire the available message.
          */
         Mailbox_pend(mbox, &msgB, BIOS_NO_WAIT);
         processMsg(&msgB);
      }
      if (events & Event_Id_01) {
         processISR();
      }
   }
}
```

4.3 Gates

A "Gate" is a module that implements the IGateProvider interface. Gates are devices for preventing concurrent accesses to critical regions of code. The various Gate implementations differ in how they attempt to lock the critical regions.

Since xdc.runtime.Gate is provided by XDCtools, the base module is documented in the online help. Implementations of Gates provided by SYS/BIOS are discussed here.

Threads can be preempted by other threads of higher priority, and some sections of code need to be completed by one thread before they can be executed by another thread. Code that modifies a global variable is a common example of a critical region that may need to be protected by a Gate.

Gates generally work by either disabling some level of preemption such as disabling task switching or even hardware interrupts, or by using a binary semaphore.

All Gate implementations support nesting through the use of a "key". For Gates that function by disabling preemption, it is possible that multiple threads would call Gate_enter(), but preemption should not be restored until all of the threads have called Gate_leave(). This functionality is provided through the use of a key. A call to Gate_enter() returns a key that must then be passed back to Gate_leave(). Only the outermost call to Gate_enter() returns the correct key for restoring preemption. (The actual module name for the implementation is used instead of "Gate" in the function name.)

Runtime example: The following C code protects a critical region with a Gate. This example uses a GateHwi, which disables and enables interrupts as the locking mechanism.

```
UInt gateKey;
GateHwi_Handle gateHwi;
GateHwi_Params prms;
Error_Block eb;

Error_init(&eb);
GateHwi_Params_init(&prms);


gateHwi = GateHwi_create(&prms, &eb);
if (gateHwi == NULL) {
    System_abort("Gate create failed");
}


/* Simultaneous operations on a global variable by multiple
 * threads could cause problems, so modifications to the global
 * variable are protected with a Gate. */
gateKey = GateHwi_enter(gateHwi);
myGlobalVar = 7;
GateHwi_leave(gateHwi, gateKey);
```

4.3.1 *Preemption-Based Gate Implementations*

The following implementations of gates use some form of preemption disabling:

- ti.sysbios.gates.GateHwi
- ti.sysbios.gates.GateSwi
- ti.sysbios.gates.GateTask

4.3.1.1 *GateHwi*

GateHwi disables and enables interrupts as the locking mechanism. Such a gate guarantees exclusive access to the CPU. This gate can be used when the critical region is shared by Task, Swi, or Hwi threads.

The duration between the enter and leave should be as short as possible to minimize Hwi latency.

4.3.1.2 *GateSwi*

GateSwi disables and enables software interrupts as the locking mechanism. This gate can be used when the critical region is shared by Swi or Task threads. This gate cannot be used by a Hwi thread.

The duration between the enter and leave should be as short as possible to minimize Swi latency.

4.3.1.3 *GateTask*

GateTask disables and enables tasks as the locking mechanism. This gate can be used when the critical region is shared by Task threads. This gate cannot be used by a Hwi or Swi thread.

The duration between the enter and leave should be as short as possible to minimize Task latency.

4.3.2 *Semaphore-Based Gate Implementations*

The following implementations of gates use a semaphore:

- ti.sysbios.gates.GateMutex
- ti.sysbios.gates.GateMutexPri

4.3.2.1 *GateMutex*

GateMutex uses a binary Semaphore as the locking mechanism. Each GateMutex instance has its own unique Semaphore. Because this gate can potentially block, it should not be used a Swi or Hwi thread, and should only be used by Task threads.

4.3.2.2 *GateMutexPri*

GateMutexPri is a mutex Gate (it can only be held by one thread at a time) that implements "priority inheritance" in order to prevent priority inversion. Priority inversion occurs when a high-priority Task has its priority effectively "inverted" because it is waiting on a Gate held by a lower-priority Task. Issues and solutions for priority inversion are described in Section 4.3.3.

Configuration example: The following example specifies a GateType to be used by HeapMem. (See section 6.8.1, *HeapMem* for further discussion.)

```
var GateMutexPri = xdc.useModule('ti.sysbios.gates.GateMutexPri');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

HeapMem.common$.gate = GateMutexPri.create();
```

4.3.3   Priority Inversion

The following example shows the problem of priority inversion. A system has three tasks—Low, Med, and High—each with the priority suggested by its name. Task Low runs first and acquires the gate. Task High is scheduled and preempts Low. Task High tries to acquire the gate, and waits on it. Next, task Med is scheduled and preempts task Low. Now task High must wait for both task Med and task Low to finish before it can continue. In this situation, task Low has, in effect, lowered task High's priority to that of Low.

Solution: Priority Inheritance
To guard against priority inversion, GateMutexPri implements priority inheritance. When task High tries to acquire a gate that is owned by task Low, task Low's priority is temporarily raised to that of High, as long as High is waiting on the gate. So, task High "donates" its priority to task Low.

When multiple tasks wait on the gate, the gate owner receives the highest priority of any of the tasks waiting on the gate.

Caveats
Priority inheritance is not a complete guard against priority inversion. Tasks only donate their priority on the call to enter a gate, so if a task has its priority raised while waiting on a gate, that priority is not carried through to the gate owner.

This can occur in situations involving multiple gates. For example, a system has four tasks: VeryLow, Low, Med, and High, each with the priority suggested by its name. Task VeryLow runs first and acquires gate A. Task Low runs next and acquires gate B, then waits on gate A. Task High runs and waits on gate B. Task High has donated its priority to task Low, but Low is blocked on VeryLow, so priority inversion occurs despite the use of the gate. The solution to this problem is to design around it. If gate A may be needed by a high-priority, time-critical task, then it should be a design rule that no task holds this gate for a long time or blocks while holding this gate.

When multiple tasks wait on this gate, they receive the gate in order of priority (higher-priority tasks receive the gate first). This is because the list of tasks waiting on a GateMutexPri is sorted by priority, not FIFO.

Calls to GateMutexPri_enter() may block, so this gate can only be used in the task context.

GateMutexPri has non-deterministic calls because it keeps the list of waiting tasks sorted by priority.

4.4   Mailboxes

The ti.sysbios.knl.Mailbox module provides a set of functions to manage mailboxes. Mailboxes can be used to pass buffers from one task to another on the same processor.

A Mailbox instance can be used by multiple readers and writers.

The Mailbox module copies the buffer to fixed-size internal buffers. The size and number of these buffers are specified when a Mailbox instance is created (or constructed). A copy is done when a buffer is sent via Mailbox_post(). Another copy occurs when the buffer is retrieved via a Mailbox_pend().

Mailbox_create() and Mailbox_delete() are used to create and delete mailboxes, respectively. You can also create mailbox objects statically.

Mailboxes can be used to ensure that the flow of incoming buffers does not exceed the ability of the system to process those buffers. The examples given later in this section illustrate just such a scheme.

You specify the number of internal mailbox buffers and size of each of these buffers when you create a mailbox. Since the size is specified when you create the Mailbox, all buffers sent and received with the Mailbox instance must be of this same size.

```
Mailbox_Handle Mailbox_create(SizeT          bufsize,
                              UInt           numBufs,
                              Mailbox_Params *params,
                              Error_Block    *eb)

Void Mailbox_delete(Mailbox_Handle *handle);
```

Mailbox_pend() is used to read a buffer from a mailbox. If no buffer is available (that is, the mailbox is empty), Mailbox_pend() blocks. The timeout parameter allows the task to wait until a timeout, to wait indefinitely (BIOS_WAIT_FOREVER), or to not wait at all (BIOS_NO_WAIT). The unit of time is system clock ticks.

```
Bool Mailbox_pend(Mailbox_Handle handle,
                  Ptr            buf,
                  UInt           timeout);
```

Mailbox_post() is used to post a buffer to the mailbox. If no buffer slots are available (that is, the mailbox is full), Mailbox_post() blocks. The timeout parameter allows the task to wait until a timeout, to wait indefinitely (BIOS_WAIT_FOREVER), or to not wait at all (BIOS_NO_WAIT).

```
Bool Mailbox_post(Mailbox_Handle handle,
                  Ptr            buf,
                  UInt           timeout);
```

Mailbox provides configuration parameters to allow you to associate events with a mailbox. This allows you to wait on a mailbox message and another event at the same time. Mailbox provides two configuration parameters to support events for the reader(s) of the mailbox—notEmptyEvent and notEmptyEventId. These allow a mailbox reader to use an event object to wait for the mailbox message. Mailbox also provides two configuration parameters for the mailbox writer(s)—notFullEvent and notFullEventId. These allow mailbox writers to use an event object to wait for room in the mailbox.

When using events, a thread calls Event_pend() and waits on several events. Upon returning from Event_pend(), the thread must call Mailbox_pend() or Mailbox_post()—depending on whether it is a reader or a writer—with a timeout value of BIOS_NO_WAIT. See Section 4.2.1 for a code example that obtains the corresponding Mailbox resource after returning from Event_pend().

4.5 Queues

The ti.sysbios.misc.Queue module provides support for creating lists of objects. A Queue is implemented as a doubly-linked list, so that elements can be inserted or removed from anywhere in the list, and so that Queues do not have a maximum size.

4.5.1 Basic FIFO Operation of a Queue

To add a structure to a Queue, its first field needs to be of type Queue_Elem. The following example shows a structure that can be added to a Queue.

A Queue has a "head", which is the front of the list. Queue_enqueue() adds elements to the back of the list, and Queue_dequeue() removes and returns the element at the head of the list. Together, these functions support a natural FIFO queue.

Run-time example: The following example demonstrates the basic Queue operations— Queue_enqueue() and Queue_dequeue(). It also uses the Queue_empty() function, which returns true when there are no more elements in the Queue.

```
/* This structure can be added to a Queue because the first field is a Queue_Elem. */
typedef struct Rec {
    Queue_Elem elem;
    Int data;
} Rec;


Queue_Handle myQ;
Rec r1, r2;
Rec* rp;


r1.data = 100;
r2.data = 200;


// No parameters or Error block are needed to create a Queue.
myQ = Queue_create(NULL, NULL);


// Add r1 and r2 to the back of myQ.
Queue_enqueue(myQ, &(r1.elem));
Queue_enqueue(myQ, &(r2.elem));


// Dequeue the records and print their data
while (!Queue_empty(myQ)) {
    // Implicit cast from (Queue_Elem *) to (Rec *)
    rp = Queue_dequeue(myQ);
    System_printf("rec: %d\n", rp->data);
}
```

The example prints:

```
rec: 100
rec: 200
```

4.5.2   Iterating Over a Queue

The Queue module also provides several APIs for looping over a Queue. Queue_head() returns the element at the front of the Queue (without removing it) and Queue_next() and Queue_prev() return the next and previous elements in a Queue, respectively.

Run-time example: The following example demonstrates one way to iterate over a Queue once from beginning to end. In this example, "myQ" is a Queue_Handle.

```
Queue_Elem *elem;

for (elem = Queue_head(myQ); elem != (Queue_Elem *)myQ;
     elem = Queue_next(elem)) {
     ...
}
```

4.5.3   Inserting and Removing Queue Elements

Elements can also be inserted or removed from anywhere in the middle of a Queue using Queue_insert() and Queue_remove(). Queue_insert() inserts an element in front of the specified element, and Queue_remove() removes the specified element from whatever Queue it is in. Note that Queue does not provide any APIs for inserting or removing elements at a given index in the Queue.

Run-time example: The following example demonstrates Queue_insert() and Queue_remove().

```
Queue_enqueue(myQ, &(r1.elem));

/* Insert r2 in front of r1 in the Queue. */
Queue_insert(&(r1.elem), &(r2.elem));

/* Remove r1 from the Queue. Note that Queue_remove() does not
 * require a handle to myQ. */
Queue_remove(&(r1.elem));
```

4.5.4   Atomic Queue Operations

Queues are commonly shared across multiple threads in the system, which might lead to concurrent modifications of the Queue by different threads, which would corrupt the Queue. The Queue APIs discussed above do not protect against this. However, Queue provides two "atomic" APIs, which disable interrupts before operating on the Queue. These APIs are Queue_get(), which is the atomic version of Queue_dequeue(), and Queue_put(), which is the atomic version of Queue_enqueue().

Timing Services

This chapter describes modules that can be used for timing purposes.

Topic **Page**

5.1 Overview of Timing Services

Several modules are involved in timekeeping and clock-related services within SYS/BIOS and XDCtools:

- **The ti.sysbios.knl.Clock module** is responsible for the periodic system tick that the kernel uses to keep track of time. All SYS/BIOS APIs that expect a timeout parameter interpret the timeout in terms of Clock ticks. The Clock module is used to schedule functions that run at intervals specified in clock ticks. By default, the Clock module uses the hal.Timer module to get a hardware-based tick. Alternately, the Clock module can be configured to use an application-provided tick source. See Section 5.2 for details. (The Clock module replaces both the CLK and PRD modules in earlier versions of DSP/BIOS.)

- **The ti.sysbios.hal.Timer module** provides a standard interface for using timer peripherals. It hides any target/device-specific characteristics of the timer peripherals. Target/device-specific properties for timers are supported by the ti.sysbios.family.xxx.Timer modules (for example, ti.sysbios.family.c64.Timer). You can use the Timer module to select a timer that calls a tickFxn when the timer expires. See Section 5.3 and Section 7.3 for details.

- **The xdc.runtime.Timestamp module** provides simple timestamping services for benchmarking code and adding timestamps to logs. This module uses a target/device-specific TimestampProvider in SYS/BIOS to control how timestamping is implemented. See Section 5.4 for details.

See the video introducing Timers and Clocks for an overview.

5.2 Clock

The ti.sysbios.knl.Clock module is responsible for the periodic system tick that the kernel uses to keep track of time. All SYS/BIOS APIs that expect a timeout parameter interpret the timeout in terms of Clock ticks.

The Clock module, by default, uses the ti.sysbios.hal.Timer module to create a timer to generate the system tick, which is basically a periodic call to Clock_tick(). See Section 5.3 for more about the Timer module.

The Clock module can be configured not to use the timer with either of the following configuration statements:

```
ti.sysbios.knl.Clock.tickSource = Clock.tickSource_USER

   or

ti.sysbios.knl.Clock.tickSource = Clock.tickSource_NULL
```

The period for the system tick is set by the configuration parameter Clock.tickPeriod. This is set in microseconds.

The Clock_tick() and the tick period are used as follows:

- **If the tickSource is Clock.tickSource_TIMER** (the default), Clock uses ti.sysbios.hal.Timer to create a timer to generate the system tick, which is basically a periodic call to Clock_tick(). Clock uses Clock.tickPeriod to create the timer. Clock.timerId can be changed to make Clock use a different timer.

- **If the tickSource is Clock.tickSource_USER**, then your application must call Clock_tick() from a user interrupt and set the tickPeriod to the approximate frequency of the user interrupt in microseconds.

- **If the tickSource is Clock.tickSource_NULL**, you cannot call any SYS/BIOS APIs with a timeout value and cannot call any Clock APIs. You can still use the Task module but you cannot call APIs that require a timeout, for example, Task_sleep(). Clock.tickPeriod values is not valid in this configuration.

Clock_getTicks() gets the number of Clock ticks that have occurred since startup. The value returned wraps back to zero after it reaches the maximum value that can be stored in 32 bits.

The Clock module provides APIs to start, stop and reconfigure the tick. These APIs allow you to make frequency changes at runtime. These three APIs are not reentrant and gates need to be used to protect them.

- **Clock_tickStop()** stops the timer used to generate the Clock tick by calling Timer_stop().

- **Clock_tickReconfig()** calls Timer_setPeriodMicroseconds() internally to reconfigure the timer. Clock_tickReconfig() fails if the timer cannot support Clock.tickPeriod at the current CPU frequency.

- **Clock_tickStart()** restarts the timer used to generate the Clock tick by calling Timer_start().

The Clock module lets you create Clock object instances, which reference functions that run when a timeout value specified in Clock ticks expires.

All Clock functions run in the context of a Swi. That is, the Clock module automatically creates a Swi for its use and run the Clock functions within that Swi. The priority of the Swi used by Clock can be changed by configuring Clock.swiPriority.

You can dynamically create clock instances using Clock_create(). Clock instances can be "one-shot" or continuous. You can start a clock instance when it is created or start it later by calling Clock_start(). This is controlled by the startFlag configuration parameter. Clock_create() can be called only from the context of a Task or the main() function.

A function and a non-zero timeout value are required arguments to Clock_create(). The function is called when the timeout expires. The timeout value is used to compute the first expiration time. For one-shot Clock instances, the timeout value used to compute the single expiration time, and the period is zero. For periodic Clock instances, the timeout value is used to compute the first expiration time; the period value (part of the params) is used after the first expiration.

Table 5–1. Timeline for One-shot and Continuous Clocks
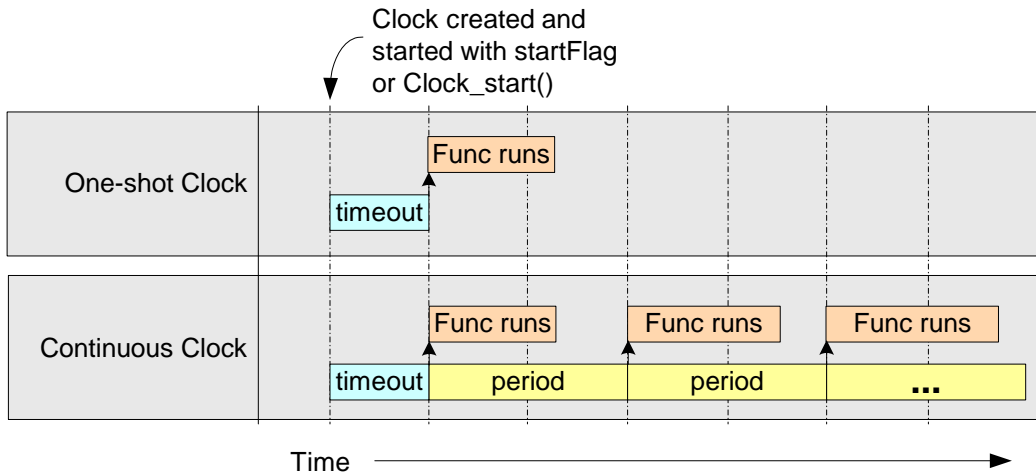
Clock instances (both one-shot and periodic) can be stopped and restarted by calling Clock_start() and Clock_stop(). Notice that while Clock_tickStop() stops the timer used to generate the Clock tick, Clock_stop() stops only one instance of a clock object. The expiration value is recomputed when you call Clock_start(). APIs that start or stop a Clock Instance—Clock_start() and Clock_stop()—can be called in any context except program startup before main() begins.

The Clock module provides the Clock_setPeriod(), Clock_setTimeout(), and Clock_setFunc() APIs to modify Clock instance properties for Clock instances that have been stopped.

Runtime example: This C example shows how to create a Clock instance. This instance is dynamic (runs repeatedly) and starts automatically. It runs the myHandler function every 5 ticks. A user argument (UArg) is passed to the function.

```
Clock_Params clockParams;
Clock_Handle myClock;
Error_Block eb;

Error_init(&eb);
Clock_Params_init(&clockParams);
clockParams.period = 5;
clockParams.startFlag = TRUE;
clockParams.arg = (UArg)0x5555;
myClock = Clock_create(myHandler1, 5, &clockParams, &eb);
if (myClock == NULL) {
    System_abort("Clock create failed");
}
```

Configuration example: This example uses XDCtools to create a Clock instance with the same properties as the previous example.

```
var Clock = xdc.useModule('ti.sysbios.knl.Clock');

var clockParams = new Clock.Params();
clockParams.period = 5;
clockParams.startFlag = true;
clockParams.arg = (UArg)0x5555;
Program.global.clockInst1 = Clock.create("&myHandler1", 5, clockParams);
```

Runtime example: This C example uses some of the Clock APIs to print messages about how long a Task sleeps.

```
UInt32 time1, time2;
. . .

System_printf("task going to sleep for 10 ticks... \n");
time1 = Clock_getTicks();
Task_sleep(10);

time2 = Clock_getTicks();
System_printf("...awake! Delta time is: %lu\n", (ULong) (time2 - time1));
```

Runtime example: This C example uses some of the Clock APIs to lower the Clock module frequency.

```
BIOS_getCpuFreq(&cpuFreq);
cpuFreq.lo = cpuFreq.lo / 2;
BIOS_setCpuFreq(&cpuFreq);

key = Hwi_disable();
Clock_tickStop();
Clock_tickReconfig();
Clock_tickStart();
Hwi_restore(key);
```

5.3   Timer Module

The ti.sysbios.hal.Timer module presents a standard interface for using the timer peripherals. This module is described in detail in Section 7.3 because it is part of the Hardware Abstraction Layer (HAL) package

You can use this module to create a timer (that is, to mark a timer for use) and configure it to call a tickFxn when the timer expires. Use this module only if you do not need to do any custom configuration of the timer peripheral.

The timer can be configured as a one-shot or a continuous mode timer. The period can be specified in timer counts or microseconds.

5.4   Timestamp Module

The xdc.runtime.Timestamp module, as the name suggests, provides timestamping services. The Timestamp module can be used for benchmarking code and adding timestamps to logs. (In previous versions of DSP/BIOS, this is the functionality provided by CLK_gethtime().)

Since xdc.runtime.Timestamp is provided by XDCtools, it is documented in the XDCtools online help.

This chapter describes issues related to memory use in SYS/BIOS.

6.1   Background

This chapter deals with the configuration of static memory (that is, memory mapping and section placement), caches, and stacks. It also provides information on dynamic memory allocation (allocating and freeing memory at runtime).

Static memory configuration relates to the "memory map" available to the executable and the placement of code and data into the memory map. The memory map is made up of internal memory regions that exist within the CPU and external memory regions located on the hardware board. See Section 6.2 for details about the memory map.

Code and data are placed in memory regions by the linker using the linker command file. The linker command file specifies a memory map. For each memory region, the linker command file specifies the origin or base address, length and attributes (read, write, and execute). A memory region specified in the linker command file is also called a "memory segment".

The following is a memory map specification from a linker command file:

```
MEMORY {
    IRAM (RWX) : org = 0x800000, len = 0x200000
    DDR : org = 0x80000000, len = 0x10000000
}
```

The linker command file also contains information on "memory section" placement, as shown in the following example. Sections are relocatable blocks of code produced by the compiler. The compiler produces some well-known sections for placements of various types of code and data, for example: .text, .switch, .bss, .far , .cinit, and .const. For details, see the appropriate compiler user's guide.

```
SECTIONS {
    .text: load >> DDR
    .switch: load >> DDR
    .stack: load > DDR
    .vecs: load >> DDR
    .args: load > DDR
    .sysmem: load > DDR
    .far: load >> DDR
    .data: load >> DDR
    .cinit: load > DDR
    .bss: load > DDR
    .const: load > DDR
    .pinit: load > DDR
    .cio: load >> DDR
}
```

The linker places "memory sections" (such as .text and .cinit) into "memory segments" (such as IRAM) as specified by SECTIONS portion of the linker command file. See Section 6.3 for details about section placement.

Section 6.2 discusses the memory map for SYS/BIOS applications. (MSP430 users should see Section 6.4 instead.)

Section 6.3 discusses section placement in a SYS/BIOS application. (MSP430 users should see Section 6.4 instead.)

Section 6.5 discusses stacks, including how to configure the system stack and task stacks.

Section 6.6 covers cache configuration specific to the C6000 and cache runtime APIs.

Section 6.7 also discusses dynamic memory allocation. Runtime code can allocate and free memory from a "heap," which is a memory pool that has been set aside and managed for the purpose of dynamic memory allocation.

Various heap implementations are described in Section 6.8.

6.2 Memory Map

> **Note:** If you are using the MSP430, see Section 6.4 instead. This section does not apply to the MSP430.

The memory map for an executable is determined by the device (which has internal memory) and the hardware board (which has external memory).

When you create a CCS project for an application that uses XDCtools and SYS/BIOS, you select a "platform" on the RTSC Configuration Settings page. The memory map for on-device and external memory is determined by this platform. The platform also sets the clock speed and specifies memory section placement.

You select a platform when you create a new project or change the project build properties, not when you create a configuration file.

Executables that need different memory maps must use different platforms even if they run on the same type of board.

The platform is tied to a particular device (CPU) and gets the internal memory map from the device—for example, IRAM and FLASH. The platform also contains the external memory specifications and cache settings. The internal and external memory segments together form the memory map.

6.2.1 Choosing an Available Platform

Before building a SYS/BIOS 6.x executable, you need to select the hardware board you will be using. You do this by selecting a platform either when you create a CCS project or in the **RTSC** tab of the project's **CCS General** properties. The **Platform** field provides a drop-down list of all available platforms that match your **Target**; these items represent various evaluation boards available for your chosen device (CPU).

Target:	ti.targets.arm.elf.M3
Platform:	ti.platforms.concertoM3:F28M35H32B1
Build-profile:	release

To view the memory map for your platform, you can open the platform wizard by choosing **Tools > RTSC Tools > Platform > Edit/View**. Select the packages repository in your XDCtools installation. For example, C:\Program Files\Texas Instruments\xdctools_3_20\packages. Then, choose the Package you are using and click **Next**.
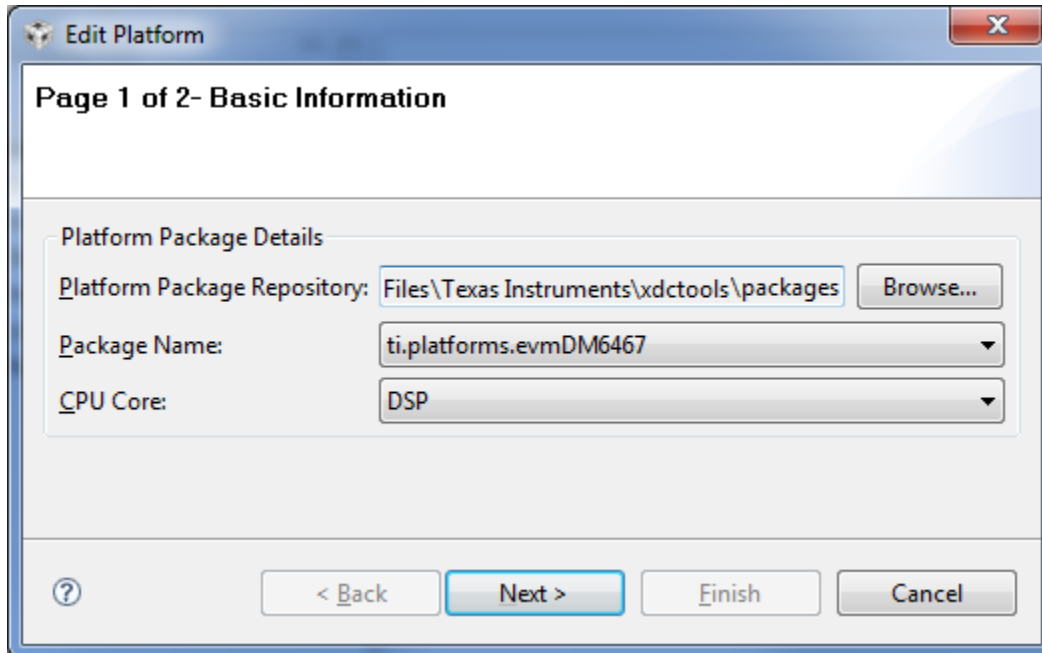
In most cases, you begin application development using one of the evaluation boards, and can select one of the standard platforms from the drop-down list. You should select one of existing platforms if *all* of the following are true:

- You are in the development phase and are using an evaluation board.

- You do not care about cache sizes and are satisfied with the defaults set by the existing platform.

- You do not want to change the default section placement.

- You want the same clock rate as the evaluation board.

If any of these statements do not apply, see Section 6.2.2.

6.2.2   *Creating a Custom Platform*

At some point in the application development process, most customers build their own boards, choosing a TI device and adding custom external memory.
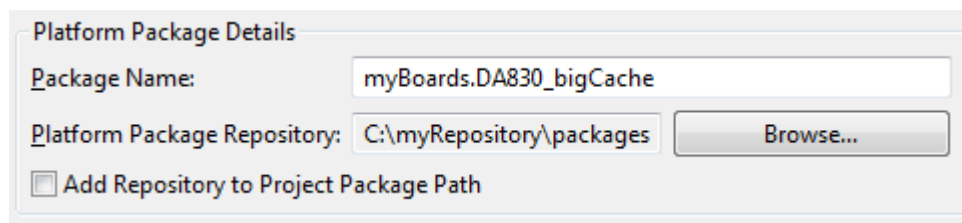
You will also need to create your own platform if any of the following items are true:

- You want to customize cache sizes.

- You want to manually override the default section placement.

For such custom boards you will need to create a platform using the platform wizard. The platform wizard is a GUI tool that allows you to easily create a custom platform. Creating a custom platform gives you a lot of flexibility in terms of defining the memory map and selecting default memory segments for section placement.
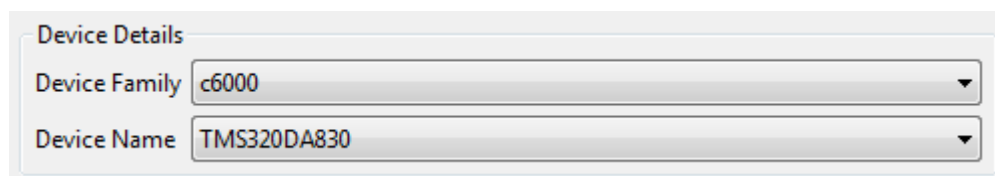
To run the platform wizard, follow these steps:

1. In CCS, choose **Tools > RTSC Tools > Platform > New** from the menus. This opens the New Platform wizard.

2. Type a name for the package. This will be the name of the directory created to contain the platform package, and will be the name you select when you choose the platform for the project to use.

   You can use a simple name or a period-delimited name. Periods correspond to directory levels when a platform package is created. For example, myBoards.DA830_bigCache will be created in C:\myRepository\packages\myBoards\DA830_bigCache if you are using C:\myRepository\packages as the repository location.

3. Next to the Platform Package Repository field, click **Browse**. Choose the location of the repository where you want to save your platform package. The default is C:\Users\<*username*>\myRepository\packages.

Platform Package Details		
Package Name:	myBoards.DA830_bigCache	
Platform Package Repository:	C:\myRepository\packages	Browse...
☐ Add Repository to Project Package Path		

   If you haven't created a package repository before, and you don't want to use the default, create a new directory to contain the repository. In the directory you choose, create a sub-directory called "packages". For example, you might use C:\myRepository\packages as the repository. The full path to the repository should not contain any spaces.

4. Optionally, if you have already created a CCS project that you want to be able to use this platform, check the **Add Repository to Project Package Path** box. Then select the project that should have access to this repository. You don't need to do this now; you can also add repositories to projects from the project's Build Properties dialog.

5. Choose the **Device Family** and **Device Name** from the drop-down lists. For example:

Device Details	
Device Family	c6000
Device Name	TMS320DA830

6. Click **Next**. You see the Device Page of the platform wizard.

Note: If you want another project to be able to use this platform, you can later add the repository that contains the platform to a project's properties by right-clicking on the project and choosing **Build Properties**. Choose the **CCS General** category and then the **RTSC** tab. Click **Add** and browse the file-system for the repository you want the project to be able to search for platforms.

See the subsections that follow for ways to specify the cache, segment, and section use for your platform. You can also visit the Demo of the RTSC Platform Wizard to watch demonstrations that use the platform wizard.

6.2.2.1 Getting and Setting the Clock Speed and Default Memory Settings

The Device Page opens with no clock speed setting, no external memory segments, and no memory section assignments. Generally, the first thing you will want to do is to import the default settings from an existing platform so that you can use those as a base for making the modifications you need.

To import defaults, follow these steps:

1. Click the **Import** button next to the Clock Speed field.

2. In the Select Platform dialog, choose the platform whose defaults you want to import, and click **OK**.

3. Click **Yes** in the confirmation dialog that asks if you are sure you want to change the settings.

4. You see the default clock speed and external memory settings. You can change these if you like.
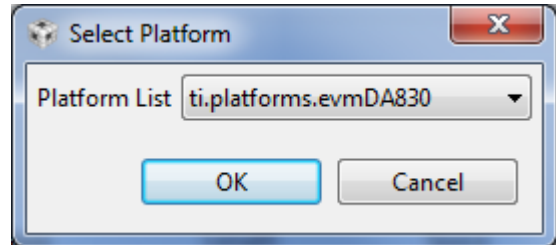
Select Platform

Platform List ti.platforms.evmDA830

OK   Cancel

Device Details

Device Name	TMS320DA830
Device Family	c6000
Clock Speed (MHz)	300.0

Import...

Device Memory

Name	Base	Length	Space	Access
IRAM	0x11800000	0x00040000	code/data	RWX
L3_CBA_RAM	0x80000000	0x00020000	code/data	RWX
IROM	0x11700000	0x00100000	code/data	RX
L1PSRAM	0x11e00000	0x00008000	code	RWX

L1D Cache: 0k   L1P Cache: 0k   L2 Cache: 0k

☐ Customize Memory

External Memory

Name	Base	Length	Space	Access
SDRAM	0xC3000000	0x01000000	code/data	RWX

Memory Sections

Code Memory: SDRAM   Data Memory: SDRAM   Stack Memory: SDRAM
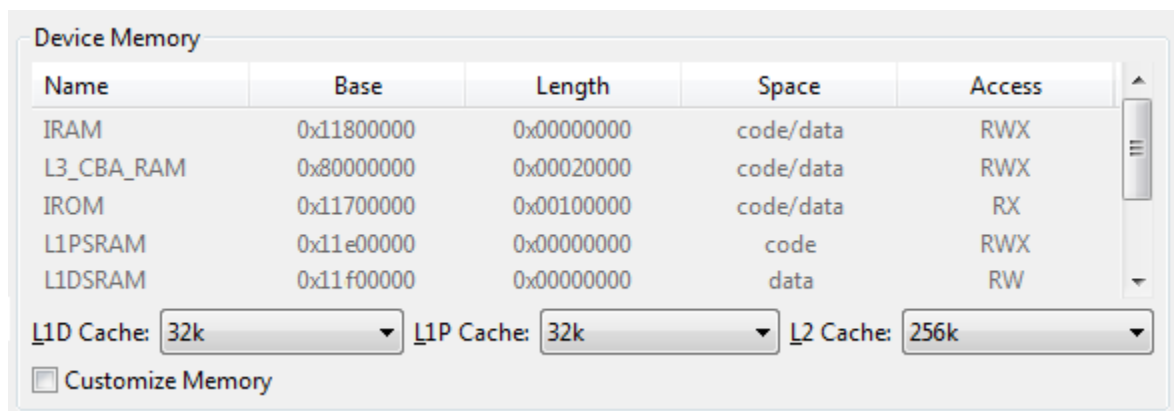
6.2.2.2 Determining Cache Sizes for Custom Platforms

Since cache sizes affect the memory map, if you are using a C6000 target, you need to decide on the sizes you want to use when creating a platform. For example, if you are using the "ti.platforms.evmDA830" platform, the L1P, L1D, and L2 cache sizes affect the size of available L1PSRAM, L1DSRAM, and IRAM.

Since cache sizes are set in the platform, executables that need different cache configurations will also need different platforms.

The following example steps use the Device Page of the platform wizard to specify the maximum cache sizes for TMS320DA830 platform:

1. Set the L1D Cache to 32K. Set the L1P Cache to 32K. Set the L2 Cache to 256K.

2. Notice that the sizes of L1PSRAM, L1DSRAM and IRAM are adjusted down to zero.

Device Memory				
Name	Base	Length	Space	Access
IRAM	0x11800000	0x00000000	code/data	RWX
L3_CBA_RAM	0x80000000	0x00020000	code/data	RWX
IROM	0x11700000	0x00100000	code/data	RX
L1PSRAM	0x11e00000	0x00000000	code	RWX
L1DSRAM	0x11f00000	0x00000000	data	RW

L1D Cache: 32k   L1P Cache: 32k   L2 Cache: 256k
☐ Customize Memory
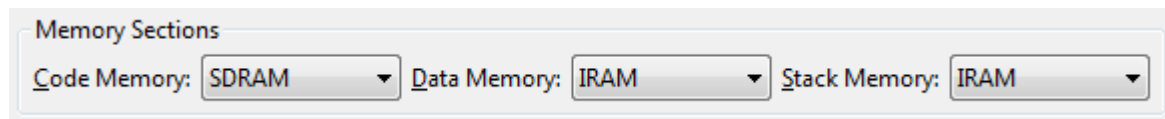
See Section 6.6 for more about cache configuration.

6.2.2.3 Selecting Default Memory Segments for Data, Code and Stack

The platform also determines the default memory segment for placement of code, data and stack. If you do not explicitly place a section, then the defaults are used. For example, if you do not configure the location of a Task stack in the .cfg file, then the task stack will be placed in the stack memory segment specified by the platform.

You can make coarse decisions on where you want your code, data, and stack to be placed by selecting values for data memory, code memory, and stack memory in your platform definition.

For example on the evmDA830, you may want your code in SDRAM and data in RAM. You can achieve this by selecting Code memory to be SDRAM and Data memory to be IRAM in the platform wizard.

Memory Sections
Code Memory: SDRAM   Data Memory: IRAM   Stack Memory: IRAM
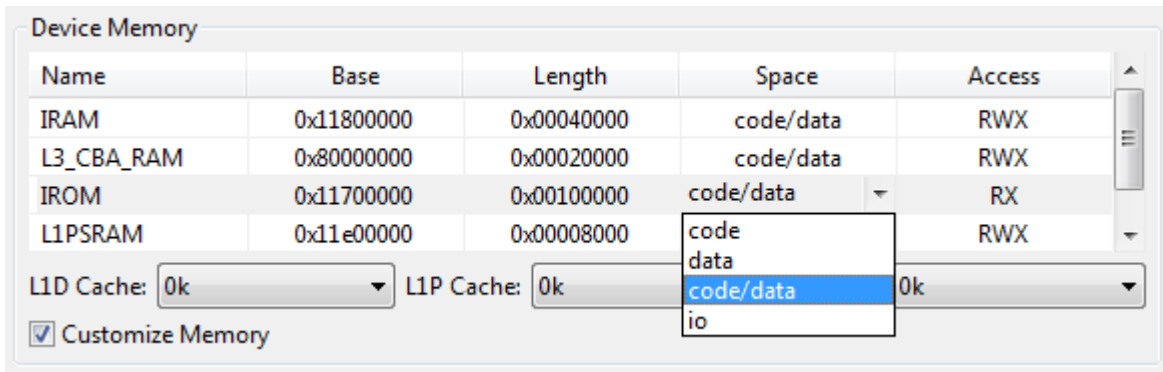
See Section 6.3 for details about how you can use the configuration files to be more specific about how memory sections should be placed in memory segments.

6.2.2.4 Setting Custom Base Addresses and Lengths for Segments

You can customize the names, locations, sizes, type, and access for both internal and external memory segments.

To customize internal memory segments, begin by checking the **Customize Memory** box in the Device Memory area. You can then click on fields in the list of memory segments and make changes. In the **Name**, **Base**, and **Length** columns, type the value you want to use. In the **Space** and **Access** columns, you can select from a list of options.
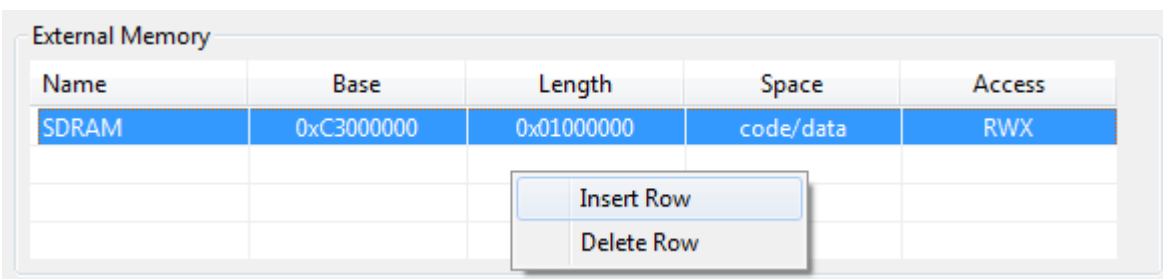
To customize external memory segments, you can right-click in the External Memory area and choose **Insert Row** or **Delete Row**.

In the new row, type a **Name**, **Base** address, and **Length** for the memory segment. Choose the type of memory **Space** and the permitted **Access** to this memory.

To watch a demo that shows how to customize memory segments, see http://rtsc.eclipse.org/docs-tip/Demo_of_Customizing_Memory_Sections.

6.3 Placing Sections into Memory Segments

> **Note:**   If you are using the MSP430, see Section 6.4 instead. This section does not apply to the MSP430.

The platform defines your application's memory map along with the default section placement in memory for those segments. (See Section 6.2 for details on platform configuration.) The platform provides general control over the placement of the "code", "data", and "stack" sections. For more fine-grained control of the sections, there are several options:

- To define and place new sections that are not managed by SYS/BIOS, you can either modify the project's configuration (.cfg) file as described in Sections 6.3.1 and 6.3.2 or provide a supplemental linker command file as described in Section 6.3.3.

- To modify the placement of sections managed by SYS/BIOS, you can either modify the project's configuration (.cfg) file as described in Sections 6.3.1 and 6.3.2 or provide your own linker command file to replace part or all of the XDCtools-generated one as described in Section 6.3.4.

Note:	To place sections into segments in the .cfg file, you will need to edit the source of your .cfg script in a text editor. Currently, you *cannot* use the XGCONF GUI editor.

6.3.1 *Configuring Simple Section Placement*

In a configuration file, section placement is done through the Program.sectMap[] array.

The simplest way to configure the segment in which a section should be placed is as follows:

```
Program.sectMap[".foo"] = "IRAM";
```

This example would cause the IRAM segment to be used both for loading and running the .foo section.

6.3.2 *Configuring Section Placement Using a SectionSpec*

The Program.sectMap[] array maps section names to structures of the type SectionSpec. If you use the simple statement syntax shown in the previous section, you don't need to create a SectionSpec structure. Using a SectionSpec structure gives you more precise control over how the run and load memory segments (or addresses) for a section are specified.

The SectionSpec structure contains the following fields.

- **runSegment.** The segment where the section is to be run.

- **loadSegment.** The segment where the section is to be loaded.

- **runAddress.** Starting address where the section is to be run. You cannot specify both the runSegment and runAddress.

- **loadAddress.** Starting address where the section is to be loaded. You cannot specify both the loadSegment and the loadAddress.

- **runAlign.** Alignment of the section specified by runSegment. If you specify the runSegment, you can also specify runAlign.

- **loadAlign.** Alignment of the section specified by loadSegment. If you specify the loadSegment, you can also specify loadAlign.

- **type.** You can use this field to define various target-specific flags to identify the section type. For example, COPY, DSECT, and NOLOAD.

- **fill.** If specified, this value is used to initialize an uninitialized section.

The following .cfg file statements specify the memory segments where the .foo section is to be loaded and run.

```
Program.sectMap[".foo"] = new Program.SectionSpec();
Program.sectMap[".foo"].loadSegment = "FLASH";
Program.sectMap[".foo"].runSegment = "RAM";
```

If you specify only the loadSegment or runSegment for a section, the default behavior is to use the specified segment for both loading and running.

The following statements place the Swi_post() function into the IRAM memory segment:

```
Program.sectMap[".text:_ti_sysbios_knl_Swi_post__F"] = new Program.SectionSpec();
Program.sectMap[".text:_ti_sysbios_knl_Swi_post__F"] = "IRAM";
```

The following statements place all static instances for the Task module into the .taskStatic section:

```
var Task = xdc.useModule('ti.sysbios.knl.Task');

Task.common$.instanceSection = ".taskStatic";
Program.sectMap[".taskStatic"] = new Program.SectionSpec();
Program.sectMap[".taskStatic"].loadSegment = "IRAM";
```

Configuration statements that specify sections using the sectMap array affect the section placement in the linker command file that is generated from the configuration.

6.3.3   *Providing a Supplemental Linker Command File*

It is possible to provide your own linker command file to supplement the one generated by XDCtools. You can do this to define new sections and to leverage all of the features available through the linker command language.

Simply add a linker command file you have written to your CCS project. The file must have a file extension of .cmd. CCS automatically recognizes such a linker command file and invokes it during the link step.

The definition of the memory map (the "MEMORY" specification of the linker command file) is handled by your platform, so this approach cannot be used to change the definition of the existing memory segments.

This approach works for defining new memory sections. If you want to change the placement of sections managed by SYS/BIOS, however, you must follow one of the approaches described in Section 6.3.1, 6.3.2, or 6.3.4.

6.3.4 *Default Linker Command File and Customization Options*

The linker command file used by a SYS/BIOS application is auto-generated when the configuration is processed. This command file is typically located with the configuration package, as shown here:
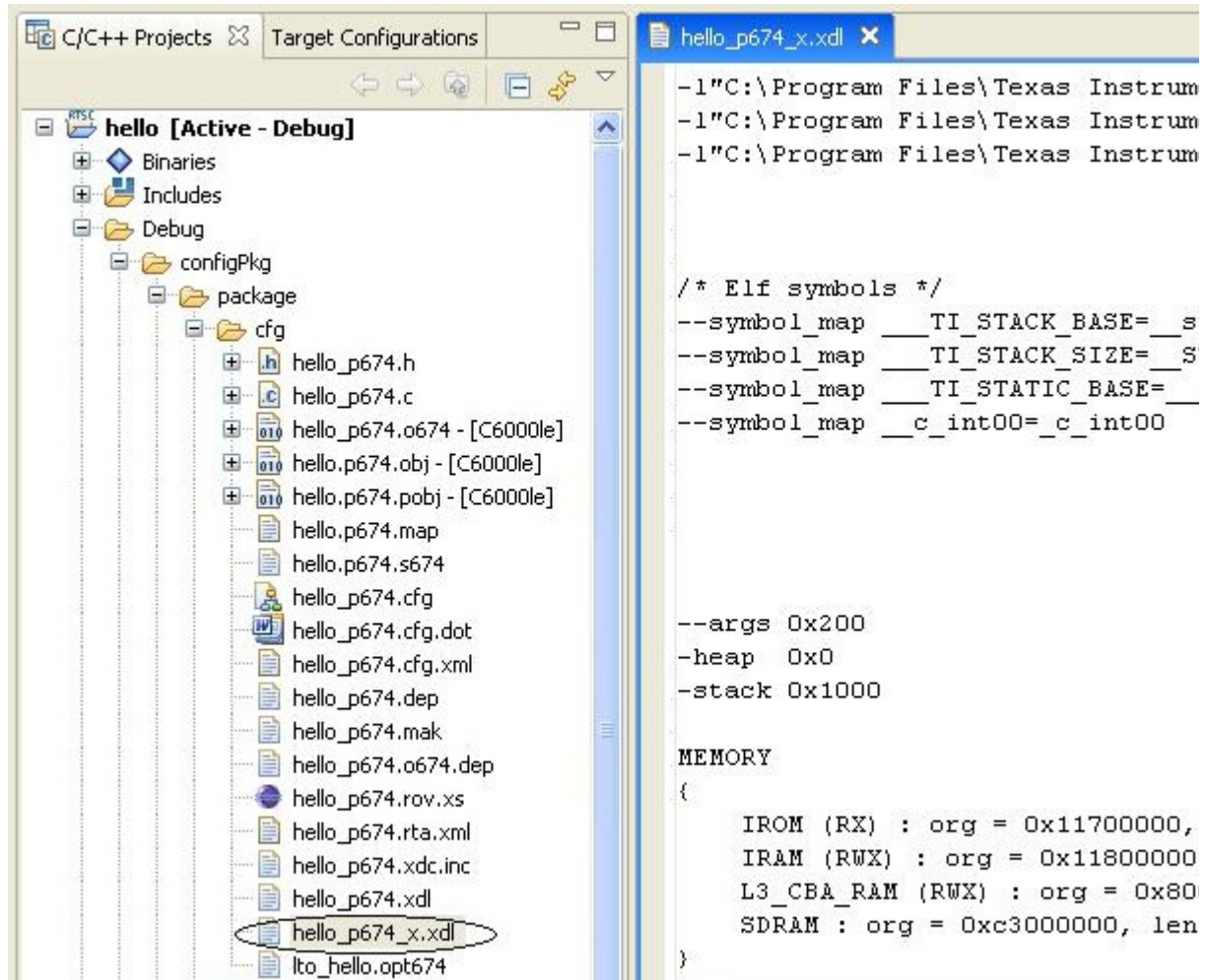
The auto-generated linker command file uses a template specified by the platform associated with the program. This command file defines MEMORY and SECTIONS as determined during the configuration process. The placement of sections in the configuration file is reflected in this auto-generated command file.

You can customize the auto-generated linker command file using any of the following techniques:

- Exclude sections from the auto-generated command file. See the page at http://rtsc.eclipse.org/cdoc-tip/xdc/cfg/Program.html#sections.Exclude for examples that configure the Program.sectionsExclude parameter.

- Replace the entire SECTIONS portion of the generated linker command file. See information about the Program.sectionsTemplate parameter at http://rtsc.eclipse.org/cdoc-tip/xdc/cfg/Program.html#sections.Template.

- Specify a template for the program's linker command file. See http://rtsc.eclipse.org/cdoc-tip/xdc/cfg/Program.html#link.Template for information and examples that use the Program.linkTemplate parameter. The simplest way to create a template for your program is to first auto-generate the linker command file, then edit it to suit your application's needs, and then set the Program.linkTemplate parameter to reference your edited file.

Important:	This technique requires that you copy the auto-generated linker command file and edit it every time you change the configuration, the platform, or install a new version of XDCtools.

6.4 Sections and Memory Mapping for MSP430, Stellaris M3, and C28x

When you create a project in CCS, you must select a device as part of the project settings (for example, MSP430F5435A) in the CCS project creation wizard. A linker command file specific to the selected device is automatically added to the project by CCS.

In the RTSC Configuration Settings page of the wizard, a Target and a Platform are automatically selected based on your previous selections. We recommend using "release" as the Build-Profile, even if you are in the debugging stage of code development. See Section 2.3.5 for more about build settings for reducing the size of your executable for the MSP430.

The platforms for the MSP430, Stellaris Cortex-M3 microcontrollers, and C28x devices differ from other platforms in that they do not define the memory map for the device. Instead, the linker command file added by the project wizard is used directly. Any changes to the memory map and section placement can be made by editing the linker command file directly. See the *MSP430 Optimizing C/C++ Compiler User's Guide* for more details on linker command file options.

Note that an additional XDCtools-generated linker command file is added to the project; this file places a few sections that are SYS/BIOS specific. This command file assumes that "FLASH" and "RAM" are part of the memory map.

Note:	For hardware-specific information about using SYS/BIOS, see links on the http://processors.wiki.ti.com/index.php/Category:SYSBIOS page.

6.5 Stacks

SYS/BIOS uses a single system stack for hardware interrupts and a separate task stack for each Task instance.

6.5.1 System Stack

You can configure the size of the System stack, which is used as the stack for hardware interrupts and software interrupts (and by the Idle functions if Task is disabled). You should set the System stack size to meet the application's needs. See Section 3.4.3 for information about system stack size requirements.

You can use the .stack section to control the location of the system stack. For example, the following configuration statements place the system stack of size 0x400 in the IRAM segment.

```
Program.stack = 0x400;
Program.sectMap[".stack"] = "IRAM";
```

Setting Program.stack generates appropriate linker options in the linker command file to allow the system stack to be allocated at link time. For example, the linker command file for a C6000 application might include the command option `-stack 0x0400`.

See your project's auto-generated linker command file for symbols related to the system stack size and location. For C6000 these include __TI_STACK_END and __STACK_SIZE.

6.5.2 Task Stacks

If the Task module is enabled, SYS/BIOS creates an additional stack for each Task instance the application contains (plus one task stack for the Idle threads). See Section 3.5.3 for information about task stack size requirements.

You can specify the size of a Task's stack in the configuration file. (You can use XGCONF to do this or edit the .cfg file directly.) For example:

```
var Task = xdc.useModule('ti.sysbios.knl.Task');

/* Set default stack size for tasks */
Task.defaultStackSize = 1024;

/* Set size of idle task stack */
Task.idleTaskStackSize = 1024;

/* Create a Task Instance and set stack size */
var tskParams = new Task.Params;
tskParams.stackSize = 1024;
var task0 = Task.create('&task0Fxn', tskParams);
```

You can control the location of task stacks for statically-created Tasks by using Program.sectMap[]. For example:

```
/* Place idle task stack section */
Program.sectMap[".idleTaskStackSection"] = "IRAM";

/* Place other static task stacks */
Program.sectMap[".taskStackSection"] = "IRAM";
```

6.5.3   *ROV for System Stacks and Task Stacks*

At runtime, you can use **Tools > RTOS Object View (ROV)** in CCS to browse for details about each of the Task instances. For information about ROV, see the RTSC-pedia page at http://rtsc.eclipse.org/docs-tip/RTSC_Object_Viewer. The Detailed tab shows the task stack information and status.
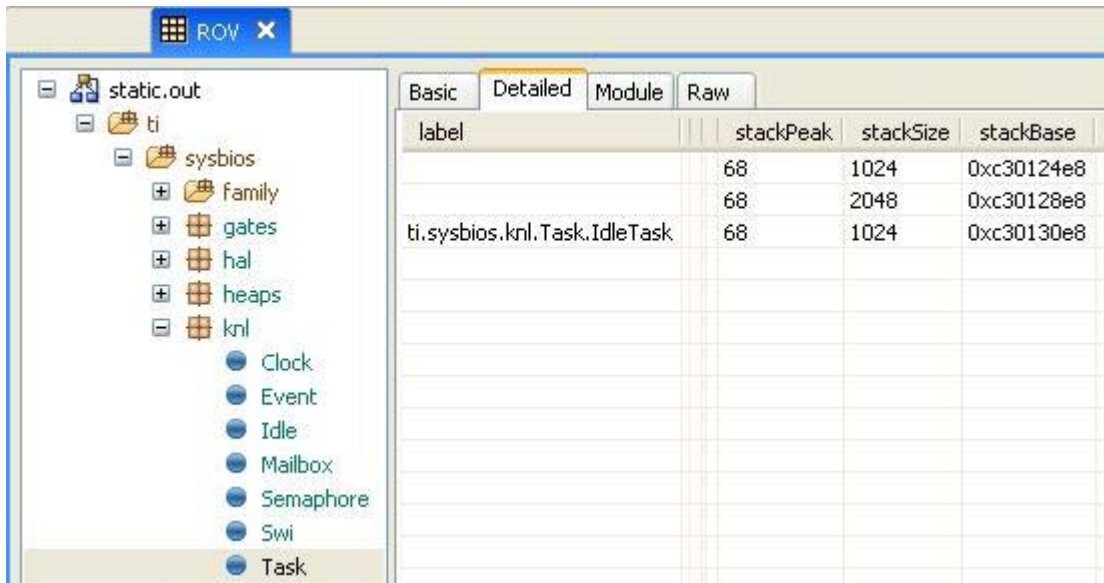
The Module tab of the Task view shows the HwiStack (which is the system stack) information and status.
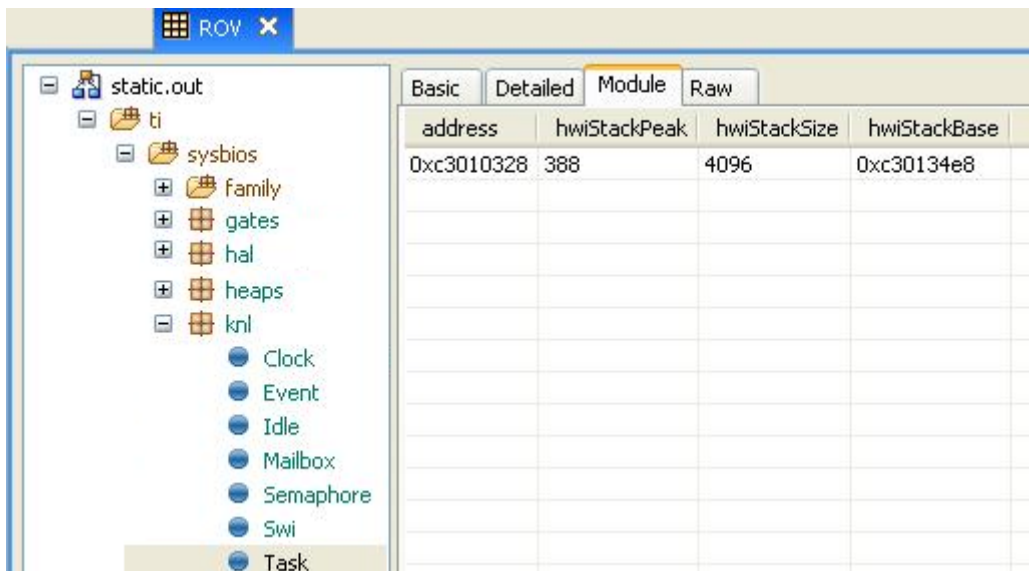
6.6 Cache Configuration

C6000 cache sizes in SYS/BIOS 6.x are determined by the platform you choose. To change cache sizes, create or modify the platform using the platform wizard as described in Section 6.2.2.2.

The following subsections describe ways you can use the family-specify Cache modules in SYS/BIOS to manipulate caching behavior.

6.6.1 Configure Cache Size Registers at Startup

For C6000 targets, the ti.sysbios.hal.Cache module gets cache sizes from the platform and sets the cache size registers at startup time. The ti.sysbios.hal.Cache module is a generic module whose implementations are provided in the family-specific ti.sysbios.family.*.Cache modules. See Section 7.4 for more about the Cache module and its implementations.

For example on the DA830, the Cache module sets the L1PCFG, L1DCFG and L2CFG registers at startup time.

6.6.2 Configure Parameters to Set MAR Registers

For C6000 targets, the ti.sysbios.family.c64p.Cache module defines Cache.MAR##-## configuration parameters that allow you to control which external memory addresses are cacheable or non-cacheable. For example, Cache_MAR128_159 is one such configuration parameter. These configuration parameters directly map to the MAR registers on the device. Each external memory address space of 16 MB is controlled by one MAR bit (0: noncacheable, 1:cacheable).

The SYS/BIOS Cache module has module-wide configuration parameters that map to the MAR registers.

By default the C64P Cache module makes all memory regions defined in the platform cacheable by setting all of the corresponding MAR bits to 0x1. To disable caching on a DA830 device for the external memory range from 8000 0000h to 80FF FFFFh, you set Cache.MAR128_159 = 0x0 as follows. This sets register MAR128 to 0.

```
var Cache = xdc.useModule('ti.sysbios.family.c64p.Cache');
Cache.MAR_128_159 = 0x0;
```

After the MAR bit is set for an external memory space, new addresses accessed by the CPU will be cached in L2 cache or, if L2 is disabled, in L1. See device-specific reference guides for MAR registers and their mapping to external memory addresses.

At system startup, the Cache module writes to the MAR registers and configures them.

6.6.3 Cache Runtime APIs

For any target that has a cache, the ti.sysbios.hal.Cache module provides APIs to manipulate caches at runtime. These include the Cache_enable(), Cache_disable(), Cache_wb(), and Cache_inv() functions. See Section 7.4.1 for details.

6.7 Dynamic Memory Allocation

A "Heap" is a module that implements the IHeap interface. Heaps are dynamic memory managers: they manage a specific piece of memory and support allocating and freeing pieces ("blocks") of that memory.

Memory allocation sizes are measured in "Minimum Addressable Units" (MAUs) of memory. An MAU is the smallest unit of data storage that can be read or written by the CPU. For the C28x, this is an 16-bit word. For the all other currently supported target families—including C6000, ARM, and MSP430—this is an 8-bit byte.

6.7.1 Memory Policy

You can reduce the amount of code space used by an application by setting the memoryPolicy on a global or per-module basis. This is particularly useful on targets where the code memory is significantly constrained.

The options are:

- **DELETE_POLICY.** This is the default. The application creates and deletes objects (or objects for this module) at runtime. You need both the *MODULE*_create() functions and the *MODULE*_delete() functions to be available to the application.

- **CREATE_POLICY.** The application creates objects (or objects for this module) at runtime. It does not delete objects at runtime. You need the *MODULE*_create() functions to be available to the application, but not the *MODULE*_delete() functions.

- **STATIC_POLICY.** The application creates all objects (or all objects for this module) in the configuration file. You don't need the *MODULE*_create() or the *MODULE*_delete() functions to be available to the application.

For example, the following configuration statements set the default memory policy for all modules to static instance creation only:

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Types = xdc.useModule('xdc.runtime.Types');
Defaults.memoryPolicy = Types.STATIC_POLICY;
```

6.7.2 Specifying the Default System Heap

The BIOS module creates a default heap for use by SYS/BIOS. When Memory_alloc() is called at runtime with a NULL heap, this system heap will be used. The default system heap created by the BIOS module is a HeapMem instance. The BIOS module provides the following configuration parameters related to the system heap:

- BIOS.heapSize can be used to set the system heap size.

- BIOS.heapSection can be used to place the system heap.

For example, you can configure the default system heap with XDCscript as follows:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.heapSize = 0x900;
BIOS.heapSection = "systemHeap";
```

If you want to use a different heap manager for the system heap, you can specify the system heap in your configuration file and SYS/BIOS will not override the setting.

> **Note:**   The SYS/BIOS system heap cannot be a HeapStd instance. The BIOS module detects
> this condition and generates an error message.

The following configuration statements specify a system heap that uses HeapBuf instead of HeapMem:

```
/* Create a heap using HeapBuf */
var heapBufParams = new HeapBuf.Params;
heapBufParams.blockSize = 128;
heapBufParams.numBlocks = 2;
heapBufParams.align = 8;
heapBufParams.sectionName = "myHeap";
Program.global.myHeap = HeapBuf.create(heapBufParams);

Program.sectMap["myHeap"] = "DDR";
Memory.defaultHeapInstance = Program.global.myHeap;
```

If you do not want a system heap to be created, you can set BIOS.heapSize to zero. The BIOS module
will then use a HeapNull instance to minimize code/data usage.

6.7.3   *Using the xdc.runtime.Memory Module*

All dynamic allocation is done through the xdc.runtime.Memory module. The Memory module provides
APIs such as Memory_alloc() and Memory_free(). All Memory APIs take an IHeap_Handle as their first
argument. The Memory module does very little work itself; it makes calls to the Heap module through the
IHeap_Handle. The Heap module is responsible for managing the memory. Using Memory APIs makes
applications and middleware portable and not tied to a particular heap implementation.

IHeap_Handles to be used with Memory APIs are obtained by creating Heap instances statically or
dynamically. When The IHeap_Handle passed to the Memory APIs is NULL, the default system heap is
used. See Section 6.7.2.

Runtime example: This example allocates and frees memory from two different heaps. It allocates from
the system heap by passing NULL to Memory_alloc as the IHeap_Handle. It allocates from a separate
heap called "otherHeap" by explicitly passing the "otherHeap" handle.

```
#include <xdc/std.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/Error.h>

extern IHeap_Handle systemHeap, otherHeap;

Void main()
{
    Ptr buf1, buf2;
    Error_Block eb;
    Error_init(&eb);
```

```
    /* Alloc and free using systemHeap */
    buf1 = Memory_alloc(NULL, 128, 0, &eb);
    if (buf1 == NULL) {
        System_abort("Memory allocation for buf1 failed");

    }
    Memory_free(NULL, buf1, 128);


    /* Alloc and free using otherHeap */
    buf2 = Memory_alloc(otherHeap, 128, 0, &eb);
    if (buf2 == NULL) {
        System_abort("Memory allocation for buf2 failed");
    }
    Memory_free(otherHeap, buf2, 128);
}
```

6.7.4   *Specifying a Heap for Module Dynamic Instances*

You can specify the default heap to be used when allocating memory for dynamically-created module instances. The configuration property that controls the default heap for all modules is Default.common$.instanceHeap.

For example, these configuration statements specify the heap for allocating instances:

```
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

var heapMemParams = new HeapMem.Params;
heapMemParams.size = 8192;
var heap1 = HeapMem.create(heapMemParams);
Default.common$.instanceHeap = heap1;
```

If you do not specify a separate heap for instances, the heap specified for Memory.defaultHeapInstance will be used (see Section 6.7.2).

To specify the heap a specific module uses when it allocates memory for dynamically-created instances, set the instanceHeap parameter for that module. For example, the following configuration statements specify the heap for the Semaphore module:

```
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

var heapMemParams = new HeapMem.Params;
heapMemParams.size = 8192;
var heap1 = HeapMem.create(heapMemParams);

Semaphore.common$.instanceHeap = heap1;
```

6.7.5 *Using malloc() and free()*

Applications can call the malloc() and free() functions. Normally these functions are provided by the RTS library supplied by the code generation tools. However, when you are using SYS/BIOS, these functions are provided by SYS/BIOS and re-direct allocations to the default system heap (see Section 6.7.2).

To change the size of the heap used by malloc(), use the BIOS.heapSize configuration parameter.

6.8 Heap Implementations

The xdc.runtime.Memory module is the common interface for all memory operations. The actual memory management is performed by a Heap instance, such as an instance of HeapMem or HeapBuf. For example, Memory_alloc() is used at runtime to dynamically allocate memory. All of the Memory APIs take a Heap instance as one of their parameters. Internally, the Memory module calls into the heap's interface functions.

The xdc.runtime.Memory module is documented in the XDCtools online help and the RTSC-pedia wiki. Implementations of Heaps provided by SYS/BIOS are discussed here.

XDCtools provides the HeapMin and HeapStd heap implementations. See the CDOC help for XDCtools for details on these implementations.

SYS/BIOS provides the following Heap implementations:

- **HeapMem.** Allocate variable-size blocks. Section 6.8.1

- **HeapBuf.** Allocate fixed-size blocks. Section 6.8.2

- **HeapMultiBuf.** Specify variable-size allocation, but internally allocate from a variety of fixed-size blocks. Section 6.8.3

- **HeapTrack.** Used to detect memory allocation and deallocation problems. Section 6.8.4

This table compares SYS/BIOS heap implementations:

Table 6–1. Heap Implementation Comparison

Module	Description/Characteristics	Limitations
ti.sysbios.heaps.HeapMem	Uses Gate to protect during allocation and freeing, accepts any block size	Slower, non-deterministic
ti.sysbios.heaps.HeapBuf	Fast, deterministic, non-blocking	Allocates blocks of a single size
ti.sysbios.heaps.HeapMultiBuf	Fast, deterministic, non-blocking, multiple-block sizes supported	Limited number of block sizes
ti.sysbios.heaps.HeapTrack	Detects memory leaks, buffer overflows, and double frees of memory.	Performance and size penalty associated with tracking.

Different heap implementations optimize for different memory management traits. The HeapMem module (Section 6.8.1) accepts requests for all possible sizes of blocks, so it minimizes internal fragmentation. The HeapBuf module (Section 6.8.2), on the other hand, can only allocate blocks of a fixed size, so it minimizes external fragmentation in the heap and is also faster at allocating and freeing memory.

6.8.1   *HeapMem*

HeapMem can be considered the most "flexible" of the Heaps because it allows you to allocate variable-sized blocks. When the size of memory requests is not known until runtime, it is ideal to be able to allocate exactly how much memory is required each time. For example, if a program needs to store an array of objects, and the number of objects needed isn't known until the program actually executes, the array will likely need to be allocated from a HeapMem.

The flexibility offered by HeapMem has a number of performance tradeoffs.

- **External Fragmentation.** Allocating variable-sized blocks can result in fragmentation. As memory blocks are "freed" back to the HeapMem, the available memory in the HeapMem becomes scattered throughout the heap. The total amount of free space in the HeapMem may be large, but because it is not contiguous, only blocks as large as the "fragments" in the heap can be allocated.

   This type of fragmentation is referred to as "external" fragmentation because the blocks themselves are allocated exactly to size, so the fragmentation is in the overall heap and is "external" to the blocks themselves.

- **Non-Deterministic Performance.** As the memory managed by the HeapMem becomes fragmented, the available chunks of memory are stored on a linked list. To allocate another block of memory, this list must be traversed to find a suitable block. Because this list can vary in length, it's not known how long an allocation request will take, and so the performance becomes "non-deterministic".

A number of suggestions can aide in the optimal use of a HeapMem.

- **Larger Blocks First.** If possible, allocate larger blocks first. Previous allocations of small memory blocks can reduce the size of the blocks available for larger memory allocations.

- **Overestimate Heap Size.** To account for the negative effects of fragmentation, use a HeapMem that is significantly larger than the absolute amount of memory the program will likely need.

When a block is freed back to the HeapMem, HeapMem combines the block with adjacent free blocks to make the available block sizes as large as possible.

Note: HeapMem uses a user-provided lock to lock access to the memory. For details, see section 4.3, *Gates*.

The following examples create a HeapMem instance with a size of 1024 MAUs.

Configuration example: The first example uses XDCtools to statically configure the heap:

```
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
/* Create heap as global variable so it can be used in C code */
var heapMemParams = new HeapMem.Params();
heapMemParams.size = 1024;
Program.global.myHeap = HeapMem.create(heapMemParams);
```

Runtime example: This second example uses C code to dynamically create a HeapMem instance:

```
HeapMem_Params prms;
static char *buf[1024];
HeapMem_Handle heap;
Error_Block eb;

Error_init(&eb);
HeapMem_Params_init(&prms);
prms.size = 1024;
prms.buf = (Ptr)buf;
heap = HeapMem_create(&prms, &eb);
if (heap == NULL) {
    System_abort("HeapMem create failed");
}
```

HeapMem uses a Gate (see the Gates section for an explanation of Gates) to prevent concurrent accesses to the code which operates on a HeapMem's list of free blocks. The type of Gate used by HeapMem is statically configurable through the HeapMem's common defaults.

Configuration example: This example configures HeapMem to use a GateMutexPri to protect critical regions of code.

```
var GateMutexPri = xdc.useModule('ti.sysbios.gates.GateMutexPri');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

HeapMem.common$.gate = GateMutexPri.create();
```

The type of Gate used depends upon the level of protection needed for the application. If there is no risk of concurrent accesses to the heap, then "null" can be assigned to forgo the use of any Gate, which would improve performance. For an application that could have concurrent accesses, a GateMutex is a likely choice. Or, if it is possible that a critical thread will require the HeapMem at the same time as a low-priority thread, then a GateMutexPri would be best suited to ensuring that the critical thread receives access to the HeapMem as quickly as possible. See section 4.3.2.2, *GateMutexPri* for more information.

6.8.2 *HeapBuf*

HeapBuf is used for allocating fixed-size blocks of memory, and is designed to be fast and deterministic. Often a program needs to create and delete a varying number of instances of a fixed-size object. A HeapBuf is ideal for allocating space for such objects, since it can handle the request quickly and without any fragmentation.

A HeapBuf may also be used for allocating objects of varying sizes when response time is more important than efficient memory usage. In this case, a HeapBuf will suffer from "internal" fragmentation. There will never be any fragmented space in the heap overall, but the allocated blocks themselves may contain wasted space, so the fragmentation is "internal" to the allocated block.

Allocating from and freeing to a HeapBuf always takes the same amount of time, so a HeapBuf is a "deterministic" memory manager.

The following examples create a HeapBuf instance with 10 memory blocks of size 128.

Configuration example: The first example uses XDCtools to statically configure the heap. In this configuration example, no buffer or bufSize parameter needs to be specified, since the configuration can compute these values and allocate the correct sections at link time.

```
var HeapBuf = xdc.useModule('ti.sysbios.heaps.HeapBuf');
/* Create heap as global variable so it can be used in C code */
var heapBufParams = new HeapBuf.Params();
heapBufParams.blockSize = 128;
heapBufParams.numBlocks = 10;
Program.global.myHeap = HeapBuf.create(heapBufParams);
```

Runtime example: This second example uses C code to dynamically create a HeapBuf instance. In this example, you must pass the bufSize and buf parameters. Be careful when specifying these runtime parameters. The blocksize needs to be a multiple of the worst-case structure alignment size. And bufSize should be equal to blockSize * numBlocks. The worst-case structure alignment is target dependent. On C6x and ARM devices, this value is 8. The base address of the buffer should also be aligned to this same size.

```
HeapBuf_Params prms;
static char *buf[1280];
HeapBuf_Handle heap;
Error_Block eb;

Error_init(&eb);
HeapBuf_Params_init(&prms);
prms.blockSize = 128;
prms.numBlocks = 10;
prms.buf = buf;
prms.bufSize = 1280;
heap = HeapBuf_create(&prms, &eb);
if (heap == NULL) {
    System_abort("HeapBuf create failed");
}
```

6.8.3   *HeapMultiBuf*

HeapMultiBuf is intended to balance the strengths of HeapMem and HeapBuf. Internally, a HeapMultiBuf maintains a collection of HeapBuf instances, each with a different block size, alignment, and number of blocks. A HeapMultiBuf instance can accept memory requests of any size, and simply determines which of the HeapBufs to allocate from.

A HeapMultiBuf provides more flexibility in block size than a single HeapBuf, but largely retains the fast performance of a HeapBuf. A HeapMultiBuf instance has the added overhead of looping through the HeapBufs to determine which to allocate from. In practice, though, the number of different block sizes is usually small and is always a fixed number, so a HeapMultiBuf can be considered deterministic by some definitions.

A HeapMultiBuf services a request for any memory size, but always returns one of the fixed-sized blocks. The allocation will not return any information about the actual size of the allocated block. When freeing a block back to a HeapMultiBuf, the size parameter is ignored. HeapMultiBuf determines the buffer to free the block to by comparing addresses.

When a HeapMultiBuf runs out of blocks in one of its buffers, it can be configured to allocate blocks from the next largest buffer. This is referred to as "block borrowing". See the online reference described in Section 1.6.1 for more about HeapMultiBuf.

The following examples create a HeapMultiBuf that manages 1024 MAUs of memory, which are divided into 3 buffers. It will manage 8 blocks of size 16 MAUs, 8 blocks of size 32 MAUs, and 5 blocks of size 128 MAUs as shown in the following diagram.

Program.global.myHeap

16 MAUs	16 MAUs	16 MAUs	16 MAUs	16 MAUs	16 MAUs	16 MAUs	16 MAUs
32 MAUs		32 MAUs		32 MAUs		32 MAUs	
32 MAUs		32 MAUs		32 MAUs		32 MAUs	
128 MAUs							
128 MAUs							
128 MAUs							
128 MAUs							
128 MAUs							

Configuration example: The first example uses XDCtools to statically configure the HeapMultiBuf instance:

```
var HeapMultiBuf = xdc.useModule('ti.sysbios.heaps.HeapMultiBuf');


/* HeapMultiBuf without blockBorrowing. */
/* Create as a global variable to access it from C Code. */
var heapMultiBufParams = new HeapMultiBuf.Params();
heapMultiBufParams.numBufs = 3;
heapMultiBufParams.blockBorrow = false;
heapMultiBufParams.bufParams =
     [{blockSize: 16, numBlocks:8, align: 0},
      {blockSize: 32, numBlocks:8, align: 0},
      {blockSize: 128, numBlocks:5, align: 0}];
Program.global.myHeap =
   HeapMultiBuf.create(heapMultiBufParams);
```

Runtime example: This second example uses C code to dynamically create a HeapMultiBuf instance:

```
HeapMultiBuf_Params prms;
HeapMultiBuf_Handle heap;
Error_Block eb;


Error_init(&eb);


/* Create the buffers to manage */
Char buf0[128];
Char buf1[256];
Char buf2[640];


/* Create the array of HeapBuf_Params */
HeapBuf_Params bufParams[3];
```

```
/* Load the default values */
HeapMultiBuf_Params_init(&prms);
prms.numBufs = 3;
prms.bufParams = bufParams;


HeapBuf_Params_init(&prms.bufParams[0]);
prms.bufParams[0].align = 0;
prms.bufParams[0].blockSize = 16;
prms.bufParams[0].numBlocks = 8;
prms.bufParams[0].buf = (Ptr) buf0;
prms.bufParams[0].bufSize = 128;


HeapBuf_Params_init(&prms.bufParams[1]);
prms.bufParams[1].align = 0;
prms.bufParams[1].blockSize = 32;
prms.bufParams[1].numBlocks = 8;
prms.bufParams[1].buf = (Ptr) buf1;
prms.bufParams[1].bufSize = 256;


HeapBuf_Params_init(&prms.bufParams[2]);
prms.bufParams[2].align = 0;
prms.bufParams[2].blockSize = 128;
prms.bufParams[2].numBlocks = 5;
prms.bufParams[2].buf = (Ptr) buf2;
prms.bufParams[2].bufSize = 640;


heap = HeapMultiBuf_create(&prms, &eb);
if (heap == NULL) {
    System_abort("HeapMultiBuf create failed");
}
```

6.8.4 *HeapTrack*

HeapTrack is a buffer management module that tracks all currently allocated blocks for any heap instance. HeapTrack is useful for detecting memory leaks, buffer overflows, and double frees of memory blocks. Any XDCtools or SYSBIOS heap instance can plugged into HeapTrack. For each memory allocation, an extra packet of data will be added. This data is used by the RTOS Object Viewer (ROV) to display information about the heap instance.

HeapTrack implements the default heap functions as well as two debugging functions. The first, HeapTrack_printHeap() prints all the memory blocks for a given HeapTrack instance. The second, HeapTrack_printTask() prints all memory blocks for a given Task handle.

HeapTrack has several asserts that detect key memory errors. These include freeing the same block of memory twice, overflowing an allocated block of memory, deleting a non-empty heap instance, and calling HeapTrack_printHeap() with a null heap object.

There is both a performance and size overhead cost when using HeapTrack. These costs should be taken into account when setting heap and buffer sizes.

You can find the amount by which the HeapTrack module increases the block size by using `sizeof(HeapTrack_Tracker)` in your C code. This is the amount by which the size is increased when your code or some other function calls Memory_alloc() from a heap managed by HeapTrack. The HeapTrack_Tracker structure is added to the end of an allocated block; HeapTrack therefore does not impact the alignment of allocated blocks.

Configuration example: This example statically configures HeapTrack with an existing heap.

```
var HeapTrack = xdc.useModule('ti.sysbios.heaps.HeapTrack');

var heapTrackParams = new HeapTrack.Params();
heapTrackParams.heap = heapHandle;
Program.global.myHeap = HeapTrack.create(heapTrackParams);
```

You can also use HeapTrack with the default BIOS module heap by setting the heapTrackEnabled configuration parameter.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.heapTrackEnabled = true;
```

Runtime example: This example uses C code to dynamically create a HeapTrack instance with an existing heap.

```
HeapTrack_Params prms;
HeapTrack_Handle heap;
Error_Block eb;

Error_init(&eb);
HeapTrack_Params_init(&prms);
prms.heap = heapHandle;
heap = HeapTrack_create(&prms, &eb);
if (heap == NULL) {
    System_abort("HeapTrack create failed");
}
```

Hardware Abstraction Layer

This chapter describes modules that provide hardware abstractions.

7.1   Hardware Abstraction Layer APIs

SYS/BIOS provides services for configuration and management of interrupts, cache, and timers. Unlike other SYS/BIOS services such as threading, these modules directly program aspects of a device's hardware and are grouped together in the Hardware Abstraction Layer (HAL) package. Services such as enabling and disabling interrupts, plugging of interrupt vectors, multiplexing of multiple interrupts to a single vector, and cache invalidation or writeback are described in this chapter.

Note:	Any configuration or manipulation of interrupts and their associated vectors, the cache, and timers in a SYS/BIOS application must be done through the SYS/BIOS HAL APIs. In earlier versions of DSP/BIOS, some HAL services were not available and developers were expected to use functions from the Chip Support Library (CSL) for a device. The most recent releases of CSL (3.0 or above) are designed for use in applications that do not use SYS/BIOS. Some of their services are not compatible with SYS/BIOS. Avoid using CSL interrupt, cache, and timer functions and SYS/BIOS in the same application, since this combination is known to result in complex interrupt-related debugging problems.

The HAL APIs fall into two categories:

- Generic APIs that are available across all targets and devices
- Target/device-specific APIs that are available only for a specific device or ISA family

The generic APIs are designed to cover the great majority of use cases. Developers who are concerned with ensuring easy portability between different TI devices are best served by using the generic APIs as much as possible. In cases where the generic APIs cannot enable use of a device-specific hardware feature that is advantageous to the software application, you may choose to use the target/device-specific APIs, which provide full hardware entitlement.

In this chapter, an overview of the functionality of each HAL package is provided along with usage examples for that package's generic API functions. After the description of the generic functions, examples of target/device-specific APIs, based on those provided for 'C64x+ devices, are also given. For a full description of the target/device-specific APIs available for a particular family or device, please refer to the API reference documentation. section 7.5, *HAL Package Organization* provides an overview of the generic HAL packages and their associated target/device-specific packages to facilitate finding the appropriate packages.

Note:	For hardware-specific information about using SYS/BIOS, see the links at http://processors.wiki.ti.com/index.php/Category:SYSBIOS.

7.2 HWI Module

The ti.sysbios.hal.Hwi module provides a collection of APIs for managing hardware interrupts. These APIs are generic across all supported targets and devices and should provide sufficient functionality for most applications. See the video introducing Hwis for an overview.

7.2.1 Associating a C Function with a System Interrupt Source

To associate a user-provided C function with a particular system interrupt, you create a Hwi object that encapsulates information regarding the interrupt required by the Hwi module.

The standard static and dynamic forms of the "create" function are supported by the ti.sysbios.hal.Hwi module.

Configuration example: The following example statically creates a Hwi object that associates interrupt 5 with the "myIsr" C function using default instance configuration parameters:

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');

Hwi.create(5, '&myIsr');
```

Runtime example: The C code required to configure the same interrupt dynamically would be as follows:

```
#include <ti/sysbios/hal/Hwi>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>

Hwi_Handle myHwi;
Error_Block eb;

Error_init(&eb);

myHwi = Hwi_create(5, myIsr, NULL, &eb);
if (myHwi == NULL) {
    System_abort("Hwi create failed");
}
```

The NULL argument is used when the default instance parameters are satisfactory for creating a Hwi object.

7.2.2 Hwi Instance Configuration Parameters

The following configuration parameters and their default values are defined for each Hwi object. For a more detailed discussion of these parameters and their values see the ti.sysbios.hal.Hwi module in the online documentation. (For information on running online help, see Section 1.6.1, *Using the API Reference Help System*, page 1-22.)

- The "maskSetting" defines how interrupt nesting is managed by the interrupt dispatcher.

```
MaskingOption maskSetting = MaskingOption_SELF;
```

- The configured "arg" parameter will be passed to the Hwi function when the dispatcher invokes it.

```
UArg arg = 0;
```

- The "enabledInt" parameter is used to automatically enable or disable an interrupt upon Hwi object creation.

```
Bool enableInt = true;
```

- The "eventId" accommodates 'C6000 devices that allow dynamic association of a peripheral event to an interrupt number. The default value of -1 leaves the eventId associated with an interrupt number in its normal (reset) state (that is, no re-association is required).

```
Int eventId = -1;
```

- The "priority" parameter is provided for those architectures that support interrupt priority setting. The default value of -1 informs the Hwi module to set the interrupt priority to a default value appropriate to the device.

```
Int priority = -1;
```

7.2.3 *Creating a Hwi Object Using Non-Default Instance Configuration Parameters*

Building on the examples given in Section 7.2.1, the following examples show how to associate interrupt number 5 with the "myIsr" C function, passing "10" as the argument to "myIsr" and leaving the interrupt disabled after creation.

Configuration example:

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');

/* initialize hwiParams to default values */
var hwiParams = new Hwi.Params;

hwiParams.arg = 10;           /* Set myIsr5 argument to 10 */
hwiParams.enableInt = false; /* override default setting */

/* Create a Hwi object for interrupt number 5
 * that invokes myIsr5() with argument 10 */
Hwi.create(5, '&myIsr', hwiParams);
```

Runtime example:

```
#include <ti/sysbios/hal/Hwi>
#include <xdc/runtime/Error.h>


Hwi_Params hwiParams;
Hwi_Handle myHwi;
Error_Block eb;

/* initialize error block and hwiParams to default values */
Error_init(&eb);
Hwi_Params_init(&hwiParams);

hwiParams.arg = 10;
hwiParams.enableInt = FALSE;

myHwi = Hwi_create(5, myIsr, &hwiParms, &eb);
if (myHwi == NULL) {
    System_abort("Hwi create failed");
}
```

7.2.4 Enabling and Disabling Interrupts

You can enable and disable interrupts globally as well as individually with the following Hwi module APIs:

- `UInt Hwi_enable();`
  Globally enables all interrupts. Returns the previous enabled/disabled state.

- `UInt Hwi_disable();`
  Globally disables all interrupts. Returns the previous enabled/disabled state.

- `Hwi_restore(UInt key);`
  Restores global interrupts to their previous enabled/disabled state. The "key" is the value returned from Hwi_disable() or Hwi_enable().

- The APIs that follow are used for enabling, disabling, and restoring specific interrupts given by "intNum". They have the same semantics as the global Hwi_enable/disable/restore APIs.:

  - `UInt Hwi_enableInterrupt(UInt intNum);`

  - `UInt Hwi_disableInterrupt(UInt intNum);`

  - `Hwi_restoreInterrupt(UInt key);`

- `Hwi_clearInterrupt(UInt intNum);`
  Clears "intNum" from the set of currently pending interrupts.

Disabling hardware interrupts is useful during a critical section of processing.

On the C6000 platform, Hwi_disable() clears the GIE bit in the control status register (CSR). On the C2000 platform, Hwi_disable() sets the INTM bit in the ST1 register.

7.2.5 A Simple Example Hwi Application

The following example creates two Hwi objects. One for interrupt number 5 and another for interrupt number 6. For illustrative purposes, one interrupt is created statically and the other dynamically. An idle function that waits for the interrupts to complete is also added to the Idle function list.

Configuration example:

```
/* Pull in BIOS module required by ALL BIOS applications */
xdc.useModule('ti.sysbios.BIOS');

/* Pull in XDC runtime System module for various APIs used */
xdc.useModule('xdc.runtime.System');


/* Get handle to Hwi module for static configuration */
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');

var hwiParams = new Hwi.Params; /* Initialize hwiParams to default values */
hwiParams.arg = 10;             /* Set myIsr5 argument */
hwiParams.enableInt = false;    /* Keep interrupt 5 disabled until later */

/* Create a Hwi object for interrupt number 5
 * that invokes myIsr5() with argument 10 */
Hwi.create(5, '&myIsr5', hwiParams);


/* Add an idle thread 'myIdleFunc' that monitors interrupts. */
var Idle = xdc.useModule(ti.sysbios.knl.Idle);

Idle.addFunc('&myIdleFunc');
```

Runtime example:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Error.h>
#include <ti/sysbios/hal/Hwi>

Bool Hwi5 = FALSE;
Bool Hwi6 = FALSE;

```

```
main(Void)    {
    Hwi_Params hwiParams;
    Hwi_Handle myHwi;
    Error_Block eb;

    /* Initialize error block and hwiParams to default values */
    Error_init(&eb);
    Hwi_Params_init(&hwiParams);

    /* Set myIsr6 parameters */
    hwiParams.arg = 12;
    hwiParams.enableInt = FALSE;

    /* Create a Hwi object for interrupt number 6
     * that invokes myIsr6() with argument 12 */
    myHwi = Hwi_create(6, myIsr6, &hwiParms, &eb);
    if (myHwi == NULL) {
        System_abort("Hwi create failed");
    }

    /* enable both interrupts */
    Hwi_enableInterrupt(5);
    Hwi_enableInterrupt(6);

    /* start BIOS */
    BIOS_start();
}

/* Runs when interrupt 5 occurs */
Void myIsr5(UArg arg)    {
    If (arg == 10) {
        Hwi5 = TRUE;
    }
}

/* Runs when interrupt 6 occurs */
Void myIsr6(UArg arg)    {
    If (arg == 12) {
        Hwi6 = TRUE;
    }
}

/* The Idle thread checks for completion of interrupts 5 & 6
 * and exits when they have both completed. */
Void myIdleFunc()
{
    If (Hwi5 && Hwi6) {
        System_printf("Both interrupts have occurred!");
        System_exit(0);
    }
}
```

7.2.6 The Interrupt Dispatcher

To consolidate code that performs register saving and restoration for each interrupt, SYS/BIOS provides an interrupt dispatcher that automatically performs these actions for an interrupt routine. Use of the Hwi dispatcher allows ISR functions to be written in C.

In addition to preserving the interrupted thread's context, the SYS/BIOS Hwi dispatcher orchestrates the following actions:

- Disables SYS/BIOS Swi and Task scheduling during interrupt processing

- Automatically manages nested interrupts on a per-interrupt basis.

- Invokes any configured "begin" Hwi Hook functions.

- Runs the Hwi function.

- Invokes any configured "end" Hwi Hook functions.

- Invokes Swi and Task schedulers after interrupt processing to perform any Swi and Task operations resulting from actions within the Hwi function.

On some platforms, such as the MSP430, the Hwi dispatcher is not provided. However, interrupt stubs are generated to provide essentially the same default functionality. However, the generated interrupt stubs do not automatically manage nested interrupts, because this is not supported on the MSP430.

Note: For hardware-specific information about using SYS/BIOS, see the links on the http://processors.wiki.ti.com/index.php/Category:SYSBIOS wiki page.

Note: The *interrupt* keyword or INTERRUPT pragma must not be used to define the C function invoked by the Hwi dispatcher (or interrupt stubs, on platforms for which the Hwi dispatcher is not provided, such as the MSP430). The Hwi dispatcher and the interrupt stubs contain this functionality, and the use of the C modifier will cause catastrophic results.

Functions that use the *interrupt* keyword or INTERRUPT pragma may not use the Hwi dispatcher or interrupt stubs and may not call SYS/BIOS APIs.

7.2.7 Registers Saved and Restored by the Interrupt Dispatcher

The registers saved and restored by the dispatcher in preparation for invoking the user's Hwi function conform to the "saved by caller" or "scratch" registers as defined in the register usage conventions section of the C compiler documents. For more information, either about which registers are saved and restored, or about the TMS320 functions conforming to the Texas Instruments C runtime model, see the *Optimizing Compiler User's Guide* for your platform.

7.2.8 Additional Target/Device-Specific Hwi Module Functionality

As described in Section 7.5, the ti.sysbios.hal.Hwi module is implemented using the proxy-delegate mechanism. All ti.sysbios.hal.Hwi module APIs are forwarded to a target/device-specific Hwi module that implements all of the ti.sysbios.hal.Hwi required APIs. Each of these Hwi module implementations provide additional APIs and functionality unique to the family/device and can be used instead of the ti.sysbios.hal.Hwi module if needed.

For example, the 'C64x+ target-specific Hwi module, ti.sysbios.family.c64p.Hwi, provides the following APIs in addition to those defined in the ti.sysbios.hal.Hwi module:

- `Hwi_eventMap(UInt intNum, UInt eventId);`
  Remaps a peripheral event number to an interrupt number.

- Bits16 Hwi_enableIER(Bits16 mask);
  Bits16 Hwi_disableIER(Bits16 mask);
  Bits16 Hwi_restoreIER(Bits16 mask);

  These three APIs allow enabling, disabling and restoring a set of interrupts defined by a "mask" argument. These APIs provide direct manipulation of the 'C64x+'s internal IER registers.

To gain access to these additional APIs, you use the target/device-specific Hwi module associated with the 'C64x+ target rather than the ti.sysbios.hal.Hwi module.

For documentation on the target/device-specific Hwi modules, see the CDOC documentation for ti.sysbios.family.*.Hwi. For example, ti.sysbios.family.c28.Hwi.

The following examples are modified versions of portions of the example in Section 7.2.5. The modifications are shown in **bold**.

Configuration example:

```
var Hwi = xdc.useModule('ti.sysbios.family.c64p.Hwi');

/* Initialize hwiParams to default values */
var hwiParams = new Hwi.Params;

/* Set myIsr5 parameters */
hwiParams.arg = 10;
hwiParams.enableInt = false;

/* Create a Hwi object for interrupt number 5
 * that invokes myIsr5() with argument 10 */
Hwi.create(5, '&myIsr5', hwiParams);
```

Runtime example:

```
#include <ti/sysbios/family/c64p/Hwi>
#include <xdc/runtime/Error.h>

main(Void)   {
    Hwi_Params hwiParams;
    Hwi_Handle myHwi;
    Error_Block eb;

    /* Initialize error block and hwiParams to default values */
    Error_init(&eb);
    Hwi_Params_init(&hwiParams);


    /* Set myIsr6 parameters */
    hwiParams.arg = 12;
    hwiParams.enableInt = FALSE;
```

```
    /* Create a Hwi object for interrupt number 6
     * that invokes myIsr6() with argument 12 */
    myHwi = Hwi_create(6, myIsr6, &hwiParms, &eb);
    if (myHwi == NULL) {
        System_abort("Hwi create failed");
    }


    /* Enable interrupts 5 & 6 simultaneously using the C64x+
     * Hwi module Hwi_enableIER() API. */
    Hwi_enableIER(0x0060);

. . .
```

7.3 Timer Module

The ti.sysbios.hal.Timer module presents a standard interface for using the timer peripherals. It hides any target/device-specific characteristics of the timer peripherals. It inherits the ti.sysbios.interfaces.ITimer interface.

You can use this module to create a timer (that is, to mark a timer for use) and configure it to call a tickFxn when the timer expires. Use this module only if you do not need to do any custom configuration of the timer peripheral.

This module has a configuration parameter called TimerProxy which is plugged by default with a target/device-specific implementation. For example, the implementation for C64x targets is ti.sysbios.family.c64.Timer.

The timer can be configured as a one-shot or a continuous mode timer. The period can be specified in timer counts or microseconds.

The timer interrupt always uses the Hwi dispatcher. The Timer tickFxn runs in the context of a Hwi thread. The Timer module automatically creates a Hwi instance for the timer interrupt.

The Timer_create() API takes a timerId. The timerId can range from zero to a target/device-specific value determined by the TimerProxy. The timerId is just a logical ID; its relationship to the actual timer peripheral is controlled by the TimerProxy.

If it does not matter to your program which timer is used, in a C program or XDCtools configuration you can specify a timerId of Timer_ANY which means "use any available timer". For example, in an XDCtools configuration use:

```
Timer.create(Timer_ANY, "&myIsr", timerParams);
```

In a C program, use:

```
myTimer = Timer_create(Timer_ANY, myIsr, &timerParams,
        &eb);
if (myTimer == NULL) {
    System_abort("Timer create failed");
}
```

The timerParams includes a number of parameters to configure the timer. For example timerParams.startMode can be set to StartMode_AUTO or StartMode_USER. The StartMode_AUTO setting indicates that statically-created timers will be started in BIOS_start() and dynamically-created timers will be started at create() time. The StartMode_USER indicates that your program starts the timer using Timer_start(). See the example in Section 7.3.1.

You can get the total number of timer peripherals by calling Timer_getNumTimers() at runtime. This includes both used and available timer peripherals. You can query the status of the timers by calling Timer_getStatus().

If you want to use a specific timer peripheral or want to use a custom timer configuration (setting timer output pins, emulation behavior, etc.), you should use the target/device-specific Timer module. For example, ti.sysbios.family.c64.Timer.

The Timer module also allows you to specify the extFreq (external frequency) property for the timer peripheral and provides an API to get the timer frequency at runtime. This external frequency property is supported only on targets where the timer frequency can be set separately from the CPU frequency.

You can use Timer_getFreq() to convert from timer interrupts to real time.

The Timer module provides APIs to start, stop, and modify the timer period at runtime. These APIs have the following side effects.

- Timer_setPeriod() stops the timer before setting the period register. It then restarts the timer.

- Timer_stop() stops the timer and disables the timer interrupt.

- Timer_start() clears counters, clears any pending interrupts, and enables the timer interrupt before starting the timer.

Runtime example: This C example creates a timer with a period of 10 microseconds. It passes an argument of 1 to the myIsr function. It instructs the Timer module to use any available timer peripheral:

```
Timer_Params timerParams;
Timer_Handle myTimer;
Error_Block eb;

Error_init(&eb);

Timer_Params_init(&timerParams);
timerParams.period = 10;
timerparams.periodType = Timer_PeriodType_MICROSECS;
timerParams.arg = 1;
myTimer = Timer_create(Timer_ANY, myIsr, &timerParams, &eb);
if (myTimer == NULL) {
    System_abort("Timer create failed");
}
```

Configuration example: This XDCtools example statically creates a timer with the same characteristics as the previous C example. It specifies a timerId of 1:

```
var timer = xdc.useModule('ti.sysbios.hal.Timer');
var timerParams = new Timer.Params();
timerParams.period = 10;
timerParams.periodType = Timer.PeriodType_MICROSECS;
timerParams.arg = 1;
timer.create(1, '&myIsr', timerParams);
```

Runtime example: This C example sets a frequency for a timer it creates. The extFreq.hi and extFreq.lo properties set the high and low 32-bit portions of the structure used to represent the frequency in Hz.

```
Timer_Params timerParams;
Timer_Handle myTimer;
Error_Block eb;

Error_init(&eb);
Timer_Params_init(&timerParams);
timerParams.extFreq.lo = 270000000; /* 27 MHz */
timerParams.extFreq.hi = 0;

myTimer = Timer_create(Timer_ANY, myIsr, &timerParams, &eb);
if (myTimer == NULL) {
    System_abort("Timer create failed");
}
```

Configuration example: This XDCtools configuration example specifies a frequency for a timer that it creates.

```
var Timer = xdc.useModule('ti.sysbios.hal.Timer');
var timerParams = new Timer.Params();
timerParams.extFreq.lo = 270000000;
timerParams.extFreq.hi = 0;
...
Timer.create(1, '&myIsr', timerParams);
```

Runtime example: This C example creates a timer that runs the tickFxn() every 2 milliseconds using any available timer peripheral. It also creates a task that, when certain conditions occur, changes the timer's period from 2 to 4 milliseconds. The tickFxn() itself prints a message that shows the current period of the timer.

```
Timer_Handle timerHandle;

Int main(Void)
{
    Error_Block eb;
    Timer_Params timerParams;
    Task_Handle taskHandle;
```

```
    Error_init(&eb);
    Timer_Params_init(&timerParams);
    timerParams.period = 2000;    /* 2 ms */
    timerHandle = Timer_create(Timer_ANY, tickFxn, &timerParams, &eb);
    if (timerHandle == NULL) {
        System_abort("Timer create failed");
    }

    taskHandle = Task_create(masterTask, NULL, &eb);
    if (taskHandle == NULL) {
        System_abort("Task create failed");
    }
}


Void masterTask(UArg arg0 UArg arg1)
{
    ...
    // Condition detected requiring a change to timer period
    Timer_stop(timerHandle);
    Timer_setPeriodMicroSecs(4000);   /* change 2ms to 4ms */
    Timer_start(timerHandle();
    ...
}


Void tickFxn(UArg arg0 UArg arg1)
{
    System_printf("Current period = %d\n",
            Timer_getPeriod(timerHandle);
}
```

7.3.1 Target/Device-Specific Timer Modules

As described in Section 7.5, the ti.sysbios.hal.Timer module is implemented using the proxy-delegate mechanism. A separate target/device-specific Timer module is provided for each supported family. For example, the ti.sysbios.timers.timer64.Timer module acts as the timer peripherals manager for the 64P family.

These target/device-specific modules provide additional configuration parameters and APIs that are not supported by the generic ti.sysbios.hal.Timer module.

In the case of the ti.sysbios.timers.timer64.Timer module, the configuration parameters controlInit, globalControlInit, and emuMgtInit are provided to configure various timer properties. This module also exposes a Hwi Params structure as part of its create parameters to allow you to configure the Hwi object associated with the Timer. This module also exposes a Timer_reconfig() API to allow you to reconfigure a statically-created timer.

Configuration example: This XDCtools configuration example specifies timer parameters, including target/device-specific parameters for a timer called myTimer that it creates.

```
var Timer = xdc.useModule('ti.sysbios.timers.timer64.Timer');

var timerParams = new Timer.Params();
timerParams.period = 2000;      //2ms
timerParams.arg = 1;
timerParams.startMode = Timer.StartMode_USER;
timerParams.controlInit.invout = 1;
timerParams.globalControlInit.chained = false;
timerParams.emuMgtInit.free = false;
timerParams.suspSrc = SuspSrc_ARM;

Program.global.myTimer = Timer.create(1, "&myIsr", timerParams);
```

Runtime example: This C example uses the myTimer created in the preceding XDCtools configuration example and reconfigures the timer with a different function argument and startMode in the program's main() function before calling BIOS_start().

```
#include <ti/sysbios/timers/timer64/Timer.h>
#include <xdc/cfg/global.h>
#include <xdc/runtime/Error.h>

Void myIsr(UArg arg)
{
    System_printf("myIsr arg = %d\n", (Int)arg);
    System_exit(0);
}

Int main(Int argc, char* argv[])
{
    Timer_Params timerParams;
    Error_Block eb;

    Error_init(&eb);
    Timer_Params_init(&timerParams);
    timerParams.arg = 2;
    timerparams.startMode = Timer_StartMode_AUTO;
    Timer_reconfig(myTimer, tickFxn, &timerParams, &eb);
    if (Error_check(&eb)) {
        System_abort("Timer reconfigure failed");
    }

    BIOS_start();

    return(0);
}
```

7.4   Cache Module

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

- **Invalidate.** Makes valid cache lines invalid and discards the content of the affected cache lines.

- **Writeback.** Writes the contents of cache lines to a lower-level memory, such as the L2 cache or external memory, without discarding the lines in the original cache.

- **Writeback-Invalidation.** Writes the contents of cache lines to lower-level memory, and then discards the contents of the lines in the original cache.

7.4.1   Cache Interface Functions

The cache interface is defined in ti.sysbios.interfaces.ICache. The Cache interface contains the following functions. The implementations for these functions are target/device-specific.

- **Cache_enable();** Enables all caches.

- **Cache_disable();** Disables all caches.

- **Cache_inv(blockPtr, byteCnt, type, wait);** Invalidates the specified range of memory. When you invalidate a cache line, its contents are discarded and the cache tags the line as "clean" so that next time that particular address is read, it is obtained from external memory. All lines in the range are invalidated in all caches.

- **Cache_wb(blockPtr, byteCnt, type, wait);** Writes back the specified range of memory. When you perform a writeback, the contents of the cache lines are written to lower-level memory. All lines within the range are left valid in caches and the data within the range is written back to the source memory.

- **Cache_wbInv(blockPtr, byteCnt, type, wait);** Writes back and invalidates the specified range of memory. When you perform a writeback, the contents of the cache lines are written to lower-level memory. When you invalidate a cache line, its contents are discarded. All lines within the range are written back to the source memory and then invalidated in all caches.

These Cache APIs operate on an address range beginning with the starting address of blockPtr and extending for the specified byte count. The range of addresses operated on is quantized to whole cache lines in each cache.

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

If the byteCnt is not equal to a whole number of cache lines, the byteCnt is rounded up to the next size that equals a whole number of cache lines.

The type parameter is a bit mask of the Cache_Type type, which allows you to specify one or more caches in which the action should be performed.

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

- **Cache_wait();** Waits for the cache wb/wbInv/inv operation to complete. A cache operation is not truly complete until it has worked its way through all buffering and all memory writes have landed in the source memory.

As described in Section 7.5, this module is implemented using the proxy-delegate mechanism. A separate target/device-specific Cache module is provided for each supported family.

Additional APIs are added to this module for certain target/device-specific implementations. For example, the ti.sysbios.family.c64p.Cache module adds APIs specific to the C64x+ caches. These extensions have functions that also have the prefix "Cache_".

Currently the C64x+, C674x, and ARM caches are supported.

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

7.5 HAL Package Organization

The three SYS/BIOS modules that reside in the ti.sysbios.hal package: Hwi, Timer, and Cache require target/device-specific API implementations to achieve their functionality. In order to provide a common set of APIs for these modules across all supported families/devices, SYS/BIOS uses the proxy-delegate module mechanism. (See the "RTSC Interface Primer: Lesson 12" for details.)

Each of these three modules serves as a proxy for a corresponding target/device-specific module implementation. In use, all Timer/Hwi/Cache API invocations are forwarded to an appropriate target/device-specific module implementation.

During the configuration step of the application build process, the proxy modules in the ti.sysbios.hal package locate and bind themselves to appropriate delegate module implementations based on the current target and platform specified in the user's config.bld file. The delegate binding process is done internally.

| Note: | For hardware-specific information about using SYS/BIOS, see the links on the http://processors.wiki.ti.com/index.php/Category:SYSBIOS wiki page. |

The following tables show currently supported (as of this document's publication) Timer, Hwi, and Cache delegate modules that may be selected based on an application's target and device. The mapping of target/device to the delegate modules used by Timer, Cache, and Hwi is accessible through a link in the ti.sysbios.hal package online help.

Table 7–1. Proxy to Delegate Mappings

Proxy Module	Delegate Modules
ti.sysbios.hal.Timer	ti.sysbios.hal.TimerNull * ti.sysbios.timers.dmtimer.Timer ti.sysbios.timers.gptimer.Timer ti.sysbios.timers.timer64.Timer ti.sysbios.family.c28.Timer ti.sysbios.family.c67p.Timer ti.sysbios.family.msp430.Timer ti.sysbios.family.arm.*<various>*.Timer

Table 7–1. Proxy to Delegate Mappings

Proxy Module	Delegate Modules
ti.sysbios.hal.Hwi	ti.sysbios.family.c28.Hwi ti.sysbios.family.c64p.Hwi ti.sysbios.family.c67p.Hwi ti.sysbios.family.msp430.Hwi ti.sysbios.family.arm.*<various>*.Hwi
ti.sysbios.hal.Cache	ti.sysbios.hal.CacheNull * ti.sysbios.family.c64p.Cache ti.sysbios.family.c67p.Cache ti.sysbios.family.arm.Cache

* For targets/devices for which a Timer or Cache module has not yet been developed, the hal.TimerNull or hal.CacheNull delegate is used. In TimerNull/CacheNull, the APIs defined in ITimer/ICache are implemented using null functions.

For the proxy-delegate mechanism to work properly, both the proxy and the delegate modules must be implementations of a common interface specification. The Timer, Hwi, and Cache interface specifications reside in ti.sysbios.interfaces and are ITimer, IHwi, and ICache respectively. These interface specifications define a minimum set of general APIs that, it is believed, will satisfy a vast majority of application requirements. For those applications that may need target/device-specific functionality not defined in these interface specifications, the corresponding Timer, Hwi, and Cache delegate modules contain extensions to the APIs defined in the interface specifications.

To access to these extended API sets, you must directly reference the target/device-specific module in your configuration file and include its corresponding header file in your C source files.

Instrumentation

This chapter describes modules and other tools that can be used for instrumentation purposes.

8.1 Overview of Instrumentation

Much of the instrumentation available to SYS/BIOS applications is provided by the XDCtools modules and APIs. See the XDCtools documentation for details about the Assert, Diags, Error, Log, LoggerBuf, and LoggerSys modules.

8.2 Load Module

The ti.sysbios.utils.Load module reports execution times and load information for threads in a system.

SYS/BIOS manages four distinct levels of execution threads: hardware interrupt service routines, software interrupt routines, tasks, and background idle functions. The Load module reports execution time and load on a per-task basis, and also provides information globally for hardware interrupt service routines, software interrupt routines, and idle functions (in the form of the idle task). It can also report an estimate of the global CPU load, which is computed as the percentage of time in the measurement window that was *not* spent in the idle loop. More specifically, the load is computed as follows.

```
global CPU load = 100 * (1 - ((x * t) / w) )
```

where:

- 'x' is the number of times the idle loop has been executed during the measurement window.

- 't' is the minimum time for a trip around the idle loop, meaning the time it takes to complete the idle loop if no work is being done in it.

- 'w' is the length in time of the measurement window.

Any work done in the idle loop is included in the CPU load. In other words, any time spent in the loop beyond the shortest trip around the idle loop is counted as non-idle time.

The Load module relies on "update" to be called to compute load and execution times from the time when "update" was last called. This is automatically done for every period specified by Load.windowInMs (default = 500 ms) in a ti.sysbios.knl.Idle function when Load.updateInIdle is set to true (the default). The benchmark time window is the length of time between 2 calls to "update".

The execution time is reported in units of xdc.runtime.Timestamp counts, and the load is reported in percentages.

By default, load data is gathered for all threads. You can use the configuration parameters Load.hwiEnabled, Load.swiEnabled, and Load.taskEnabled to select which type(s) of threads are monitored.

8.2.1 *Load Module Configuration*

The Load module has been setup to provide data with as little configuration as possible. Using the default configuration, load data is gathered and logged for all threads roughly every 500 ms.

The following code configures the Load module to write Load statistics to a LoggerBuf instance.

```
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
var Load = xdc.useModule('ti.sysbios.utils.Load');
var Diags = xdc.useModule('xdc.runtime.Diags');

var loggerBuf = LoggerBuf.create();
Load.common$.logger = loggerBuf;
Load.common$.diags_USER4 = Diags.ALWAYS_ON;
```

For information on advanced configuration and caveats of the Load module, see the online reference documentation.

8.2.2 *Obtaining Load Statistics*

Load statistics recorded by the Load module can be obtained in one of two ways:

- **Load module logger.** If you configure the Load module with a logger and have turned on the diags_USER4, the statistics gathered by the Load module are recorded to the load module's logger instance. You can use the RTOS Analyzer tools to visualize the Load based on these Log records. See the System Analyzer User's Guide (SPRUH43) and the System Analyzer wiki page for more information.

  Alternatively, you can configure the logger to print the logs to the console. The global CPU load log prints a percentage. For example:

  ```
  LS_cpuLoad: 10
  ```

  The global Swi and Hwi load logs print two numbers: the time in the thread, and the length of the measurement window. For example:

  ```
  LS_hwiLoad: 13845300,158370213
  LS_swiLoad: 11963546,158370213
  ```

  These evaluate to loads of 8.7% and 7.6%.

  The Task load log uses the same format, with the addition of the Task handle address as the first argument. For example:

  ```
  LS_taskLoad: 0x11802830,56553702,158370213
  ```

  This evaluates to a load of 35.7%.

- **Runtime APIs.** You can also choose to call Load_getTaskLoad(), Load_getGlobalSwiLoad(), Load_getGlobalHwiLoad() or Load_getCPULoad() at any time to obtain the statistics at runtime.

  The Load_getCPULoad() API returns an actual percentage load, whereas Load_getTaskLoad(), Load_getGlobalSwiLoad(), and Load_getGlobalHwiLoad() return a Load_Stat structure. This structure contains two fields, the length of time in the thread, and the length of time in the

measurement window. The load percentage can be calculated by dividing these two numbers and multiplying by 100%. However, the Load module also provides a convenience function, Load_calculateLoad(), for this purpose. For example, the following code retrieves the Hwi Load:

```
Load_Stat stat;
UInt32 hwiLoad;

Load_getGlobalHwiLoad(&stat);
hwiLoad = Load_calculateLoad(&stat);
```

8.3 Error Handling

A number of SYS/BIOS APIs—particularly those that create objects and allocate memory—have an argument that expects an Error_Block. This type is defined by the xdc.runtime.Error module provided by XDCtools. The following example shows the recommended way to declare and use an error block when creating a Swi:

```
#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Swi.h>

Swi_Handle swi0;
Swi_Params swiParams;
Error_Block eb;

Error_init(&eb);
Swi_Params_init(&swiParams);

swi0 = Swi_create(swiFunc, &swiParams, &eb);
if (swi0 == NULL) {
    System_abort("Swi create failed");
}
```

Notice that in the previous example, the test to determine whether to call System_abort() compares the value returned by Swi_create() to NULL as follows:

```
if (swi0 == NULL) {
    System_abort("Swi create failed");
}
```

Most of the SYS/BIOS APIs that expect an error block also return a handle to the created object or the allocated memory. For this reason, it is simplest and provides the best performance to check the value returned by these APIs.

For APIs that get passed an Error_Block but do not return a handle or other status value, you can use the Error_check() function to test the Error_Block as in the following example, which calls Timer_reconfig() to reconfigure a statically-created Timer object:

```
Timer_reconfig(myTimer, tickFxn, &timerParams, &eb);
if (Error_check(&eb)) {
    System_abort("Timer reconfigure failed");
}
```

You may see examples in other documentation that pass NULL in place of the Error_Block argument. If an error occurs when creating an object or allocating memory and NULL was passed instead of an Error_Block, the application aborts and a reason for the error is output using System_printf(). This may be the best behavior in systems where any error is fatal, and you do not want to do any error checking.

The advantage to passing and testing an Error_Block is that your program can have control over when it aborts. For example, instead of aborting when memory cannot be allocated, you might want to try to release other memory and try again or switch to a mode with more limited memory needs.

Note that the System_abort() function calls a hook function from its System.SupportProxy module (for example, xdc.runtime.SysStd). You might want to use a different abort() hook function that performs a reset and starts the application over.

8.4 Instrumentation Tools in Code Composer Studio

The RTOS Object Viewer (ROV) is a stop-mode debugging tool provided by XDCtools. For information, see the RTSC-pedia page on ROV at http://rtsc.eclipse.org/docs-tip/RTSC_Object_Viewer.

The System Analyzer tools, which are available from the **Tools** menu, allow you to access instrumentation data within CCS. This tool suite includes the target-side UIA package for instrumenting applications. Once you enable the UIA package, SYS/BIOS thread execution and loads are instrumented by default. You can add additional UIA events to benchmark code and gather additional data. Tools are provided in CCS to view this data in tables and graphs. Multicore data correlation is also provided as a part of System Analyzer. For details about using System Analyzer, see the System Analyzer User's Guide (SPRUH43) and the System Analyzer wiki page.

By default, UIA is disabled in SYS/BIOS examples. To enable the use of UIA instrumentation in a SYS/BIOS application, follow these steps:

1. In the Project Explorer pane of CCS, right-click on the project and choose **Properties**.

2. In the Properties dialog, choose the **General** category, then the **RTSC** tab.

3. In the Products and Repositories area, check the box next to **System Analyzer (UIA Target)** and make sure the most recent version is selected. This causes your application to link with the necessary parts of UIA and makes the UIA modules available within XGCONF.

4. Click **OK**.
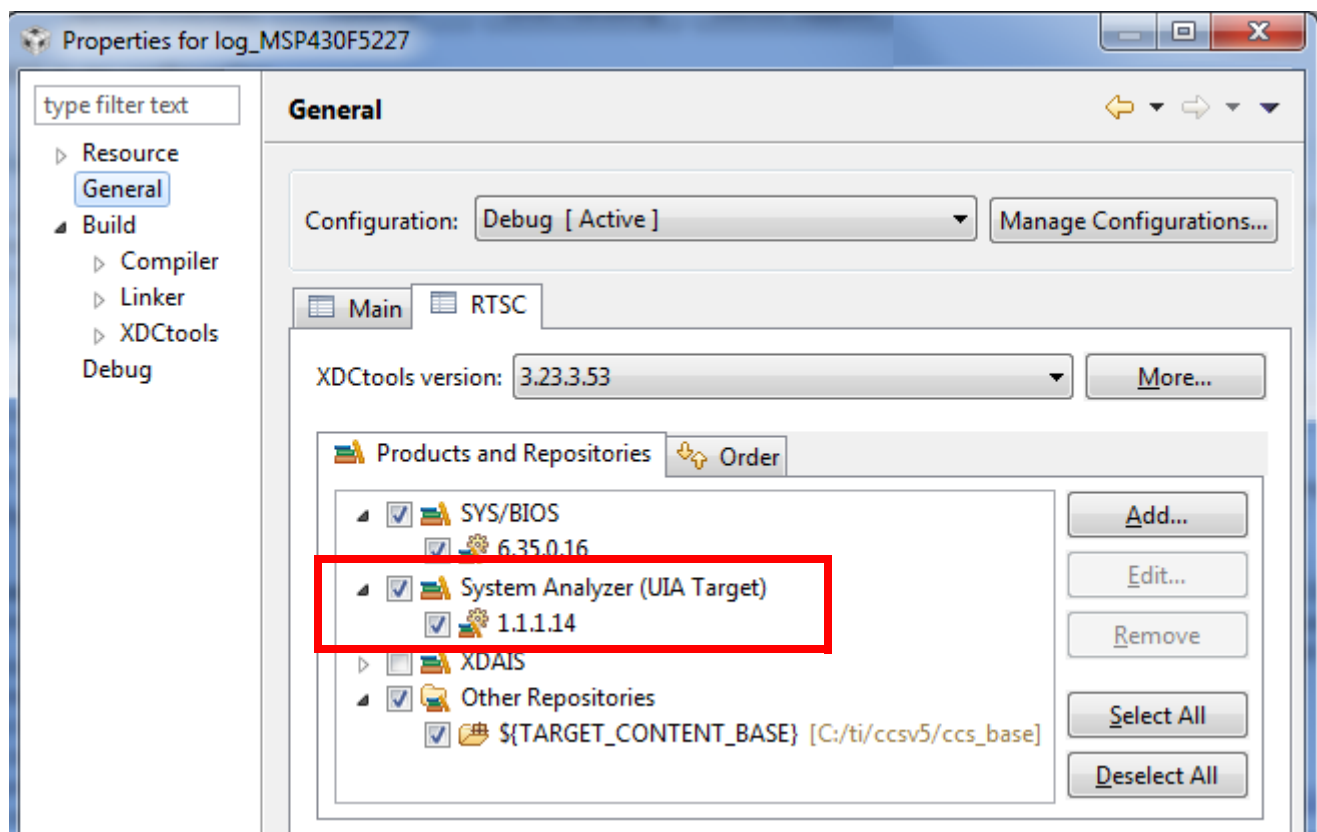
UIA runtime data can be gathered via Ethernet for any target that has an NDK driver. Runtime data collection via JTAG is also provided for all C64x+ and C66x targets. Stop-mode data collection is provided for the additional targets listed in the UIA release notes.

8.5 Performance Optimization

This section provides suggestions for optimizing the performance of SYS/BIOS applications. This is accomplished in two ways: by using compiler and linker optimizations, and by optimizing the configuration of SYS/BIOS.

8.5.1 Configuring Logging

Logging can significantly impact the performance of a system. You can reduce the impact of logging by optimizing the configuration. There are two main ways to optimize the logging used in your application:

- **No logging.** In SYS/BIOS, logging is not enabled by default.

- **Optimizing logging.** If you need some logging enabled in your application, there are some configuration choices you can make to optimize performance. These are described in the following subsections.

8.5.1.1 Diags Settings

There are four diagnostics settings for each diagnostics level: RUNTIME_OFF, RUNTIME_ON, ALWAYS_OFF, and ALWAYS_ON.

The two runtime settings (RUNTIME_OFF and RUNTIME_ON) allow you to enable or disable a particular diagnostics level at runtime. However, a check must be performed to determine whether logging is enabled or disabled every time an event is logged.

If you use ALWAYS_OFF or ALWAYS_ON instead, you cannot change the setting at runtime. The call will either be a direct call to log the event (ALWAYS_ON) or will be optimized out of the code (ALWAYS_OFF).

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');

/* 'RUNTIME' settings allow you to turn it off or on at runtime,
 * but require a check at runtime. */
Defaults.common$.diags_USER1 = Diags.RUNTIME_ON;
Defaults.common$.diags_USER2 = Diags.RUNTIME_OFF;

/* These settings cannot be changed at runtime, but optimize out
 * the check for better performance. */
Defaults.common$.diags_USER3 = Diags.ALWAYS_OFF;
Defaults.common$.diags_USER4 = Diags.ALWAYS_ON;
```

8.5.1.2 Choosing Diagnostics Levels

SYS/BIOS modules only log to two levels: USER1 and USER2. They follow the convention that USER1 is for basic events and USER2 is for more detail.

To improve performance, you could only turn on USER1, or turn on USER2 for particular modules only.

Refer to each module's documentation to see which events are logged as USER1 and which are logged as USER2.

8.5.1.3 Choosing Modules to Log

To optimize logging, enable logging only for modules that interest you for debugging.

For example, Hwi logging tends to be the most expensive in terms of performance due to the frequency of hardware interrupts. Two Hwi events are logged on every Clock tick when the Clock's timer expires.

8.5.2 Configuring Diagnostics

By default, ASSERTS are enabled for all modules. SYS/BIOS uses asserts to check for common user mistakes such as calling an API with an invalid argument or from an unsupported context. Asserts are useful for catching coding mistakes that may otherwise lead to confusing bugs.

To optimize performance after you have done basic debugging of API calls, your configuration file can disable asserts as follows:

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');

/* Disable asserts in all modules. */
Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;
```

8.5.3 Choosing a Heap Manager

SYS/BIOS provides three different heap manager implementations: HeapMem, HeapBuf, and HeapMultiBuf. Each of these has various performance trade-offs when allocating and freeing memory.

HeapMem can allocate a block of any size, but is the slowest of the three. HeapBuf can only allocate blocks of a single configured size, but is very quick. HeapMultiBuf manages a pool of HeapBuf instances and balances the advantages of the other two. HeapMultiBuf is quicker than HeapMem, but slower than HeapMem.

See the documentation for each of these modules for a detailed discussion of the trade-offs of each module.

Consider also using different heap implementations for different roles. For example, HeapBuf is ideally suited for allocating a fixed-size object that is frequently created and deleted. If you were creating and deleting many Task instances, you could create a HeapBuf instance just for allocating Tasks.

8.5.4 Hwi Configuration

The hardware interrupt dispatcher provides a number of features by default that add to interrupt latency. If your application does not require some of these features, you can disable them to reduce interrupt latency.

- **dispatcherAutoNestingSupport.** You may disable this feature if you don't need interrupts enabled during the execution of your Hwi functions.

- **dispatcherSwiSupport.** You may disable this feature if no Swi threads will be posted from any Hwi threads.

- **dispatcherTaskSupport.** You may disable this feature if no APIs are called from Hwi threads that would lead to a Task being scheduled. For example, Semaphore_post() would lead to a Task being scheduled.

- **dispatcherIrpTrackingSupport.** This feature supports the Hwi_getIrp() API, which returns an interrupt's most recent return address. You can disable this feature if your application does not use that API.

8.5.5 Stack Checking

By default, the Task module checks to see whether a Task stack has overflowed at each Task switch. To improve Task switching latency, you can disable this feature the Task.checkStackFlag property to false.

Input/Output

This chapter describes modules that can be used to handle input and output data.

9.1 Overview

This chapter describes how to use the GIO (General Purpose I/O Manager) module (ti.sysbios.io.GIO) and drivers written to implement the IOM interface for input and output. You use the GIO module to send data to an output channel or receive data from an input channel.

The GIO module is an abstraction of device drivers. It allows you to develop applications that do not directly interact with driver code. This allows the application to be developed in a driver-independent and platform-independent manner.

The GIO module also allows your application to process one buffer while the driver is working on another buffer. This improves performance and minimizes the copying of data. The typical use case for the GIO module is that an application gets a buffer of data from an input channel, processes the data and sends the data to an output channel.

The following figure shows a typical flow of data for an output channel:
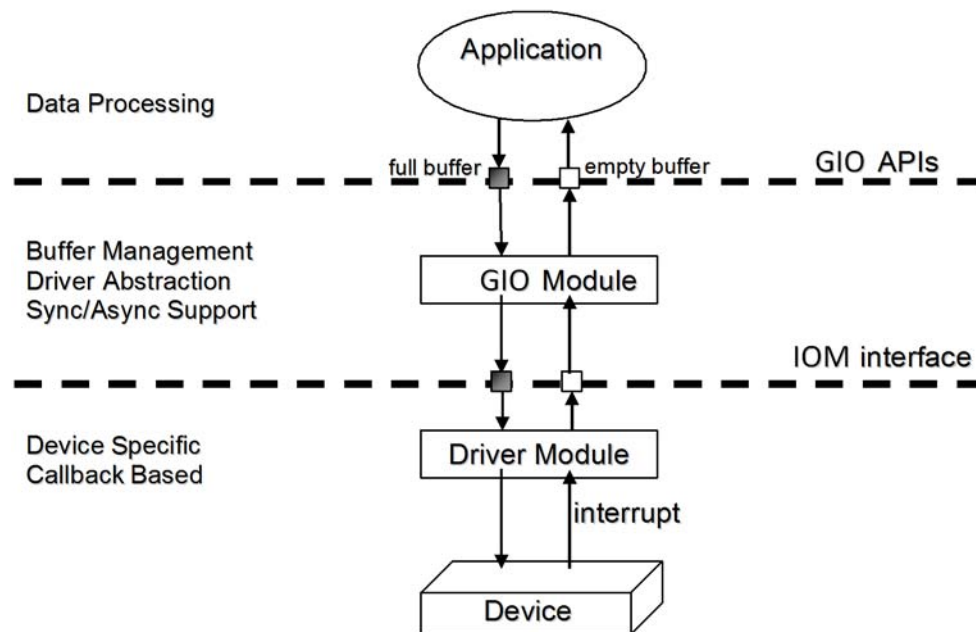
In this use case, the application calls GIO module APIs. Those APIs manage buffers and the driver abstraction. Internally, the GIO module APIs calls the implementation of the IOM interface for the driver. As this figure shows, the high-level application does not interact with the drivers directly. Instead, the application uses the GIO module to interface with the drivers.

IOM drivers implement the IOM interface provided in <bios_install>/packages/ti/sysbios/io/IOM.h to manage a peripheral. Such drivers are provided in some certain other software components (for example, the PSP and SDK products), or you can write your own. See Appendix E for details about the IOM interface. IOM drivers are sometimes called "mini-drivers".

The are two types of objects that applications must create in order to use GIO channels and IOM drivers:

- **DEV table:** The DEV module maintains a table of IOM driver names for use by the GIO module. You can manage this table statically using the XGCONF configuration tool or at runtime using C APIs. See Section 9.2.

- **GIO channels:** The GIO module manages GIO objects that correspond to input and output channels. You can create GIO channels using C APIs, but they cannot be created statically. See Section 9.3.2.

The GIO module supports both synchronous (blocking) APIs and asynchronous (non-blocking) APIs:

- **Synchronous APIs** can yield to other threads while waiting for a buffer to be ready or for the timeout to occur. GIO calls this model GIO_Model_STANDARD. APIs can only be used in the context of Task threads, because Hwi and Swi threads cannot use blocking calls. The main GIO APIs used with the synchronous model are GIO_read and GIO_write. In addition, drivers typically implement GIO_submit, GIO_flush, GIO_abort, and GIO_control as synchronous APIs.

- **Asynchronous APIs** cannot yield to other threads. If the buffer is not ready, these functions return a failure status. GIO calls this model GIO_Model_ISSUERECLAIM. APIs can be used in any thread context. The main GIO APIs used with the asynchronous model are GIO_issue, GIO_reclaim, and GIO_prime.

9.2   Configuring Drivers in the Device Table

The DEV module manages a table of IOM drivers. This a table of devices that can be found by name, allowing GIO channels to be created using the name. To configure the driver table in XGCONF, follow these steps:

1. Open your application's *.cfg file with XGCONF.

2. In the Available Products area, select the **SYS/BIOS > I/O > DEV** module.

3. In the Device Manager -- Module Settings page, check the box to **Add DEV to my configuration**. You can change the size of the device table to the maximum number of devices you will create both statically and dynamically.
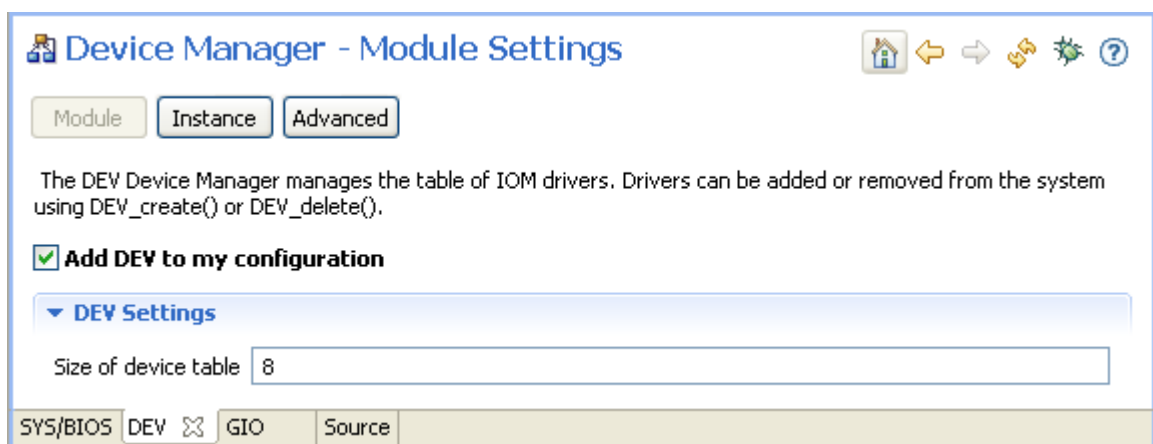
4. Click the **Instance** button near the top of the Device Manager page.

5. Click the **Add** button next to the DEVs list. Type a name for the driver. This is the name that you will use when creating GIO channels, either with XGCONF or the GIO_create() API.

6. Set the required and additional settings for the selected DEV driver. The information to specify should be provided in the documentation for the IOM driver you are using. See the CDOC online help for the DEV module for information about the properties.
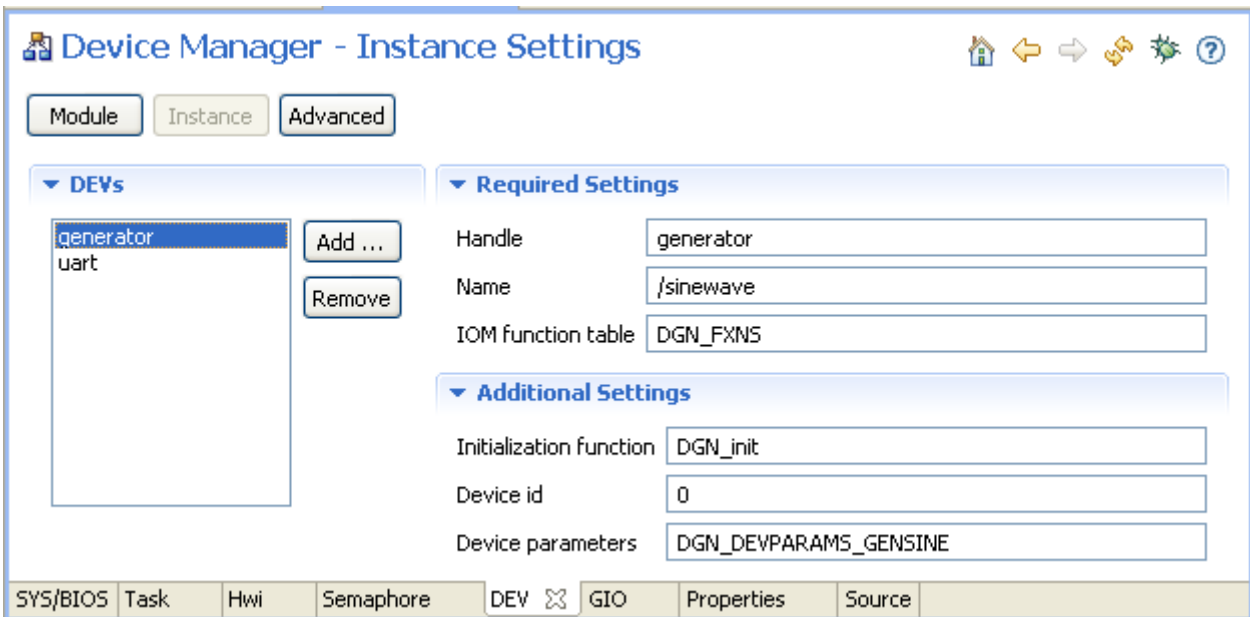
7. Create an item for each IOM driver your application will use. You can create multiple GIO channels for a single driver (if the driver supports it). So, for example, you do not need to specify two separate drivers if the application will use an IOM driver on two channels.

If you prefer to edit *.cfg files using the Source tab or a text editor, the following statements create the "generator" driver in the DEV table as shown in the previous figure:

```
var DEV = xdc.useModule('ti.sysbios.io.DEV');

...

DEV.tableSize = 4;

var dev0Params = new DEV.Params();
dev0Params.instance.name = "generator";
dev0Params.initFxn = "&DGN_init";
dev0Params.deviceParams = "&DGN_DEVPARAMS_GENSINE";
Program.global.generator = DEV.create("/sinewave", "&DGN_FXNS", dev0Params);
```

The runtime APIs for managing the table maintained by the DEV module are:

- DEV_create()
- DEV_delete()

For example, the following C code fragment creates the same "generator" driver in the DEV table at runtime:

```
#include <ti/sysbios/io/DEV.h>

DEV_Params params;
Error_Block eb;

DEV_Params_init(&params);
Error_init(&eb);

DEV_create("/sinewave", &DGN_FXNS, &params, &eb);
```

If you use a variable for the name of the driver (instead of a hardcoded string as used in the previous example), that variable must remain defined with the driver name as long as the DEV instance exists.

The DEV module provides additional APIs for matching a device name to a driver handle and for getting various information from the DEV table. These APIs are using internally by the GIO module; your application will probably not need to use them. Details are available in the CDOC online help system.

9.2.1 *Configuring the GIO Module*

The GIO module manages a set of channel instances. You can create these instances only with C APIs; they cannot be created statically. However, you do need to enable the GIO module in the configuration by following these steps:

1. Open your application's *.cfg file with XGCONF.

2. In the Available Products area, select the **SYS/BIOS > I/O > GIO** module.

3. In the General Purpose I/O Manager -- Module Settings page, check the box to **Add GIO module to my configuration**.
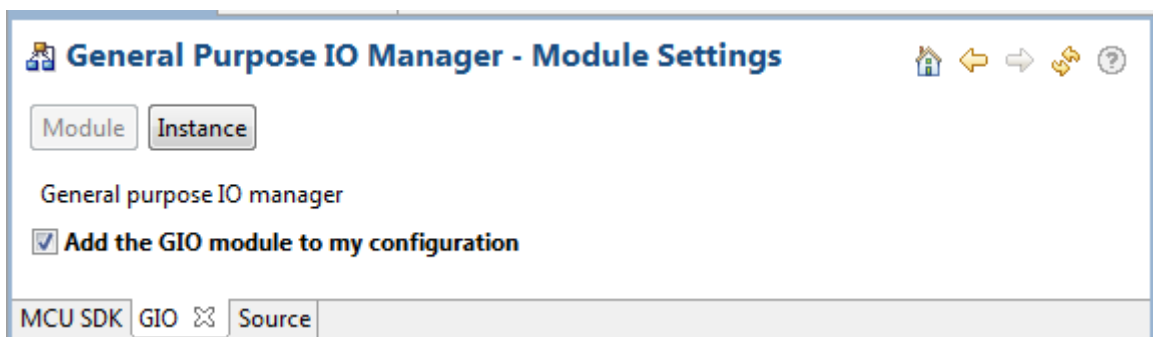
If you prefer to edit *.cfg files using the Source tab or a text editor, the following statement enables the GIO module:

```
var GIO = xdc.useModule('ti.sysbios.io.GIO');
```

For the runtime C APIs used to create and delete GIO channels, see Section 9.3.2.

9.3 Using GIO APIs

This section describes how to use the GIO module as an abstraction for drivers after you have associated a driver handle with a driver name as described in the previous section. The ti.sysbios.io.GIO module provides the following API functions to create and delete devices and channels:

API Function	Description
GIO_create()	Allocate and initialize a GIO_Obj object and open a communication channel.
GIO_delete()	Finalize and free a previously-allocated GIO_Obj instance.
GIO_Params_init	Initialize a config-params structure with supplied values before creating an instance.

The ti.sysbios.io.GIO module provides the following API functions to control sending data over channels and to control the behavior of channels:

API Function	Description	Standard Model	Issue/Reclaim Model
GIO_abort()	Abort all pending I/O.	X	X
GIO_control()	Send a control command to the driver.	X	X
GIO_flush()	Drain output buffers and discard any pending input.	X	
GIO_issue()	Issue a buffer to the channel.		X
GIO_prime()	Prime an output channel instance.		X
GIO_read()	Synchronously read from a GIO instance.	X	
GIO_reclaim()	Reclaim a buffer that was previously issued by calling issue		X
GIO_submit()	Submit an I/O job to the mini-driver.	X	
GIO_write()	Synchronously write to a GIO instance.	X	

The Standard model is described in Section 9.3.3, and the Issue/Reclaim model is described in Section 9.3.4.

9.3.1 Constraints When Using GIO APIs

- A GIO instance can be used only by a single Task.

- GIO_issue() and GIO_reclaim() can only be called by a single thread (Task or Swi) or in the callback context.

- GIO_issue(), GIO_reclaim(), GIO_read(), and GIO_write() cannot be called during Module startup. Some drivers may require that these APIs not be called even from main() if they require hardware interrupts to enable their peripherals.

9.3.2   *Creating and Deleting GIO Channels*

The GIO module manages GIO instances, which correspond to communication channels. Such channels are assigned a mode when you create them; they can be input, output, or in/out channels. You also specify the device driver a channel should use when you create it.

GIO channels can only be created using C APIs. They cannot be created statically with XGCONF.

Several GIO instances can be created to use the same driver instance. If the driver instance supports two channels, then two GIO instances can be created using the same driver.

The GIO_create() API creates a GIO instance. It has the following parameters:

```
GIO_Handle GIO_create(
    String          name,
    UInt            mode,
    const GIO_Params  *params,
    Error_Block     *eb)
```

The GIO_Params structure allows you to specify the following parameters:

```
struct GIO_Params {
    IInstance_Params    *instance;
    Ptr                 chanParams;
    GIO_Model           model;
    Int                 numPackets;
    ISync_Handle        sync;
    Uint                timeout;
};
```

The driver **name** and **mode** are required parameters for creating a GIO instance. The driver name must match a string that is present in the driver table. See Section 9.2.

The **mode** must be GIO_INPUT, GIO_OUPUT, or GIO_INOUT. Note that some drivers only support certain modes. See the documentation for your driver for details.

GIO_Params allow you to set optional parameters.

- The "instance" field allows you to specify a name for the GIO instance. The name will show up in the the ROV debugging tool.

- The "chanParams" is a pointer to driver-specific parameters used to configure the driver. These parameters are sent to the driver in the driver's open() call. Typically drivers have default values for such parameters.

- The "model" field can be set to GIO_Model_STANDARD or GIO_Model_ISSUERECLAIM. The Standard model is the default and the simplest option to use; it is used with synchronous GIO APIs that can block while waiting for a buffer, and so can only be called in the context of a Task thread. The IssueReclaim model must be used with asynchronous GIO APIs that cannot block, but can be called in the context of Hwi, Swi, and Task threads.

- The "numPackets" field specifies the number of packets that can be used if you are using the GIO_Model_ISSUERECLAIM asynchronous model. This is 2 by default. If you are using the GIO_Model_STANDARD model and the GIO_read()/GIO_write() functions, you should set this parameter to 1 to save the memory used by the extra packet.

- The "sync" field selects a synchronization mechanism to use if the GIO_Model_ISSUERECLAIM model is selected. GIO uses the xdc.runtime.knl.Sync module for synchronization when the GIO channel waits for the driver. If you pass NULL for this field, Sync uses a Semaphore for synchronization. Most applications will not need to change this default. See Section 9.5 for more on synchronization mechanisms.

- The "timeout" field specifies how long to wait (in Clock module ticks) for a buffer to be ready when a blocking call is made. This field is used only with the GIO_Model_STANDARD model. The default is BIOS_WAIT_FOREVER.

GIO_create() also takes an **Error_Block** and can fail. GIO_create() returns a GIO_Handle that is passed to many other GIO APIs to send or receive data.

This example creates two GIO channels that will use the Issue/Reclaim model:

```
#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <xdc/cfg/global.h>
#include <ti/sysbios/io/GIO.h>

GIO_Handle handleIn, handleOut;
Error_Block eb;
Error_init(&eb);


/*
 * ======== main ========
 */
Int main(Int argc, Char* argv[])
{
    GIO_Params gioParams;


    /* Create input GIO instance */
    GIO_Params_init(&gioParams);
    gioParams.model = GIO_Model_ISSUERECLAIM;
    handleIn = GIO_create("sampleDriver", GIO_INPUT, &gioParams, &eb);


    /* Create output GIO instance */
    gioParams.timeout = 5000;
    handleOut = GIO_create("sampleDriver", GIO_OUTPUT, &gioParams, &eb);


    BIOS_start();
    return(0);
}
```

Important: A GIO_Handle cannot be used by more than one Task simultaneously. One Task must finish a GIO_read() or GIO_write() or GIO_issue()/GIO_reclaim() before another Task can use the same GIO_Handle. It is safer for only one Task to use each GIO_Handle.

GIO_delete() deletes a GIO instance and frees all resources allocated during GIO_create(). This frees up the driver channel in use.

GIO_control() can be called to configure or perform control functionality on the communication channel. The calling syntax is as follows:

```
Int GIO_control(
     GIO_Handle    handle,
     UInt          cmd,
     Ptr           args);
```

The cmd argument may be one of the command code constants listed in ti/sysbios/io/iom.h. Drivers may add command codes to provide additional functionality.

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

GIO_control() returns IOM_COMPLETED upon success. If an error occurs, the device returns a negative value. For a list of error values, see ti/sysbios/io/iom.h. The underlying driver IOM function call is typically a blocking call, so calling GIO_control() can result in blocking.

9.3.3   *Using GIO_read() and GIO_write() — The Standard Model*

The Standard GIO model is the simplest way to use GIO. It waits for a buffer to be ready and allows other threads to run while waiting. The Standard model can only be used if you are reading from and writing to the device in the context of a Task thread. For a more complicated model that uses double-buffering, see Section 9.3.4.

When you are using GIO_Model_STANDARD, your application should call GIO_read() and GIO_write() from the context of Task threads. GIO_read() is used to get a buffer from an input channel. GIO_write() is used to write to a buffer for an output channel.

GIO_read() has the following calling syntax:

```
Int GIO_read(
     GIO_Handle    handle,
     Ptr           buf,
     SizeT         *pSize);
```

An application calls GIO_read() to read a specified number of MADUs (minimum addressable data units) from a GIO channel. The GIO_Handle is a handle returned by GIO_create().

The buf parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the read data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data. For example, for video capture devices, this structure may contain pointers to RGB buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The pSize parameter points to the size of the buffer or data structure pointed to by the buf parameter. When GIO_read() returns, this parameter points to the number of MADUs actually read from the device. This parameter is relevant only if the buf parameter points to a raw data buffer. In cases where it points to a device-defined structure, it is redundant—the size of the structure is known to the mini-driver and the application.

GIO_read() returns IOM_COMPLETED upon successfully reading the requested number of MADUs from the device. If an error occurs, the device returns a negative value. For a list of error values, see ti/sysbios/io/iom.h.

GIO_read() blocks until the buffer is filled or the timeout period specified when this GIO instance was created occurs.

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

GIO_write() has the following parameters:

```
Int GIO_write(
      GIO_Handle    handle,
      Ptr           buf,
      SizeT         *pSize);
```

An application calls GIO_write() to write a specified number of MADUs (minimum addressable data units) to the GIO channel.

The handle, buf, and pSize parameters are the same as those passed to GIO_read(). When you call GIO_write(), the buf parameter should already contain the data to be written to the device. When GIO_write() returns, the pSize parameter points to the number of MADUs written to the device.

GIO_write() returns IOM_COMPLETED upon successfully writing the requested number of MADUs to the device. If an error occurs, the device returns a negative value. For a list of error values, see ti/sysbios/io/iom.h.

GIO_write() blocks until the buffer can be written to the device or the timeout period specified when this GIO instance was created occurs.

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

GIO_submit() is called internally by GIO_read() and GIO_write(). Typically, your application will not call it directly. The only reason to call it directly is if your application needs to call the mdSubmit function of the associated driver in some specialized way.

9.3.4 *Using GIO_issue(), GIO_reclaim(), and GIO_prime() — The Issue/Reclaim Model*

GIO has another model for using channels called GIO_Model_ISSUERECLAIM. This model uses asynchronous API calls, which cannot yield to other threads. If the buffer is not ready, the functions return a failure status. These APIs can be used in any thread context. The main GIO APIs used with the synchronous model are GIO_issue, GIO_reclaim, and GIO_prime.

- A GIO_issue()/GIO_reclaim() pair on an input channel is similar to a call to GIO_read().

- A GIO_issue()/GIO_reclaim() pair on an output channel is similar to a call to GIO_write().

GIO_issue() sends a buffer to a channel. Full buffers are sent to output channels, and empty buffers are sent to input channels. No buffer is returned during the call to GIO_issue(), and the application no longer owns the buffer. GIO_issue() returns control to the thread from which it was called without blocking. The buffer has been given to the driver for processing.

Later, when the application is ready to get the buffer back, it calls GIO_reclaim(). By default, this call to GIO_reclaim() blocks if the driver has not completed processing the I/O packet. If you want GIO_reclaim() to be non-blocking, you can specify that it should use a different Sync implementation when you create the GIO channel. In normal operation each GIO_issue() call is followed by a GIO_reclaim() call.

The GIO_issue/GIO_reclaim model provides flexibility by allowing your application to control the number of outstanding buffers at runtime. Your application can send multiple buffers to a channel without blocking by using GIO_issue(). This allows the application to choose how deep to buffer a device.

The GIO_issue/GIO_reclaim APIs guarantee that the client's buffers are returned in the order in which they were issued. This allows a client to use memory for streaming. For example, if a SYS/BIOS Task receives a large buffer, that Task can pass the buffer to the channel in small pieces simply by advancing a pointer through the larger buffer and calling GIO_issue() for each piece. The pieces of the buffer are guaranteed to come back in the same order they were sent. In this case, the number of buffer pieces to be passed should be less than or equal to the numPackets parameter you set when creating the channel.

Short bursts of multiple GIO_issue() calls can be made without an intervening GIO_reclaim() call followed by short bursts of GIO_reclaim() calls. However, over the life of the channel GIO_issue() and GIO_reclaim() must be called the same number of times. The number of GIO_issue() calls can exceed the number of GIO_reclaim() calls by the value specified by the gioParams.numPackets parameter when creating the channel (2 by default).

GIO_issue() has the following calling syntax:

```
Int GIO_issue(
    GIO_Handle    handle,
    Ptr           buf,
    SizeT         size,
    UArg          arg);
```

The GIO_Handle is a handle returned by GIO_create() and the buf argument is a pointer to a buffer to issue to the channel.

The size argument is a direction dependent logical size value. For an output channel, use the size argument to specify the number of MADUs (minimum addressable data units) of data the buffer contains. For an input channel, the size argument indicates the number of MADUs being requested. In either case, this logical size value must be less than or equal to the physical size of the buffer.

The arg argument is not interpreted by GIO, but can be used to pass associated information along with the buffer to the driver. All compliant device drivers preserve the value of arg and maintain its association with the data it was issued with. For example, arg could be used to send a timestamp to an output device, indicating exactly when the data is to be rendered. The driver treats the arg as a read-only field, and the arg is returned by GIO_reclaim().

If GIO_issue() returns a failure value, the channel was not able to accept the buffer being issued or there was a error from the underlying driver. Note that the error could be driver-specific. If GIO_issue() fails because of an underlying driver problem, your application should call GIO_abort() before attempting to perform more I/O through the channel.

GIO_reclaim() requests a buffer back from a channel. The buffers are returned in the same order that they were issued. GIO_reclaim() has the following calling syntax:

```
Int GIO_reclaim(
    GIO_Handle   handle,
    Ptr          *pBuf,
    SizeT        *pSize,
    UArg         *pArg);
```

For output channels, GIO_reclaim() returns a processed buffer, and the size is zero. For input channels, GIO_reclaim() returns a full buffer, and the size is the number of MADUs of data in the buffer.

As with GIO_issue(), the GIO_reclaim() API does not modify the contents of the pArg argument.

GIO_reclaim() calls Sync_wait() with the timeout you specified when you created the GIO channel. Sync_wait() used the Sync implementation you specified when you created the GIO channel. If this is a blocking Sync implementation, then GIO_reclaim() can block. If you are calling this API from a Swi thread, you should use a non-blocking Sync. See Section 9.5 for more about Sync implementations.

If GIO_reclaim() returns a failure status (!= IOM_COMPLETED), no buffer was returned. Therefore, if GIO_reclaim() fails, your application should not attempt to de-reference pBuf, since it is not guaranteed to contain a valid buffer pointer.

If you attempt to call GIO_reclaim() without having a previous unmatched call to GIO_issue(), the GIO_reclaim() call will either block forever or fail. The error is not detected at run-time.

GIO_prime() sends a buffer to an output channel to prime it at startup. It has the following calling syntax:

```
Int GIO_prime(
    GIO_Handle   handle,
    Ptr          buf,
    SizeT        size,
    UArg         arg);
```

This API is useful, for example, if an application uses double-buffering and you want to issue a buffer an output channel, but the driver cannot handle dummy buffers. GIO_prime() makes a buffer available for GIO_reclaim() without actually sending the buffer to the driver.

This API is non-blocking, and can be called from any thread context.

If GIO_prime() returns a failure status (!= IOM_COMPLETED), the GIO channel was not able to accept the buffer being issued due to the un-availability of GIO packets.

See the CDOC online reference described in Section 1.6.1 for more about these APIs.

9.3.5   *GIO_abort() and Error Handling*

Your application can delete all pending buffers being sent to a channel by calling GIO_abort(). If the error is less serious, you may instead call GIO_flush() to finish all pending output requests but discard any pending input requests. One of these actions may need to be done in response to the following situations:

- The application has decided to cancel current I/O and work on something else.

- A GIO API returned an error.

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

GIO_abort() returns IOM_COMPLETED after successfully aborting all input and output requests. If an error occurs, the device returns a negative value. The list of error values are defined in the ti/sysbios/io/iom.h file. The underlying call sent to the driver is typically implemented as a blocking call, so calling GIO_abort() can result in the thread blocking.

If you are using the ISSUERECLAIM model, the underlying device connected to the channel is idled as a result of calling GIO_abort(), and all buffers are made ready for GIO_reclaim(). The application needs to call GIO_reclaim() to get back the buffers. However the client will not block when calling GIO_reclaim() after GIO_abort().

GIO_abort() has the following calling syntax:

```
Int GIO_abort(
    GIO_Handle    handle
);
```

GIO_flush() discards all buffers pending for input channels and completes any pending requests to output channels. All pending input calls are completed with a status of IOM_FLUSHED, and all output calls are completed routinely.

This call returns IOM_COMPLETED upon successfully flushing all input and output. If an error occurs, the device returns a negative value. For a list of error values, see ti/sysbios/io/iom.h. The underlying call sent to the driver is typically implemented as a blocking call, so calling GIO_flush() can result in the thread blocking.

GIO_flush() has the following calling syntax:

```
Int GIO_flush(
    GIO_Handle    handle
);
```

9.4 Using GIO in Various Thread Contexts

The GIO module can be used in a number of different thread context and with various synchronization mechanisms. The following subsections discuss special cases.

9.4.1 Using GIO with Tasks

GIO_read and GIO_write can be used only with Task threads. In addition, GIO_issue() and GIO_reclaim() can be used with either Task or Swi threads. By default GIO uses SyncSem for synchronization.

9.4.1.1 Using a Semaphore Instance Created by the Application with a GIO Instance

There may be cases where you do not want GIO to create a Semaphore for synchronization, and instead wants to supply your own Semaphore to GIO. The following code snippet shows how to do this.

```
#include <ti/sysbios/syncs/SyncSem.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/io/GIO.h>


/*
* ======== main ========
*/
Int main(Int argc, Char* argv[])
{
    GIO_Params gioParams;
    SyncSem_Params syncParams;
    SyncSem_Handle syncSem;


    /* Create input channel */
    SyncSem_Params_init(&syncParams);
    syncParams.sem = Semaphore_create(0, NULL, NULL);
    syncSem = SyncSem_create(&syncParams, NULL);


    GIO_Params_init(&gioParams);
    gioParams.chanParams = (UArg)&genParamsIn;
    gioParams.sync = SyncSem_Handle_upCast(syncSem);
    handleIn = GIO_create("/genDevice", GIO_INPUT, &gioParams, &eb);
}
```

9.4.2   Using GIO with Swis

GIO_issue() and GIO_reclaim() can also be used with Swi threads. The client first creates a GIO Instance. The application has to populate the "sync" field in the GIO_Params struct. This sync field needs to be assigned to a SyncSwi instance.

The application needs to follow these steps to get a SyncSwi_Handle.

1. Create a Swi instance.

2. Populate the "swi" field of a SyncSwi_Params struct with the Swi_Handle received in the previous step.

3. Create a SyncSwi instance using the SyncSwi_Params struct.

4. Populate the "sync" field in GIO_Params struct with the SyncSwi_Handle received from the previous step.

The GIO module calls ISync_signal() in the callback path when I/O completes. This results in a Swi_post(). The swi runs and calls GIO_reclaim() to get back the processed buffer.

SyncSwi_wait() does nothing and returns FALSE for timeout.

9.4.3   Using GIO with Events

There are cases where a Task needs to wait on multiple events. For example, a Task may need to wait for a buffer from an input channel or a message from the host. In such cases, the Task should use the Event_pend() API to "wait on multiple" items. See Section 4.2 for more information on Events.

In order to use GIO with the Event module, the GIO instance needs to be created using the following steps:

1. Create an Event instance.

2. Populate the "event" field of the SyncEvent_Params struct with the Event_Handle received from the previous step.

3. Create a SyncEvent instance using the SyncEvent_Params struct.

4. Populate the "sync" field in the GIO_Params struct with the SyncEvent_Handle received from the previous step.

For the example later in this section, the application first primes the channel by calling GIO_issue(). When the worker task runs, it calls Event_pend() and BLOCKS waiting for I/O completion.

When I/O is complete, the GIO module calls ISync_signal(), which results in an Event_post() call. The Task wakes up, checks which event happened, and calls GIO_reclaim() to get the processed buffer.

9.5 GIO and Synchronization Mechanisms

The xdc.runtime.knl.ISync module specifies the interface for two functions that are used internally by a GIO channel instance: ISync_signal() and ISync_wait(). Various implementations of these interfaces handle synchronization of GIO in different ways. SYS/BIOS contains several implementations of the ISync interface:

- SyncSem (the default) is based on Semaphores and is BLOCKING. This module is found in ti.sysbios.syncs.

- SyncSwi is based on Swis and is NON-BLOCKING. This module is found in ti.sysbios.syncs.

- SyncEvent is based on Events and is NON-BLOCKING. This module is found in ti.sysbios.syncs.

- SyncGeneric can be used with any two functions that provide the ISync_signal() and ISync_wait() functionality. This module is found in XDCtools in xdc.runtime.knl.

- SyncNull performs no synchronization. This module is found in XDCtools in xdc.runtime.knl.

Instead of tying itself to a particular module for synchronization, the GIO module allows you to pick an ISync implementation module. You can select an ISync implementation for all GIO instances using the module-level configuration parameter "GIO_SyncProxy". You can select an ISync implementation for a specific GIO channel instance using the instance-level configuration parameter "sync" (see Section 9.3.2).

By default GIO_SyncProxy is bound to SyncSem by the BIOS module. If you pass NULL for the "sync" parameter to GIO_create(), the GIO module creates a SyncSem instance for the GIO instance. This translates to a Semaphore instance for this GIO instance.

The GIO module calls ISync_signal() in the callback path when I/O completes. It calls ISync_wait() from GIO_reclaim() when I/O has not been completed.

9.5.1 Using GIO with Generic Callbacks

It is possible for the application to provide two functions equivalent to ISync_signal() and ISync_wait() using the xdc.runtime.knl.SyncGeneric module.

One use case for this is when the application will use GIO with a simple callback. The application then provides its callback function as the signal function to SyncGeneric_create(). The GIO module then invokes the application callback when I/O completes. The application can reclaim the buffer within the callback function.

Rebuilding SYS/BIOS

This appendix describes how to rebuild the SYS/BIOS source code.

A.1 Overview

The SYS/BIOS product includes source files and build scripts that allow you to modify the SYS/BIOS sources and rebuild its libraries. You can do this in order to modify, update, or add functionality. If you edit the SYS/BIOS source code and/or corresponding build scripts, you must also rebuild SYS/BIOS in order to create new libraries containing these modifications.

Note that starting with SYS/BIOS v6.32, you can cause the SYS/BIOS libraries to be rebuilt as part of the application build within CCS. See Section 2.3.5 for details. The custom-built libraries will be stored with your CCS project and will contain only modules and APIs that your application needs to access. You can cause such a custom build to occur by configuring the BIOS.libType property as follows.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
```

Caution: This appendix provides details about rebuilding the SYS/BIOS source code. We strongly recommend that you copy the SYS/BIOS installation to a directory with a different name and rebuild that copy, rather than rebuilding the original installation.

For information about building SYS/BIOS applications (that is, applications that use SYS/BIOS), see Section 2.3.

A.2 Prerequisites

In order to rebuild SYS/BIOS, the SYS/BIOS and XDCtools products must both be installed. It is important to build SYS/BIOS with a compatible version of XDCtools. To find out which versions are compatible, see the "Dependencies" section of the Release Notes in the top-level directory of your SYS/BIOS installation.

Note: You should generally avoid installing the various Texas Instruments tools and source distributions in directories that have spaces in their paths.

A.3 Building SYS/BIOS Using the bios.mak Makefile

Rebuilding SYS/BIOS itself from the provided source files is straightforward, whether you are using the TI compiler toolchain or the GNU GCC toolchain.

SYS/BIOS ships with a `bios.mak` file in the top-level installation directory. This makefile enables you to easily (re)build SYS/BIOS using your choice of compilers and desired "targets". A target incorporates a particular ISA and a runtime model; for example, cortex-M3 and the GCC compiler with specific options.

The instructions in this section can be used to build SYS/BIOS applications on Windows or Linux. If you are using a Windows machine, you can use the regular DOS command shell provided with Windows. However, you may want to install a Unix-like shell, such as Cygwin.

For Windows users, the XDCtools top-level installation directory contains `gmake.exe`, which is used in the commands that follow to run the Makefile. The gmake utility is a Windows version of the standard GNU "make" utility provided with Linux.

If you are using Linux, change the "gmake" command to "make" in the commands that follow.

For these instructions, suppose you have the following directories:

- `$BASE/sysbios/bios_6_33_01_##` — The location where you installed SYS/BIOS.

- `$BASE/sysbios/copy-bios_6_33_01_##` — The location of a copy of the SYS/BIOS installation.

- `$BASE/xdctools_3_23_##_##` — The location where you installed XDCtools.

- `$TOOLS/gcc/bin/arm-none-eabi-gcc` — The location of a compiler, in this case a GCC compiler for M3.

The following steps refer to the top-level directory of the XDCtools installation as *<xdc_install_dir>*. They refer to the top-level directory of the *copy* of the SYS/BIOS installation as *<bioscopy_install_dir>*.

Follow these steps to rebuild SYS/BIOS:

1. If you have not already done so, install XDCtools and SYS/BIOS.

2. Make a copy of the SYS/BIOS installation that you will use when rebuilding. This leaves you with an unmodified installation as a backup. For example, use commands similar to the following on Windows:

```
mkdir c:\sysbios\copy-bios_6_33_01_##

copy c:\sysbios\bios_6_33_01_## c:\sysbios\copy-bios_6_33_01_##
```

Or, use the a command similar to the following on Linux:

```
cp -r $BASE/sysbios/bios_6_33_01_##/* $BASE/sysbios/copy-bios_6_33_01_##
```

3. Make sure you have access to compilers for any targets for which you want to be able be able to build applications using the rebuilt SYS/BIOS. Note the path to the directory containing the executable for each compiler. These compilers can include Texas Instruments compilers, GCC compilers, and any other command-line compilers for any targets supported by SYS/BIOS.

4. If you are using Windows and the gmake utility provided in the top-level directory of the XDCtools installation, you should add the *<xdc_install_dir>* to your PATH environment variable so that the gmake executable can be found.

5. You may remove the top-level doc directory located in *<bioscopy_install_dir>*/docs if you need to save disk space.

6. At this point, you may want to add the remaining files in the SYS/BIOS installation tree to your Software Configuration Management (SCM) system.

7. Open the `<bioscopy_install_dir>`/bios.mak file with a text editor, and make the following changes for any options you want to hardcode in the file. (You can also set these options on the command line if you want to override the settings in the bios.mak file.)

— Ignore the following lines near the beginning of the file. These definitions are used internally, but few users will have a need to change them.

```
#
# Where to install/stage the packages
# Typically this would point to the devkit location
#
DESTDIR ?= <UNDEFINED>

prefix ?= /
docdir ?= /docs/bios
packagesdir ?= /packages
```

— Specify the location of XDCtools. For example:

```
XDC_INSTALL_DIR ?= $(BASE)/xdctools_3_23_##_##
```

— Specify the location of the compiler executable for all targets you want to be able to build for with SYS/BIOS. Use only the directory path; do not include the name of the executable file. Any targets for which you do not specify a compiler location will be skipped during the build. For example, on Linux you might specify the following:

```
ti.targets.C28_float ?= /opt/ti/ccsv5/tools/compiler/c2000
ti.targets.arm.elf.M3 ?= /opt/ti/ccsv5/tools/compiler/tms470
gnu.targets.arm.M3 ?= $TOOLS/gcc/bin/arm-none-eabi-gcc
```

Similarly, on Windows you might specify the following compiler locations:

```
ti.targets.C28_float ?= c:/ti/ccsv5/tools/compiler/c2000
ti.targets.arm.elf.M3 ?= c:/ti/ccsv5/tools/compiler/tms470
gnu.targets.arm.M3 ?= c:/tools/gcc/bin/arm-none-eabi-gcc
```

— If you need to add any repositories to your XDCPATH (for example, to reference the packages directory of another component), you should edit the XDCPATH definition.

— You can uncomment the line that sets XDCOPTIONS to "v" if you want more information output during the build.

8. Clean the SYS/BIOS installation with the following commands. (If you are running the build on Linux, change all "gmake" commands to "make".)

```
cd <bioscopy_install_dir>

gmake -f bios.mak clean
```

9. Run the bios.mak file to build SYS/BIOS as follows. (Remember, if you are running the build on Linux, change all "gmake" commands to "make".)

```
gmake -f bios.mak
```

10. If you want to specify options on the command line to override the settings in bios.mak, use a command similar to the following.

```
gmake -f bios.mak XDC_INSTALL_DIR=<xdc_install_dir> gnu.targets.arm.M3=<compiler_path>
```

A.4   Building Your Project Using a Rebuilt SYS/BIOS

To build your application using the version of SYS/BIOS you have rebuilt, you must point your project to this rebuilt version by following these steps:

1. Open CCS and select the application project you want to rebuild.

2. Right-click on your project and choose **Properties**. If you have a configuration project that is separate from your application project, open the properties for the configuration project.

3. In the **CCS General** category of the Properties dialog, choose the **RTSC** tab.

4. Under the **Products and Repositories** tab, uncheck *all* the boxes for SYS/BIOS (and DSP/BIOS if there are any). This ensures that no version is selected.
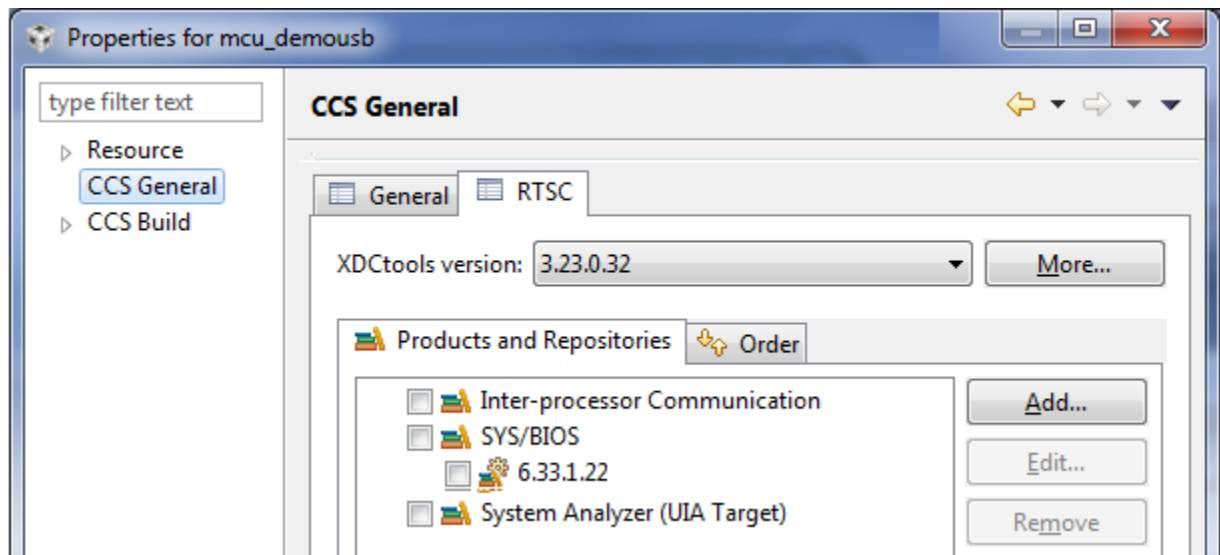
5. Click the **Add** button next to the **Products and Repositories** tab.

6. Choose **Select repository from file-system**, and browse to the "packages" directory of the location where you copied and rebuilt SYS/BIOS. For example, the location may be `C:\myBiosBuilds\custom_bios_6_##_##-##\packages` on Windows or `$BASE/sysbios/copy-bios_6_33_01_##/packages` on Linux.
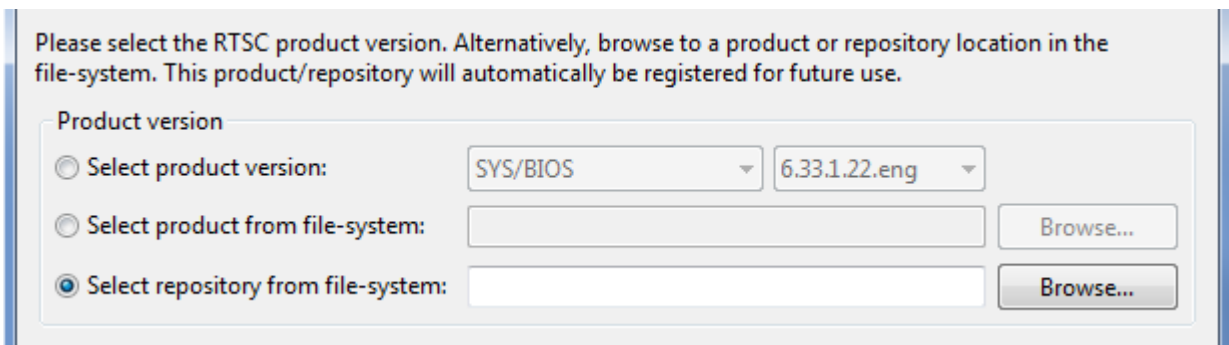
7. Click **OK** to apply these changes to the project.

8. You may now rebuild your project using the re-built version of SYS/BIOS.

Timing Benchmarks

This appendix describes SYS/BIOS timing benchmark statistics.

B.1   Timing Benchmarks

This appendix describes the timing benchmarks for SYS/BIOS functions, explaining the meaning of the values as well as how they were obtained, so that designers may better understand their system performance.

The sections that follow explain the meaning of each of the timing benchmarks. The name of each section corresponds to the name of the benchmark in the actual benchmark data table.

The explanations in this appendix are best viewed along side the actual benchmark data. Since the actual benchmark data depends on the target and the memory configuration, and is subject to change, the data is provided in HTML files in the ti.sysbios.benchmarks package (that is, in the BIOS_INSTALL_DIR\packages\ti\sysbios\benchmarks directory).

The benchmark data was collected with the Build-Profile set to "release" and the BIOS.libType configuration parameter set to BIOS.LibType_Custom. See Section 2.3.5 for more on these settings.

B.2   Interrupt Latency

The Interrupt Latency benchmark is the maximum number of instructions during which the SYS/BIOS kernel disables maskable interrupts. Interrupts are disabled in order to modify data shared across multiple threads. SYS/BIOS minimizes this time as much as possible to allow the fastest possible interrupt response time.

The interrupt latency of the kernel is measured across the scenario within SYS/BIOS in which maskable interrupts will be disabled for the longest period of time. The measurement provided here is the cycle count measurement for executing that scenario.

B.3   Hwi-Hardware Interrupt Benchmarks

Hwi_enable(). This is the execution time of a Hwi_enable() function call, which is used to globally enable hardware interrupts.

Hwi_disable(). This is the execution time of a Hwi_disable() function call, which is used to globally disable hardware interrupts.

Hwi dispatcher. These are execution times of specified portions of Hwi dispatcher code. This dispatcher handles running C code in response to an interrupt. The benchmarks provide times for the following cases:

- **Interrupt prolog for calling C function.** This is the execution time from when an interrupt occurs until the user's C function is called.

- **Interrupt epilog following C function call.** This is the execution time from when the user's C function completes execution until the Hwi dispatcher has completed its work and exited.

Hardware interrupt to blocked task. This is a measurement of the elapsed time from the start of an ISR that posts a semaphore, to the execution of first instruction in the higher-priority blocked task, as shown in Figure B–1.
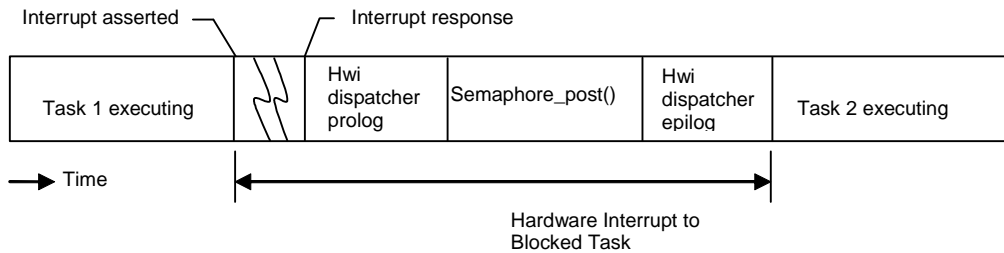
Figure B–1 Hardware Interrupt to Blocked Task

Hardware interrupt to software interrupt. This is a measurement of the elapsed time from the start of an ISR that posts a software interrupt, to the execution of the first instruction in the higher-priority posted software interrupt.

This duration is shown in Figure B–2. Swi 2, which is posted from the ISR, has a higher priority than Swi 1, so Swi 1 is preempted. The context switch for Swi 2 is performed within the Swi executive invoked by the Hwi dispatcher, and this time is included within the measurement. In this case, the registers saved/restored by the Hwi dispatcher correspond to that of "C" caller saved registers.
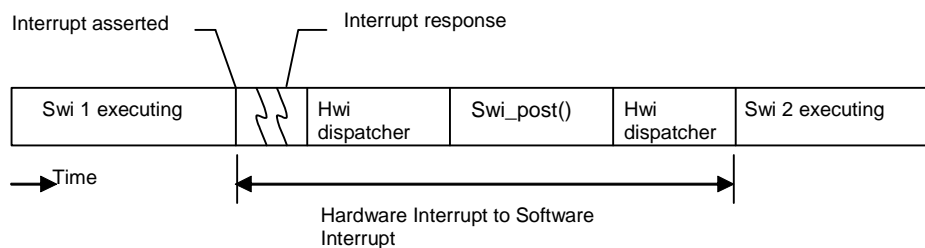
Figure B–2 Hardware Interrupt to Software Interrupt

B.4 Swi-Software Interrupt Benchmarks

Swi_enable(). This is the execution time of a Swi_enable() function call, which is used to enable software interrupts.

Swi_disable(). This is the execution time of a Swi_disable() function call, which is used to disable software interrupts.

Swi_post(). This is the execution time of a Swi_post() function call, which is used to post a software interrupt. Benchmark data is provided for the following cases of Swi_post():

- **Post software interrupt again.** This case corresponds to a call to Swi_post() of a Swi that has already been posted but hasn't started running as it was posted by a higher-priority Swi. Figure B–3 shows this case. Higher-priority Swi1 posts lower-priority Swi2 twice. The cycle count being measured corresponds to that of second post of Swi2.

Swi 1 executing	Swi_post() of Swi 2	Swi 1 executing	Swi_post() of Swi 2 again	Swi 1 executing

→ Time

Post a Swi that has already been posted
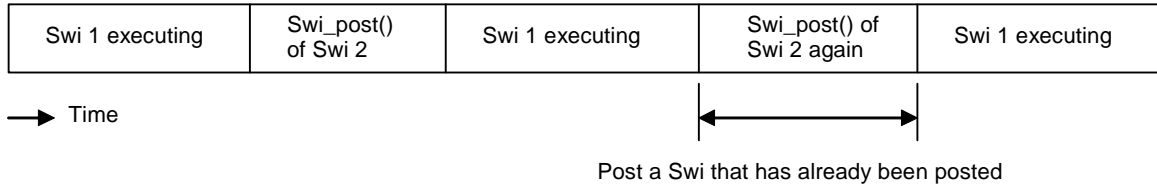
Figure B–3 Post of Software Interrupt Again

- **Post software interrupt, no context switch.** This is a measurement of a Swi_post() function call, when the posted software interrupt is of lower priority then currently running Swi. Figure B–4 shows this case.

Swi 1 executing	Swi_post() of Swi 2	Swi 1 executing

→ Time

Post Software Interrupt,
No Context Switch
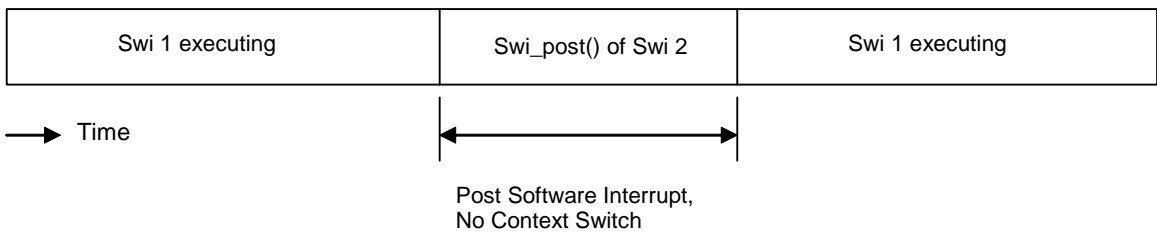
Figure B–4 Post Software Interrupt without Context Switch

- **Post software interrupt, context switch.** This is a measurement of the elapsed time between a call to Swi_post() (which causes preemption of the current Swi) and the execution of the first instruction in the higher-priority software interrupt, as shown in Figure B–5. The context switch to Swi2 is performed within the Swi executive, and this time is included within the measurement.

Swi 1 executing	Swi_post() of Swi 2	Swi 2 executing

→ Time

Post Software Interrupt,
Context Switch
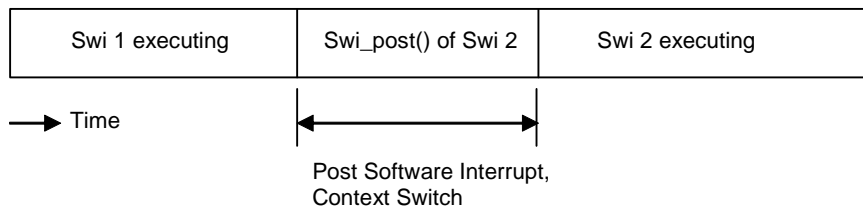
Figure B–5 Post Software Interrupt with Context Switch

B.5   Task Benchmarks

Task_enable(). This is the execution time of a Task_enable() function call, which is used to enable SYS/BIOS task scheduler.

Task_disable(). This is the execution time of a Task_disable() function call, which is used to disable SYS/BIOS task scheduler.

Task_create(). This is the execution time of a Task_create() function call, which is used to create a task ready for execution. Benchmark data is provided for the following cases of Task_create():

- **Create a task, no context switch.** The executing task creates and readies another task of lower or equal priority, which results in no context switch. See Figure B–6.

Task 1 executing	Task_create() (Readies lower priority new Task 2)	Task 1 executing

→ Time

Create a Task, No Context Switch
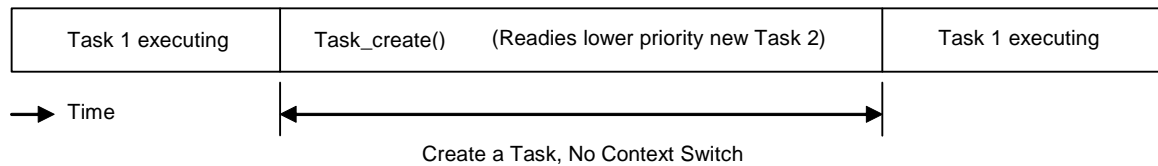
Figure B–6 Create a New Task without Context Switch

- **Create a task, context switch.** The executing task creates another task of higher priority, resulting in a context switch. See Figure B–7.

Task 1 executing	Task_create() (Readies higher priority new Task 2, Task Context Switch)	Task 2 executing

→ Time

Create a Task, Context Switch
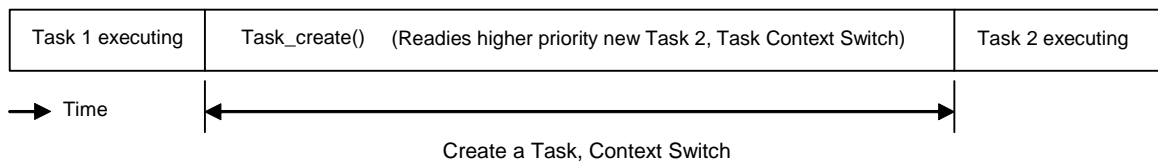
Figure B–7 Create a New Task with Context Switch

Note:	The benchmarks for Task_create() assume that memory allocated for a Task object is available in the first free list and that no other task holds the lock to that memory. Additionally the stack has been pre-allocated and is being passed as a parameter.

Task_delete(). This is the execution time of a Task_delete() function call, which is used to delete a task. The Task handle created by Task_create() is passed to the Task_delete() API.

Task_setPri(). This is the execution time of a Task_setPri() function call, which is used to set a task's execution priority. Benchmark data is provided for the following cases of Task_setPri():

- **Set a task priority, no context switch.** This case measures the execution time of the Task_setPri() API called from a task Task1 as in Figure B–8 if the following conditions are all true:

  — Task_setPri() sets the priority of a lower-priority task that is in ready state.

  — The argument to Task_setPri() is less then the priority of current running task.

Task 1 executing	Task_setPri()	Task 1 executing

→ Time

Set a Task Priority,
No Context Switch
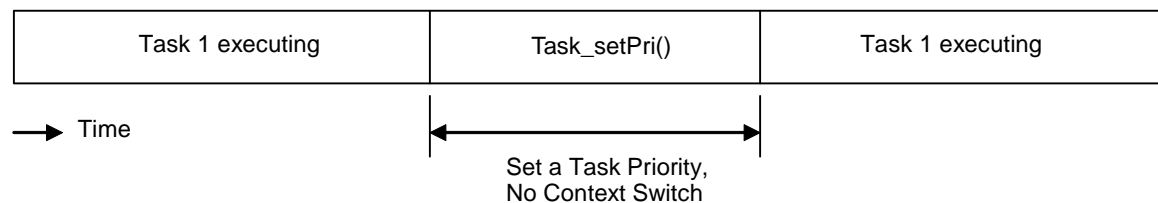
Figure B–8 Set a Task's Priority without a Context Switch

- **Lower the current task's own priority, context switch.** This case measures execution time of Task_setPri() API when it is called to lower the priority of currently running task. The call to Task_setPri() would result in context switch to next higher-priority ready task. Figure B–9 shows this case.

Task 1 executing	Task_setPri() (Lower Task 1's priority, Task Context Switch)	Task 2 executing

→ Time

Lower the Current Task's Own Priority, Context Switch
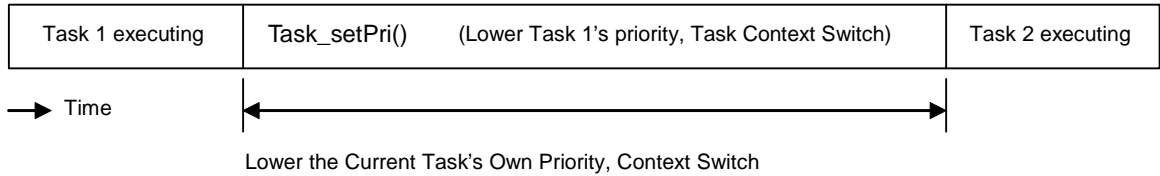
Figure B–9 Lower the Current Task's Priority, Context Switch

- **Raise a ready task's priority, context switch.** This case measures execution time of Task_setPri() API called from a task Task1 if the following conditions are all true:

  — Task_setPri() sets the priority of a lower-priority task that is in ready state.

  — The argument to Task_setPri() is greater then the priority of current running task.

  The execution time measurement includes the context switch time as shown in Figure B–10.

Task 1 executing	Task_setPri() (Raise Task 2's priority, Task Context Switch)	Task 2 executing

→ Time

Raise a Task's Priority, Context Switch
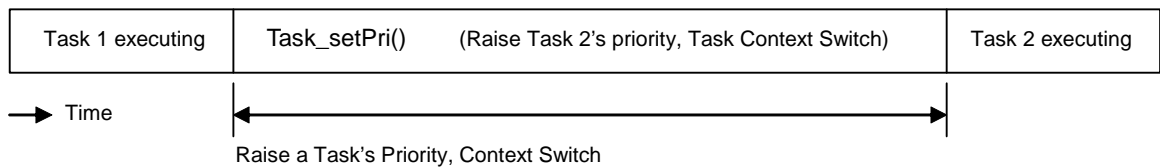
Figure B–10 Raise a Ready Task's Priority, Context Switch

- **Task_yield().** This is a measurement of the elapsed time between a function call to Task_yield() (which causes preemption of the current task) and the execution of the first instruction in the next ready task of equal priority, as shown in Figure B–11.

Task 1 executing	Task_yield() (Task Context Switch)	Task 2 executing
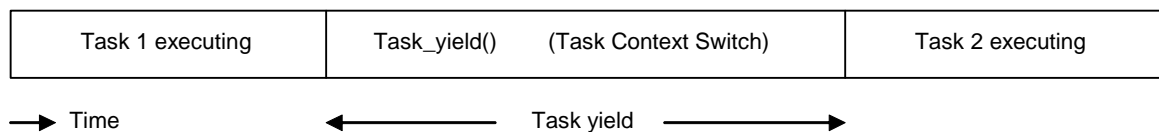
→ Time   ←   Task yield   →

Figure B–11 Task Yield

B.6   Semaphore Benchmarks

Semaphore benchmarks measure the time interval between issuing a Semaphore_post() or Semaphore_pend() function call and the resumption of task execution, both with and without a context switch.

Semaphore_post(). This is the execution time of a Semaphore_post() function call. Benchmark data is provided for the following cases of Semaphore_post():

- **Post a semaphore, no waiting task.** In this case, the Semaphore_post() function call does not cause a context switch as no other task is waiting for the semaphore. This is shown in Figure B–12.

Task 1 executing		Task 1 executing

Time →

Post Semaphore,
No Waiting Task
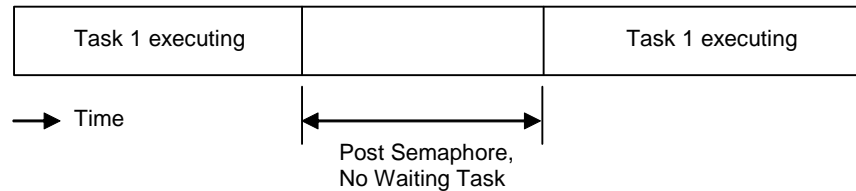
Figure B–12 Post Semaphore, No Waiting Task

- **Post a semaphore, no context switch.** This is a measurement of a Semaphore_post() function call, when a lower-priority task is pending on the semaphore. In this case, Semaphore_post() readies the lower-priority task waiting for the semaphore and resumes execution of the original task, as shown in Figure B–13.

Task 1 executing	Semaphore_post()	Task 1 executing

Time →

Post Semaphore,
No Context Switch
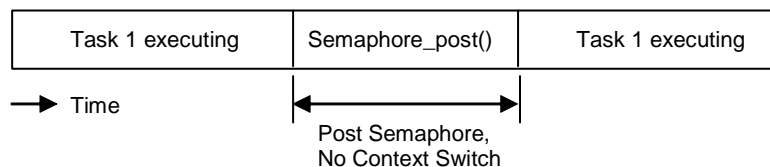
Figure B–13 Post Semaphore, No Context Switch

- **Post a semaphore, context switch.** This is a measurement of the elapsed time between a function call to Semaphore_post() (which readies a higher-priority task pending on the semaphore causing a context switch to higher-priority task) and the execution of the first instruction in the higher-priority task, as shown in Figure B–14.

Task 1 executing	Semaphore_post() (Readies higher priority Task 2, Context Switch)	Task 2 executing

Time →

Post Semaphore, Context Switch
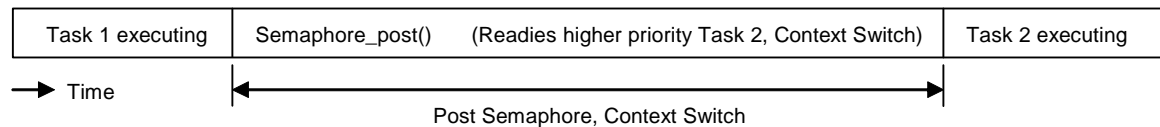
Figure B–14 Post Semaphore with Task Switch

Semaphore_pend(). This is the execution time of a Semaphore_pend() function call, which is used to acquire a semaphore. Benchmark data is provided for the following cases of Semaphore_pend():

- **Pend on a semaphore, no context switch.** This is a measurement of a Semaphore_pend() function call without a context switch (as the semaphore is available.) See Figure B–15.

Task 1 executing		Task 1 executing

Time →

Pend on Semaphore,
No Context Switch
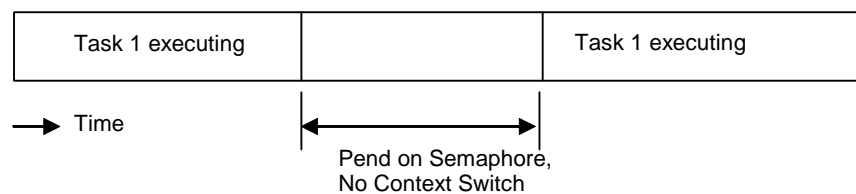
Figure B–15 Pend on Semaphore, No Context Switch

- **Pend on a semaphore, context switch.** This is a measurement of the elapsed time between a function call to Semaphore_pend() (which causes preemption of the current task) and the execution of first instruction in next higher-priority ready task. See Figure B–16.

Task 1 executing	(Task 1 suspends, Task Context Switch)	Task 2 executing

→ Time

Pend on Semaphore, Task Switch
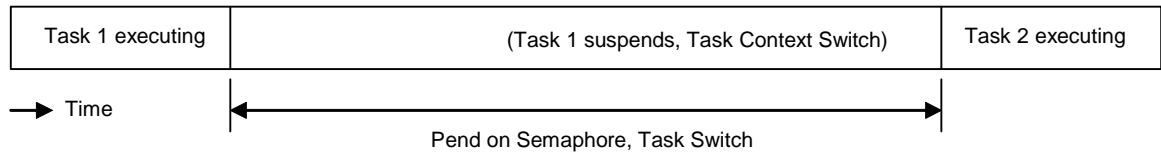
Figure B–16 Pend on Semaphore with Task Switch

Size Benchmarks

This appendix describes SYS/BIOS size benchmark statistics.

C.1 Overview

This appendix contains information on the size impact of using SYS/BIOS modules in an application.

Tradeoffs between different SYS/BIOS 6 modules and their impact on system memory can be complex, because applying a module usually requires support from other modules.

Also, even if one module's code is linked in by another module, it does not necessarily link in the entire module, but typically only the functions referenced by the application—an optimization that keeps the overall size impact of the SYS/BIOS 6 kernel to a minimum.

Because of the complexity of these tradeoffs, it is important to understand that this appendix does not provide an analytical model of estimating SYS/BIOS 6 overhead, but rather gives sizing information for a number of SYS/BIOS configurations.

The size benchmarks are a series of applications that are built on top of one another. Moving down Table C–1, each application includes all of the configuration settings and API calls in the previous applications. Applications lower on the table generally require the modules in the applications above them (The Clock module, for example, requires the Hwi module), so this progression allows for measuring the size impact of a module by subtracting the sizes of all of the other modules it depends upon. (The data in the table, however, is provided in absolute numbers.)

The actual size benchmark data is included in the SYS/BIOS 6 installation in the ti.sysbios.benchmarks package (that is, in the BIOS_INSTALL_DIR\packages\ti\sysbios\benchmarks directory). There is a separate HTML file for each target. For example, the 'C64x sizing information is in the c6400Sizing.html file. The sections that follow should be read alongside the actual sizing information as a reference.

The benchmark data was collected with the Build-Profile set to "release" and the BIOS.libType configuration parameter set to BIOS.LibType_Custom. See Section 2.3.5 for more on these settings.

For each benchmark application, the table provides four pieces of sizing information, all in 8-bit bytes.

- **Code Size** is the total size of all of the code in the final executable.
- **Initialized Data Size** is the total size of all constants (the size of the .const section).
- **Uninitialized Data Size** is the total size of all variables.
- **C-Initialization Size** is the total size of C-initialization records.

C.2 Comparison to DSP/BIOS 5

Where possible, SYS/BIOS 6 size benchmarks have been designed to match the DSP/BIOS 5 benchmarks so that the results can be compared directly. The following table shows which data to compare.

Table C–1 Comparison of Benchmark Applications

DSP/BIOS 5	SYS/BIOS 6
Default configuration	Default configuration
Base configuration	Basic configuration
HWI application	Hwi application
CLK application	Clock application

DSP/BIOS 5	SYS/BIOS 6
CLK Object application	Clock Object application
SWI application	Swi application
SWI Object application	Swi Object application
PRD application	None[1]
PRD Object application	None[1]
TSK application	Task application
TSK Object application	Task Object application
SEM application	Semaphore application
SEM Object application	Semaphore Object application
MEM application	Memory application
Dynamic TSK application	Dynamic Task application
Dynamic SEM application	Dynamic Semaphore application
RTA application	None[2]
None[3]	Timing Application

[1] SYS/BIOS 6 does not have a PRD module. Instead, the SYS/BIOS 6 Clock module supports the functionality of both the DSP/BIOS 5 CLK and PRD modules.

[2] The RTA application is not implemented for SYS/BIOS 6.

[3] The new benchmark is the application used to generate the timing benchmarks for SYS/BIOS 6 (see Appendix B). This application leverages all of the key components of the operating system in a meaningful way. It does not utilize any of the size-reducing measures employed in the base configuration of the size benchmarks.

C.3 Default Configuration Sizes

There are two minimal configurations provided as base size benchmarks:

- **Default Configuration.** This is the true "default" configuration of SYS/BIOS. The configuration script simply includes the BIOS module as follows:

```
xdc.useModule('ti.sysbios.BIOS');
```

This shows the size of an empty application with everything left at its default value; no attempts have been made here to minimize the application size.

- **Basic Configuration.** This configuration strips the application of all unneeded features and is essentially the smallest possible SYS/BIOS application. Appendix D details tactics used to reduce the memory footprint of this configuration. This is the configuration that the size benchmarks will be built off of.

C.4   Static Module Application Sizes

This section is the focus of the size benchmarks. Each application builds on top of the applications above it in Table C–1.

For each module there are generally two benchmarks. For example, there is the "Clock application" benchmark and the "Clock Object application" benchmark. The first of the two benchmarks (Clock application) does three things:

- In the configuration script, it includes the module.

- In the configuration script, it creates a static instance of the module.

- In the C code, it makes a call to one of the module's APIs.

The second benchmark (the "object" application) creates a second static instance in the configuration script. This demonstrates the size impact of creating an instance of that object. For example, if the Clock application requires x bytes of initialized data, and the Clock Object application requires y bytes of initialized data, then the impact of one Clock instance is (y - x) bytes of data.

The code snippets for each application apply to all targets, except where noted.

C.4.1   Hwi Application

The Hwi Application configuration script creates a Hwi instance, and the C code calls the Hwi_plug() API.

Configuration Script Addition

```
// Use target/device-specific Hwi module.
var Hwi = xdc.useModule('ti.sysbios.family.c64.Hwi');
var hwi5 = Program.global.hwi5 = Hwi.create(5, '&oneArgFxn');
```

C Code Addition

```
Hwi_plug(7, (Hwi_PlugFuncPtr)main);
```

C.4.2   Clock Application

The Clock Application enables the Clock module, creates a Clock instance, and pulls in the modules necessary to call the Timestamp_get32() API in the C code.

Configuration Script Addition

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
xdc.useModule('xdc.runtime.Timestamp');

BIOS.clockEnabled = true;
var Clock = xdc.useModule('ti.sysbios.knl.Clock');
Clock.create("&oneArgFxn", 5, {startFlag:true,arg:10});
```

C Code Addition

```
Timestamp_get32();
```

C.4.3 Clock Object Application

The Clock Object Application statically creates an additional Clock instance to illustrate the size impact of each Clock instance.

Configuration Script Addition

```
Clock.create("&oneArgFxn", 5, {startFlag:true,arg:10});
```

C.4.4 Swi Application

The Swi Application enables the Swi module and creates a static Swi instance in the configuration script. It calls the Swi_post() API in the C code.

Configuration Script Addition

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.swiEnabled = true;

var Swi = xdc.useModule('ti.sysbios.knl.Swi');
Program.global.swi0 = Swi.create('&twoArgsFxn');
```

C Code Addition

```
Swi_post(swi0);
```

C.4.5 Swi Object Application

The Swi Object Application creates an additional Swi instance to illustrate the size impact of each new Swi instance.

Configuration Script Addition

```
Program.global.swi1 = Swi.create('&twoArgsFxn');
```

C.4.6 Task Application

The Task Application configuration script enables Tasks and creates a Task instance. It also configures the stack sizes to match the sizes in the DSP/BIOS 5 benchmarks (for comparison). In the C code, the Task application makes a call to the Task_yield() API.

Configuration Script Addition

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.taskEnabled = true;

var Task = xdc.useModule('ti.sysbios.knl.Task');
Program.global.tsk0 = Task.create("&twoArgsFxn");

Task.idleTaskStackSize = 0x200;
Program.global.tsk0.stackSize = 0x200;
```

C Code Addition

```
Task_yield();
```

C.4.7 Task Object Application

The Task Object Application creates an additional Task instance to illustrate the size impact of each new Task instance.

Configuration Script Addition

```
Program.global.tsk1 = Task.create("&twoArgsFxn");
Program.global.tsk1.stackSize = 0x200;
```

C.4.8 Semaphore Application

The Semaphore Application configuration script creates a Semaphore instance and disables support for Events in the Semaphore for an equitable comparison with the DSP/BIOS 5 SEM module.

In the C code, the Semaphore application makes a call to the Semaphore_post() and Semaphore_pend() APIs.

Configuration Script Addition

```
var Sem = xdc.useModule('ti.sysbios.knl.Semaphore');
Sem.supportsEvents = false;
Program.global.sem0 = Sem.create(0);
```

C Code Addition

```
Semaphore_post(sem0);
Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
```

C.4.9 Semaphore Object Application

The Semaphore Object Application configuration script creates an additional Semaphore instance to illustrate the size impact of each new Semaphore instance.

Configuration Script Addition

```
Program.global.sem1 = Sem.create(0);
```

C.4.10 Memory Application

The Memory Application configuration script configures the default heap used for memory allocations. It creates a HeapMem instance to manage a 4 KB heap, places the heap into its own section in memory, then assigns the HeapMem instance as the default heap to use for Memory.

In the C code, the Memory application makes calls to the Memory_alloc() and Memory_free() APIs. It allocates a block from the default heap by passing NULL as the first parameter to Memory_alloc(), then frees the block back to the default heap by again passing NULL as the first parameter to Memory_free().

Configuration Script Addition

```
var mem = xdc.useModule('xdc.runtime.Memory');

var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
var heap0 = HeapMem.create();
heap0.sectionName = "myHeap";
Program.sectMap["myHeap"] = Program.platform.dataMemory;
heap0.size = 0x1000;

mem.defaultHeapInstance = heap0;
```

C Code Addition

```
Ptr *buf;
Error_Block eb;

Error_init(&eb);

buf = Memory_alloc(NULL, 128, 0, &eb);
if (buf == NULL) {
    System_abort("Memory allocation failed");
}

Memory_free(NULL, buf, 128);
```

C.5   Dynamic Module Application Sizes

The following application demonstrate the size effects of creating object dynamically (in the C code).

C.5.1   Dynamic Task Application

The Dynamic Task Application demonstrates the size impact of dynamically (in the C code) creating and deleting a Task instance. This application comes after the Memory application because it must use the Memory module to allocate space for the new Task instance.

C Code Addition

```
Task_Handle task;
Error_Block eb;

Error_init(&eb);

task = Task_create((Task_FuncPtr)main, NULL, &eb);
if (task == NULL) {
    System_abort("Task create failed");
}
Task_delete(&task);
```

C.5.2   Dynamic Semaphore Application

The Dynamic Semaphore Application demonstrates the size impact of dynamically (in the C code) creating and deleting a Semaphore instance. This application comes after the Memory application because it must use the Memory module to allocate space for the new Semaphore instance.

C Code Addition

```
Semaphore_Handle sem;
Error_Block eb;

Error_init(&eb);

sem = Semaphore_create(1, NULL, &eb);
if (sem == NULL) {
    System_abort("Semaphore create failed");
}
Semaphore_delete(&sem);
```

C.6   Timing Application Size

The timing application is the application used to generate the timing benchmarks for SYS/BIOS 6 (see Appendix B). This application leverages all of the key components of the operating system in a meaningful way, and does not utilize any of the size-reducing measures employed in the base configuration of the size benchmarks. Therefore, this is the largest application provided as a benchmark.

Minimizing the Application Footprint

This appendix describes how to minimize the size of a SYS/BIOS application.

D.1   Overview

This section provides tips and suggestions for minimizing the memory requirements of a SYS/BIOS-based application. This is accomplished by disabling features of the operating system that are enabled by default and by reducing the size of certain buffers in the system.

Most of the tips described here are used in the "Kernel only" configuration for the size benchmarks. The final section of this chapter presents a full configuration script that minimizes memory requirements.

The actual size benchmark data is included in the SYS/BIOS 6 installation in the ti.sysbios.benchmarks package (that is, in the BIOS_INSTALL_DIR\packages\ti\sysbios\benchmarks directory). There is a separate HTML file for each target. For example, the 'C64x sizing information can be found in the c6400Sizing.html file.

The following sections simply describe different configuration options and their effect on reducing the application size. For further details on the impact of these settings, refer to the documentation for the relevant modules.

Because the code and data sections are often placed in separate memory segments, it may be more important to just reduce either code size data size. Therefore the suggestions are divided based on whether they reduce code or data size. In general, it is easier to reduce data size than code size.

D.2   Reducing Data Size

D.2.1   Removing the malloc Heap

Calls to malloc are satisfied by a separate heap, whose size is configurable. The following code removes the heap to minimize to data footprint. Applications that remove the heap cannot dynamically allocate memory. Therefore, such applications should not use the SYS/BIOS Memory module APIs or other SYS/BIOS APIs that internally allocate memory from a heap.

```
BIOS.heapSize = 0;
```

D.2.2   Reducing Space for Arguments to main()

A special section in memory is created to store any arguments to the main() function of the application. The size of this section is configurable, and can be reduced depending on the application's needs.

```
Program.argSize = 0x4;
```

The Program variable is automatically available to all scripts. It defines the "root" of the configuration object model. It comes from the xdc.cfg.Program module, and is implicitly initialized as follows:

```
var Program = xdc.useModule('xdc.cfg.Program');
```

D.2.3 Reducing the Size of Stacks

See Section 3.4.3 for information about system stack size requirements and Section 3.5.3 for information about task stack size requirements.

The size of the System stack, which is used as the stack for interrupt service routines (Hwis and Swis), is configurable, and can be reduced depending on the application's needs.

```
Program.stack = 0x400;
```

Likewise, the size of each Task stack is individually configurable, and can be reduced depending on the application's needs. See Section 3.5.3 for information about task stack sizes.

```
var tskParams = new Task.Params;
tskParams.stackSize = 512;
var task0 = Task.create('&task0Fxn', tskParams);
```

You can configure the size of the Idle task stack as follows:

```
Task.idleTaskStackSize = 512;
```

D.2.4 Disabling Named Modules

The space used to store module name strings can be reclaimed with the following configuration setting:

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
Defaults.common$.namedModule = false;
```

D.2.5 Leaving Text Strings Off the Target

By default, all of the text strings in the system, such as module and instance names and error strings, are loaded into the target's memory. These strings can be left out of the application using the following settings.

```
var Text = xdc.useModule('xdc.runtime.Text');
Text.isLoaded = false;
```

D.2.6 Disabling the Module Function Table

Modules that inherit from an interface (such as GateHwi, which inherits from IGate) by default have a generated function table that is used in supporting abstract handles for instances. For example, an API takes a handle to an IGate as one of its parameters. Because the type of the gate is abstract, it must use a function table to access the module functions for that gate. Likewise, several Memory module APIs expect an IHeap Handle as an parameter.

If the instances of one of these modules are never used in an abstract way, however, then the function table for the module is unnecessary and can be removed. For example, if none of the GateHwi instances in a system are ever cast to IGate instances, then the GateHwi function table can be disabled as follows.

```
var GateHwi = xdc.useModule('ti.sysbios.gates.GateHwi');
GateHwi.common$.fxntab = false;
```

It is generally not safe to set the common$.fxntab property to false for Heap implementation modules, because access to their function table may be needed when internal calls to Memory APIs by the Heap implementation use abstract handles.

D.2.7   Reduce the Number of atexit Handlers

By default, up to 8 System_atexit() handlers can be specified that will be executed when the system is exiting. You can save data space by reducing the number of handlers that can be set at runtime to the number you are actually intending to use. For example:

```
System.maxAtexitHandlers = 0;
```

D.3   Reducing Code Size

D.3.1   Use the Custom Build SYS/BIOS Libraries

Set the **Build-Profile** (shown when you are creating a new CCS project or modifying the **CCS General > RTSC** settings) to "release."

In the configuration settings, change the BIOS.libType parameter to BIOS.LibType_Custom. This causes the SYS/BIOS libraries to be recompiled with optimizations that reduce code size.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
```

See Section 2.3.5 for more on these settings.

D.3.2   Disabling Logging

Logging and assertion handling can be disabled with the following configuration settings:

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtimg.Diags');
Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;
BIOS.assertsEnabled = false;
BIOS.logsEnabled = false;
```

D.3.3   Setting Memory Policies

The Memory module supports different "memory policies" for creating and deleting objects. If all of the objects in an application can be statically created in the configuration script, then all of the code associated with dynamically creating instances of modules can be left out of the application. This is referred to as a static memory policy.

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Types = xdc.useModule('xdc.runtime.Types');
Defaults.common$.memoryPolicy = Types.STATIC_POLICY;
```

D.3.4 Disabling Core Features

Some of the core features of SYS/BIOS can be enabled or disabled as needed. These include the Swi, Clock, and Task modules.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.swiEnabled = false;
BIOS.clockEnabled = false;
BIOS.taskEnabled = false;
```

Applications typically enable at least one of the Swi and Task handlers. Some applications may not need to use both Swi and Task, and can disable the unused thread type.

D.3.5 Eliminating printf()

There is no way to explicitly remove printf from the application. However, the printf code and related data structures are not included if the application is free of references to System_printf(). This requires two things:

- The application code cannot contain any calls to System_printf().

- The following configuration settings need to be made to SYS/BIOS:

```
var System = xdc.useModule('xdc.runtime.System');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
SysMin.bufSize = 0;
SysMin.flushAtExit = false;
System.SupportProxy = SysMin;

//Remove Error_raiseHook, which brings System_printf
var Error = xdc.useModule('xdc.runtime.Error');
Error.raiseHook = null;
```

See the module documentation for details. Essentially, these settings will eliminate all references to the printf code.

D.3.6 Disabling RTS Thread Protection

If an application does not require the RTS library to be thread safe, it can specify to not use any Gate module in the RTS library. This can prevent the application from bringing in another type of Gate module.

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.rtsGateType = BIOS.NoLocking;
```

D.3.7   Disable Task Stack Overrun Checking

If you are not concerned that any of your Task instances will overrun their stacks, you can disable the checks that make sure the top of the stack retains its initial value. This saves on code space. See Section 3.5.3 for information about task stack sizes.

By default, stack checking is performed. Use these statements if you want to disable stack checking for all Tasks:

```
Task.checkStackFlag = false;
Task.initStackFlag = false;
```

D.4   Basic Size Benchmark Configuration Script

The basic size benchmark configuration script puts together all of these concepts to create an application that is close to the smallest possible size of a SYS/BIOS application.

The values chosen for Program.stack and Program.argSize are chosen to match the settings used in generating the DSP/BIOS 5 benchmarks in order to support a fair comparison of the two systems.

Note that in a real-world application, you would want to enable at least either Swi or Task handlers so that your application could use some threads.

This configuration script works on any target.

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var System = xdc.useModule('xdc.runtime.System');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
var Error = xdc.useModule('xdc.runtime.Error');
var Text = xdc.useModule('xdc.runtime.Text');
var Types = xdc.useModule('xdc.runtime.Types');
var Diags = xdc.useModule('xdc.runtime.Diags');
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var GateHwi = xdc.useModule('ti.sysbios.gates.GateHwi');
var HeapStd = xdc.useModule('xdc.runtime.HeapStd');
var Memory = xdc.useModule('xdc.runtime.Memory');


// Assumption: app needs SysMin at a minimum,
// but may not use printf, so buf can be zero.
SysMin.bufSize = 0;
SysMin.flushAtExit = false;
System.SupportProxy = SysMin;


// Get rid of Error_raiseHook which brings in System_printf
Error.raiseHook = null;


// Remove the heap used by SYS/BIOS
BIOS.heapSize = 0;

```

```
// arg and stack size made same as BIOS 5.00
Program.argSize = 0x4;
Program.stack = 0x400;


// Logger disabled for benchmarking
Defaults.common$.logger = null;


//Set isLoaded for Text module
Text.isLoaded = false;


// Recompile SYS/BIOS libraries with optimization
BIOS.libType = BIOS.LibType_Custom;


// Remove unneeded atexit handlers
System.maxAtexitHandlers = 0;


// Set STATIC_POLICY
Defaults.common$.memoryPolicy = Types.STATIC_POLICY;
Defaults.common$.diags_ASSERT = Diags.ALWAYS_OFF;
Defaults.common$.namedModule = false;


BIOS.swiEnabled = false;
BIOS.clockEnabled = false;
BIOS.taskEnabled = false;
BIOS.logsEnabled = false;
BIOS.assertsEnabled = false;
BIOS.rtsGateType = BIOS.NoLocking;


// App not using abstract GateHwi instances
GateHwi.common$.fxntab = false;
HeapStd.common$.fxntab = false;
```

IOM Interface

This appendix provides reference details for the IOM (I/O Mini-driver) interface.

E.1 Mini-Driver Interface Overview

The mini-driver interface specifies how to implement a mini-driver.

Functions

A mini-driver should implement the following functions:

- mdBindDev. Bind device to mini-driver.

- mdControlChan. Perform channel control command.

- mdCreateChan. Create a device channel.

- mdDeleteChan. Delete a channel.

- mdSubmitChan. Submit a packet to a channel for processing.

- mdUnBindDev. Unbind device from mini-driver.

Description

A mini-driver contains the device-specific portions of the driver. Once you create the specified functions for your mini-driver, application integrators can easily use your mini-driver through GIO channels.

The sections that follow describe how to implement the mini-driver functions in detail. Once implemented, these functions should be referenced in an interface table of type IOM_Fxns, which applications will reference to integrate the mini-driver. For example:

```
IOM_Fxns UART_FXNS = {
    mdBindDev,
    IOM_UNBINDDEVNOTIMPL,
    mdControlChan,
    mdCreateChan,
    mdDeleteChan,
    mdSubmitChan
};
```

Note: Any mini-driver functions you choose not to implement should either plug the mini-driver function table with IOM_xxxNOTIMPL, where xxx corresponds to the function name. Alternately, you may implement a function that returns a status of IOM_ENOTIMPL.

Constants, Types, and Structures

The following code can be found in <bios_install>/packages/ti/sysbios/io/IOM.h.

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


/* IOM Status Codes */
#define IOM_COMPLETED    0  /* I/O successful */
#define IOM_PENDING      1  /* I/O queued and pending */
#define IOM_FLUSHED      2  /* I/O request flushed */
#define IOM_ABORTED      3  /* I/O aborted */


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


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


/* Command codes for GIO_control and mdControlChan */
#define IOM_CHAN_RESET     0 /* reset channel only */
#define IOM_CHAN_TIMEDOUT  1 /* channel timeout occurred */
#define IOM_DEVICE_RESET   2 /* reset entire device */


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

```
#define IOM_BINDDEVNOTIMPL (IOM_TmdBindDev)IOM_mdNotImpl
#define IOM_UNBINDDEVNOTIMPL (IOM_TmdUnBindDev)IOM_mdNotImpl
#define IOM_CONTROLCHANNOTIMPL (IOM_TmdControlChan)IOM_mdNotImpl
#define IOM_CREATECHANNOTIMPL (IOM_TmdCreateChan)IOM_mdNotImpl
#define IOM_DELETECHANNOTIMPL (IOM_TmdDeleteChan)IOM_mdNotImpl
#define IOM_SUBMITCHANNOTIMPL (IOM_TmdSubmitChan)IOM_mdNotImpl


typedef struct IOM_Packet {   /* frame object */
   Queue_Elem   link;        /* queue link */
   Ptr          addr;        /* buffer address */
   SizeT        size;        /* buffer size */
   UArg         arg;         /* user argument */
   UInt         cmd;         /* mini-driver command */
   Int          status;      /* status of command */
   UArg         misc;        /* reserved for driver */
} IOM_Packet;


/* Mini-driver's callback function. */
Void (*IOM_TiomCallback)(Ptr arg, IOM_Packet *packet);
```

mdBindDev

Bind device to mini-driver

C Interface

Syntax

status = mdBindDev(*devp, devid, devParams);

Parameters

Ptr	*devp;	/* address for global device data pointer */
Int	devid;	/* device id */
Ptr	devParams;	/* pointer to config parameters */

Return Value

Int	status;	/* success or failure code */

Description

The mdBindDev function is called by SYS/BIOS during device initialization. It is called once per configured device and is called after the mini-driver's initialization function.

This function is typically used to specify device-specific global data, such as interrupts IDs and global data structures (for ROM-ability). Additional system resources may be allocated by the mini-driver at runtime.

The devp parameter provides the address where the function should place the global device data pointer.

The devid parameter is used to identify specific devices for systems that have more than one device of a specific type. For example, several McBSP mini-drivers use the devid parameter to specify which McBSP port to allocate and configure.

The devParams parameter is a pointer to the configuration parameters to be used to configure the device.

This function should return IOM_COMPLETED if it is successful. If a failure occurs, it should return one of the a negative error codes listed in section E.1, *Mini-Driver Interface Overview*. If this function returns a failure code, the SYS/BIOS initialization fails with a call to System_abort().

mdControlChan   *Perform channel control command*

C Interface

Syntax

status = mdControlChan (chanp, cmd, arg);

Parameters

Ptr	chanp;	/* channel handle */
UInt	cmd;	/* control functionality to perform */
Ptr	arg;	/* optional device-defined data structure */

Return Value

| Int | status; | /* success or failure code */ |

Description

A class driver calls this function to cause the mini-driver to perform some type of control functionality. For example, it may cause the mini-driver to reset the device or get the device status. Calling GIO_control results in execution of the appropriate mini-driver's mdControlChan function.

The chanp parameter provides a channel handle to identify the device instance.

The cmd parameter indicates which control functionality should be carried out.

```
/* Command codes for GIO_control and mdControlChan */
#define IOM_CHAN_RESET     0  /* reset channel only */
#define IOM_CHAN_TIMEDOUT  1  /* channel timeout occurred */
#define IOM_DEVICE_RESET   2  /* reset entire device */
```

The arg parameter is an optional device-defined data structure used to pass control information between the device and the application

If successful, this function should return IOM_COMPLETED. If the cmd value provided is unsupported, this function should return a status of IOM_ENOTIMPL.

mdCreateChan   *Create a device channel*

C Interface

> **Syntax**
>> status = mdCreateChan (*chanp, devp, name, mode, chanParams, cbFxn, cbArg);

> **Parameters**
>> | | | |
>> |---|---|---|
>> | Ptr | *chanp; | /* channel handle */ |
>> | Ptr | devp; | /* device global data structure */ |
>> | String | name | /* name of device instance */ |
>> | Int | mode | /* input or output mode */ |
>> | Ptr | chanParams | /*pointer to channel parameters */ |
>> | IOM_TiomCallback | cbFxn | /* pointer to callback function */ |
>> | Ptr | cbArg | /* callback function argument */ |

> **Return Value**
>> | | | |
>> |---|---|---|
>> | Int | status; | /* success or failure code */ |

Description

> A class driver calls this function to create a channel instance. Calling GIO_create results in execution of the appropriate mini-driver's mdCreateChan function.

> The chanp parameter provides an address at which this function should place a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See Section 5.2.3, *mdBindDev Function*, page 5-8 for an example.

> The devp parameter is a pointer to the device's global data structure. This is the value returned by the mini-driver's mdBindDev call.

> The name parameter is the name of the device instance. This is the remainder of the device name after getting a complete match from the SYS/BIOS device driver table. For example, this might contain channel parameters.

> The mode parameter specifies whether the device is being opened in input mode, output mode, or both. The mode may be IOM_INPUT, IOM_OUTPUT, or IOM_INOUT. If your driver does not support one or more modes, this function should return IOM_EBADMODE for unsupported modes.

> The chanParams parameter is used to pass device- or domain-specific arguments to the mini-driver.

> The cbFxn parameter is a function pointer that points to the callback function to be called by the mini-driver when it has completed a request.

> The cbArg parameter is an argument to be passed back by the mini-driver when it invokes the callback function.

> Typically, the mdCreateChan function places the callback function and its argument in the device-specific data structure. For example:

```
chan->cbFxn = cbFxn;
chan->cbArg = cbArg;
```

> If successful, this function should return IOM_COMPLETED. If unsuccessful, this function should return one of the a negative error codes listed in section E.1, *Mini-Driver Interface Overview*.

mdDeleteChan   *Delete a channel*

C Interface

Syntax
    status = mdDeleteChan (chanp)

Parameters
    Ptr   chanp;   /* channel handle */

Return Value
    Int   status;   /* success or failure code */

Description

A class driver calls this function to delete the specified channel instance. Calling GIO_delete results in execution of the appropriate mini-driver's mdDeleteChan function.

The chanp parameter provides a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See the mdBindDev topic for an example.

If successful, this function should return IOM_COMPLETED. If unsuccessful, this function should return one of the a negative error codes listed in section E.1, *Mini-Driver Interface Overview*.

mdSubmitChan — *Submit a packet to a channel for processing*

C Interface

Syntax
status = mdSubmitChan (chanp, *packet);

Parameters
```
Ptr                 chanp;        /* channel handle */
IOM_Packet          *packet;      /* pointer to IOM_Packet */
```

Return Value
```
Int                 status;       /* success or failure code */
```

Description

A class driver calls this function to cause the mini-driver to process the IOM_Packet. Calls to GIO_submit, GIO_read, GIO_write, GIO_abort, and GIO_flush result in execution of the appropriate mini-driver's mdSubmitChan function.

Note:	The mini-driver function mdSubmitChan must be written to be reentrant to allow it to be called from multiple thread contexts.

The chanp parameter provides a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See the mdBindDev topic for an example.

The packet parameter points to a structure of type IOM_Packet. This structure is defined as follows:

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

The value for the cmd code may be one of the following:

```
#define IOM_READ        0
#define IOM_WRITE       1
#define IOM_ABORT       2
#define IOM_FLUSH       3
```

Additional cmd codes may be added for domain-specific commands. Such codes should be constants with values greater than 127. See the iom.h file for these cmd codes.

If the cmd code is IOM_READ or IOM_WRITE, this function should queue the packet on the pending list. If the cmd code is IOM_ABORT, this function should abort both read and write packets. If the cmd code is IOM_FLUSH, this function should complete queued writes, but abort queued reads.

If this function successfully completes a read or write IOM_Packet request, it should return IOM_COMPLETED. If this function queues up a read or write request, it should return IOM_PENDING. If this function successfully aborts or flushes a packet, it should return IOM_COMPLETED. If unsuccessful, this function should return one of the a negative error codes listed in Section E.1.

mdUnBindDev *Unbind device from mini-driver*

C Interface

Syntax
    status = mdUnBindDev(devp);

Parameters
    Ptr   devp;   /* global device data pointer */

Return Value
    Int   status;   /* success or failure code */

Description

This function should free resources allocated by the mdBindDev function.

Currently, this function is not called as a result of any GIO functions. It may be used in the future to support dynamic device driver loading and unloading.

The devp parameter is a pointer to the device's global data structure. This is the value returned by the mini-driver's mdBindDev call.

If successful, this function should return IOM_COMPLETED. If unsuccessful, this function should return a negative error code.

Index

compiler options   47
Configuration Results view   35
configuration script
   XDCtools technology used for   12
configuration size
   basic size benchmark configuration script   223
   default   212
configuro tool   42
counting semaphores   108
Create hook function
   for hardware interrupts   61, 62
   for software interrupts   76, 77
   for tasks   88, 89
create() function
   Clock module   127
   GIO module   187
   Hwi module   60
   Mailbox module   121
   memory policy   145
   Semaphore module   108
   Swi module   68
   Task module   84, 205
   Timer module   164
critical regions, protecting (see gates)
custom libType   47
customCCOpts parameter   47
Cygwin shell   43

D

data size, reducing   219
debug build profile   46
debug libType   47
debugging   173
dec() function, Swi module   71, 72, 74
Defaults module   17
delegate modules   170
Delete hook function
   for hardware interrupts   62
   for software interrupts   76, 77
   for tasks   88, 89
delete() function
   GIO module   189
   Mailbox module   121
   memory policy   145
   Semaphore module   108
   Swi module   76
   Task module   84, 206
dequeue() function, Queue module   123
deterministic performance   149
DEV module   183
   configuring   183
DEV_create() function   184
DEV_delete() function   184
device drivers   182
device-specific modules   170
Diags module   17, 173
   optimizing   178
disable() function
   Cache interface   169

   Hwi module   56, 159, 203
   Swi module   56, 76, 204
   Task module   56, 205
disableInterrupt() function, Hwi module   56
dispatcher   162
   optimization   179
documents (resources)   21
download   21
drivers   182
   table of names   183
DSP/BIOS 5
   size benchmark comparisons   211
dynamic configuration   12
dynamic module application sizes   217

E

enable() function
   Cache interface   169
   Hwi module   60, 159, 203
   Swi module   204
   Task module   205
End hook function
   for hardware interrupts   62
   for software interrupts   76, 77
enqueue() function, Queue module   123
enter() function, Gate module   119
error block   175
Error module   17, 173, 175
Error_check() function   176
Error_init() function   175
errors
   finding in configuration   40
   handling   175
Event module   18, 113
   used with Mailbox   122
Event object   13
Event_create() function   114, 117
Event_pend() function   113, 118, 122
Event_post() function   113, 118
events   113
   associating with mailboxes   122
   channels used with   195
   creating dynamically   114
   creating statically   114
   examples of   114
   posting   113, 114
   posting implicitly   116
   waiting on   113, 114
execution states of tasks   85
   Task_Mode_BLOCKED   85, 86
   Task_Mode_INACTIVE   85
   Task_Mode_READY   85, 86
   Task_Mode_RUNNING   85, 86
   Task_Mode_TERMINATED   85, 86
execution states of threads   54
Exit hook function, for tasks   88, 90
exit() function, Task module   86
eXpress Dsp Components (see XDCtools)
external memory   132

F

family-specific modules   170
flush() function, GIO module   193
forum, E2E community   21
fragmentation, memory   149
free() function   148
  C++   18
  Memory module   146
function names   19
functions
  (see also hook functions)

G

Gate module   17, 119
Gate object   13
Gate_enter() function   119
Gate_leave() function   119
GateHwi module   120
GateHwi_create() function   119
GateMutex module   120
GateMutexPri module   120, 121
gates   119
  preemption-based implementations of   120
  priority inheritance with   121
  priority inversion, resolving   121
  semaphore-based implementations of   120
GateSwi module   120
GateTask module   120
GCC (GNU Compiler Collection)   43
get() function, Queue module   124
getFreq() function, Timer module   165
getHookContext() function
  Hwi module   59
getHookContext() function, Swi module   77
getNumTimers() function, Timer module   165
getStatus() function, Timer module   165
getTicks() function, Clock module   127
getTrigger() function, Swi module   72
GIO module   182
  APIs   186
  channels   183
  configuring   185
  synchronization   196
GIO_abort() function   193
GIO_control() function   189
GIO_create() function   187, 194
GIO_delete() function   189
GIO_flush() function   193
GIO_issue() function   191
GIO_Params structure   187
GIO_prime() function   192
GIO_read() function   189
GIO_reclaim() function   192
GIO_submit() function   190
GIO_write() function   190
global namespace   41
gmake utility   43, 198

H

hal package   170
hardware interrupts   52, 59
  application size   217
  compared to other types of threads   53
  creating   60
  disabling   159
  enabled at startup   50
  enabling   159
  hook functions for   61, 63
  interrupt dispatcher for   162, 179
  priority of   55
  registers saved and restored by   162
  timing benchmarks for   203
  when to use   53
head() function, Queue module   124
Heap implementation
  used by Memory module   148
HeapBuf module   18, 148, 150, 179
HeapMem module   18, 148, 149, 179
  system heap   145
HeapMultiBuf module   18, 148, 151, 179
heaps   148
  default   147
  HeapBuf implementation   150
  HeapMem implementation   149
  HeapMultiBuf implementation   151
  implementations of   148
  module-specific   147
  optimizing   179
  system   145
HeapTrack module   148, 154
help system   22
hook context pointer   58
hook functions   54, 58
  for hardware interrupts   61, 63
  for software interrupts   76
  for tasks   88, 90
hook sets   58
Hwi dispatcher   162
Hwi module   18, 59, 61, 171
  logging and performance   179
Hwi threads (see hardware interrupts)
Hwi_create() function   60
Hwi_delete() function
  hook function   62
Hwi_disable() function   56, 129
Hwi_disableInterrupt() function   56
Hwi_enable() function   60
Hwi_getHookContext() function   59, 62, 65
Hwi_plug() function   60
Hwi_restore() function   129
Hwi_setHookContext() function   62, 64

I

I/O modules   182
ICache interface   169
Idle Loop   52, 101
  compared to other types of threads   53
  priority of   55

X

xdc.cfg package   17
xdc.runtime package   17
xdc.runtime.Gate module   119
xdc.runtime.knl package   196
xdc.useModule() statement   16, 30
XDCscript   22
XDCtools
  build settings   42
  command line   42
  configuration using   12

other documentation   21
relationship to SYS/BIOS   13
XGCONF   30
  opening   31
  saving   32
  views   33

Y

yield() function, Task module   207
yielding   56, 86

Products		Applications	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		