SYS/BIOS (TI-RTOS Kernel) v7.x

User's Guide

Literature Number: SPRUIX7
November 2020

Texas Instruments

Contents

Preface 8

1 About SYS/BIOS 10
- 1.1 What is SYS/BIOS? 11
- 1.2 How is SYS/BIOS Installed? 11
- 1.3 SYS/BIOS as a Set of Packages 12
- 1.4 SYS/BIOS Packages and APIs 12
- 1.5 Configuring SYS/BIOS Using SysConfig 13
- 1.6 SYS/BIOS Object Creation 15
  - 1.6.1 POSIX Thread Support 17
- 1.7 Using C++ with SYS/BIOS 17
  - 1.7.1 Memory Management 17
  - 1.7.2 Name Mangling 17
  - 1.7.3 Calling Class Methods from the Configuration 18
  - 1.7.4 Class Constructors and Destructors 18
- 1.8 For More Information 19

2 SYS/BIOS Configuration and Building 20
- 2.1 Creating Projects that Use SYS/BIOS 21
- 2.2 Configuring SYS/BIOS Applications 22
  - 2.2.1 Opening a Configuration File with SysConfig 23
  - 2.2.2 Performing Tasks with SysConfig 24
  - 2.2.3 Adding and Removing Modules in the Configuration 24
  - 2.2.4 Setting Property Values in the Configuration 25
  - 2.2.5 Saving the Configuration 25
  - 2.2.6 Viewing Configuration Code 26
  - 2.2.7 Finding and Fixing Errors 27
- 2.3 Building SYS/BIOS Applications 28
  - 2.3.1 Understanding the Build Flow 28

3 Threading Modules 29
- 3.1 SYS/BIOS Startup Sequence 30
- 3.2 Overview of Threading Modules 31
  - 3.2.1 Types of Threads 32
  - 3.2.2 Choosing Which Types of Threads to Use 33
  - 3.2.3 A Comparison of Thread Characteristics 34
  - 3.2.4 Thread Priorities 35
  - 3.2.5 Yielding and Preemption 36
  - 3.2.6 Hooks 38
- 3.3 Hardware Interrupts 40
  - 3.3.1 Creating Hwi Objects 40
  - 3.3.2 Hardware Interrupt Nesting and System Stack Size 41
  - 3.3.3 Hwi Hooks 41
- 3.4 Software Interrupts 46

- 3.4.1 Creating Swi Objects 47
- 3.4.2 Setting Software Interrupt Priorities 48
- 3.4.3 Software Interrupt Priorities and System Stack Size 48
- 3.4.4 Execution of Software Interrupts 48
- 3.4.5 Using a Swi Object's Trigger Variable 49
- 3.4.6 Benefits and Tradeoffs 52
- 3.4.7 Synchronizing Swi Functions 53
- 3.4.8 Swi Hooks 53
- 3.5 Tasks 60
- 3.5.1 Creating Tasks 61
- 3.5.2 Task Execution States and Scheduling 61
- 3.5.3 Task Stacks 63
- 3.5.4 Testing for Stack Overflow 64
- 3.5.5 Task Hooks 64
- 3.5.6 Task Yielding for Time-Slice Scheduling 71
- 3.6 The Idle Loop 77
- 3.7 Example Using Hwi, Swi, and Task Threads 78

4 Synchronization Modules 82
- 4.1 Semaphores 83
- 4.1.1 Semaphore Example 84
- 4.2 Event Module 88
- 4.2.1 Implicitly Posted Events 91
- 4.3 Gates 94
- 4.3.1 Preemption-Based Gate Implementations 95
- 4.3.2 Semaphore-Based Gate Implementations 96
- 4.3.3 Priority Inversion 96
- 4.3.4 Setting the SYS/BIOS Gate Type 97
- 4.4 Mailboxes 97
- 4.5 Queues 99
- 4.5.1 Basic FIFO Operation of a Queue 99
- 4.5.2 Iterating Over a Queue 100
- 4.5.3 Inserting and Removing Queue Elements 100
- 4.5.4 Atomic Queue Operations 100

5 Timing Services 101
- 5.1 Overview of Timing Services 102
- 5.2 Clock 102
- 5.3 Timer Module 105
- 5.4 Seconds Module 105
- 5.5 Timestamp Module 106

6 Support Modules 108
- 6.1 Modules for Application Support and Management 109
- 6.2 BIOS Module 109
- 6.3 System Module 110
- 6.3.1 SysMin Module 111
- 6.3.2 SysCallback Module 112
- 6.4 Startup Module 112
- 6.5 Error Module 112

7 Memory **114**
7.1 Background........ 115
7.2 Stacks........ 115
7.2.1 System Stack........ 115
7.2.2 Task Stacks........ 115
7.3 Dynamic Memory Allocation........ 116
7.3.1 Specifying the Default System Heap........ 116
7.3.2 Using the Memory Module........ 117
7.3.3 Using malloc() and free()........ 117
7.4 Heap Implementations........ 118
7.4.1 HeapMin........ 119
7.4.2 HeapMem........ 119
7.4.3 HeapBuf........ 120
7.4.4 HeapMultiBuf........ 121
7.4.5 HeapCallback........ 123
7.4.6 HeapTrack........ 123

8 Hardware Abstraction Layer **125**
8.1 Hardware Abstraction Layer APIs........ 126
8.2 Hwi Module........ 127
8.2.1 Associating a C Function with a System Interrupt Source........ 127
8.2.2 Hwi Instance Parameters........ 127
8.2.3 Creating a Hwi Object Using Non-Default Instance Parameters........ 128
8.2.4 Enabling and Disabling Interrupts........ 128
8.2.5 A Simple Example Hwi Application........ 129
8.2.6 The Interrupt Dispatcher........ 130
8.2.7 Registers Saved and Restored by the Interrupt Dispatcher........ 131

9 Optimization **132**
9.1 Overview........ 133
9.2 Load Module........ 133
9.2.1 Load Module Configuration........ 133
9.2.2 Obtaining Load Statistics........ 134
9.3 Performance Optimization........ 134
9.3.1 Choosing a Heap Manager........ 134
9.3.2 Hwi Module Configuration........ 135
9.3.3 Stack Checking........ 135
9.3.4 Zero Latency Interrupts........ 135
9.4 Memory Optimization........ 135
9.4.1 Reducing Data Size........ 135
9.4.2 Reducing Code Size........ 136
9.4.3 Basic Size Benchmark Configuration........ 137

A Device Addendum........ **138**
A.1 ARM Cortex-M Introduction........ 139
A.2 Cortex-M Hardware Interrupt (Hwi) Handling........ 139
A.2.1 Hwi MaskingOptions and Priorities........ 139
A.2.2 Zero Latency Interrupt Support........ 140
A.3 Cortex-M Operating Modes and Stack Usage........ 141
A.3.1 Operating Modes........ 141
A.3.2 Stacks........ 141

A.4 Cortex-M Timer Usage by SYS/BIOS Modules . . . 142
A.4.1 High-Level SYS/BIOS Timing Modules . . . 142
A.4.2 Low-Level SYS/BIOS Peripheral Timer Implementations . . . 142
A.4.3 Clock Module . . . 142
A.4.4 HAL Timer Module . . . 144
A.4.5 HAL Seconds Module . . . 144
A.4.6 Timestamp Module . . . 144
A.4.7 POSIX Timing Services . . . 144
A.5 Cortex-M Timer Device Details . . . 145
A.5.1 CC13xx/CC26xx . . . 145
A.5.2 CC32xx . . . 146
A.6 FAQ . . . 147
A.6.1 Configure Divide-by-Zero to be an Exception . . . 147

B Timing Benchmarks . . . 148
B.1 Timing Benchmarks . . . 149
B.2 Interrupt Latency . . . 149
B.3 Hwi-Hardware Interrupt Benchmarks . . . 149
B.4 Swi-Software Interrupt Benchmarks . . . 150
B.5 Task Benchmarks . . . 151
B.6 Semaphore Benchmarks . . . 153

C Size Benchmarks . . . 156
C.1 Overview . . . 157
C.2 Constructed Application Sizes . . . 158
C.2.1 Constructed Task Application . . . 158
C.2.2 Constructed Semaphore Application . . . 158
C.2.3 Constructed Mutex Application . . . 159
C.2.4 Constructed Clock Application . . . 159
C.3 Created Module Application Sizes . . . 160
C.3.1 Created Task Application . . . 160
C.3.2 Created Semaphore Application . . . 160
C.3.3 Created Mutex Application . . . 161
C.3.4 Created Clock Application . . . 161
C.4 POSIX Application Sizes . . . 162
C.4.1 POSIX Pthread Application . . . 162
C.4.2 POSIX Semaphore Application . . . 162
C.4.3 POSIX Mutex Application . . . 163
C.4.4 POSIX Timer Application . . . 163

Index . . . 164

List of Figures

3-1 Thread Priorities 35
3-2 Preemption Scenario 37
3-3 Using Swi_inc() to Post a Swi 50
3-4 Using Swi_andn() to Post a Swi 51
3-5 Using Swi_dec() to Post a Swi 51
3-6 Using Swi_or() to Post a Swi. 52
3-7 Execution Mode Variations 62
4-1 Trace Window Results from Example 4-4 88
B–1 Hardware Interrupt to Blocked Task 150
B–2 Hardware Interrupt to Software Interrupt 150
B–3 Post of Software Interrupt Again 151
B–4 Post Software Interrupt without Context Switch. 151
B–5 Post Software Interrupt with Context Switch 151
B–6 Create a New Task without Context Switch. 152
B–7 Create a New Task with Context Switch 152
B–8 Set a Task's Priority without a Context Switch. 152
B–9 Lower the Current Task's Priority, Context Switch. 153
B–10 Raise a Ready Task's Priority, Context Switch 153
B–11 Task Yield 153
B–12 Post Semaphore, No Waiting Task 154
B–13 Post Semaphore, No Context Switch. 154
B–14 Post Semaphore with Task Switch 154
B–15 Pend on Semaphore, No Context Switch 154
B–16 Pend on Semaphore with Task Switch 155

List of Tables

1–1 Packages and Modules Provided by SYS/BIOS 12
1–2 Benefits of Various Object Creation Styles 16
3-1 Comparison of Thread Characteristics 34
3-2 Thread Preemption 36
3–3 Hook Functions by Thread Type 38
3-4 Swi Object Function Differences 49
3–5 Task Stack Use by Target Family 63
5–1 Timeline for One-shot and Continuous Clocks 104
7–1 Heap Implementation Comparison 118
C–1 SYS/BIOS 6 Benchmark Applications 157

Read This First

About This Manual

This manual describes SYS/BIOS, which is an OS kernel provided with the core SimpleLink SDK. SYS/BIOS is called “TI-RTOS Kernel” in some documents and was previously called DSP/BIOS. This document was published in conjunction with the release of SYS/BIOS 7.0; it may be used with later versions of SYS/BIOS if changes to the software do not cause this document to become incorrect.

SYS/BIOS 7.0 is provided as part of the SimpleLink SDK for SimpleLink targets only. SYS/BIOS 7.0 uses SysConfig .syscfg files for configuration. Previous versions of SYS/BIOS used XGCONF with .cfg files for configuration. No automated migration path is provided to convert .cfg files to .syscfg files.

For non-SimpleLink device families (C28x, C66x, C7x, Cortex-A, and others), you should continue to use SYS/BIOS 6.x, which is documented in SPRUEX3V.

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 <ti/sysbios/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 1.8.

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, SimpleLink, 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.

November 2, 2020

Texas Instruments

Chapter 1

SPRUIX7—November 2020

About SYS/BIOS

This chapter provides an overview of SYS/BIOS and describes its general organization.

Topic | Page

1.1 What is SYS/BIOS? 11
1.2 How is SYS/BIOS Installed? 11
1.3 SYS/BIOS as a Set of Packages 12
1.4 SYS/BIOS Packages and APIs 12
1.5 Configuring SYS/BIOS Using SysConfig 13
1.6 SYS/BIOS Object Creation 15
1.7 Using C++ with SYS/BIOS 17
1.8 For More Information 19

1.1 What is SYS/BIOS?

SYS/BIOS is a scalable real-time kernel. It is designed for use in applications that require real-time scheduling and synchronization or real-time instrumentation. SYS/BIOS provides preemptive multi-threading, hardware abstraction, and real-time analysis. SYS/BIOS helps minimize memory and CPU requirements on the target.

SYS/BIOS is sometimes called *TI-RTOS Kernel*. Previously, it was called DSP/BIOS. SYS/BIOS includes support for the TI Code Generation Tools, IAR, and GNU compiler tool chains.

SYS/BIOS provides the following benefits:

- SYS/BIOS objects are configured dynamically using module-specific create() or construct() methods.
- 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.
- 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.
- 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 Installed?

SYS/BIOS is installed as part of the *SimpleLink SDK*. Separate SimpleLink SDK installers are available for use with various SimpleLink device families. SYS/BIOS requires no up-front or run-time license fees.

SYS/BIOS is not provided as part of the Code Composer Studio installation. SYS/BIOS is provided with full source code.

This document refers to the directory where the SimpleLink SDK is installed as SDK_INSTALL_DIR. This directory has a path like `C:\ti\simplelink_<target>_#_##_##_##`, where C:\ti may be different if you installed CCS in some other location, `<target>` is the device family, and `#` is a digit in the version number.

Other documents may refer to BIOS_INSTALL_DIR as the directory where SYS/BIOS is installed. This environment variable is no longer defined, because SYS/BIOS is integrated into the SDK's `kernel`, `source`, `docs`, and `examples` directories. To use such other documents, treat BIOS_INSTALL_DIR as a pointer to a location such as `C:\ti\simplelink_<target>_#_##_##_##\kernel\tirtos`.

1.3 SYS/BIOS as a Set of Packages

SYS/BIOS is organized into modules. Modules are grouped together in directories based on functionality categories. For example, kernel-related modules reside in the `ti/sysbios/knl` directory, and heap-related modules reside in the `ti/sysbios/heaps` directory. (Previous versions of SYS/BIOS called these directories "packages" and that name remains in use in the directory hierarchy.)

SYS/BIOS uses a hierarchical package-naming convention; each level is separated by a period ("."). The top level is `ti.sysbios`, which is followed by module and submodule names. For example, `ti.sysbios.knl.Task`.

These names reflect the physical layout of the packages 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:

```
SDK_INSTALL_DIR\kernel\tirtos\packages\ti\sysbios\knl
```

See Section 1.6 for a partial list of the packages provided by SYS/BIOS.

1.4 SYS/BIOS Packages and APIs

SYS/BIOS provides the following packages:

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

Package	Description	See
ti.sysbios.family.*	Contains target/device-specific functions.	Section
ti.sysbios.gates	Contains several implementations of the IGateProvider interface for use in various situations. These include GateHwi, GateSwi, GateTask, GateMutex, and GateMutexPri.	Section 4.3
ti.sysbios.hal	Contains the Hwi, Timer, and Seconds modules	Section 8.2, Section 5.3, Section 5.4
ti.sysbios.heaps	Provides several heap implementations for use with the ti/sysbios/runtime/Memory module. These include HeapBuf (fixed-size buffers), HeapMem (variable-sized buffers), HeapMultiBuf (multiple fixed-size buffers), and HeapCallback (customizable heap interface).	Chapter 7
ti.sysbios.knl	Contains modules for the SYS/BIOS kernel, including Swi, Task, Idle, and Clock. Also contains modules related to inter-process communication: Event, Mailbox, and Semaphore.	Chapter 3, Chapter 4, Chapter 5
ti.sysbios.runtime	Contains several modules that provide basic system services your SYS/BIOS application will need to operate successfully. These include the Error, Memory, Startup, System, and Timestamp modules.	Section 6.5, Section 7.3, Section 6.4, Section 6.3, Section 5.5
ti.sysbios.utils	Contains the Load module, which provides global CPU load as well as thread-specific load.	Section 9.2

Each SYS/BIOS package provides one or more modules. Each module, in turn, provides APIs for working with that module. APIs have function names of the form *Module_actionDescription*(). For example, Task_setPri() sets the priority of a Task thread.

In order to use SYS/BIOS modules, your application code must include the SYS/BIOS header file each module used. For example:

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

The API functions provided by each module differ.

1.5 Configuring SYS/BIOS Using SysConfig

SYS/BIOS is configured using SysConfig, an easy-to-use tool for configuring pins, peripherals, radios, subsystems, and other components. SysConfig is delivered integrated with Code Composer Studio (CCS), as a standalone installer, or as part of the CCS Cloud tools. SYS/BIOS with SysConfig integration should be used with CCS v10.1 or higher.

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 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.

For information about using SysConfig, see the following:

- SysConfig product page
- SysConfig Basics
- *SimpleLink SDK User's Guide* (linked to by the Documentation Overview in your SimpleLink SDK installation)
- E2E article on SysConfig

SysConfig manages a number of files related to your application's configuration. Files that contain configuration related to SYS/BIOS are:

- ti_sysbios_config.h
- ti_sysbios_config.c
- <project>.syscfg

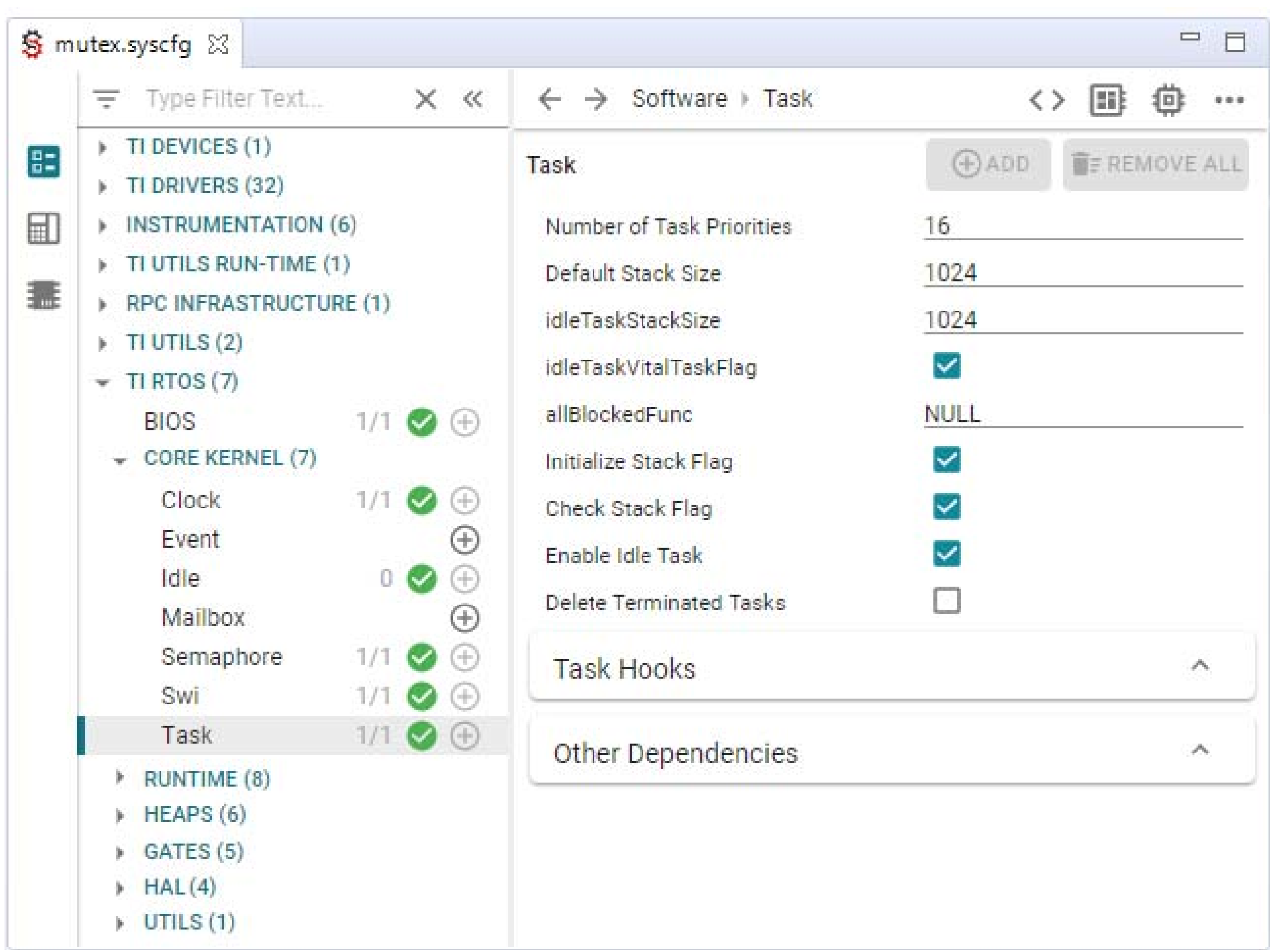
It is possible to edit these files with a text editor. However, SysConfig performs checking to prevent inconsistencies and conflicts. Therefore, it is recommended that you use the SysConfig interface to configure applications unless you are experienced with the generated configuration files.

You should not edit the `ti_sysbios_config.h` and `ti_sysbios_config.c` files using a text editor, because SysConfig will generate new files and overwrite your changes when the <project>.syscfg file is changed or the project is rebuilt.

For example, to configure SYS/BIOS Task threads, follow these steps:

1. Open the <project>.syscfg file in your project.
2. Expand the **TI RTOS** category and then the **Core Kernel** category.
3. Select **Task** and set properties for the Task module.
4. Additional information is available for most properties. Hover your mouse cursor over a property to see the full property name. Click the small, circled **?** icon to see additional help.
5. When you save the <project>.syscfg file and build the project, the ti_sysbios_config.h and ti_sysbios_config.c files are generated, compiled, and linked with the rest of the project.

The SysConfig UI for managing Task properties looks like the following:



1.6 SYS/BIOS Object Creation

Several modules support the creation of instance objects during runtime *Module*_create() and *Module*_construct() calls. Such modules include Hwi, Task, Swi, Semaphore, Mailbox, Queue, Event, Clock, Timer, and various types of Gate and Heap modules. For example, the Task module allows you to create several Task objects. Each Task object corresponds to a thread that has its own function, priority, and timing.

Instance objects can be created in the following ways.

- ***Module*_create()** The create APIs require a heap to allow dynamic memory allocation of the object. For example, this code fragment creates and posts a Semaphore object:

```
#include <ti/sysbios/knl/Semaphore.h>
Semaphore_Handle semaphore0;
Semaphore_Params semaphoreParams;

...

Semaphore_Params_init(&semaphoreParams);
semaphoreParams.mode = Semaphore_Mode_BINARY;
semaphore0 = Semaphore_create(0, &semaphoreParams, Error_IGNORE);
if (semaphore0 == NULL) {
    ... }
Semaphore_post(semaphore0);
```

  Objects that were created with a *Module*_create() API should be destroyed with the corresponding *Module*_delete() API.

- ***Module*_construct()** The construct APIs must be passed an object structure instead of dynamically allocating the object from a heap. Avoiding dynamic memory allocation helps reduce the code footprint. Some *Module*_construct() APIs may allocate memory internally. For example, Task_construct() allocates a task stack if you do not provide one. For example, this code fragment constructs and posts a Semaphore object:

```
#include <ti/sysbios/knl/Semaphore.h>
Semaphore_Struct semaphore0Struct;
Semaphore_Params semaphoreParams;

...

Semaphore_Params_init(&semaphoreParams);
semaphoreParams.mode = Semaphore_Mode_BINARY;
Semaphore_construct(&semaphore0Struct, 0, &semaphoreParams);
...
Semaphore_post(Semaphore_handle(&semaphore0Struct));
```

  Objects that were created with a *Module*_construct() API should be destroyed with the corresponding *Module*_destruct() API.

The SYS/BIOS examples use the *Module*_construct() mechanism. However, each style of object creation has its own pros and cons as listed in Table 1–2.

Table 1–2. Benefits of Various Object Creation Styles

	Module_create()	Module_construct()
Programming Flow	Simple	Application must manage object Structs (instead of just Handles). Most construct calls cannot fail (see specific APIs for details).
Dynamic Memory Allocation	Yes	Goal is to minimize or eliminate dynamic memory allocation.
Code Footprint	Larger	Smaller
Object Destruction	Yes, with *Module*_delete()	Yes, with *Module*_destruct()

See Appendix C of this document for size comparisons.

Notice that each of the methods of object creation uses a *Module*_Params structure. These structures contain instance parameters to control how the instance behaves. For example, the Semaphore_Params structure is defined by SYS/BIOS as follows:

```
typedef struct Semaphore_Params {    // Instance params structure
    Event_Handle      event;         // Event instance to use if non-NULL
    unsigned int      eventId;       // eventId if using Events
    Semaphore_Mode    mode;          // Semaphore mode: COUNTING or BINARY
} Semaphore_Params;
```

The following application code creates the Semaphore parameters structure, initializes the structure with the default parameters, and sets the “mode” parameter to BINARY to specify that this is a binary semaphore. Following this code, you would use either Semaphore_create() or Semaphore_construct() to create an instance.

```
#include <ti/sysbios/knl/Semaphore.h>
Semaphore_Handle semaphore0;
Semaphore_Params semParams;

Semaphore_Params_init(&semaphoreParams);
semParams.mode = Semaphore_Mode_BINARY;
semaphore0 = Semaphore_create(0, &semParams, Error_IGNORE);
if (semaphore0 == NULL) {
    ...
}
```

For information about the parameter structure and its individual parameters for any module, see the Doxygen API Reference described in Section 1.8.

The Runtime Object View (ROV) tool inn CCS show objects created with both *Module*_create() and *Module*_construct() APIs.

1.6.1 POSIX Thread Support

POSIX Thread support is available in many of the TI software SDKs. POSIX support is provided separately and is documented as part of the SDK documentation.

1.7 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++.

1.7.1 Memory Management

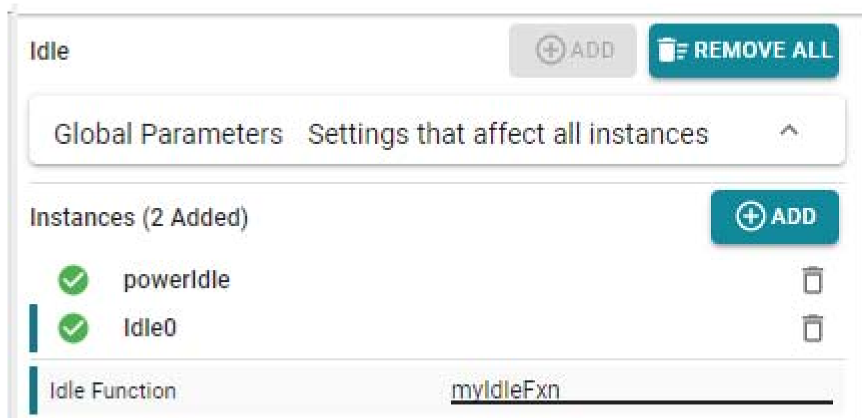
The functions new and delete are the C++ operators for dynamic memory allocation and deallocation. For TI targets, these operators use malloc() and free(). SYS/BIOS provides reentrant versions of malloc() and free() that internally use the ti.sysbios.runtime.Memory module and the ti.sysbios.heaps.HeapMem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

1.7.2 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 identify function names within SysConfig for Idle functions, hook functions for the Hwi, Swi, Task, Heap modules, and callback functions for the HeapCallback and SysCallback modules.

For example, suppose you reference a C++ function in SysConfig like this:



In the C++ source file, you should declare functions that are referenced in SysConfig using an extern “C” block as shown in the following code fragment. The extern “C” blocks are not subject to name mangling, so they allow functions identified within the SysConfig configuration to name functions in the C++ code.

```
extern "C" {
void myIdleFxn(void);
} // end extern "C"

void myIdleFxn(void)
{
    // actions performed by function
    return;
}
```

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
}
```

1.7.3 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.

1.7.4 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 Doxygen API Reference.

1.8 For More Information

API reference information for SYS/BIOS APIs is available in the following locations:

- **Doxygen API Reference:** View `SDK_INSTALL_DIR/docs/tirtos/doxygen.html` in your browser for C API information.
- **SysConfig help:** Hover your mouse cursor over a property to see the full property name. Click the small, circled **?** icon to see additional help.

You can use the following additional sources to learn more about the SYS/BIOS, the SimpleLink SDK, and Code Composer Studio:

- **SYS/BIOS (TI-RTOS Kernel)**
  - SysConfig Basics in TI Resource Explorer
  - Runtime Object View (ROV) User's Guide in TI Resource Explorer
  - TI's E2E Community including the SYS/BIOS (TI-RTOS) FAQ thread
  - TI-RTOS Kernel Product Folder on TI.com
  - Embedded Software Download Page
- **SimpleLink SDK**
  - Documentation Overview (`SDK_INSTALL_DIR/docs/Documentation_Overview.html`)
  - *SimpleLink MCU SDK User's Guide* (HTML file included in the SDK installation)
  - TI Resource Explorer (online or installed within CCS) contains links to many other documents
- **Code Composer Studio (CCS)**
  - *CCS online help.* Choose **Help > Help Contents** in CCS.
  - Code Composer forum on TI's E2E Community

TEXAS INSTRUMENTS

Chapter 2

SPRUIX7—November 2020

SYS/BIOS Configuration and Building

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

Topic **Page**

2.1 Creating Projects that Use SYS/BIOS 21

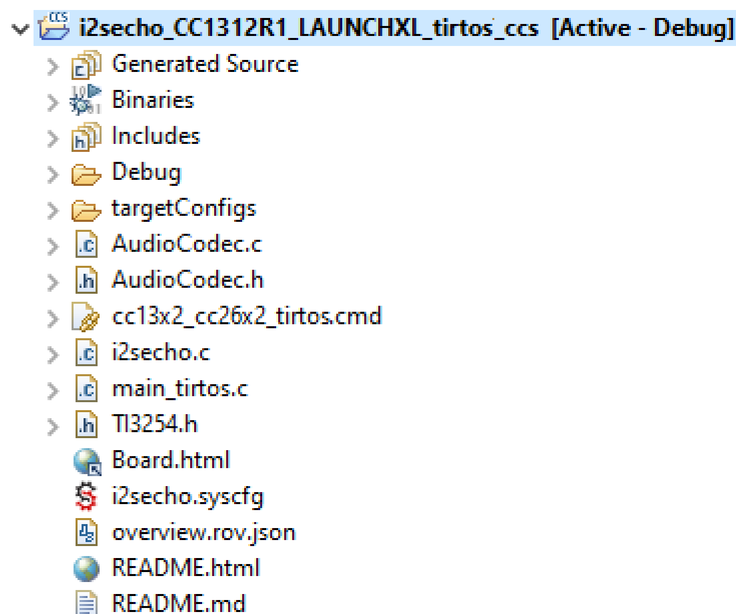
2.2 Configuring SYS/BIOS Applications 22

2.3 Building SYS/BIOS Applications 28

2.1 Creating Projects that Use SYS/BIOS

You can create a CCS project that allows you to use SYS/BIOS by importing an example provided with the SimpleLink SDK. To import an example, follow these steps:

1. In Code Composer Studio, choose **Project > Import CCS Projects**.
2. In the Import CCS Projects dialog, click **Browse** next to “Select search-directory”.
3. Browse to the folder directory where you installed the SimpleLink SDK, then browse within that tree to either of the following. There are projects available for use with CCS, IAR, and GCC.
   - `examples\rtos\<target>\demos\<app>\tirtos`
   - `examples\rtos\<target>\drivers\<app>\tirtos`
4. Click **Select Folder**.
5. Check the boxes next to projects you want to import. Versions of each project that use the TI compiler, GNU compiler, or IAR toolchain are available.
6. Click **Finish** to import the selected projects. You can expand the project to view or change the source code and configuration file.



7. Build the project using CCS. If you want to change any build options, right click on the project and select **Properties** from the context menu.
8. Connect to your target and load the program.

2.2 Configuring SYS/BIOS Applications

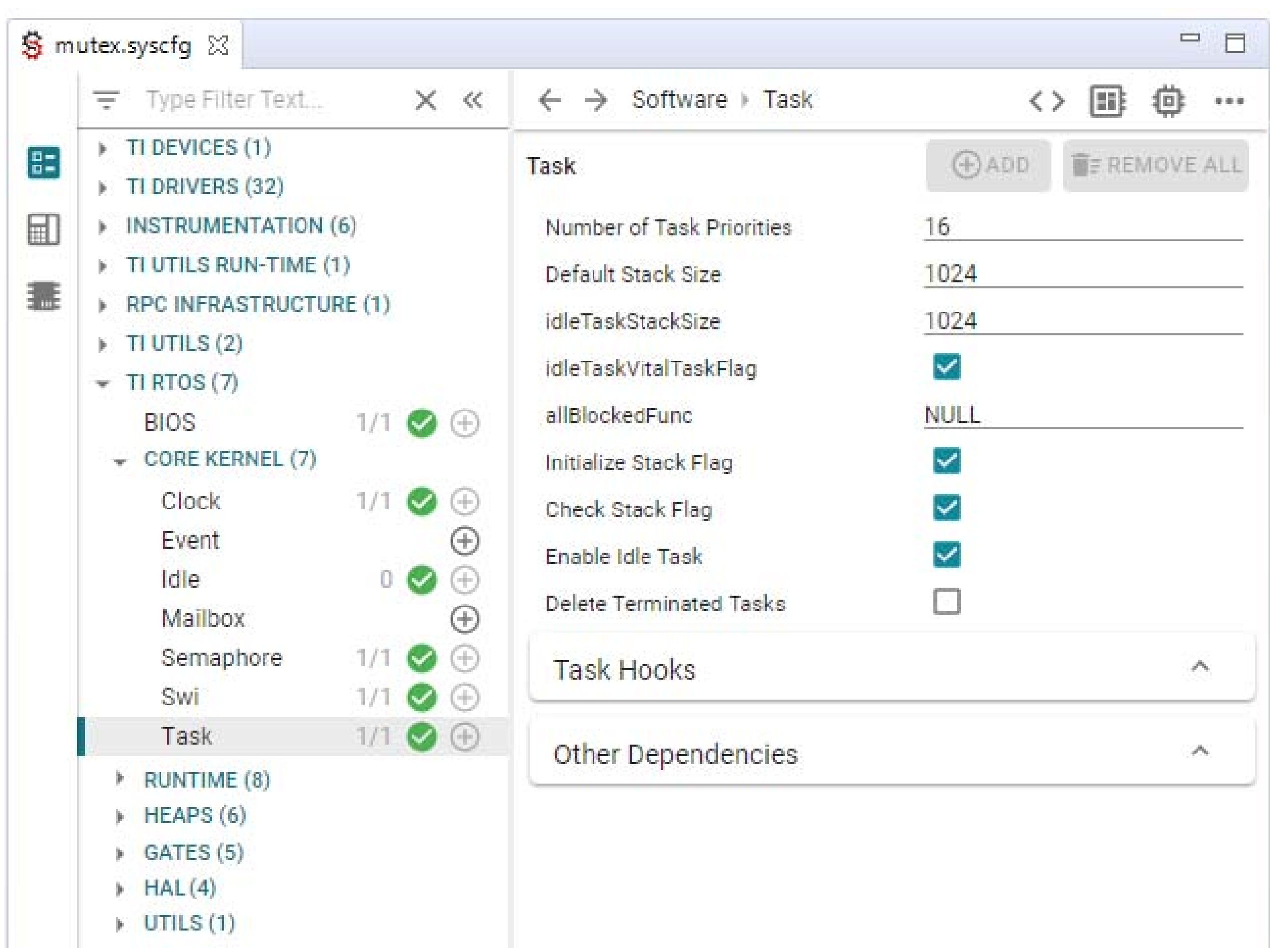
You configure SYS/BIOS applications using the SysConfig graphical editor, which modifies the *.syscfg configuration file, which is stored as JavaScript. When you build an application, the *.syscfg file is used to generate C code that is compiled and linked as part of the application.

The SysConfig editor is available as an integrated tool within the CCS or IAR IDE environments. You can also use SysConfig as a standalone tool outside of an IDE. See SysConfig Basics for more information.

You can edit the `<project>.syscfg` file using a text editor outside SysConfig. However, SysConfig performs checking to prevent inconsistencies and conflicts. Therefore, it is recommended that you use the SysConfig interface to configure applications unless you are experienced with the generated configuration files. You should not edit the `ti_sysbios_config.h` and `ti_sysbios_config.c` files with a text editor, because SysConfig will generate new files and overwrite your changes when the <project>.syscfg file is changed or the project is rebuilt.

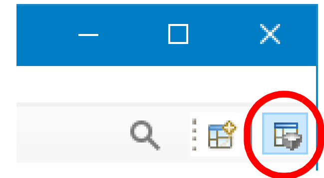
Within SysConfig, you can enable use of modules and set module-wide parameters. Instances for a module must be created using runtime C code.

For example, the SysConfig UI for managing Task properties looks like the following:

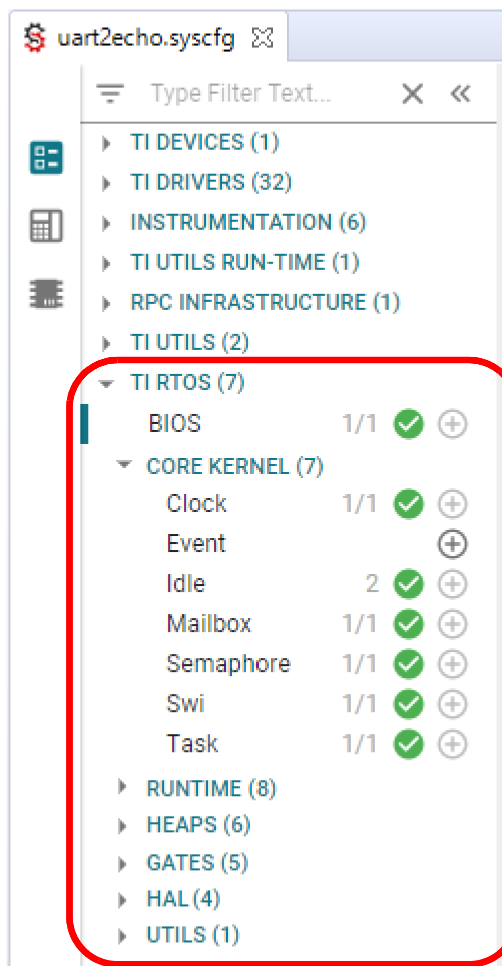


2.2.1 Opening a Configuration File with SysConfig

To open SysConfig using Code Composer Studio, follow these steps:



1. Make sure you are in the **CCS Edit** perspective of CCS. If you are not in that perspective, click the perspective link in the upper-right corner to switch back.
2. Double-click on the <project>.syscfg configuration file in the **Project Explorer** tree.
3. When SysConfig opens, you see the **Device Configuration** properties. This sheet provides links to SYS/BIOS documentation resources.
4. Click the arrows in the list of components to collapse the items above **TI RTOS**. Expand **TI RTOS** to see the SYS/BIOS packages and modules.



If you are using the IAR IDE, the steps to open a configuration file are similar.

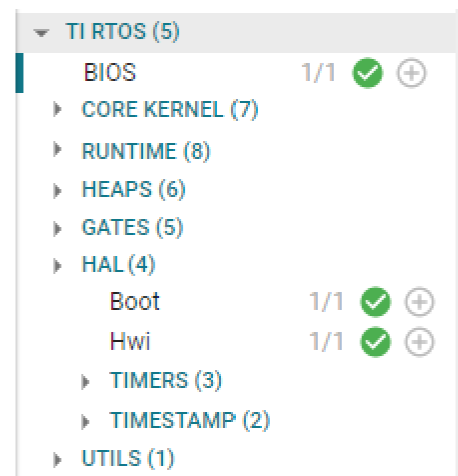
2.2.2 Performing Tasks with SysConfig

The following list shows configuration tasks you can perform with SysConfig and links to more details:

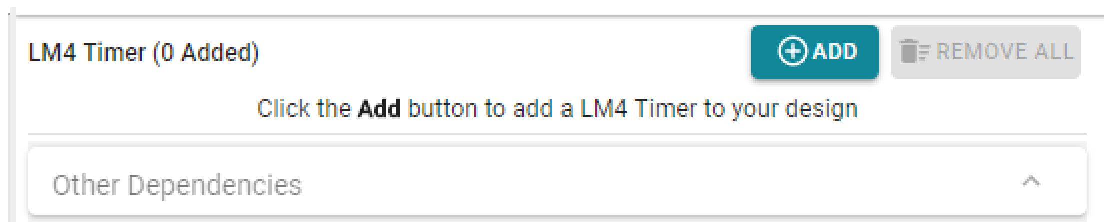
- Find, add, and delete modules. See Section 2.2.3.
- Change property values. See Section 2.2.4.
- Get help about a module. Hover your mouse cursor over a property to see the full property name. Click the small, circled **?** icon to see additional help.
- Configuring the memory map and section placement. Memory mapping and section placement are described in Chapter 7.
- Save the configuration or revert to the last saved file. See Section 2.2.5.
- View the code in the configuration file. See Section 2.2.6.

2.2.3 Adding and Removing Modules in the Configuration

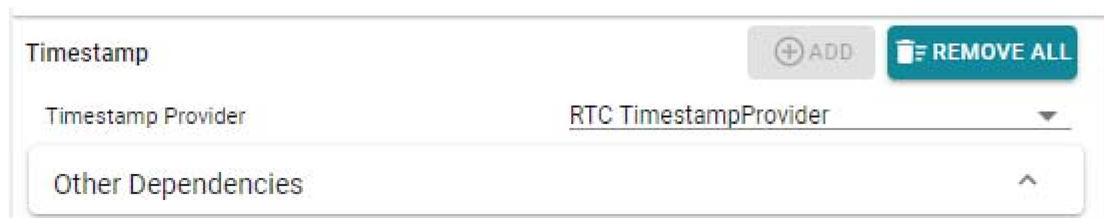
You can find modules by expanding components to browse the list. Or type a module name in the "Type Filter Text" area at the top of the component list. The SYS/BIOS modules are listed under the **TI RTOS** component heading.



To add a module to a configuration, click the **+** sign next to a module in the list or highlight a module and click **+Add** in the right pane.

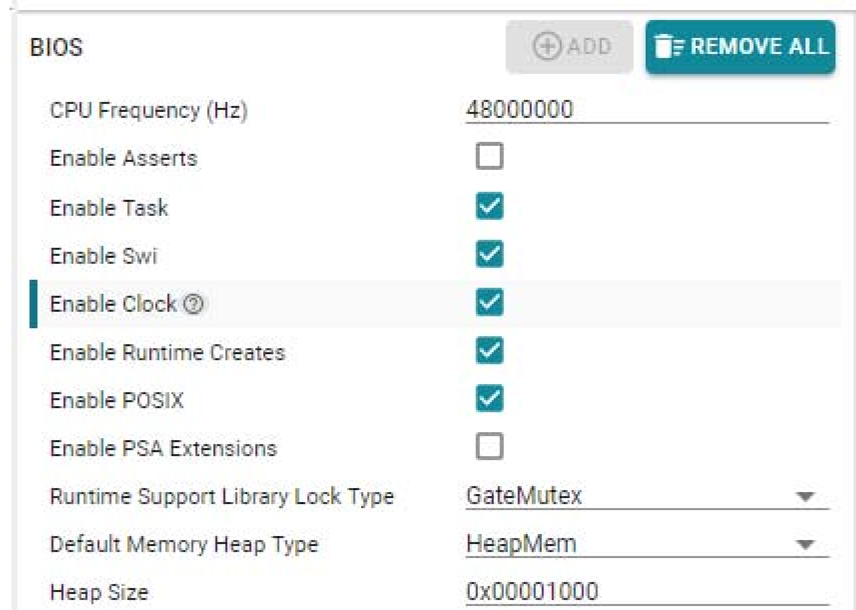


To remove a module from a configuration, highlight a module in the list and click **Remove All** in the right pane. If the **Remove All** button is gray, the module is enabled by a property in the BIOS module or is required by another enabled module that is dependent on this module.



2.2.4 Setting Property Values in the Configuration

To configure a module, set properties as needed after enabling the module. Hover your mouse cursor over a property to see the full property name. Click the small, circled **?** icon to see additional information about the property.



Types of properties include the following:

- **Boolean values:** Enable these properties by checking the box.
- **Enumerated types:** Select one of the available values.
- **Numeric values:** Type addresses using hex notation. Type sizes using hex or decimal notation. Type most other numeric values, such a priority levels, as decimal values.
- **Functions names and function sets:** Several modules allow you to create functions managed by the module. Examples include a list of Idle functions, user-defined Startup functions, and sets of Hwi, Swi, and Task hook functions. These functions are not the same as instances created with *Mod*_create() or *Mod*_construct() APIs in C code. To configure these functions, click **+Add** and type the name of the function in your C code to be called. The files generated by SysConfig will define this as an extern. If you are using C++, see Section 1.7 for information about name mangling.

Note: Previous versions of SYS/BIOS supported the static configuration of instances for a number of modules. SYS/BIOS no longer supports static instance creation. Use runtime C code to create objects. See Section 1.6.

Several modules have advanced properties that you can view by clicking the category heading to expand the list. For example, the Hwi module has additional properties to set Core Exception Handlers and Hwi Hook functions. See the relevant chapters in this document and the Doxygen API Reference described in Section 1.8 for details about such properties.

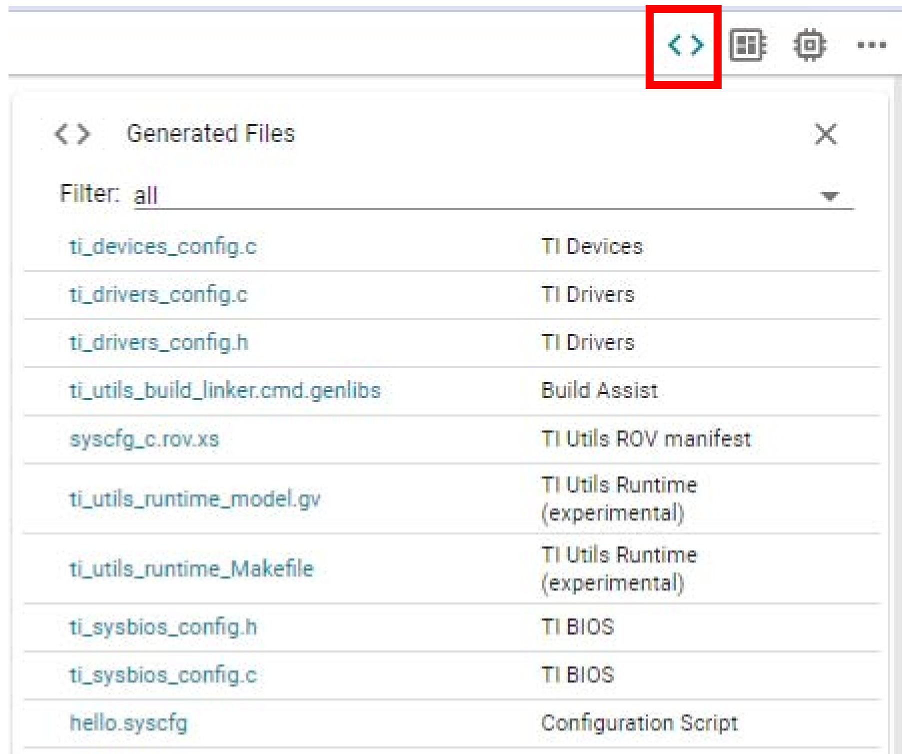
Most modules have an expandable list of Other Dependencies. These lists identify relationships between modules. Properties are often shown in gray for dependent modules. Typically to set properties for a dependent module, you will need to open that module or the BIOS module.

2.2.5 Saving the Configuration

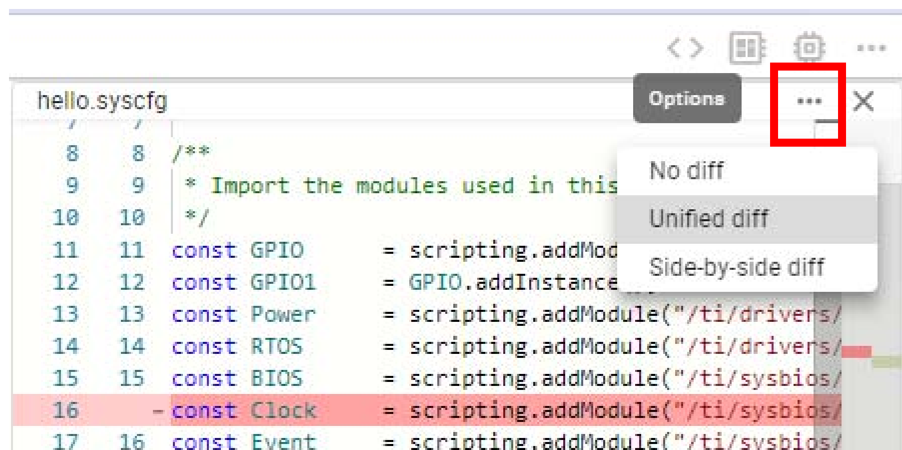
If you have modified the configuration, you can press Ctrl+S to save the file or choose **File > Save** from the CCS menu bar.

2.2.6 Viewing Configuration Code

SysConfig lets you view the code being modified by your changes to the settings. To see the list of files, click the Show Generated Files icon in the upper-right corner of the SysConfig window.



Click a filename to open that file. Use the options (...) button in the upper-right to choose how you want differences between the last saved version of the file and the current settings to be displayed.



SysConfig manages a number of files related to your application's configuration. Files that contain configuration related to SYS/BIOS are:

- `<project>.syscfg`
- `ti_sysbios_config.h`
- `ti_sysbios_config.c`
- `syscfg_c.rov` (contains information used by ROV to show SYS/BIOS module information)

You can edit the `<project>.syscfg` file using a text editor outside SysConfig. However, SysConfig performs checking to prevent inconsistencies and conflicts. Therefore, it is recommended that you use the SysConfig interface to configure applications unless you are experienced with the generated configuration files.

You should not edit files other than `<project>.syscfg` using a text editor, because SysConfig will generate new files and overwrite any changes to files such as `ti_sysbios_config.h` and `ti_sysbios_config.c` when the `<project>.syscfg` file is changed or the project is rebuilt.

2.2.7 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
- Save the configuration

You can force the configuration to be validated by saving the configuration.

Validation means that semantic checks are performed to make sure that there are no conflicts between modules and properties. These checks also make sure that values are within the valid ranges for a property. Some datatype checking is also performed whenever you set a property.

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 Project Explorer. Expand the Debug folder to see the files that were generated by the build process.

2.3.1 *Understanding the Build Flow*

When a build occurs, SysConfig tool uses the `<project>.syscfg` file to generate a number of files, including the following files related to SYS/BIOS:

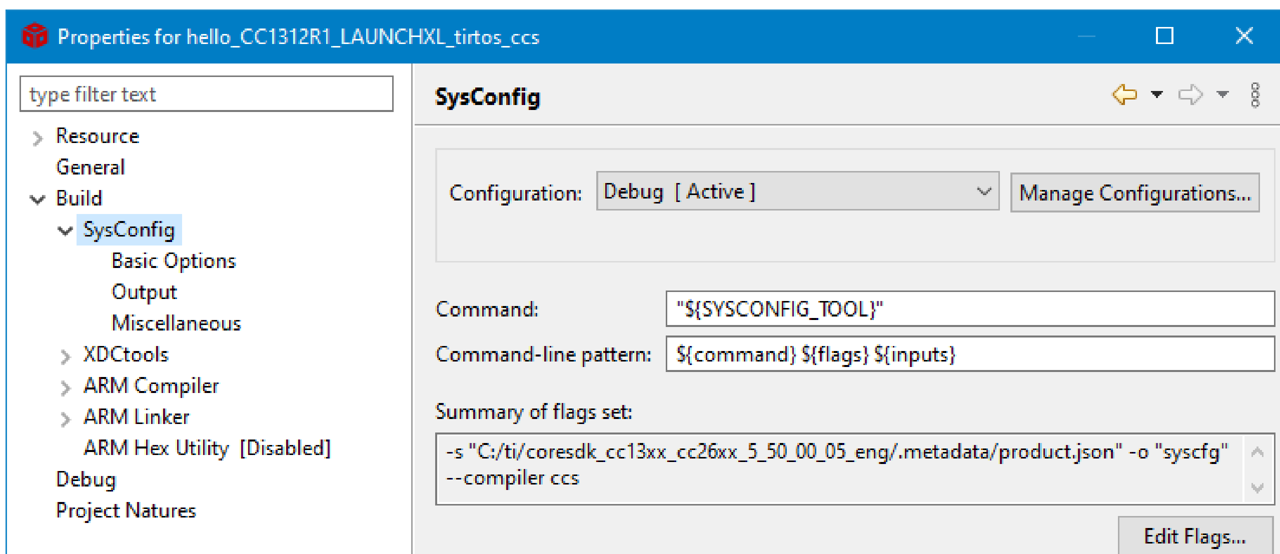
- `ti_sysbios_config.c`, which contains code and #includes the relevant C files needed for a given configuration.
- `ti_sysbios_config.h`, which contains #defines that are used in ti_sysbios_config.c.

Your project compiles `ti_sysbios_config.c` along with all of your application C files.

In a CCS project, the files generated by SysConfig are listed in **Generated Source > SysConfig** in the Project Explorer. The files generated by SysConfig during a build are placed in the **syscfg** folder under the build folder (**Debug** by default). These files are generated by SysConfig; you should not edit them directly, because they will be overwritten.

Notice that one of the generated files is `ti_utils_build_linker.cmd.genlibs`. This linker command file includes the `sysbios.a` library, which must be linked with in order for an application to use SYS/BIOS.

To control the behavior of SysConfig during a build, you can choose **Project > Properties** in CCS. Select the **Build > SysConfig** category and make settings as needed. See Section 1.8 for additional information sources.



TEXAS INSTRUMENTS

Chapter 3

SPRUIX7—November 2020

Threading Modules

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

Topic **Page**

3.1 SYS/BIOS Startup Sequence 30

3.2 Overview of Threading Modules 31

3.3 Hardware Interrupts 40

3.4 Software Interrupts 46

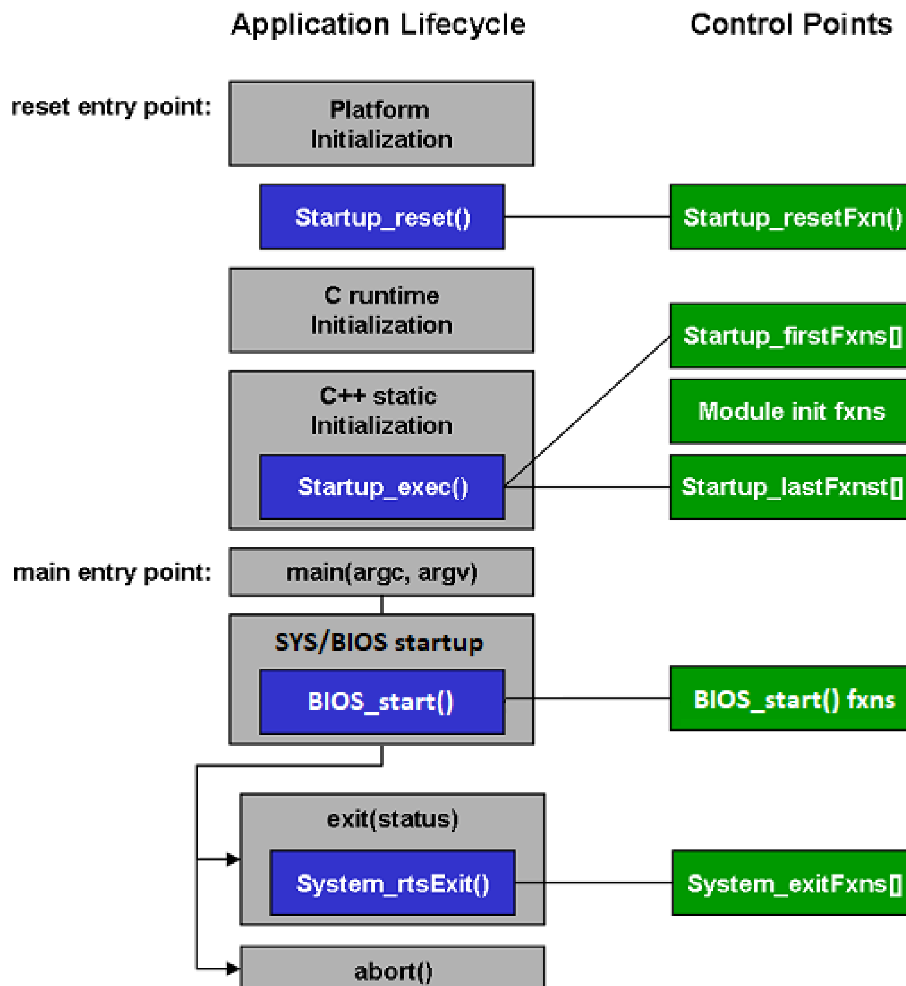
3.5 Tasks 60

3.6 The Idle Loop 77

3.7 Example Using Hwi, Swi, and Task Threads 78

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 sequence phases for user startup functions to be inserted.



The "before main()" startup sequence is as follows:

1. Immediately after CPU reset, perform target/device-specific CPU initialization (beginning at c_int00). See the "Program Loading and Running" chapter in the *Assembly Language Tools User's Guide* for your target family for details on this step and the cinit() step.
2. Prior to cinit(), run any user-supplied "Reset functions". The ti.sysbios.runtime.Startup module lets you configure these functions. These reset functions are called only on platforms where a reset is performed before running a program.
3. Run cinit() to initialize C runtime environment.
4. Run any user-supplied "First functions". The ti.sysbios.runtime.Startup module lets you configure these functions.
5. Run all the module initialization functions.

6. Run any user-supplied “Last functions”. The ti.sysbios.runtime.Startup module lets you configure these functions.
7. Run pinit().
8. Run main().

The “after main()” startup sequence 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. **BIOS Startup Functions.** Run any user-supplied BIOS_start() functions. The ti.sysbios.BIOS module lets you configure these functions.
2. **Enable Hardware Interrupts.**
3. **Enable Software Interrupts.** If the system supports software interrupts (Swis) (see BIOS.swiEnabled), then the SYS/BIOS startup sequence enables Swis at this point.
4. **Task Startup.** If the system supports Tasks (see BIOS.taskEnabled), then task scheduling begins here. If there are no Tasks in the system, then execution proceeds directly to the idle loop.

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 40, 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 Hwis, Swi 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 46, 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 60, 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 77, 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.

Note: The SimpleLink SDK also provides a subset of the POSIX thread (pthread) APIs. These include pthread threads, mutexes, read-write locks, barriers, and condition variables. The pthread APIs can simplify porting applications from a POSIX environment to SYS/BIOS, as well as allowing code to be compiled to run in both a POSIX environment and with SYS/BIOS. As the pthread APIs are built on top of the SYS/BIOS Task and Semaphore modules, some POSIX APIs can be called from SYS/BIOS Tasks.

For details about supported POSIX thread APIs, see the *TI-POSIX User's Guide* in the `/docs/tiposix/Users_Guide.html` file in the SDK installation. For more, see the official POSIX specification and the generic POSIX implementation.

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.

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. Periodic functions run at the priority of the Clock Swi.	Up to 32. 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.
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 synchronization 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

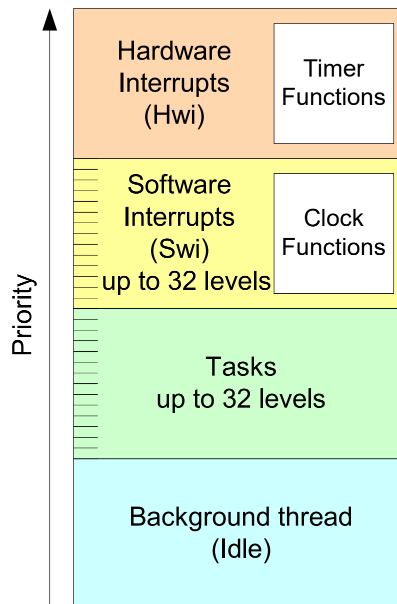
Characteristic	Hwi	Swi	Task	Idle
Runtime object creation	Yes	Yes	Yes	No
Dynamically change priority	See Note 2	Yes	Yes	No

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) 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). Tasks can be preempted by any higher-priority thread. Tasks can block while waiting for resource availability and lower-priority threads.

For Swis and Tasks, higher numbers equal higher priorities. That is, zero is the lowest priority level within the set of Swis and within the set of Tasks.

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 (highest priority number) 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

	Running Thread			
Newly Posted 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

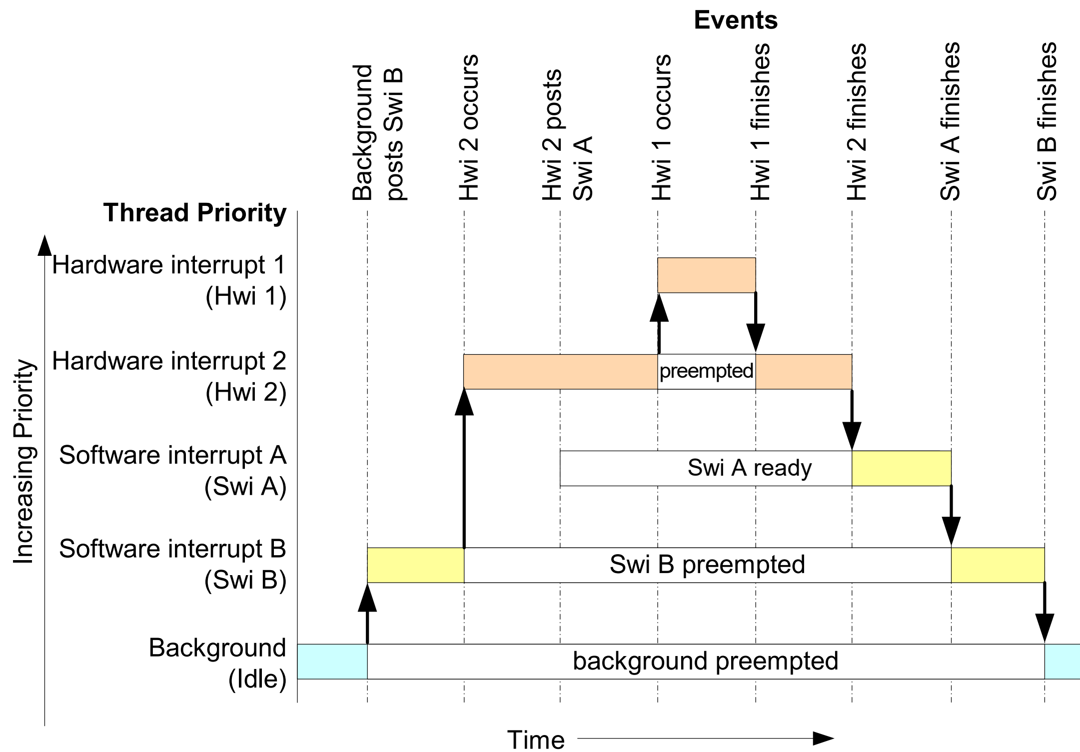
	Running Thread			
Enabled, higher-priority Task	Waits	Waits	Preempts	Preempts
Low-priority Task	Waits	Waits	Waits	Preempts

* 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 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 SysConfig) 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.

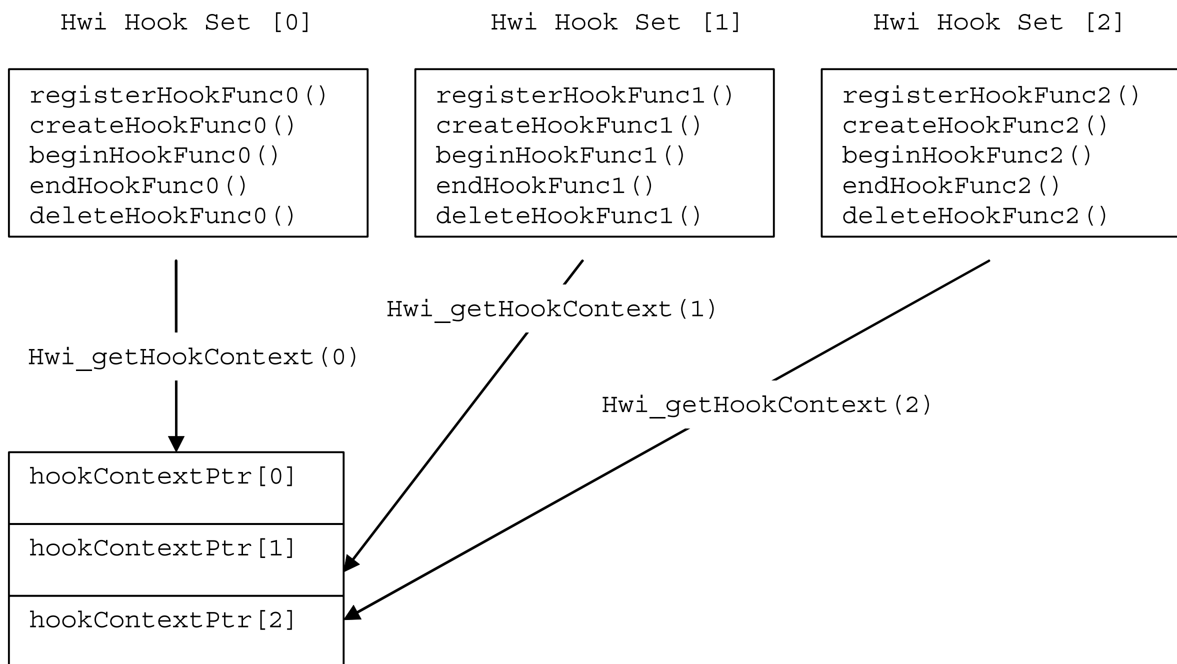
You can define as many hook sets as necessary for your application. When more than one hook set is defined, the individual hook functions within each set are invoked in hook ID order for a particular hook type. For example, during Task_create() the order that the Create hook within each Task hook set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



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.

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++.

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 8 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 dispatcherTo create a Hwi object dynamically, use calls similar to these:

```
#include <ti/sysbios/family/arm/m3/Hwi.h>
...
Hwi_Handle hwi0;
Hwi_Params hwiParams;
...

Hwi_Params_init(&hwiParams);

hwiParams.arg = 5;
hwi0 = Hwi_create(id, hwiFunc, &hwiParams, Error_IGNORE);
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.

See Section 8.2 for more about creating Hwi objects.

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. To change the system stack size, edit the stack_size in the linker command file.

For Cortex-M, the amount of system stack space required for interrupts by the first Hwi is about 176 bytes. For subsequent Hwis, the worst-case Hwi interrupt nesting requires about 80 bytes of additional stack space per additional Hwi.

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 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.
- **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 with SysConfig.

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 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);
```

See Section 6.5 for information about the Error_Block argument.

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 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 SysConfig settings 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 <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/runtime/Timestamp.h>

#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/hal/Timer.h>
#include <ti/sysbios/family/arm/m3/Hwi.h>

volatile bool myEnd2Flag = false;
int myHookSetId1, myHookSetId2;
Timer_Handle myTimer;
Task_Handle myTask;
```

```
/* 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 Hwis */
void myCreate1(Hwi_Handle hwi, Error_Block *eb)
{
    void *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, (void *)0xdead1);
}

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

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

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

    pEnv = Hwi_getHookContext(hwi, myHookSetId1);
    System_printf("myEnd1: pEnv = 0x%x, time = %d\n", pEnv, Timestamp_get32());
    Hwi_setHookContext(hwi, myHookSetId1, (void *)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  */
void myCreate2(Hwi_Handle hwi, Error_Block *eb)
{
    void *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, (void *)0xdead2);
}

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

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

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

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

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

```
/* ======== myTaskFunc ======== */
void myTaskFunc(uintptr_t arg0, uintptr_t 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[])
{
    Task_Params taskParams;
    Timer_Params timerParams;

    Task_Params_init(&taskParams);
    taskParams.name = "myTask";
    myTask = Task_create(myTaskFunc, &taskParams, Error_IGNORE);
    if (myTask == NULL) {
        System_abort("myTsk0 create failed");
    }

    Timer_Params_init(&timerParams);
    timerParams.period = 1000;
    timerParams.startMode = Timer.StartMode_USER;
    timerParams.runMode = Timer.RunMode_ONESHOT;
    myTimer = Timer_create(Timer_ANY, myTimerFunc, &timerParams, Error_IGNORE);
    if (myTimer == NULL) {
        System_abort("Timer create failed");
    }

    System_printf("Starting HwiHookExample...\n");
    BIOS_start();
    return (0);
}
```

The SysConfig configuration for this example should do the following:

- Enable the BIOS, Timestamp, Task, Timer, and Hwi modules.
- Set **Enable Clock** to false in the BIOS module.
- Add an instance for the Idle module and set its **Idle Function** to `myIdleFunc`.
- For the Hwi module, which is in the HAL category, create two sets of Hwi Hooks. The functions for Hwi Hooks 0 should be `myRegister1`, `myCreate1`, `myBegin1`, and `myEnd1`. The functions for Hwi Hooks 1 should be `myRegister2`, `myCreate2`, myBegin2, and `myEnd2`.

The program output is as follows, though the number of myCreate calls may be different on different devices:

```
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. 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 create Swi objects dynamically—with a call to Swi_create() or Swi_construct(). Swis can also be deleted during program execution.

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, properties, and objects, see the Swi module in the SYS/BIOS Doxygen API Reference described in Section 1.8.

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

```
Swi_Handle swi0;
Swi_Params swiParams;

...

Swi_Params_init(&swiParams);

swi0 = Swi_create(swiFunc, &swiParams, Error_IGNORE);
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.

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 (higher priority number) 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 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 specified in the linker command file (.cmd).

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).

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. 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 trigger variable. The trigger can be used either to determine whether to post the Swi or to provide values that can be evaluated within the Swi function. The trigger is stored as an integer, which is 32-bits on SimpleLink devices, but may have a different size on other devices.

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 3-4 summarizes the differences between these functions.

Table 3-4. 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.

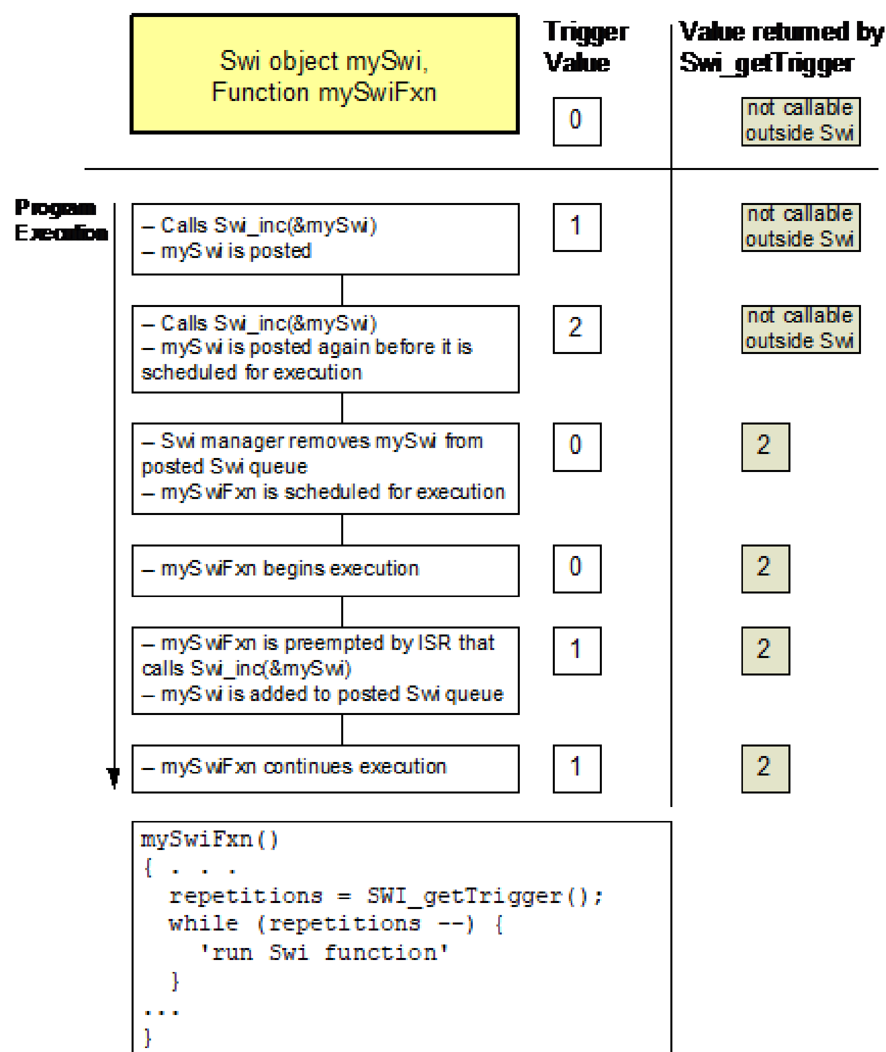
To access the value of its trigger, a Swi function can call Swi_getTrigger(), which can be called only from the Swi object's function. The value returned by Swi_getTrigger() is the value of the trigger before the Swi object was removed from the posted Swi queue and the Swi function was scheduled for 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. 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.

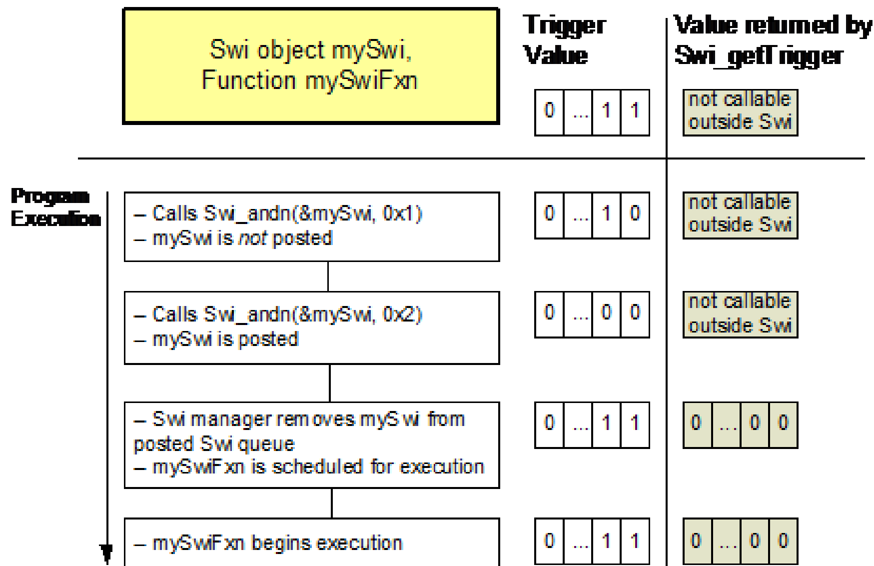
When a Swi has been posted using Swi_inc(), once the Swi manager calls the corresponding Swi function for execution, the Swi function can access the Swi object trigger to know how many times it was posted before it was scheduled to run, and proceed to execute the same function as many times as the value of the trigger.

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



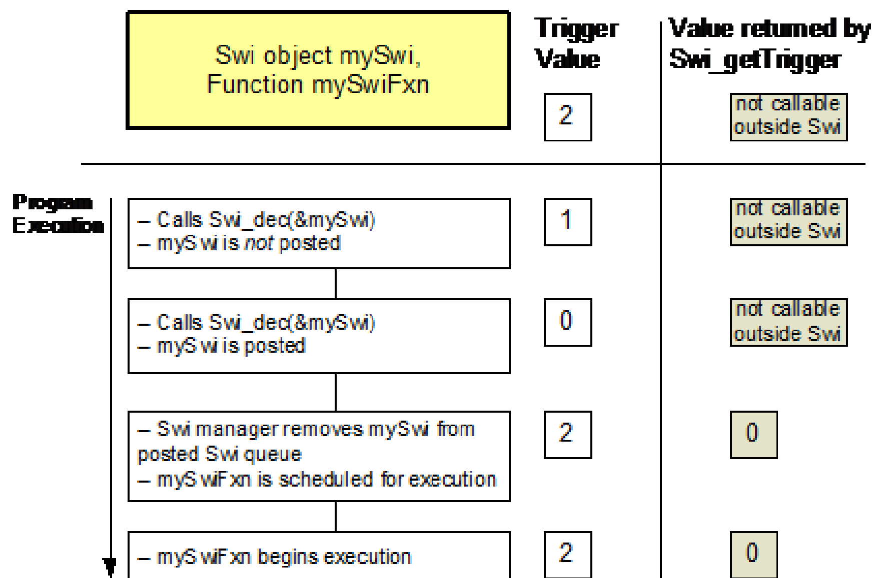
If more than one event must always happen for a given Swi to be triggered, Swi_andn() should be used to post the corresponding Swi object as shown in Figure 3-4. For example, if a Swi must wait for input data from two different devices before it can proceed, its trigger should have two set bits when the Swi object is created. 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



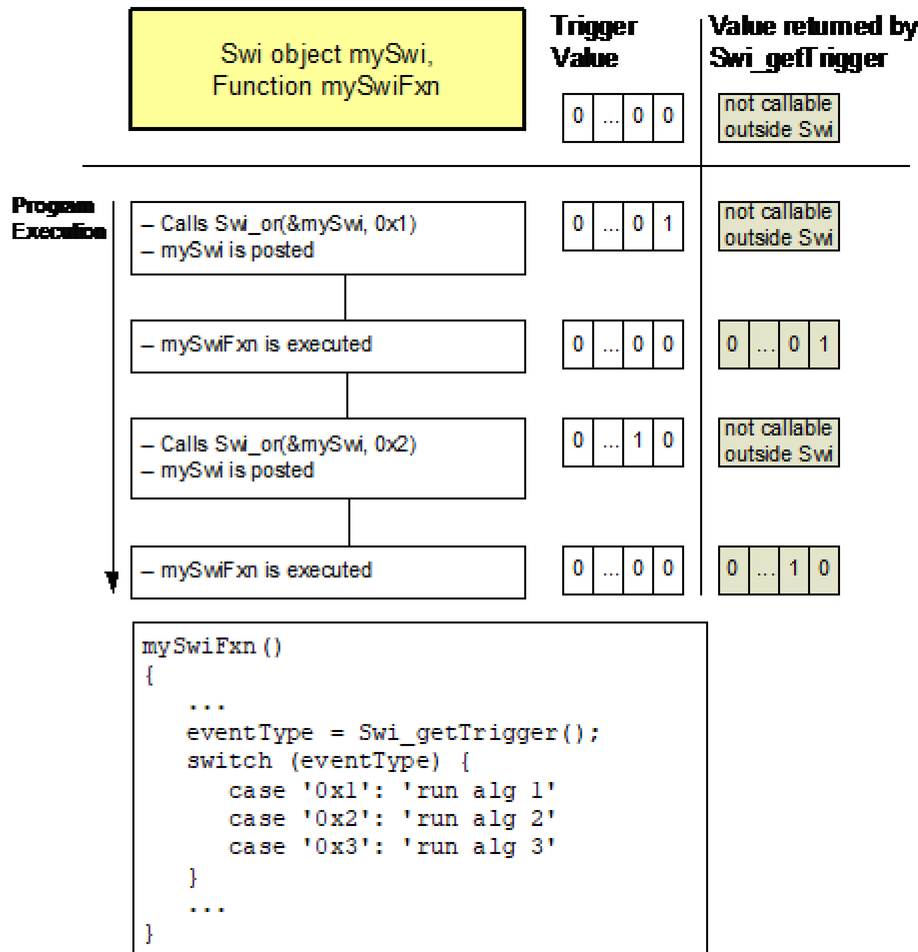
If the program execution requires that multiple occurrences of the same event must take place before a Swi is posted, Swi_dec() should be used to post the Swi as shown in Figure 3-5. By configuring the Swi trigger to be equal to the number of occurrences of the event before the Swi should be posted and 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.



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 Section 3.4.7). 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 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 created using Swi_create() and Swi_construct().
- **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 using SysConfig.

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 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);
```

See Section 6.5 for information about the Error_Block argument.

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 configuration settings and program output are shown after the C code listing.

This is the C code for the example:

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

#include <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/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;
Task_Handle myTask;
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)
{
    void *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, (void *)0xdead1);
}
```

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

    pEnv = Swi_getHookContext(swi, myHookSetId1);

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

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

    pEnv = Swi_getHookContext(swi, myHookSetId1);

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

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

    pEnv = Swi_getHookContext(swi, myHookSetId1);

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

/* ======== myDelete1 ========
 * invoked upon Swi deletion */
void myDelete1(Swi_Handle swi)
{
    void *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)
{
    void *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, (void *)0xdead2);
}

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

    pEnv = Swi_getHookContext(swi, myHookSetId2);

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

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

    pEnv = Swi_getHookContext(swi, myHookSetId2);

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

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

    pEnv = Swi_getHookContext(swi, myHookSetId2);

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

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

    pEnv = Swi_getHookContext(swi, myHookSetId2);

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

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

/* ======== myTaskFunc ======== */
void myTaskFunc(uintptr_t arg0, uintptr_t 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[])
{
    Task_Params taskParams;

    Task_Params_init(&taskParams);

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

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

    /* Create myTask */
    taskParams.name = "myTask";
    myTask = Task_create(myTaskFunc, &taskParams, Error_IGNORE);
    if (myTask == NULL) {
        System_abort("myTsk0 create failed");
    }

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

The SysConfig configuration for this example should do the following:

- Enable the BIOS, Idle, Swi, Task, and Timestamp modules.
- Set **Enable Clock** to false in the BIOS module.
- Add an instance for the Idle module and set its **Idle Function** to `myIdleFunc`.
- For the Swi module, create two sets of Swi Hooks. The functions for Swi Hooks 0 should be `myRegister1`, `myCreate1`, `myReady1`, `myBegin1`, `myEnd1`, and `myDelete1`. The functions for Swi Hooks 1 should be `myRegister2`, `myCreate2`, `myReady2`, `myBegin2`, `myEnd2`, and `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 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.

You can configure general behaviors of the Task module in SysConfig. First, enable the module, which is listed in the **Core Kernel** category. Click the small, circled **?** icons in SysConfig for information about properties. See the Doxygen API Reference described in Section 1.8 for details about the APIs.

3.5.1 Creating Tasks

You can create Task objects with a call to Task_create() or Task_construct(). Tasks can be deleted during program execution.

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;
...

/* Create 1 task with priority 15 */
Task_Params_init(&taskParams);
taskParams.stackSize = 512;
taskParams.priority = 15;
task0 = Task_create((Task_FuncPtr)hiPriTask, &taskParams, Error_IGNORE);
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. See Section 3.5.3 for information about task stack sizes.

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

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);
```

3.5.2 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.

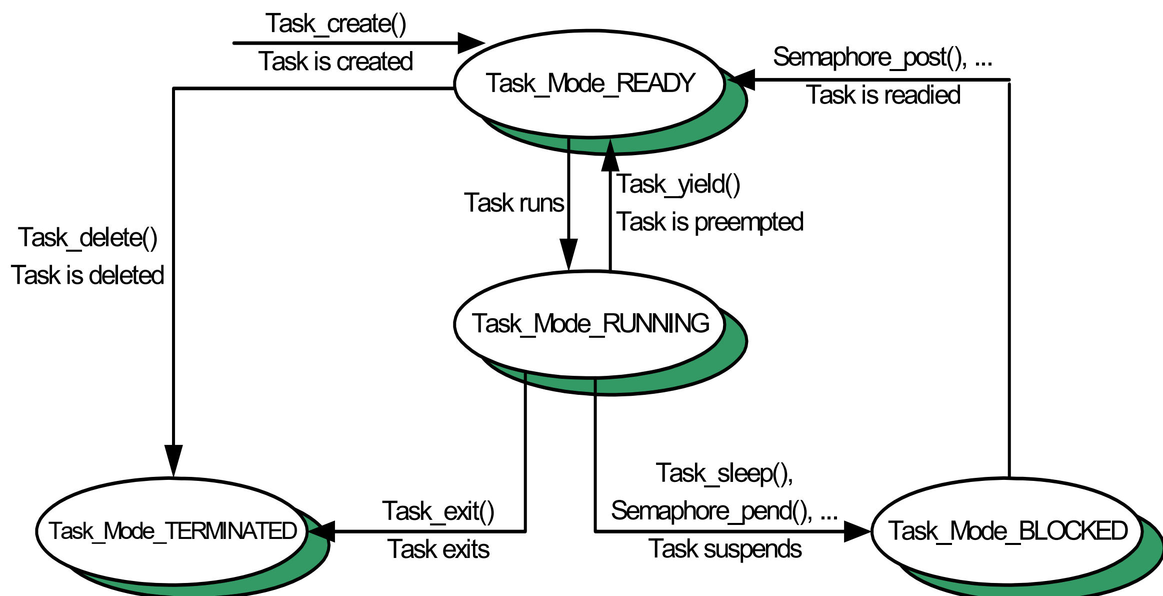
- **Task_Mode_INACTIVE,** which means the task has a priority equal to -1 and is in a pre-Ready state. This priority can be set when the task is created or by calling the Task_setPri() API at runtime.

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

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

Note that when the kernel is configured without Task hooks (with **Minimize Latency** set to false in the Task module), each task stack must only support a single full Hwi interrupt context in addition to the task scheduler's local variables. This configuration is equal to the “Minimum Stack Consumed” in Table 3–5.

Table 3–5. Task Stack Use by Target Family

Target	Minimum Stack Consumed	Maximum Stack Consumed	Units
GCC_M4	39	159	8-bit bytes
IAR_M4	32	152	8-bit bytes
TI_M4	32	112	8-bit bytes
GCC_M4F	111	303	8-bit bytes
IAR_M4F	104	296	8-bit bytes
TI_M4F	104	256	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)) {
   System_printf("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 Runtime Object View (ROV) to examine runtime Task stack usage.

3.5.5 Task Hooks

The Task module supports the following set of Hook functions:

- **Register.** A function called before any Tasks are initialized at runtime. The register hook is called at boot time before main() and before interrupts are enabled. It is used to register the hookset with the Task module.
- **Create.** A function called when a Task is created. This includes Tasks that are created using either 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 with SysConfig.

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 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);
```

See Section 6.5 for information about the Error_Block argument.

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 configuration settings 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 <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/runtime/Memory.h>
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/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;
    void *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, (void *)0xdead);
}

/* ======== myReady ========
 * invoked when Task is made ready to run */
void myReady(Task_Handle task)
{
    string name;
    void *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, (void *)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;
    void *pPrevEnv;
    void *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, (void *)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, (void *)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;
    void *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, (void *)0xdeadbeef);
}
```

```
/* ======== myDelete ========
 * invoked upon Task deletion */
void myDelete(Task_Handle task)
{
    string name;
    void *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(uintptr_t arg0, uintptr_t arg1)
{
    System_printf("myTsk0 Entering\n");
    System_printf("myTsk0 Calling Task_yield\n");
    Task_yield();
    System_printf("myTsk0 Exiting\n");
}

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

void myTsk2Func(uintptr_t arg0, uintptr_t 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;

    Task_Params_init(&params);
```

```
    params.name = "myTsk0";
    myTsk0 = Task_create(myTsk0Func, &params, Error_IGNORE);
    if (myTsk0 == NULL) {
        System_abort("myTsk0 create failed");
    }

    params.name = "myTsk1";
    myTsk1 = Task_create(myTsk1Func, &params, Error_IGNORE);
    if (myTsk1 == NULL) {
        System_abort("myTsk1 create failed");
    }

    params.name = "myTsk2";
    myTsk2 = Task_create(myTsk2Func, &params, Error_IGNORE);
    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 SysConfig configuration for this example should do the following:

- Enable the SysMin, Idle, and Task modules.
- Set **Output Buffer Size** to 4096 in the SysMin module.
- Add an instance for the Idle module and set its **Idle Function** to `myIdleFunc`.
- For the Task module, create one set of Task Hooks. The functions for this hook set should be `myRegister`, `myCreate`, `myReady`, `mySwitch`, `myExit`, and `myDelete1`.

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

```
/*
 * ======== 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 <ti/sysbios/runtime/System.h>
#include <ti/sysbios/runtime/Error.h>


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


void hiPriTask(uintptr_t arg0, uintptr_t arg1);
void task(uintptr_t arg0, uintptr_t arg1);
void clockHandler1(uintptr_t arg);
void clockHandler2(uintptr_t arg);
Semaphore_Handle sem;


/* ======== main ======== */
void main()
{
    Task_Params taskParams;
    Task_Handle myTsk0, myTski;
    Clock_Params clockParams;
    Clock_Handle myClk0, myClk1;
    unsigned int i;


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

```
/* Create 1 task with priority 15 */
Task_Params_init(&taskParams);
taskParams.stackSize = 512;
      // Note: Larger stack needed for some targets
taskParams.priority = 15;
myTsk0 = Task_create((Task_FuncPtr)hiPriTask, &taskParams, Error_IGNORE);
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, Error_IGNORE);
    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, Error_IGNORE);
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, Error_IGNORE);
if (myClk1 == NULL) {
    System_abort("Clock1 create failed");
}
```

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

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

/* ======== clockHandler1 ======== */
void clockHandler1(uintptr_t arg)
{
    /* Call Task_yield every 4 ms */
    Task_yield();
}

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

/* ======== task ======== */
void task(uintptr_t arg0, uintptr_t arg1)
{
    int time;
    int prevtime = -1;
    unsigned int 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(uintptr_t arg0, uintptr_t 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, use SysConfig to set the Task module **Enable Idle Task** property to false and the **allBlockedFunc** property to `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 configuration settings and program output are shown after the C code listing. This is the C code for the example:

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

#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/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>

typedef struct {
    Queue_Elem elem;
    uint32_t timeout;
    uint32_t period;
    void (*fxn)(uintptr_t);
    uintptr_t 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(uintptr_t arg)
{
    Swi_post(swi);
}

/* Swi thread to handle Timer interrupt */
void swiFxn(uintptr_t arg1, uintptr_t 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(uintptr_t arg1, uintptr_t 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(uintptr_t arg)
{
    System_printf("In clk1Fxn, arg = %d.\n", arg);
    clk1.arg++;
}

/* Second Clock function, invoked every 20 timer interrupts */
void clk2Fxn(uintptr_t 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;

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

    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, Error_IGNORE);
```

```
    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, Error_IGNORE);
    if (task == NULL) {
        System_abort("Task create failed");
    }

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

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

    BIOS_start();

    return(0);
}
```

The SysConfig configuration for this example should do the following:

- Enable the BIOS, Clock, task, Semaphore, Hwi, and HeapMem modules.
- In the BIOS module, set **Heap Size** to 0x2000.
- In the System module, set **SystemSupport Module** to SysMin and **Maximum Atexit Handlers** to 4.
- Set **Output Buffer Size** to 0x400 in the SysMin module.

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
```

TEXAS INSTRUMENTS

Chapter 4

SPRUIX7—November 2020

Synchronization Modules

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

Topic		Page
4.1	Semaphores	83
4.2	Event Module	88
4.3	Gates	94
4.4	Mailboxes	97
4.5	Queues	99

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 and as either simple (FIFO) or priority-aware semaphores. Semaphores can be used for task synchronization and mutual exclusion. The same APIs are used for both counting and binary semaphores.

By default, semaphores are simple 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 count value for a semaphore is limited only by the size of the 16-bit counter. If Asserts are disabled, the count rolls over with no notification if the count is incremented from the maximum 16 bit value.

To configure a counting semaphore, set the mode parameter as follows:

```
semParams.mode = Semaphore_Mode_COUNTING;
```

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.

To configure a binary semaphore, set the mode parameter as follows:

```
semParams.mode = Semaphore_Mode_BINARY;
```

Tasks wait for simple counting and binary semaphores in FIFO order without regard to the priority of the tasks. Optionally, you can create "priority" semaphores that insert pending tasks into the waiting list before the first task that has a lower priority. As a result, tasks of equal priority pend in FIFO order, but tasks of higher priority are readied before tasks of lower priority.

To configure a counting or binary priority semaphore, set the mode parameter using one of the following constants:

```
semParams.mode = Semaphore_Mode_COUNTING_PRIORITY;

semParams.mode = Semaphore_Mode_BINARY_PRIORITY;
```

Note that using priority semaphores can increase the interrupt latency in the system, since interrupts are disabled while the list of tasks waiting on the semaphore is scanned for the proper insertion point. This is typically about a dozen instructions per waiting task. For example, if you have 10 tasks of higher priority waiting, then all 10 will be checked with interrupts disabled before the new task is entered onto the list.

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

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

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);
```

See Section 6.5 for information about the Error_Block argument.

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,
    unsigned int       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 first 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);
```

Calling Semaphore_post() may result in hardware interrupts being re-enabled in specific circumstances. See Section 8.2.4 for details.

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 SysConfig configuration for this example should do the following:

- Enable the BIOS, Clock, Task, Semaphore, Hwi, and HeapMem modules.
- In the BIOS module, set Heap Size to 0x2000.
- In the SysMin module, set Output Buffer Size to 0x400.

Three writer tasks, a reader task, a semaphore, and a queue used in this example are created using the following C code:

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 <ti/sysbios/runtime/Memory.h>
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/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 */

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 in main() */
Semaphore_Handle sem;
Queue_Handle msgQueue;
Queue_Handle freeQueue;
```

```
/* ======== main ======== */
int main(int argc, char* argv[])
{
    int i;
    MsgObj *msg;
    Task_Params taskParams;

    Task_Params_init(&taskParams);

    /* reader */
    taskParams.priority = 5;
    Task_create(reader, &taskParams, NULL);

    /* writer0 */
    taskParams.priority = 3
    taskParams.arg0 = 0;
    Task_create(writer, &taskParams, NULL);

    /* writer1 */
    taskParams.priority = 3
    taskParams.arg0 = 1;
    Task_create(writer, &taskParams, NULL);

    /* writer2 */
    taskParams.priority = 3
    taskParams.arg0 = 2;
    Task_create(writer, &taskParams, NULL);

    sem = Semaphore_create(0, NULL);

    msgQueue = Queue_create(NULL);
    freeQueue = Queue_create(NULL);

    msg = (MsgObj *) Memory_alloc(NULL, NUMMSGS * sizeof(MsgObj), 0, Error_IGNORE);
    if (msg == NULL) {
        System_abort("Memory allocation failed");
    }

    msg = (MsgObj *) Memory_alloc(NULL, NUMMSGS * sizeof(MsgObj), 0, Error_IGNORE);
    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, the 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. However, see Section 4.2.1 for details about events that are implicitly posted from a Semaphore or Mailbox.

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

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:

```
unsigned int Event_pend(Event_Handle   event,
                        unsigned int   andMask,
                        unsigned int   orMask,
                        unsigned int   timeout);
```

The Event_post() prototype is as follows:

```
void Event_post(Event_Handle   event,
                unsigned int   eventIds);
```

Calling Event_post() may result in hardware interrupts being re-enabled in specific circumstances. See Section 8.2.4 for details.

Runtime example: The following C code creates an Event object with an Event_Handle named "myEvent".

```
Event_Handle myEvent;

...

/* Default instance params */
myEvent = Event_create(NULL, Error_IGNORE);
if (myEvent == NULL) {
    System_abort("Event create failed");
}
```

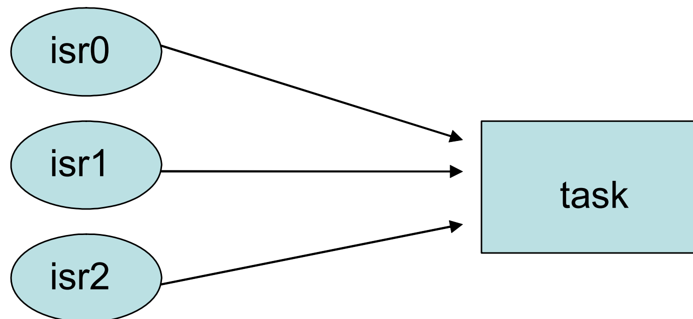
Runtime example: The following C code blocks on an event. It wakes the task only when both events 0 and 6 have occurred. It sets the andMask to enable both Event_Id_00 and Event_Id_06. It sets the orMask to Event_Id_NONE.

```
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, Error_IGNORE);
    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()
{
    unsigned int 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 set 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. To enable the ability to post events from Mailbox and Semaphore objects, you must set the supportsEvents property of the Semaphore module to True. For example, the configuration could include the following statement:

```
Semaphore.supportsEvents = true;
```

SYS/BIOS objects that support implicit event posting must have 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 used for implicit event posting should only be waited on by a single Task at a time.

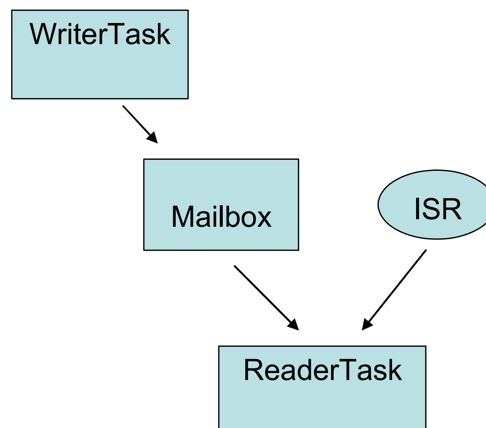
When Event_pend() is used to acquire a resource from implicitly posting objects, the BIOS_NO_WAIT timeout parameter should be used to subsequently retrieve the resource from the object.

Recall that Events are binary, so upon returning from an Event_pend() all matching events are consumed. If a task is blocked on an Event that is implicitly posted from a Semaphore, the corresponding Event ID is consumed. An immediate call to Event_pend() again on that same Event ID would block.

However, if `Semaphore_pend(sem, BIOS_NO_WAIT)` is called on the corresponding Semaphore *prior* to calling Event_pend(), the matching Event ID in the Event object is internally refreshed prior to returning from Semaphore_pend() to match the newly updated Semaphore "count". If the Semaphore count is non-zero after calling Semaphore_pend(), the corresponding Event ID is set within the Event object. If the count is zero after calling Semaphore_pend(), the corresponding Event ID in the event object is cleared. In this way, the Event object is kept in sync with the Semaphore object that implicitly calls Event_post(). But the application *must* call `Semaphore_pend(sem, BIOS_NO_WAIT)` in order for this synchronization to take place.

Since Mailboxes are built upon Semaphores, the same logic applies to Event objects that register with Mailboxes. Following the return from Event_pend() of an implicitly posted Event by either Mailbox_post() or Mailbox_pend(), the corresponding Event object is refreshed by an intervening call to `Mailbox_pend(mbx, mbuf, BIOS_NO_WAIT)` or `Mailbox_post(mbx, mbuf, BIOS_NO_WAIT)`.

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 {
   unsigned int id;
   char buf[10];
}
```

```
main()
{
   Mailbox_Params mboxParams;

   myEvent = Event_create(NULL, Error_IGNORE);
   if (myEvent == NULL) {
      System_abort("Event create failed");
   }
   Mailbox_Params_init(&mboxParams);
   mboxParams.readerEvent = myEvent;

   /* Assign Event_Id_00 to Mailbox "not empty" event */
   mboxParams.readerEventId = Event_Id_00;
   mbox = Mailbox_create(sizeof(msg), 50, &mboxParams, Error_IGNORE);
   if (mbox == NULL) {
      System_abort("Mailbox create failed");
   }

   /* Mailbox_create() sets Mailbox's readerEvent 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

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.

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.

As shown in the examples that follow, 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.

```
unsigned int gateKey;
GateHwi_Handle gateHwi;
GateHwi_Params prms;

...

GateHwi_Params_init(&prms);

gateHwi = GateHwi_create(&prms, Error_IGNORE);
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);
```

For information about the parameter structure and individual parameters for instances of these modules, see the Doxygen API Reference described in Section 1.8.

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.

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.

4.3.4 Setting the SYS/BIOS Gate Type

The application sets the type of gate to be used by calls in the TI RTS library (Chapter 6). The type of gate selected is used to guarantee re-entrancy of the RTS APIs. In SysConfig, the **Runtime Support Library Lock Type** property of the BIOS module controls this behavior.

The type of gate depends on the type of threads that are going to be calling RTS library functions. For example, if both Swi and Task threads are going to be calling the RTS library's System_printf() function, GateSwi should be used. In this case, Hwi threads are not disabled during System_printf() calls from the Swi or Task threads.

If NoLocking is used, the RTS lock is not plugged, and re-entrancy for the TI RTS library calls is not guaranteed. The application can plug the RTS locks directly if it wants.

The types of gates available are listed in Section 4.3.1 and Section 4.3.2. Note that GateTask is not supported as a SYS/BIOS RTS gate type.

- **GateHwi:** Interrupts are disabled and restored to maintain re-entrancy. Use if making any RTS calls from a Hwi.
- **GateSwi:** Swis are disabled and restored to maintain re-entrancy. Use if not making RTS calls from any Hwis but making such calls from Swis.
- **GateMutex:** A single mutex is used to maintain re-entrancy. Use if only making RTS calls from Tasks. Blocks only Tasks that are also trying to execute critical regions of the RTS library.
- **GateMutexPri:** A priority-inheriting mutex is used to maintain re-entrancy. Blocks only Tasks that are also trying to execute critical regions of the RTS library. Raises the priority of the Task that is executing the critical region in the RTS library to the level of the highest priority Task that is blocked by the mutex.

The appropriate **Runtime Support Library Lock Type** in the BIOS module depends on which types of threads are enabled. If both **Enable Swi** and **Enable Task** are set to false, use “No lock” (GateNull) as the lock type. If **Enable Task** is set to true, GateMutex is recommended. If **Enable Swi** is true and **Enable Task** is false, GateSwi is recommended.

If **Enable Task** is false, you should not select GateMutex (or other Task level gates). Similarly, if both **Enable Task** and **Enable Swi** are false, you should not select GateSwi or the Task level gates.

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.

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(size_t         bufsize,
                              unsigned int   numBufs,
                              Mailbox_Params *params,
                              Error_Block    *eb)

void Mailbox_delete(Mailbox_Handle *handle);
```

See Section 6.5 for information about the Error_Block argument.

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

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,
                  void           *buf,
                  unsigned int   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,
                  void           *buf,
                  unsigned int   timeout);
```

Mailbox provides 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 parameters to support events for the reader(s) of the mailbox—readerEvent and readerEventId. These allow a mailbox reader to use an event object to wait for the mailbox message. Mailbox also provides two parameters for the mailbox writer(s)—writerEvent and writerEventId. These allow mailbox writers to use an event object to wait for room in the mailbox.

Note that the names of these event handles can be misleading. The readerEvent is the Event that a Mailbox reader should pend on, but it is posted by the Mailbox writer within the Mailbox_post() call. The writerEvent is the Event that the Mailbox writer should pend on waiting for the Mailbox to become not full so that it can successfully perform a Mailbox_post() without pending because the Mailbox is full. However, the writerEvent is posted by the Mailbox reader whenever the Mailbox is successful read from (that is, Mailbox_pend() returns true).

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().

Calling Mailbox_post() may result in hardware interrupts being re-enabled in specific circumstances. See Section 8.2.4 for details.

4.5 Queues

The ti.sysbios.knl.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().

Chapter 5

SPRUIX7—November 2020

Timing Services

This chapter describes modules that can be used for timing purposes.

Topic **Page**

5.1 Overview of Timing Services 102

5.2 Clock. 102

5.3 Timer Module 105

5.4 Seconds Module 105

5.5 Timestamp Module 106

5.1 Overview of Timing Services

Several modules are involved in timekeeping and clock-related services within SYS/BIOS:

- **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.arm.m3.Timer). You can use the Timer module to select a timer that calls a tickFxn when the timer expires. See Section 5.3 for details.
- **The ti.sysbios.hal.Seconds module** provides a means for maintaining the current time and date, as defined by the number of seconds since 1970 (the Unix epoch). This module generates a custom time() function that calls Seconds_get(), overriding the C standard library's time() function. See Section 5.4 for details.
- **The ti.sysbios.runtime.Timestamp module** provides simple timestamping services for benchmarking code. This module uses a target/device-specific TimestampProvider in SYS/BIOS to control how timestamping is implemented. See Section 5.5 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 period for the system tick is set by the **Clock Tick Period in microseconds** property.

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 in SysConfig not to use the timer by setting the **Clock Tick Source** property of the Clock module to either Clock_tickSource_USER or Clock_TickSource_NULL.

The Clock_tick() and the tick period are used as follows:

- **If the Clock Tick Source 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 Clock Tick Source 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 Clock Tick Source 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.

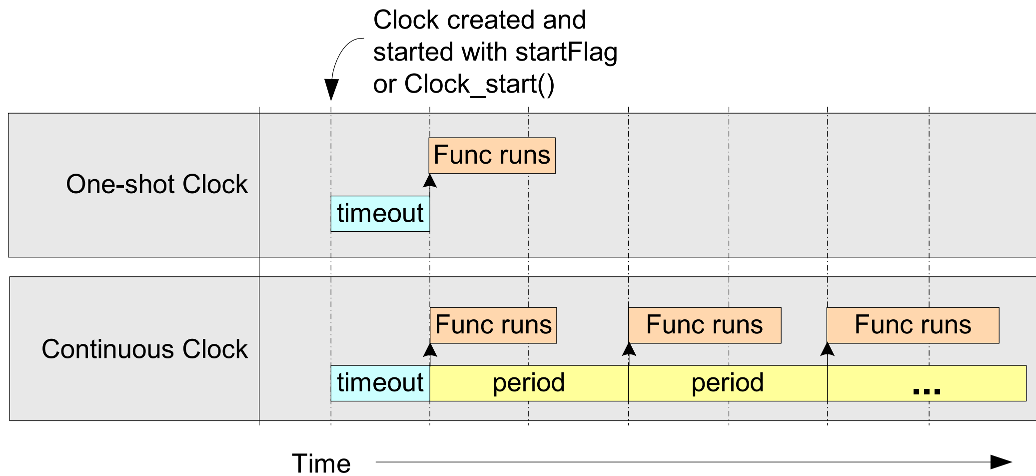
All Clock functions run in the context of a Swi. That is, the Clock module automatically creates a Swi for its use and run the Clock functions within that Swi. The priority of the Swi used by Clock can be changed by configuring the **Clock Swi Priority** property.

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 parameter. Clock_create() can be called only from the context of a Task or the main() function.

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

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 (uintptr_t) is passed to the function.

```
Clock_Params clockParams;
Clock_Handle myClock;

...

Clock_Params_init(&clockParams);
clockParams.period = 5;
clockParams.startFlag = true;
clockParams.arg = (uintptr_t)0x5555;
myClock = Clock_create(myHandler1, 5, &clockParams, Error_IGNORE);
if (myClock == NULL) {
   System_abort("Clock create failed");
}
```

Runtime example: This C example uses some of the Clock APIs to print messages about how long a Task sleeps.

```
uint32_t 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. Target-specific implementations are provided within the ti.sysbios.family tree.

You can use SysConfig to configure timers for an application to use. Examples of timers include the LM4, RTC, and SysTick timers.

Timers can be configured as one-shot or continuous mode timers. The period can be specified in timer counts or microseconds.

For information about the Timer APIs, see the Doxygen API Reference described in Section 1.8.

5.4 Seconds Module

The ti.sysbios.hal.Seconds module provides a way to set and get the number of seconds elapsed since Jan 1 00:00:00 GMT 1970 (the Unix epoch). The Seconds module maintains the time through a device-specific Seconds implementation, if available.

The Seconds module APIs are:

```
void Seconds_set(uint32_t seconds);

uint32_t Seconds_get(void);
```

An application should call Seconds_set() to initialize the seconds count. Seconds_set() can be called again, if needed, to update or reset the seconds count. An application must call Seconds_set() at least once before any calls to Seconds_get(). Otherwise, the result returned by Seconds_get() is meaningless. The Seconds_set() function is non-reentrant.

The Seconds module includes a time() function that calls Seconds_get(). This overrides the C Standard Library time() function. You can use this time() function in conjunction with other time functions in the C Standard header file, time.h, to view the current date and time in a readable format.

Examples

This example initializes the Seconds module, sets the date, gets the current date, and displays the current time and date in human readable format:

```
#include <time.h>
#include <ti/sysbios/hal/Seconds.h>

uint32_t t;
time_t t1;
struct tm *ltm;
char *curTime;

/* set to today's date in seconds since Jan 1, 1970 */
Seconds_set(1412800000); /* Wed, 08 Oct 2014 20:26:40 GMT */

/* retrieve current time relative to Jan 1, 1970 */
t = Seconds_get();

/* Use overridden time() function to get the current time.
 * Use standard C RTS library functions with return from time().
 * Assumes Seconds_set() has been called as above */
t1 = time(NULL);
ltm = localtime(&t1);
curTime = asctime(ltm);
System_printf("Time(GMT): %s\n", curTime);
```

Note: For some compiler run-time libraries, including TI's, the time() function returns the number of seconds elapsed since Jan 1, 1900. In this case, the Seconds module's time() function adds an offset to the value returned by Seconds_get(), in order to be consistent with other APIs in the run-time support library.

5.5 Timestamp Module

The ti.sysbios.runtime.Timestamp module, as the name suggests, provides timestamping services. The Timestamp module can be used for benchmarking code. Calls to the Timestamp module function are forwarded to a platform-specific TimestampProvider implementation.

In SysConfig, you can enable the Timestamp module, which is listed within the **Runtime** category. You can set the **Timestamp Provider** property to one of four providers. The provider modules are listed within the **HAL** category in SysConfig.

The package path to the Timestamp module is ti.sysbios.runtime.Timestamp, so SYS/BIOS applications should contain the following #include statement:

```
#include <ti/sysbios/runtime/Timestamp.h>
```

The following Timestamp module APIs are useful in SYS/BIOS applications:

- `Timestamp_get32()` — Return a 32-bit timestamp.
- `Timestamp_get64()` — Return a 64-bit timestamp if 64-bit resolution is supported by the target.
- `Timestamp_getFreq()` — Get the timestamp timer's frequency in Hz. You can use this function to convert a timestamp value into units of real time.

For information about the Timestamp APIs, see the Doxygen API Reference described in Section 1.8.

If you want a platform-independent version of Timestamp, you can use the TimestampStd modules, which uses the ANSI C clock() function.

Platform-specific TimestampProvider modules are located in the ti.sysbios.family package. For example, ti.sysbios.family.arm.cc26xx.TimestampProvider and ti.sysbios.family.arm.m3.TimestampProvider. Most TimestampProvider modules have configuration parameters that you can use to control the hardware clock source and the behavior if the timestamp counter overflows.

This example calculates the factor needed to correlate Timestamp delta to the CPU cycles:

```
Types_FreqHz freq1;   /* Timestamp frequency */
Types_FreqHz freq2;   /* BIOS frequency */
float        factor;  /* Clock ratio cpu/timestamp */


Timestamp_getFreq(&freq1);
BIOS_getCpuFreq(&freq2);
factor = (float)freq2.lo / freq1.lo;

System_printf("%lu\t%lu\t%lu\t Timestamp Freq, BIOS Freq, Factor\n",
                               freq1.lo, freq2.lo, (uint32_t) factor);
```

TEXAS INSTRUMENTS

Chapter 6

SPRUIX7—November 2020

Support Modules

This chapter describes modules that provide APIs for some basic support features that manage overall application behavior.

Topic		Page
6.1	Modules for Application Support and Management	109
6.2	BIOS Module	109
6.3	System Module	110
6.4	Startup Module	112
6.5	Error Module	112

6.1 Modules for Application Support and Management

SYS/BIOS provides a number of modules with functions intended to be used to support and manage overall application behavior. The following modules provide APIs and configuration properties that fall into this category:

- **BIOS Module (ti.sysbios.BIOS):** Responsible for SYS/BIOS startup and global parameter maintenance. See Section 6.2.
- **System (ti.sysbios.runtime.System):** provides low-level "system" services. See Section 6.3.
- **SysMin (ti.sysbios.runtime.SysMin):** Recommended support module for System module. See Section 6.3.1.
- **SysCallback (ti.sysbios.runtime.SysCallback):** Support module for custom System behavior. See Section 6.3.2.
- **Startup (ti.sysbios.Startup):** Manages application reset, startup, and initialization. See Section 6.4.
- **Memory (ti.sysbios.Memory):** Manages memory use. See Chapter 7.

6.2 BIOS Module

The BIOS module is responsible for setting up global parameters for use by SYS/BIOS and for performing the SYS/BIOS startup sequence. See Section 3.1 for details about the startup sequence.

To configure SYS/BIOS startup and overall behavior in SysConfig, set properties for the BIOS module. Click the small, circled **?** icons in SysConfig for information about properties. See the Doxygen API Reference described in Section 1.8 for more information.

The package path to the BIOS module is ti.sysbios.BIOS, so SYS/BIOS applications should contain the following #include statement:

```
#include <ti/sysbios/BIOS.h>
```

Typically, you begin editing the SYS/BIOS configuration in SysConfig by enabling the **BIOS** module. This automatically enables a number of other modules because the BIOS module properties that enable those modules are set to true by default.

The following BIOS module APIs are useful in SYS/BIOS applications:

- `BIOS_start()` — An application's main() function must call this function as the final statement to be executed before giving control to the SYS/BIOS thread scheduler. This call should be performed after all initialization required by the application has been performed. This function performs any remaining SYS/BIOS initialization and then transfers control to the highest priority Task that is ready to run. If Tasks are not enabled, control is transferred directly to the Idle Loop. The BIOS_start() function does not return.
- `BIOS_getCpuFreq()` — This function returns the CPU frequency in Hz. See Chapter 5 for information about timing. An example that uses this function is provided in Section 5.2.
- `BIOS_setCpuFreq()` — This function sets the CPU frequency in Hz. See Chapter 5 for information about timing. An example that uses this function is provided in Section 5.2.
- `BIOS_exit()` — Call this function when you want a SYS/BIOS application to terminate. BIOS_exit() does not return to the function that called BIOS_start(). Instead, BIOS_exit() calls System_exit() and aborts. All functions bound via System_atexit() or the ANSI C Standard Library atexit() function are then executed. Calling BIOS_exit() is recommended over calling System_exit() for SYS/BIOS

applications, because BIOS_exit() performs internal cleanup before calling System_exit(). If an application needs to terminate with an error condition, it should call System_abort() or System_abortSpin().

- `BIOS_getThreadType()` — This function returns the type of thread from which this function call is made. The available thread types are BIOS_ThreadType_Hwi, BIOS_ThreadType_Swi, BIOS_ThreadType_Task, and BIOS_ThreadType_Main.

The BIOS module defines the BIOS_WAIT_FOREVER and BIOS_NO_WAIT constants, which can be used with APIs that have a timeout argument.

The BIOS module provides several configuration parameters that control global SYS/BIOS behavior. For example, **Heap Size** is described in Section 7.3.1. Various ways to reduce the size of a SYS/BIOS application by setting BIOS module configuration parameters are shown in Section 9.4 and Appendix C.

6.3 System Module

The System module provides basic low-level "system" services, such as character output, printf-like output, and exit handling.

You can configure properties for the System module in SysConfig. First, enable the module, which is listed in the **Runtime** category. Click the small, circled **?** icons in SysConfig for information about properties. See the Doxygen API Reference described in Section 1.8 for details about the APIs.

The package path to the System module is ti.sysbios.runtime.System, so SYS/BIOS applications should contain the following #include statement:

```
#include <ti/sysbios/runtime/System.h>
```

The following System module APIs are useful in SYS/BIOS applications:

- `System_printf()` — Several functions similar to printf are provided. These include System_putch(), System_sprintf(), System_vprintf(), System_aprintf(), and System_vsnprintf() (print a specified number of characters to a character buffer using a varargs list to pass the arguments).

  We strongly recommend that SYS/BIOS applications call System_printf() and its related functions in place of the standard printf() function. The System module provides familiar printf-like functionality but with fewer options. The memory footprint is much smaller than traditional printf(). System_printf() allows users to specify the handling of the character output using a System provider (see Section 6.3.1 and Section 6.3.2).

- `System_abort()` — Your application can call this function when it needs to abort abnormally and return an error message. This function allows you to return a string describing the error that occurred. When this function is called, the System gate is entered, the SystemSupport module's abort function is called, and then System.abortFxn is called. No exit functions bound via System_atexit() or the ANSI C Standard Library atexit() functions are called.

- `System_flush()` — This function sends any buffered output characters to the output device. It also issues a break point to the IDE. Because this call halts the target, calling it can affect the details of real-time execution. The destination for the output characters is determined by the SystemSupport module.

- `System_exit()` — It is recommended that SYS/BIOS applications call BIOS_exit() instead of the System_exit() function. The BIOS_exit() function performs internal cleanup before calling System_exit(). The System_exit() may be called from a Task, but cannot be called from a Swi or Hwi.

- `System_atexit()` — Call this function to add an exit handler to the internal stack of functions to be executed when System_exit() is called. The **Maximum Atexit Handlers** property of the System module controls how many exit handlers can be stacked. The default is 8.

The System module uses a support module to implement low-level services needed by SYS/BIOS. The implementation to use is specified by the **SystemSupport Module** property of the System module. The default is the SysMin module (Section 6.3.1). Another option is the SysCallback module (Section 6.3.2).

Exit Functions

By default, when an application exits with no error condition, the System module calls its standard exit function, which in turn calls the ANSI C Standard exit() function. If you would like your application to spin indefinitely, so that you can debug the application's state, change the **exit function** property of the System module to System_exitSpin.

Configuring this "spin" function also reduces the code size of your application. The System_exitSpin() API should not be called directly by applications.

You can also use the System_atexit() function in C code to specify multiple exit handlers at runtime. The prototype for a custom exit handler is:

```
typedef void (*System_AtexitHandler)(int);
```

Abort Functions

By default, when an application aborts due to an error condition, the System module calls its standard abort function, which in turn calls the ANSI C Standard abort() function. If you would like your application to spin indefinitely, so that you can debug the application's state, change the **abort function** property of the System module to System_abortSpin.

Configuring this "spin" function also reduces the code size of your application. The System_abortSpin() API should not be called directly by applications.

The prototype for a custom abort handler is:

```
typedef void (*System_AbortFxn)();
```

6.3.1 SysMin Module

The SysMin module is the SystemSupport module used by most SYS/BIOS examples and templates. This module provides implementations of the functions required for system support. This includes functions to flush buffered characters, output a single character, and perform exit and abort actions.

The SysMin module is recommended for most applications, because it places characters into a circular buffer that the Runtime Object View (ROV) tool knows how to find and present.

The module maintains an internal circular buffer on the target to store the "output" characters. When the buffer is full, data is over-written. When a function is called that flushes the buffer, characters in the internal circular buffer are "output" using the configured **Character Output function callback**. Unless you provide a custom output function, on TI targets the HOSTwrite() function in the TI C Run Time Support library is used to output the character buffer. On non-TI targets, the ANSI C Standard Library function fwrite() is used.

To use the SysMin module, in SysConfig set the **SystemSupport Module** property of the System module to SysMin. Expand the SysMin section of the System module and set properties as needed. You can modify the **Output Buffer Size** to control the size of the buffer used to store output internally. Click the small, circled **?** icons in SysConfig for information about properties.

An application should not call SysMin APIs directly. See the Doxygen API Reference described in Section 1.8 for details about the APIs.

6.3.2 SysCallback Module

The SysCallback module is an alternative to the SysMin module. It requires you to provide custom functions to handle abort, exit, flush, putch, and ready actions. Use this module if you need to customize the output behavior of your application.

To use the SysCallback module, in SysConfig set the **SystemSupport Module** property of the System module to SysCallback. Expand the SysCallback section of the System module and specify callback function names. Click the small, circled **?** icons in SysConfig for information about properties.

An application should not call SysCallback APIs directly. See the Doxygen API Reference described in Section 1.8 for details about the callback function signatures.

6.4 Startup Module

The Startup module manages the very early startup initialization that occurs before C's main() function is invoked. This initialization typically consists of setting hardware specific registers that control watchdog timers, access to memory, clock speeds, etc.

In addition to configuring custom startup functions, this module also provides services that allow modules to automatically add initialization functions to the startup sequence.

In SysConfig, enable the System module if you want to modify its properties. This module is listed under the **Runtime** category.

See Section 3.1 for an overview of the startup process, including the sequence in which functions configured for the Startup module are run.

The Startup module also allows you to specify functions to run when a reset occurs. Reset functions are called as early as possible in the application startup and are intended for platform-specific hardware initialization.

Note that only certain target families perform a device reset before running a program. As a result, the reset function is *not* supported on all platforms. Do not place code that you intend to be portable in this function.

6.5 Error Module

To configure error handling in SysConfig, set properties for the Error Handling module, which is listed in the **Runtime** category. Click the small, circled **?** icons in SysConfig for information about properties. See the Doxygen API Reference described in Section 1.8 for details about the APIs.

A number of SYS/BIOS APIs—particularly those that create objects and allocate memory—expect an Error_Block argument. These Error_Block arguments are supported in SYS/BIOS 7.x primarily to remain API compatible with SYS/BIOS 6.x. While the legacy Error_Block/Error_init()/Error_check() mechanism, continues to work, we recommend passing one of the following values for this argument:

- **Error_IGNORE** bypasses the Error subsystem and immediately returns back to the calling function. The calling function should simply check the return value of the function.
- **Error_ABORT** (or NULL) causes the Error subsystem to perform the following sequence of actions:
  - Print the error if the printDetails configuration property is set to true.
  - Call the Error.raiseHook if it is set to a non-empty value in the configuration.
  - Call System_abort(), which halts the processor. Alternately, you can configure your own System abort function to perform custom abort handling. For example, a function might log the abort before jumping to the reset vector or to c_int00 to restart the application.

The path to the Error module is ti.sysbios.runtime.Error, so SYS/BIOS applications that use Error APIs should contain the following #include statement:

```
#include <ti/sysbios/runtime/Error.h>
```

The following Error module APIs may be useful in your SYS/BIOS applications:

- `Error_init()` — Puts an error block into its initial state.
- `Error_check()` — Returns true if an error was raised.
- `Error_print()` — Prints an error using System_printf().
- `Error_raise()` — Raises an error.

This example raises an error in response to an error condition:

```
if (val % 2) {
    Error_raise(eb, Error_E_generic, "Value is not a multiple of 2", 0);
}
```

You can configure how SYS/BIOS applications respond to raised errors with the Error.policy, Error.policyFxn, and Error.raiseHook configuration parameters.

By default, the Error.policy is to return errors to the calling function (Error_UNWIND).

If the Error.policy is Error_TERMINATE, and the error block parameter is not Error_IGNORE, all raised errors are fatal and calls to Error_raise() do not return to the caller. Recall that Error_IGNORE bypasses the Error module and returns immediately.

By default, the Error.policyFxn is Error_policyDefault(), which processes the error and logs it before returning to the caller or aborting, depending on the Error.policy. Alternately, you can use Error_policySpin(), which simply loops infinitely, to minimize the target footprint.

You can specify an Error.raiseHook function to be called whenever an error is raised, even if the Error.policy is TERMINATE. By default, this function is set to Error_print(), which causes the error to be formatted and output by System_printf(). Setting this configuration parameter to null indicates that no function hook should be called.

Texas Instruments

Chapter 7

SPRUIX7—November 2020

Memory

This chapter describes issues related to memory use in SYS/BIOS.

Topic **Page**

7.1	**Background**	**115**
7.2	**Stacks**	**115**
7.3	**Dynamic Memory Allocation**	**116**
7.4	**Heap Implementations**	**118**

7.1 Background

This chapter deals with the Memory API and the various heap modules provided with SYS/BIOS. Code and data are placed in memory regions by the linker using standard linker command file syntax.

- Section 7.2 discusses stacks, including how to configure the system stack and task stacks.
- Section 7.3 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.
- Section 7.4 describes various heap implementations.

7.2 Stacks

SYS/BIOS uses a single system stack for hardware and software (Hwi and Swi) interrupts and a separate task stack for each Task instance.

7.2.1 System Stack

The system stack size and location are controlled in the application's linker .cmd file. See the SDK examples for stack size and placement examples for the various toolchains. 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.

See your project's auto-generated linker command file for symbols related to the system stack size and location.

7.2.2 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.

In SysConfig, enable the Task module if you want to modify its properties. This module is listed under the **Core Kernel** category. For example, you can set the maximum **Number of Task Priorities**, the **Default Stack Size**, and the **idleTaskStackSize**.

7.3 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.

7.3.1 Specifying the Default System Heap

The BIOS module creates a default system heap for use by SYS/BIOS. When Memory_alloc() is called at runtime with a NULL heap, this system heap will be used. By default, this system heap created by the BIOS module is a HeapMem instance.

In SysConfig, you can set the following properties for the BIOS module to control the system heap:

- **Default Memory Heap Type:** Choose a heap implementation to manage the heap. The options are HeapMem, HeapMin, HeapCallback, and HeapUser.

  If you want to use the HeapBuf or HeapMultiBuf implementation, select HeapUser. Then in your main() function, create a HeapBuf or HeapMultiBuf instance and call Memory_setDefaultHeap() with the heapBuf or HeapMultiBuf handle. This call to Memory_setDefaultHeap() needs to occur before any call to Memory_alloc() or any *Mod*_create().
- **Heap Size:** The size of the default system heap. If you do not set this property, you can set the Base address and End address properties instead.
- **Base address of the ’Primary Heap’ buffer:** The starting location of the default system heap specified using an address or symbol defined in the linker command file. If you set both the Base and End addresses, the Heap Size is ignored. Set both the Base address and the End address if you set either one.
- **End address of the ’Primary Heap’ buffer, plus one:** The ending location of the default system heap specified using an address or symbol defined in the linker command file.
- **Use HeapTrack with system default heap:** Enabling this property is useful when you are trying to debug memory overflow issues. A tracker packet is added to each allocated buffer, and the information can be viewed in RTOS Object Viewer (ROV). Asserts are raised if buffer overflows, memory leaks, allocation/deallocation problems, or other issues occur.

For example, if you want to use HeapMin as the **Default Memory Heap Type**, you can configure properties in SysConfig as follows:

- **Default Memory Heap Type:** HeapMin
- **Heap Size:** 0x00002000
- **Use HeapTrack with system default heap:** false

If you do not want a system heap to be created, you can set **Heap Size** to zero. The BIOS module will not create a heap, which minimizes code/data usage.

7.3.2 Using the Memory Module

All dynamic allocation is done through the ti.sysbios.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 selected Heap implementation is responsible for managing the memory. Calling Memory APIs rather than Heap implementation APIs makes applications and middleware portable and not tied to a particular heap implementation.

IHeap_Handles used with Memory APIs are obtained by creating Heap instances. When The IHeap_Handle passed to the Memory APIs is NULL, the default system heap is used. See Section 7.3.1.

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 <ti/sysbios/runtime/IHeap.h>
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/runtime/Memory.h>
#include <ti/sysbios/runtime/Error.h>

extern IHeap_Handle systemHeap, otherHeap;

void main()
{
    void *buf1, *buf2;

    /* Alloc and free using systemHeap */
    buf1 = Memory_alloc(NULL, 128, 0, Error_IGNORE);
    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, Error_IGNORE);
    if (buf2 == NULL) {
        System_abort("Memory allocation for buf2 failed");
    }
    Memory_free(otherHeap, buf2, 128);
}
```

7.3.3 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 7.3.1).

To change the size of the heap used by malloc(), set either the **Heap size** property or the **Base address** and **End address** properties of the BIOS module.

7.4 Heap Implementations

The ti.sysbios.runtime.Memory module is the common interface to all memory operations. Actual memory management is performed by a Heap instance, such as an instance of HeapMem or HeapBuf. For example, Memory_alloc() can be called to allocate memory. All Memory APIs take a Heap instance as one of their parameters. Internally, the Memory module calls the heap's interface functions.

SYS/BIOS provides the following Heap implementations:

- **HeapMin.** Very small code footprint implementation. Supports memory allocation, but does not support freeing memory. See Section 7.4.1.
- **HeapMem.** Allocate variable-size blocks. See Section 7.4.2.
- **HeapBuf.** Allocate fixed-size blocks. See Section 7.4.3.
- **HeapMultiBuf.** Specify variable-size allocation, but internally allocate from a variety of fixed-size blocks. See Section 7.4.4.
- **HeapTrack.** Used to detect memory allocation and deallocation problems. See Section 7.4.6.
- **HeapCallback.** Allows you to fully customize heap behavior by specifying functions to run for all heap-related events. See Section 7.4.5.

Different heap implementations optimize for different memory management traits. The HeapMem module (Section 7.4.2) accepts requests for all possible sizes of blocks, so it minimizes internal fragmentation. The HeapBuf module (Section 7.4.3), 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. This table compares SYS/BIOS heap implementations. See the Doxygen API Reference described in Section 1.8 for details.

Table 7–1. Heap Implementation Comparison

Module	Description/Characteristics	Limitations
ti.sysbios.runtime.HeapMin	Very small code size, non-blocking. No support for freeing heap memory.	free() not supported
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
ti.sysbios.heaps.HeapCallback	Fully customizable heap implementation.	Requires creating of callback functions.

7.4.1 HeapMin

HeapMin is a minimal footprint heap implementation. This memory implementation is designed for applications that generally allocate memory and create module instances at runtime, but never delete created instances or free memory explicitly.

HeapMin does not support freeing memory. By default, an application aborts with an error status if it calls HeapMin_free(). The HeapMin **Raise Error if free() is called** property can be set to “false” to cause HeapMin_free() to simply return without raising an error.

If you call HeapMin_create() at runtime, the C code is responsible for specifying the buffer that the heap will manage and aligning the buffer.

7.4.2 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*.

You cannot create HeapMem instances beyond the default system heap in SysConfig. Other heaps must be created in runtime C code. However, you can configure the **Module Gate Type** used by HeapMem. Options are GateHwi, GateSwi, GateMutex, and GateMutexPri. See Section 4.3 for information about Gate implementations. Other Heap implementations do not use a Gate internally; instead they use Hwi_disable/Hwi_restore to protect the heap.

Runtime example: This example uses C code to dynamically create a HeapMem instance with a size of 1024 MAUs:

```
HeapMem_Params prms;
static char buf[1024];
HeapMem_Handle heap;

...

HeapMem_Params_init(&prms);
prms.size = 1024;
prms.buf = (void *)buf;
heap = HeapMem_create(&prms, Error_IGNORE);
if (heap == NULL) {
    System_abort("HeapMem create failed");
}
```

HeapMem uses a Gate (see Section 4.3 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 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 for more information.

7.4.3 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.

To use the HeapBuf implementation, in the BIOS module of SysConfig, set the **Runtime Support Library Lock Type** to HeapUser. No system heap will be created at startup. Your C code should create a heap at runtime and use the Memory_setDefaultHeap() API to register the heap with the Memory module. You cannot create HeapMultiBuf instances beyond the default system heap in SysConfig. Other heaps must be created in runtime C code.

You cannot create HeapBuf instances beyond the default system heap in SysConfig. Other heaps must be created in runtime C code. However, you can configure the **Track maximum number of outstanding alloc(0s** property. Setting this property to true enables tracking of the maximum number of allocations during the lifetime of each HeapBuf instance.

Runtime example: This second example uses C code to dynamically create a HeapBuf instance with 10 memory blocks of size 128. 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 devices with a 32-bit architecture, the 8-byte alignment is used. The base address of the buffer should also be aligned to this same size.

```
HeapBuf_Params prms;
static char buf[1280];
HeapBuf_Handle heap;

...

HeapBuf_Params_init(&prms);
prms.blockSize = 128;
prms.numBlocks = 10;
prms.buf = (void *)buf;
prms.bufSize = 1280;
heap = HeapBuf_create(&prms, Error_IGNORE);
if (heap == NULL) {
    System_abort("HeapBuf create failed");
}
```

7.4.4 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.

To use the HeapMultiBuf implementation, in the BIOS module of SysConfig, set the **Runtime Support Library Lock Type** to HeapUser. No system heap will be created at startup. Your C code should create a heap at runtime and use the Memory_setDefaultHeap() API to register the heap with the Memory module. You cannot create HeapMultiBuf instances beyond the default system heap in SysConfig. Other heaps must be created in runtime C code.

A HeapMultiBuf instance 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 Doxygen API Reference described in Section 1.8 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.

<table>
<tr><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td><td>16 MAUs</td></tr>
<tr><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td></tr>
<tr><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td><td colspan="2">32 MAUs</td></tr>
<tr><td colspan="8">128 MAUs</td></tr>
<tr><td colspan="8">128 MAUs</td></tr>
<tr><td colspan="8">128 MAUs</td></tr>
<tr><td colspan="8">128 MAUs</td></tr>
<tr><td colspan="8">128 MAUs</td></tr>
</table>

Runtime example: This example uses C code to dynamically create a HeapMultiBuf instance:

```
HeapMultiBuf_Params prms;
HeapMultiBuf_Handle heap;

/* 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 = (void *)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 = (void *)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 = (void *)buf2;
prms.bufParams[2].bufSize = 640;

heap = HeapMultiBuf_create(&prms, Error_IGNORE);
if (heap == NULL) {
    System_abort("HeapMultiBuf create failed");
}
```

7.4.5 HeapCallback

If you configure the **Default Memory Heap Type** in the BIOS module to use HeapCallback, you must also configure the HeapCallback module with callback functions to call when heap create, init, delete, alloc, free, getStats, and isBlocking functions need to run. Your create function will be called at initialization time with a NULL HeapCallback_Params argument.

See the Doxygen API Reference described in Section 1.8 for the HeapCallback module for details about the signatures for these functions.

7.4.6 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 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 Runtime Object View (ROV) to display information about the heap instance.

To enable HeapTrack for the default system heap, use SysConfig to set the **Use HeapTrack with system default heap** property of the BIOS module to true. To enable HeapTrack for other heaps, use runtime C code.

HeapTrack implements standard heap functions and two debugging functions. HeapTrack_printHeap() prints all the memory blocks for a given HeapTrack instance. HeapTrack_printTask() prints all memory blocks for a given Task handle.

ROV is the best tool for examining HeapTrack. The runtime print functions are not used often. See the video on Debugging Applications that use TI-RTOS for more about using HeapTrack. (Note that the configuration information in this video is outdated. However, the ROV information is current.)

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.

Runtime example: This example uses C code to dynamically create a HeapTrack instance with an existing heap.

```
HeapTrack_Params prms;
HeapTrack_Handle heap;

HeapTrack_Params_init(&prms);
prms.heap = heapHandle;
heap = HeapTrack_create(&prms, Error_IGNORE);
if (heap == NULL) {
    System_abort("HeapTrack create failed");
}
```

Texas Instruments

Chapter 8
SPRUIX7—November 2020

Hardware Abstraction Layer

This chapter describes modules that provide hardware abstractions.

Topic **Page**

8.1 Hardware Abstraction Layer APIs **126**

8.2 Hwi Module **127**

8.1 Hardware Abstraction Layer APIs

SYS/BIOS provides services for configuration and management of interrupts 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, and multiplexing of multiple interrupts to a single vector are described in this chapter.

Note: Any configuration or manipulation of interrupts, their associated vectors, 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 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 are also given. For a full description of the target/device-specific APIs available for a particular family or device, please refer to the Doxygen API Reference described in Section 1.8.

The SYS/BIOS modules in the ti.sysbios.hal package: Hwi, Timer, and Seconds require target/device-specific API implementations to achieve their functionality. SysConfig simplifies and hides access to target-specific implementations where possible, but you can find the implementations in the ti.sysbios.family package. To access to the extended API sets available for the target-specific APIs, your C code must include its header file.

For hardware-specific information about using SYS/BIOS, see Appendix A.

8.2 Hwi Module

The ti.sysbios.family.arm.m3.Hwi module provides APIs for managing hardware interrupts. These APIs should provide sufficient functionality for most applications. See Section 3.3 for more about Hwi interrupts and their interactions with other types of threads.

You can configure general behaviors of the Hwi module in SysConfig. First, enable the module, which is listed in the **HAL** category. Click the small, circled **?** icons in SysConfig for information about properties. See the Doxygen API Reference described in Section 1.8 for details about the APIs.

8.2.1 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 Hwi_create() function is supported by the ti.sysbios.family.arm.m3.Hwi module. However, Hwi_construct() is not supported because the Hwi module is implemented through device-specific modules.

You cannot create Hwi instances in SysConfig. Use runtime C code to create Hwi instances.

Runtime example: The following C code creates a Hwi object that associates interrupt 5 with the “myIsr” C function using default instance parameters:

```
#include <ti/sysbios/family/arm/m3/Hwi.h>
#include <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/runtime/System.h>

Hwi_Handle myHwi;

...

myHwi = Hwi_create(5, myIsr, NULL, Error_IGNORE);
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.

8.2.2 Hwi Instance Parameters

The following 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.family.arm.m3.Hwi module in the online documentation.

- The "maskSetting" defines how interrupt nesting is managed by the interrupt dispatcher.

```
MaskingOption maskSetting = MaskingOption_SELF;
```

- The “arg” parameter will be passed to the Hwi function when the dispatcher invokes it.

```
uintptr_t arg = 0;
```

- The "enabledInt" parameter is used to automatically enable or disable an interrupt upon Hwi object creation.

```
bool enableInt = true;
```

- 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;
```

For information about the parameter structure and individual parameters for instances of this module, see the Doxygen API Reference described in Section 1.8.

8.2.3 Creating a Hwi Object Using Non-Default Instance Parameters

Building on the example given in Section 8.2.1, the following example shows 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.

Runtime example:

```
#include <ti/sysbios/family/arm/m3/Hwi.h>
#include <ti/sysbios/runtime/Error.h>

Hwi_Params hwiParams;
Hwi_Handle myHwi;

...

/* initialize hwiParams to default values */
Hwi_Params_init(&hwiParams);

hwiParams.arg = 10;
hwiParams.enableInt = false;

myHwi = Hwi_create(5, myIsr, &hwiParms, Error_IGNORE);
if (myHwi == NULL) {
    System_abort("Hwi create failed");
}
```

8.2.4 Enabling and Disabling Interrupts

You can enable and disable interrupts globally as well as individually with the following Hwi module APIs:

- `unsigned int Hwi_enable();`
  Globally enables all interrupts. Returns the previous enabled/disabled state.
- `unsigned int Hwi_disable();`
  Globally disables all interrupts. Returns the previous enabled/disabled state.
- `Hwi_restore(unsigned int 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.:
  - `unsigned int Hwi_enableInterrupt(unsigned int intNum);`
  - `unsigned int Hwi_disableInterrupt(unsigned int intNum);`
  - `Hwi_restoreInterrupt(unsigned int key);`
- `Hwi_clearInterrupt(unsigned int intNum);`
  Clears "intNum" from the set of currently pending interrupts.

Disabling hardware interrupts is useful during a critical section of processing.

On SimpleLink devices, Hwi_disable() raises the interrupt priority level to disable all interrupts of lower priority. Hwi_restore() restores the previous priority. This max priority level is configurable. See Section A.2.2 for information about creating zero-latency interrupts on Cortex-M devices.

There is a situation in which interrupts become re-enabled when you would expect them to remain disabled. This occurs when a Task thread calls Mailbox_post(), Semaphore_post(), or Event_post()—or a similar call not including Swi_post()—while hardware interrupts are disabled but the task scheduler is enabled. All XXX_post() APIs (except Swi_post()) call Task_restore() internally, which in this case ends by calling Hwi_enable().

8.2.5 A Simple Example Hwi Application

The following example creates an Hwi object for interrupt number 6. An idle function that waits for the interrupts to complete is also added to the Idle function list.

The SysConfig configuration for this example should do the following:

- Enable the BIOS, System, Hwi, and Idle modules.
- For the Idle module, add an instance that calls `myIdleFunc` as the Idle Function.

Runtime example:

```
#include <ti/sysbios/runtime/System.h>
#include <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/family/arm/m3/Hwi.h>

bool Hwi6 = false;


main(void)   {
    Hwi_Params hwiParams;
    Hwi_Handle myHwi;

    /* Initialize hwiParams to default values */
    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, Error_IGNORE);
    if (myHwi == NULL) {
        System_abort("Hwi create failed");
    }

    /* enable both interrupts */
    Hwi_enableInterrupt(6);

    /* start BIOS */
    BIOS_start();
}

/* Runs when interrupt 6 occurs */
void myIsr6(uintptr_t 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 (Hwi6) {
        System_printf("Both interrupts have occurred!");
        System_exit(0);
    }
}
```

8.2.6 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.

For hardware-specific information about using SYS/BIOS, see Appendix A.

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.

8.2.7 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.

TEXAS INSTRUMENTS

Chapter 9

SPRUIX7—November 2020

Optimization

This chapter describes modules and other tools that can be used for instrumentation purposes.

Topic **Page**

9.1 Overview **133**
9.2 Load Module **133**
9.3 Performance Optimization **134**
9.4 Memory Optimization **135**

9.1 Overview

Several modules discussed in this chapter are provided in the ti.sysbios.runtime package. See the Doxygen API Reference described in Section 1.8 for details.

When you are debugging an application, there are several items in the **Tools** menu that are especially useful for debugging SYS/BIOS applications. **Tools > Runtime Object View (ROV)** is a stop-mode debugging tool for use with SYS/BIOS applications. For information, see the Runtime Object View (ROV) User's Guide in TI Resource Explorer.

9.2 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 as the count stored by the ti.sysbios.runtime.Timestamp module (see Section 5.5). The CPU load is reported in percentages.

By default, load data is gathered for all threads. You can configure the Load module to select which type(s) of threads are monitored.

9.2.1 Load Module Configuration

The Load module has been set up to provide data with as little configuration as possible. Using the default configuration, load data is gathered roughly every 500 ms.

To enable the collection of Load data, make the following settings in SysConfig:

- Enable the Load module, which is listed in the **Utils** category.
- Set properties to true to enable load monitoring for the Hwi, Swi, and Task modules as needed.

For more information about the Load module, see the help in SysConfig and the Doxygen API Reference described in Section 1.8.

9.2.2 Obtaining Load Statistics

Load statistics recorded by the Load module can be obtained in the following ways:

- **Runtime Object View (ROV.)** Load statistics can be viewed in ROV. For information about viewing this information, see the Runtime Object View (ROV) User's Guide in TI Resource Explorer.
- **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_t hwiLoad;

Load_getGlobalHwiLoad(&stat);
hwiLoad = Load_calculateLoad(&stat);
```

9.3 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.

9.3.1 Choosing a Heap Manager

SYS/BIOS provides several different heap manager implementations. Each of these has various performance trade-offs when allocating and freeing memory. See Section 7.4 for a detailed discussion of the trade-offs of each module.

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 HeapBuf. HeapMin uses minimal memory but cannot deallocate memory.

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 allocating and freeing many fixed sized data buffers, you could create a HeapBuf instance just for allocating these data buffers.

9.3.2 Hwi Module 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 in SysConfig to reduce interrupt latency.

- **Dispatcher Auto Nesting Support.** You may disable this feature if you don't need interrupts enabled during the execution of your Hwi functions.
- **Dispatcher Swi Support.** You may disable this feature if no Swi threads will be posted from any Hwi threads.
- **Dispatcher Task Support.** 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.

9.3.3 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.

9.3.4 Zero Latency Interrupts

See Section A.2.2 for target-specific information about configuring zero latency interrupts.

9.4 Memory Optimization

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 actual benchmark data is provided in the `SDK_INSTALL_DIR\docs\tirtos\benchmarks.html` file.

The following sections describe 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.

9.4.1 Reducing Data Size

9.4.1.1 Removing the malloc Heap

Calls to malloc are satisfied by a separate heap, whose size is configurable. You can remove this heap to minimize the 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.

To remove this heap in SysConfig, set the **Heap Size** property of the BIOS module to 0.

9.4.1.2 Reducing the Size of Stacks

The system stack size is specified as the HEAPSIZE in the project's linker command file (.cmd). The system stack is used as the stack for interrupt service routines (Hwis and Swis) and can be reduced depending on the application's needs. See Section 3.4.3 for information about system stack size requirements and Section 3.5.3 for information about task stack size requirements.

Each Task has its own stack. You can configure the default size of Task stacks in SysConfig by changing the value of the **Default Stack Size** property of the Task module. Configure the size of the stack that serves all the Idle threads by changing the value of the **idleTaskStackSize** property of the Task module.

The sizes of individual Task stacks are configurable when you create Tasks at runtime, and can be reduced depending on the application's needs.

```
Task_Params_init(&taskParams);
taskParams.stackSize = 512;
Task_create(task0Fxn, &taskParams, NULL);
```

9.4.1.3 Leaving Text Strings Off the Target

Text strings are used by the Assert and Error modules. To prevent strings from being stored on the target, in SysConfig, set both the **Use System_printf() to print details** and the **Leave strings in memory (flash)** properties of the Error Handling module to false. In addition, set the **Add file and line info to assert messages** property of the Assertion Handling module to false.

9.4.1.4 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 actually intend to use. To reduce this number, set the **Maximum Atexit Handlers** property of the System module in SysConfig.

9.4.2 Reducing Code Size

9.4.2.1 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. These can be disabled in the BIOS module page in SysConfig.

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.

9.4.2.2 Eliminating printf()

There is no way to explicitly remove printf from the application. However, printf code and related data structures are not included if an application has no references to System_printf(). To reduce code size:

- Remove any calls to System_printf().
- Configure the application as follows in SysConfig to reduce code size:
  - Enable the SysMin module. Set **Output Buffer Size** to 0 and disable the **Flush At Exit** property.
  - Set the **Optional function to call when an error is raised** property of the Error Handling module to NULL.

Essentially, these settings will eliminate all references to the printf code.

9.4.2.3 Disabling RTS Thread Protection

If an application does not require the RTS library to be thread safe, it can set the **Runtime Support Library Lock Type** property of the BIOS module to false. This causes no Gate module to be used.

9.4.2.4 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. Disable the **Initialize Stack Flag** and **Check Stack Flag** properties of the Task module in SysConfig if you want to disable stack checking for all Tasks:

9.4.2.5 Cortex-M3/M4 Exception Management

The Cortex-M exception module uses System_printf() to send useful information about an exception. This is useful when debugging, but it uses some code and data space. In SysConfig, toggling the **Enable Exception Decoding at runtime** property of the Hwi module off causes the exception module to simply spin in an infinite loop when an exception occurs.

9.4.3 Basic Size Benchmark Configuration

The .syscfg configuration files for the driver examples include descriptions of individual properties and their settings with regards to their effects on memory footprint.

The basic size benchmark configuration combines the techniques described in previous sections to create an application that is close to the smallest possible size of a SYS/BIOS application. The following settings are configured in SysConfig:

- Enable the BIOS, Hwi, Task, SysMin, and System modules.
- Set **Heap Size** to 0 in the BIOS module.
- Set **Maximum Atexit Handlers** to 4 in the System module.
- Set **Output Buffer Size** to 128 and disable **Flush At Exit** in the SysMin module.
- Set **Default Stack Size** to 1024 and **Idle Task Stack Size** to 1024 in the Task module.

In the linker command file for the project, set HEAPSIZE to 1024.

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.

TEXAS INSTRUMENTS

Appendix A

Device Addendum

This appendix highlights device-specific information about using SYS/BIOS on a Cortex-M device.

Topic **Page**

A.1 ARM Cortex-M Introduction 139
A.2 Cortex-M Hardware Interrupt (Hwi) Handling 139
A.3 Cortex-M Operating Modes and Stack Usage. 141
A.4 Cortex-M Timer Usage by SYS/BIOS Modules 142
A.5 Cortex-M Timer Device Details. 145
A.6 FAQ. 147

A.1 ARM Cortex-M Introduction

SYS/BIOS runs on several different Texas Instrument devices that have an ARM Cortex-M core.

- SimpleLink Devices (e.g., CC13xx, CC26xx, CC32xx)

A.2 Cortex-M Hardware Interrupt (Hwi) Handling

The Cortex-M CPU natively supports efficient interrupt handlers written in C. In order to support the three-tiered threading model provided by SYS/BIOS (Hwis, Swis, and Tasks), the native interrupt support provided by the Cortex-M core must be augmented with a few housekeeping functions that enforce the expected thread execution behavior.

To consolidate the overhead of these housekeeping functions and provide a standardized set of instrumentation features, SYS/BIOS uses a centralized interrupt dispatcher that invokes the application's custom ISR (Hwi) functions.

The interrupt dispatcher provides the following functionality:

- The interaction of Hwi, Swi, and Task threads is carefully orchestrated:
  - Hwi threads can schedule execution of Swi threads by means of the various Swi_post() APIs.
  - Hwi threads can schedule execution of Task threads by means of the Semaphore_post() and Event_post() APIs.
- Application-specific Hwi begin and end hook functions are safely invoked at standardized times.

By default, all SYS/BIOS managed interrupts are routed to the interrupt dispatcher, which subsequently invokes the user's interrupt handler.

The low-level Hwi module used on the Cortex-M devices is the ti.sysbios.family.arm.m3.Hwi module. For more details, please read the Doxygen API Reference described in Section 1.8.

A.2.1 Hwi MaskingOptions and Priorities

A.2.1.1 Supported MaskingOptions

The Cortex-M's Nested Vectored Interrupt Controller (NVIC) natively supports configurable interrupt priorities.

The SYS/BIOS interrupt dispatcher used in Cortex-M devices is designed to support the native interrupt nesting functionality of the NVIC. Consequently, only the Hwi.MaskingOption_LOWER enumeration is honored for an individual Hwi's params.maskingOption setting. No support is provided for the following Hwi masking options:

- Hwi.MaskingOption_NONE
- Hwi.MaskingOption_ALL
- Hwi.MaskingOption_SELF
- Hwi.MaskingOption_BITMASK

A.2.1.2 Supported Priority Values

While the Cortex-M can support up to 256 priority settings, the amount of internal chip real-estate required to support this number of priorities is exceedingly large. Since most embedded system applications can be fully supported with much fewer interrupt priorities, only 8 priority levels are supported on the TI Cortex-M devices (except the Ducati and Benelli cores which support 16).

The 3 bits of priority required to define the 8 priority values occupy bits [7:5] of the 8-bit priority value rather than bits [2:0] as one might expect. Consequently, the values of the 8 supported priorities are not 0x00 through 0x07 but 0x00, 0x20, 0x40, 0x60, 0x80, 0xa0, 0xc0, and 0xe0, where 0x00 is the highest priority and 0xe0 the lowest priority.

Additionally, the NVIC design defines interrupt priority groups within a given priority setting. The Hwi.priGroup configuration parameter allows you to specify which of the 3 significant bits of the priority setting are group bits and which are priority bits. For a full discussion of NVIC's priority and priority-group behavior, see the "NVIC register descriptions" section of the ARM Cortex-M3 documentation.

No validity checking is performed on the Hwi_params.priority setting. You should make sure the configured interrupt priorities will be effective for your application.

A.2.2 Zero Latency Interrupt Support

The Hwi module used by Cortex-M devices supports Zero Latency (unmanaged) Interrupts. When configured, a Zero Latency Interrupt is never disabled internally by SYS/BIOS functions, not even during critical thread execution (thus the name Zero Latency). Thus, a zero latency interrupt handler should never call any BIOS APIs.

To define an interrupt that is never disabled by SYS/BIOS, use the Hwi_create() C API and set the params.priority value to anything less than the Hwi_disablePriority setting. By default, priority 0 is reserved for zero latency interrupts. You can reserve additional priority levels for zero latency interrupts by changing the setting of the **Hwi_disable() priority** property of the Hwi module in SysConfig.

Remember that zero latency interrupts are not handled by the SYS/BIOS interrupt dispatcher and are never disabled. Consequently, in order to guarantee the integrity of the SYS/BIOS thread scheduler, zero latency interrupt handlers are severely restricted in terms of the SYS/BIOS APIs they can invoke. Specifically the following APIs cannot be called within a zero latency interrupt handler:

- Semaphore_post()
- Event_post()
- Swi_post(), nor any of its derivatives
- System_printf(), unless the interrupt handler is the only user of this API

This code example maps the function myZeroLatencyHwi() to the GPIO Port A interrupt (id = 16):

```
#include <ti/sysbios/runtime/Error.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/family/arm/m3/Hwi.h>

int main(int argc, char* argv[]) {
    Hwi_Params params;

    Hwi_Params_init(&params);
    params.priority = 0;
    Hwi_create(16, myZeroLatencyHwi, &params, Error_IGNORE);

    BIOS_start();
}
```

A.3 Cortex-M Operating Modes and Stack Usage

A.3.1 Operating Modes

Internally, the Cortex-M supports two operating modes: Thread mode and Handler mode. Within Thread mode, the Cortex-M code can run in either Privileged or User mode. By definition, Handler mode code always runs in Privileged mode.

Within SYS/BIOS:

- Swi and Task threads run in Thread mode.
- Hwi threads run in Handler mode.
- All SYS/BIOS internal code as well as application code (Hwi, Swi, and Task threads) runs in Privileged mode.
- For performance reasons, SYS/BIOS does not support User mode.

A.3.2 Stacks

The Cortex-M internally supports two stacks: the Main stack and the Process stack.

After reset and throughout the entire boot up sequence prior to the BIOS_start() invocation in main(), the Cortex-M core runs in Thread mode using the Main stack.

The Main stack always refers to the stack buffer that is defined and managed by the Codegen Tools. This is the buffer that is placed in the .stack section with its size configured using the -stack XXX linker command option.

If Tasks are supported by the application (because BIOS.taskEnabled = true), then within the Task_startup() function, which is called by BIOS_start(), the Cortex-M processor is configured to use the Process stack while in Thread mode and to automatically switch to the Main stack when an interrupt occurs (that is, when transitioning to Handler mode). When execution is subsequently passed to a Task thread, the Process stack then becomes dedicated to Task threads and the Main stack becomes dedicated to Hwi and Swi threads.

As opposed to Hwi threads, which automatically switch to the Main stack when Handler mode is entered, the Swi scheduler manually switches the processor to and from the Main stack before and after Swi threads are run.

If Tasks are not supported by the application (because BIOS.taskEnabled = false), the split between the Process stack and Main stack is not made. All threads run on the Main stack. Consequently, all configured Idle functions (that is, the application's background functions) share the same stack as Hwi and Swi threads.

Within SYS/BIOS:

- Hwi and Swi threads always use the Main stack.
- Task threads always use the Process stack.

A.4 Cortex-M Timer Usage by SYS/BIOS Modules

There are high-level timing modules and low-level timer implementations modules. The following sections provide an overview of these two groups of modules. For details, please refer to the Doxygen API Reference described in Section 1.8.

A.4.1 High-Level SYS/BIOS Timing Modules

This section talks generically about SYS/BIOS timing services. Other sections will go into specific details about which timers are used on the various devices.

Module	Description
ti.sysbios.knl.Clock	The Clock module is responsible for timed services in SYS/BIOS. It generates the periodic system tick. The application can plug in functions that will be called once or periodically by the Clock module.
ti.sysbios.hal.Timer	This module presents a standard interface for using all timer peripherals.
ti.sysbios.hal.Seconds	This module generates a custom time() function in the configuration-generated .c file.
ti.sysbios.runtime.Timestamp	This module provides timestamping services.
POSIX	The POSIX interface offers several timing services (clock, timer, sleep). These reside on top of the SYS/BIOS timing services.

A.4.2 Low-Level SYS/BIOS Peripheral Timer Implementations

The device-specific timer modules available for use are:

Module	Description
ti.sysbios.family.arm.m3.Timer	SysTick
ti.sysbios.family.arm.lm4.Timer	General Purpose (GP) timers
ti.sysbios.family.arm.cc26xx.Timer	RTC timer

These timers can be enabled in SysConfig.

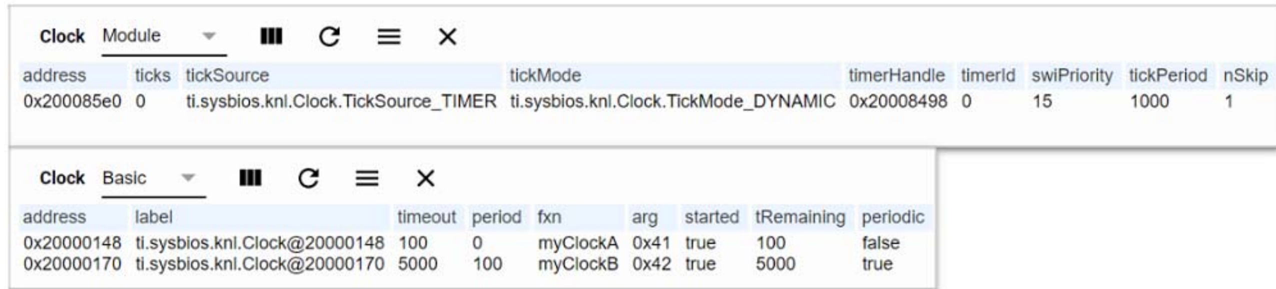
A.4.3 Clock Module

SYS/BIOS, by default, uses one timer peripheral to drive system timings (e.g. Task_sleep(), Semaphore_pend() with a timeout). The timer is managed by the Clock module (ti.sysbios.knl.Clock). The Clock module allows the multiplexing of that single timer for application usage. With the Clock Module, applications can create one-shot and/or periodic functions that will be called by SYS/BIOS.

The basic unit of the Clock module is the tick. For most devices, the default duration of a tick is 1ms. This value is controlled by the **Clock Tick Period** property of the Clock module in SysConfig. The CC13xx/CC26xx devices the Clock period is generally set to 10 microseconds.

All timed kernel actions take place on a tick. For example, when a task sleeps for three ticks (Task_sleep(3)), it will be made ready on the third tick after calling Task_sleep().

Let's look at an example in which the application creates two Clock functions: myClockA and myClockB. myClockA is a one-shot function that will execute once in 100 ticks. myClockB is a periodic function that will first execute in 5000 ticks and then every 100 ticks after that.



Clock Module

address	ticks	tickSource	tickMode	timerHandle	timerId	swiPriority	tickPeriod	nSkip
0x200085e0	0	ti.sysbios.knl.Clock.TickSource_TIMER	ti.sysbios.knl.Clock.TickMode_DYNAMIC	0x20008498	0	15	1000	1

Clock Basic

address	label	timeout	period	fxn	arg	started	tRemaining	periodic
0x20000148	ti.sysbios.knl.Clock@20000148	100	0	myClockA	0x41	true	100	false
0x20000170	ti.sysbios.knl.Clock@20000170	5000	100	myClockB	0x42	true	5000	true

The previous ROV picture was taken when running the following code and halting at BIOS_start().

```
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/runtime/Error.h>
void myClockA(uintptr_t arg1)
{
    // my one-shot timer. arg1 will be 'A'
    // do processing...
    return;
}

void myClockB(uintptr_t arg1)
{
    // my periodic timer. arg1 will be 'B'
    // do processing...
    return;
}

int main(void)
{
    Clock_Params clockParams;

    Clock_Params_init(&clockParams);
    clockParams.period = 0;        // 0 is the default, but added for completeness
    clockParams.startFlag = true;
    clockParams.arg = 'A';
    Clock_create(myClockA, 100, &clockParams, Error_IGNORE);

    clockParams.period = 100;
    clockParams.startFlag = true;
    clockParams.arg = 'B';
    Clock_create(myClockB, 5000, &clockParams, Error_IGNORE);
```

Low power is a key area for all the SimpleLink devices. The Clock module for the CC13xx and CC26xx devices uses a timer that supports “dynamic” mode. Dynamic mode is also called “tickless” mode. When in dynamic mode, the timer interrupt only fires when there is an expected action (e.g. a Task_sleep() should expire) instead of firing at the configured Clock period. Please refer to the docs/tidrivers/Power_Management.pdf file in your SimpleLink SDK for more details.

As shown above, applications can create Clock functions that will be called once or periodically. Doing so allows an application to share the kernel's timer instead of using a different one. If two functions expire on the same Clock tick, the order of execution will match the creation order. All timing must be a multiple of the Clock tick period (e.g. Clock.tickPeriod * N when N is a natural number). Please note, there is jitter with using Clock functions: the time when a Clock function executes is between N-1 and N Clock ticks. For example, let's look at the myClockA Clock function mentioned above. Let's say the Clock_create()

API is called when the current Clock tick was 50. The 50th tick might have just happened or the 51st tick might just be ready to happen. So the actual execution of the myClockA function will be during the 150th Clock tick, but that duration between creation and execution is between 99ms and 100ms. If the time variance is not acceptable for a specific timing requirement, use the Timer module to get a dedicated timer instead of sharing the Clock's timer.

Additionally, drift can occur when using a timer that does not supply an exact Clock tick period. For example, when the Clock tick is set to 1ms and a 2768Hz timer is used, drift will occur.

The Clock module can be configured to not allocate a timer. The application must then call Clock_tick() to provide the timing for the kernel. Please refer to the Clock module in the Doxygen API Reference described in Section 1.8.

In summary, the Clock module can allow an application to essentially share the timer used by the kernel.

A.4.4 HAL Timer Module

The Hardware Abstract Layer (HAL) Timer module allows an application to generically use a Timer. The full path for this module is ti.sysbios.hal.Timer. Internally, it uses a device-specific "family" Timer, such as ti.sysbios.family.arm.m3.Timer.

Please note, a device-specific timer can be created instead of using the HAL Timer module. This is generally done when the device-specific Timer module has features not exposed in the HAL Timer module. This is done in SysConfig, where you can enable various types of timers.

A.4.5 HAL Seconds Module

Much like the Timer module, the Seconds (ti.sysbios.hal.Seconds) module uses device-specific "family" Seconds modules, such as ti.sysbios.family.arm.cc26xx.Seconds.

A.4.6 Timestamp Module

The Timestamp (ti.sysbios.runtime.Timestamp) module provides 32-bit and 64-bit timestamps. The Timestamp module uses a device-specific TimestampProvider module, such as ti.sysbios.family.arm.m3.TimestampProvider. That provider may create a new timer to drive the timestamps or use an existing timer depending on its configuration. Please refer to the Timestamp module in the Doxygen API Reference described in Section 1.8.

A.4.7 POSIX Timing Services

There are various timing support services in the POSIX shim on top of SYS/BIOS. This table shows the underlying SYS/BIOS module for each of the support POSIX timing services.

POSIX API	Underlying SYS/BIOS Module
timer_create	Clock
sleep/usleep	Clock (via Task_sleep)
clock_gettime (CLOCK_MONOTONIC)	Clock
clock_gettime (CLOCK_REALTIME)	Seconds

A.5 Cortex-M Timer Device Details

Now we have the basics of the *timing* modules and the actual timer implementations, let's look at each Cortex-M device specifically to understand how it works.

The tables in the sections that follow show the default timers used for each of the kernel's timing services (e.g. Clock). Please refer to the documentation on the timing service for details on how to configure the module to use a different timer.

A.5.1 CC13xx/CC26xx

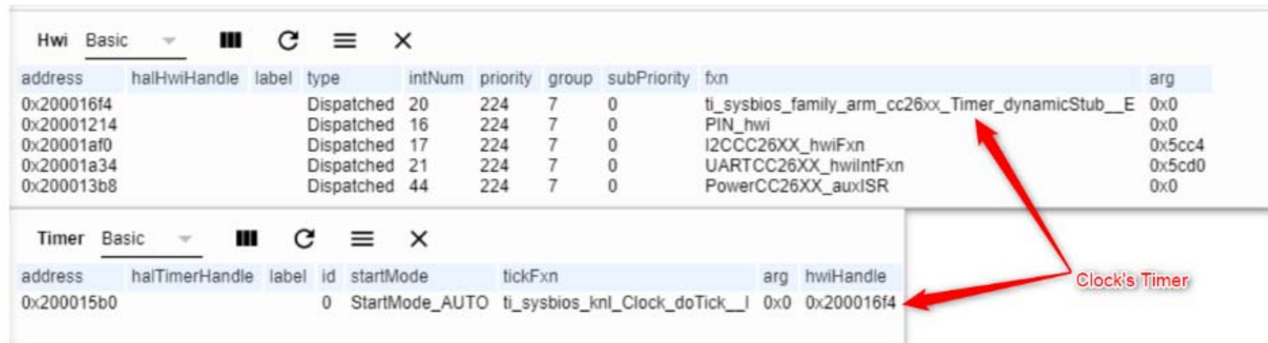
The sub-1GHz and BLE devices are focused on low power while also having strict timing requirements. The RTC peripheral (ti.sysbios.family.arm.cc26xx.Timer) is designed to meet these requirements.

The Clock module uses the RTC peripheral in a "dynamic" mode with a period of 10us. Please note that the 10us period does not imply that the interrupt rate is 10us. It means that the peripheral should support 10us granularity. TI recommends that customers use the RTC for the Clock module and leave the period at 10us for optimal accuracy for the radio stacks. Please refer to the docs/tidrivers/Power_Management.pdf file in your SimpleLink SDK for more details.

Module	Default Timer	Notes
Clock	ti.sysbios.family.arm.cc26xx.Timer	RTC timer supports dynamic mode
Timer	ti.sysbios.family.arm.m3.Timer	SysTick (so only one instance is available). We recommend you use the Clock module for your timing services since the SysTick timer will be reset when going in/out of standby sleeps cycles.
Seconds	ti.sysbios.family.arm.cc26xx.Timer	Shares the Clock's RTC timer.
Timestamp	ti.sysbios.family.arm.cc26xx.Timer	By default the Timestamp module shares the Clock's RTC timer. This is configurable in the ti.sysbios.family.arm.cc26xx.TimestampProvider module.

The image below shows the Runtime Object View for the demos/port example for the CC1350 LaunchPad. The portable example uses the default Clock module settings, Timestamp and creates a POSIX timer to be used by the application. The image shows one timer (and multiple drivers) in the Hwi view and then one timer in the Timers view.

- Clock module's timer (ti.sysbios.family.arm.cc26xx)
- The application-created POSIX timer shares the Clock's timer
- The Timestamp timer shares the Clock's timer



Note: The Clock timer function is ti_sysbios_knl_Clock_doTick__I, but the Hwi stub is ti_sysbios_family_arm_cc26xx_Timer_dynamicStub__E. SYS/BIOS plugs the stub function into the vector table. When that interrupt is asserted, the stub function runs and then calls the Clock's timer function.

A.5.2 CC32xx

Low power is a key area for the CC32xx devices.

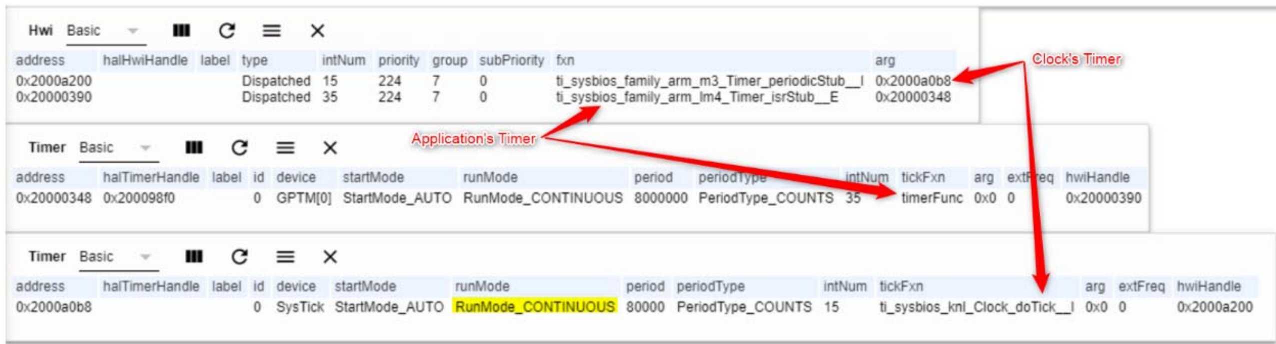
Module	Default Timer	Notes
Clock	ti.sysbios.family.arm.m3.Timer	SysTick
Timer	ti.sysbios.family.arm.lm4.Timer	Device general purpose timer.
Seconds	RTC	The Seconds module for the CC32xx devices uses the RTC timer. It does not use the ti.sysbios.family.arm.cc32xx.Timer module. Instead it directly interfaces with the peripheral to minimize code footprint and optimize performance.
Timestamp	ti.sysbios.family.arm.m3.Timer	By default, the Timestamp module shares the Clock's SysTick timer. This is configurable in the ti.sysbios.family.arm.m3.TimestampProvider module.

The SysTick timer does not continue to increment in LPDS. The kernel uses the RTC to re-adjust the Clock tick after the device comes out of LPDS. Please refer to the docs/tidrivers/Power_Management.pdf document for more details.

The image below shows the Runtime Object View for the sysbios/StairStep example. The StairStep example uses the default Clock module settings, Timestamp and creates a timer to be used by the application. It shows two timers in the Hwi view and then one timer for the ti.sysbios.family.arm.lm4 view and one ti.sysbios.family.arm.m3 view. More specifically, the timers used are as follows:

Clock module's timer (ti.sysbios.family.arm.m3). This is shared by the Timestamp module.

The application-created timer (ti.sysbios.family.arm.lm4).



Note: The runMode for the Clock's timer is RunMode_CONTINUOUS. This is because the SysTick does not support "dynamic" mode.

Note: The application's timer function is timerFunc, but the Hwi stub is ti_sysbios_family_arm_lm4_Timer_isrStub__E. SYS/BIOS plugs the stub function into the vector table. When that interrupt is asserted, the stub function runs and then calls the timer function.

A.6 FAQ

For answers to some frequently asked questions about using SYS/BIOS on Cortex-M, see the SYS/BIOS (TI-RTOS) FAQ thread on the E2E Support Forums.

A.6.1 Configure Divide-by-Zero to be an Exception

By default, out of reset, the M3 core does not treat Divide-by-Zero as an exception. To configure the M3 to treat Divide-by-Zero as an exception, use SysConfig to set the **NVIC CCR Register Setting** property of the Hwi module to DIV_0_TRP.

See the Hwi_CCR struct in the Doxygen API Reference described in Section 1.8 for information about other settings for this property.

Appendix B

SPRUIX7—November 2020

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 application used to generate the following timing benchmarks can be built and run by using the source files in the `SDK_INSTALL_DIR\kernel\tirtos\packages\ti\sysbios\benchmarks` directory.

Topic **Page**

B.1 Timing Benchmarks 149
B.2 Interrupt Latency 149
B.3 Hwi-Hardware Interrupt Benchmarks 149
B.4 Swi-Software Interrupt Benchmarks 150
B.5 Task Benchmarks 151
B.6 Semaphore Benchmarks 153

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 alongside 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 `SDK_INSTALL_DIR\kernel\tirtos\packages\ti\sysbios\benchmarks` directory).

The benchmark data was collected with the Build-Profile set to “release”.

B.2 Interrupt Latency

The Interrupt Latency benchmark is the maximum number of cycles during which SYS/BIOS 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 is measured within the context of a SYS/BIOS application containing SYS/BIOS API calls that internally disable interrupts. A timer is triggered to interrupt on each instruction boundary within the application, and the latency is calculated to be the time, in cycles, to vector to the first instruction of the interrupt dispatcher. Note that this is not the time to the first instruction of the user’s ISR.

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.

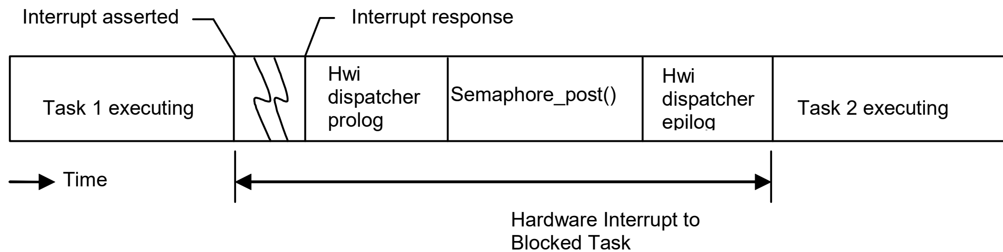


Figure B–1 Hardware Interrupt to Blocked Task

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.

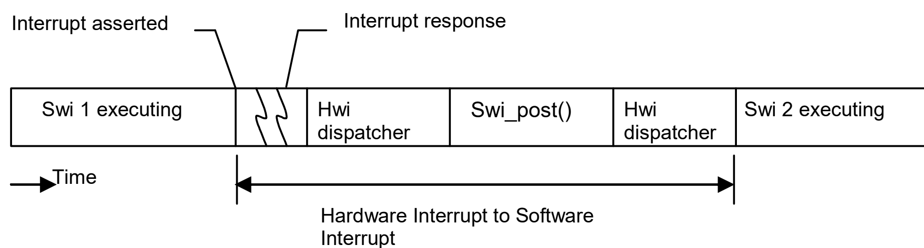


Figure B–2 Hardware Interrupt to Software Interrupt

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.

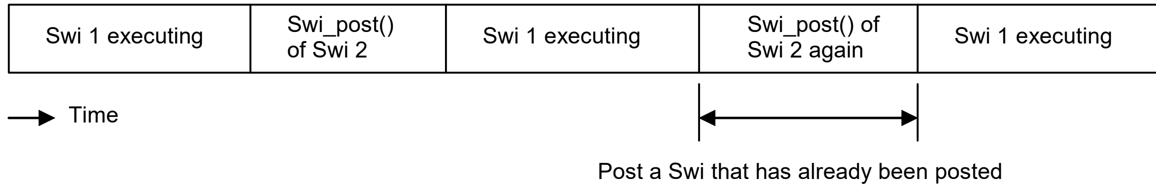


Figure B–3 Post of Software Interrupt Again

- **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.

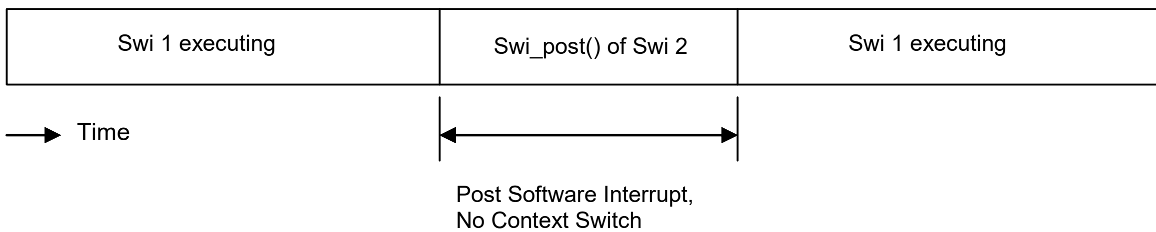


Figure B–4 Post Software Interrupt without Context Switch

- **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.

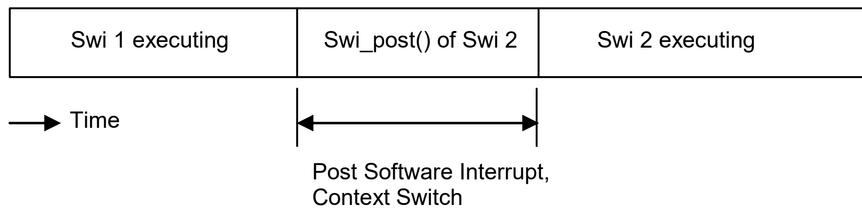


Figure B–5 Post Software Interrupt with Context Switch

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.

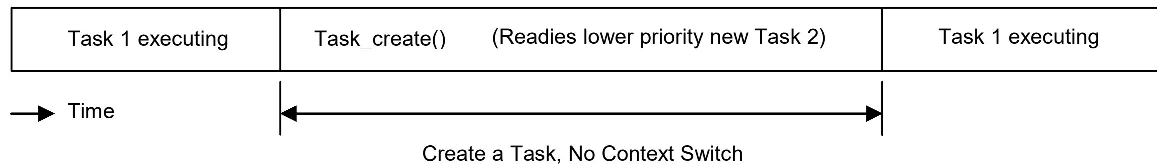


Figure B–6 Create a New Task without Context Switch

- **Create a task, context switch.** The executing task creates another task of higher priority, resulting in a context switch. See Figure B–7.

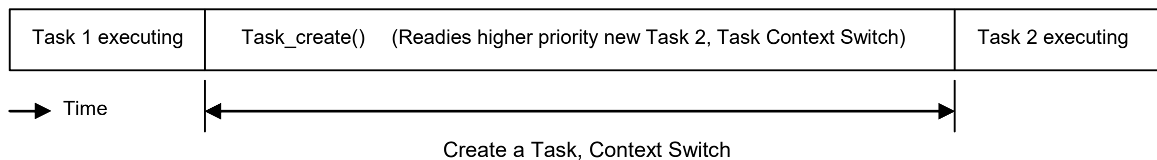


Figure B–7 Create a New Task with Context Switch

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 than the priority of current running task.

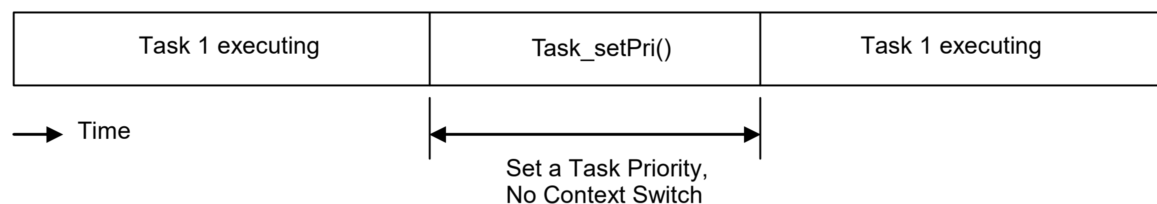


Figure B–8 Set a Task's Priority without a Context Switch

- **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.

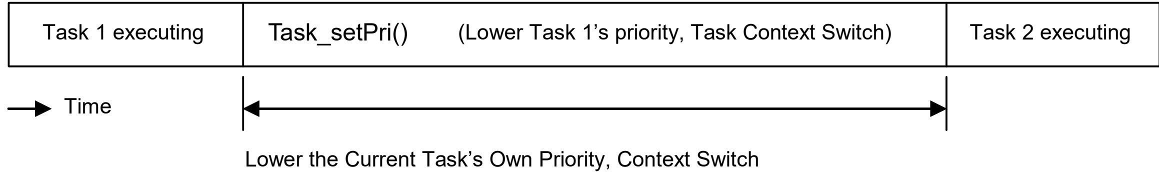


Figure B–9 Lower the Current Task's Priority, Context Switch

- **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 than the priority of current running task.

  The execution time measurement includes the context switch time as shown in Figure B–10.

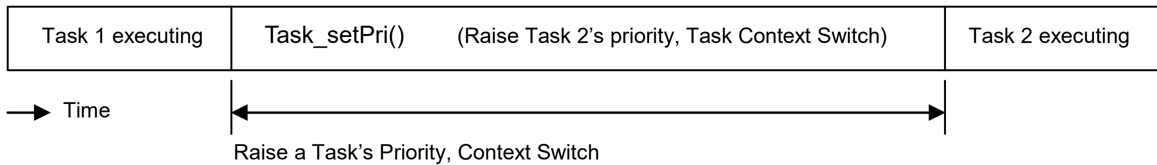


Figure B–10 Raise a Ready Task's Priority, Context Switch

- **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.

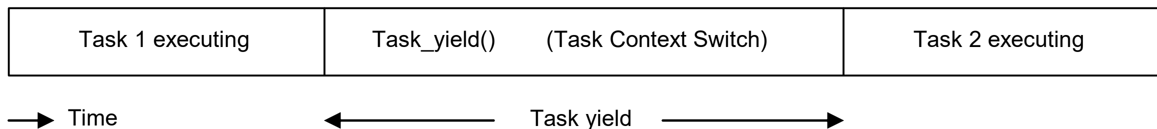


Figure B–11 Task Yield

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.

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.

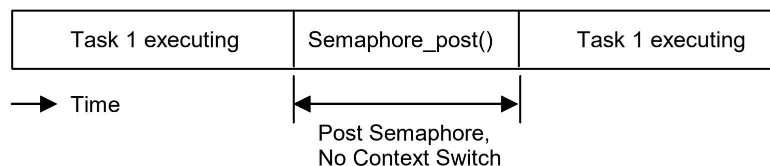


Figure B–13 Post Semaphore, No Context Switch

- **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.

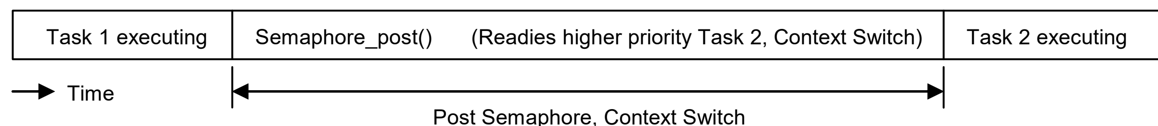


Figure B–14 Post Semaphore with Task Switch

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.

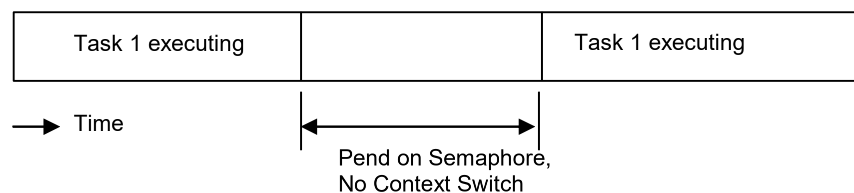


Figure B–15 Pend on Semaphore, No Context Switch

- **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.

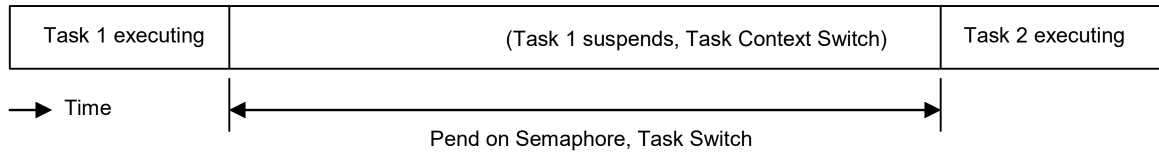


Figure B–16 Pend on Semaphore with Task Switch

TEXAS INSTRUMENTS

Appendix C

SPRUIX7—November 2020

Size Benchmarks

This appendix describes SYS/BIOS size benchmark statistics.

Topic **Page**

C.1 Overview 157

C.2 Constructed Application Sizes 158

C.3 Created Module Application Sizes. 160

C.4 POSIX Application Sizes. 162

C.1 Overview

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. All of the test applications, by default, include the commonly used Hwi and Swi modules. In turn, this pulls in all related code for that module while excluding most other non-dependent module code.

The applications are split into three categories:

- Those which construct SYS/BIOS objects with *Mod*_construct() API calls.
- Those which create SYS/BIOS objects with *Mod*_create() API calls.
- Those which create POSIX objects.

Applications lower on the table generally require the modules in the applications above them, (The Semaphore module, for example, requires the Task module.) 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, are provided in absolute numbers.

In addition to the test application size benchmarks, module-specific size benchmarks are provided. These benchmarks demonstrate how much memory is necessary to create a single instance of a specific SYS/BIOS object.

The actual size benchmark data are included in the SYS/BIOS 6 installation in the ti.sysbios.benchmarks package (that is, in the `SDK_INSTALL_DIR\kernel\tirtos\packages\ti\sysbios\benchmarks` directory). There is a separate HTML file for each target that can be reached through the `benchmarks.html` page. 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”.

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 code built against the TI compiler.

Table C–1 SYS/BIOS 6 Benchmark Applications

Constructed Applications	Created Applications	POSIX Applications
Task	Task	Pthread
Semaphore	Semaphore	Semaphore
Mutex	Mutex	Mutex
Clock	Clock	Timer

C.2 Constructed Application Sizes

Each application builds on top of the applications above it in Table C–1 and for each module there are generally two benchmarks: the application size and the amount of memory taken up by the modules' object. The following applications construct their objects and do not use any memory allocation.

Note: All of the following applications include the Hwi and Swi Modules within their respective configuration files.

The code snippets for each application apply to all targets, except where noted.

C.2.1 Constructed Task Application

The Task Application configuration enables Task scheduling while attempting to minimize code size by disabling the task stack check debug feature. The configuration also reduces the number of allowed priorities task to reduce RAM usage.

Configure the following settings in SysConfig:

- Enable the BIOS and Task modules.
- Set **Enable Task** to true in the BIOS module.
- Set **Check Stack Flag** to false in the Task module.
- Set **Number of Task Priorities** to 2 in the Task module.

C Code Addition

```
Task_Struct taskStruct;
Task_Handle taskHandle;
Task_Params taskParams;

Task_Params_init(&taskParams);
taskParams.stackSize = TASKSTACKSIZE;
taskParams.stack = &taskStack;

Task_construct(&taskStruct, twoArgsFxn, &taskParams, NULL);
taskHandle = Task_handle(&taskStruct);

Task_getPri(taskHandle);
Task_destruct(&taskStruct);
```

C.2.2 Constructed Semaphore Application

This application also includes the constructed Task application described in Section C.2.1. The Semaphore Application configuration disables support for Events in the Semaphore in order to reduce code size.

Configure the following settings in SysConfig:

- Enable the Semaphore module.
- Set **Supports Events** to false in the BIOS module.

C Code Addition

```
Semaphore_Struct semStruct;
Semaphore_Handle semHandle;

Semaphore_construct(&semStruct, 0, NULL);
semHandle = Semaphore_handle(&semStruct);

Semaphore_post(semHandle);
Semaphore_pend(semHandle, BIOS_WAIT_FOREVER);
Semaphore_destruct(&semStruct);
```

C.2.3 Constructed Mutex Application

This application includes the constructed Semaphore application described in Section C.2.2. No Mutex module specific configuration settings were changed in this application.

Configure the following settings in SysConfig:

- Set **Runtime Support Library Lock Type** to GateMutex in the BIOS module.

C Code Addition

```
GateMutex_Struct mutexStruct;
GateMutex_Handle mutexHandle;
intptr_t mutexKey;

GateMutex_construct(&mutexStruct, NULL);
mutexHandle = GateMutex_handle(&mutexStruct);

mutexKey = GateMutex_enter(mutexHandle);
GateMutex_leave(mutexHandle, mutexKey);
GateMutex_destruct(&mutexStruct);
```

C.2.4 Constructed Clock Application

This application includes the Mutex application described in Section C.2.3.

The Clock Application configuration enables the SYS/BIOS Clock services, which has the effect of registering a timer Hwi to generate the system tick.

Configure the following settings in SysConfig:

- Enable the BIOS and Clock modules.
- Set **Enable Clock** to true in the BIOS module.

C Code Addition

```
Clock_Struct clockStruct;
Clock_Handle clockHandle;

Clock_construct(&clockStruct, oneArgFxn, 10, NULL);
clockHandle = Clock_handle(&clockStruct);

Clock_start(clockHandle);
Clock_stop(clockHandle);
Clock_destruct(&clockStruct);
```

C.3 Created Module Application Sizes

The created applications demonstrate the size effects of dynamically creating objects (in the C code). The configuration files are identical to their constructed counterparts, as seen in Section C.2:

The .c files are also identical to their constructed counterparts apart from the *Mod*_construct() calls being replaced with *Mod*_create calls.

Note: All applications that use *Mod*_create() APIs internally pull in code from the Memory module to allocate space for the new object instances.

C.3.1 Created Task Application

The Created Task Application demonstrates the size impact of dynamically creating and deleting a Task instance.

C Code Addition

```
Task_Handle dynamicTask;
Task_Params taskParams;

Task_Params_init(&taskParams);
taskParams.stackSize = TASKSTACKSIZE;
taskParams.stack = &taskStack;

dynamicTask = Task_create(twoArgsFxn, &taskParams, NULL);

Task_getPri(dynamicTask);
Task_delete(&dynamicTask);
```

C.3.2 Created Semaphore Application

The Created Semaphore Application demonstrates the size impact of dynamically creating and deleting a Semaphore instance.

C Code Addition

```
Semaphore_Handle dynamicSem;

dynamicSem = Semaphore_create(0, NULL, NULL);

Semaphore_post(dynamicSem);
Semaphore_pend(dynamicSem, BIOS_WAIT_FOREVER);
Semaphore_delete(&dynamicSem);
```

C.3.3 Created Mutex Application

The Created Mutex Application demonstrates the size impact of dynamically creating and deleting a GateMutex instance.

C Code Addition

```
GateMutex_Handle dynamicMutex;

intptr_t dynamicMutexKey;

dynamicMutex = GateMutex_create(NULL, NULL);

dynamicMutexKey = GateMutex_enter(dynamicMutex);
GateMutex_leave(dynamicMutex, dynamicMutexKey);
GateMutex_delete(&dynamicMutex);
```

C.3.4 Created Clock Application

The Created Clock Application demonstrates the size impact of dynamically creating and deleting a Clock instance.

C Code Addition

```
Clock_Handle dynamicClock;

dynamicClock = Clock_create(oneArgFxn, 10, NULL, NULL);

Clock_start(dynamicClock);
Clock_stop(dynamicClock);
Clock_delete(&dynamicClock);
```

C.4 POSIX Application Sizes

The POSIX size benchmarks use a set of applications that dynamically create their respective objects.

C.4.1 POSIX Pthread Application

The POSIX Pthread Application demonstrates the size impact of dynamically initializing and creating a POSIX thread. Since Pthreads are built on top of the native SYS/BIOS Task module, their size can be compared to benchmarks for the Created Task application.

C Code Addition

```
pthread_t threadHandle = (pthread_t)NULL;
int threadArg = 1;
uint32_t retVal;

/* Set up stack buffer, pthread_create calls Task_create internally */
pthread_attr_t pattrs;

pthread_attr_init(&pattrs);
pthread_attr_setstack(&pattrs,(void *)taskStack, TASKSTACKSIZE);

pthread_create(&threadHandle, &pattrs, threadFxn,
   (void *)&threadArg);
pthread_join(threadHandle, (void **)&retVal);
```

C.4.2 POSIX Semaphore Application

The POSIX Semaphore Application demonstrates the size impact of dynamically creating an instance of a POSIX Semaphore. This application includes the PThread module. Its size can be compared to benchmarks for the Created Semaphore application.

C Code Addition

```
static sem_t sem1;

sem_init(&sem1, 0, 0);

sem_trywait(&sem1);
sem_destroy(&sem1);
```

C.4.3 POSIX Mutex Application

The POSIX Mutex Application demonstrates the size impact of dynamically creating an instance of a POSIX Mutex. This application includes the PThread module. Its size can be compared to benchmarks for the Created Mutex application.

C Code Addition

```
int getVal;
pthread_mutexattr_t  attrs;

/* Create minimal mutex(No priority) */
pthread_mutexattr_setprotocol(&attrs, PTHREAD_PRIO_NONE);
pthread_mutex_init(&mutex, &attrs);

pthread_mutexattr_getprotocol(&attrs, &getVal);

pthread_mutexattr_destroy(&attrs);
pthread_mutex_destroy(&mutex);
```

C.4.4 POSIX Timer Application

The POSIX Timer Application demonstrates the size impact of dynamically creating an instance of a POSIX Timer. This application includes the PThread module. Its size can be compared to benchmarks for the Created Clock application.

C Code Addition

```
sigevent    sev;
timer_t     timer;

timer_create(CLOCK_MONOTONIC, &sev, &timer);
```

Index

A

alloc() function
  Memory module 117
alloc() function, Memory module 118
andn() function, Swi module 48, 49, 51
application
  configuring 22
application stack size 48
atexit() handlers 136

B

background thread (see Idle Loop)
Begin hook function
  for hardware interrupts 41, 42
  for software interrupts 53, 54
binary semaphores 83
BIOS module 109, 116
BIOS_exit() function 109
BIOS_getCpuFreq() function 109
BIOS_getThreadType() function 110
BIOS_setCpuFreq() function 109
BIOS_start() function 30, 31, 109
BIOS_Start() functions 31
blocked state 36, 61, 63
books (resources) 19
buffer overflows 123

C

C++ 17
CCS
  building 28
  creating a project 21
  other documentation 19
Chip Support Library (CSL) 40
cinit() function 30
class constructor 18
class destructor 18
class methods 18
Clock module 12, 102
Clock_create() function 73, 103, 104
Clock_getTicks() function 74, 103, 105
Clock_setFunc() function 104
Clock_setPeriod() function 104
Clock_setTimeout() function 104
Clock_start() function 104
Clock_stop() function 104
Clock_tick() function 102
Clock_tickReconfig() function 103, 105
Clock_tickStart() function 103, 105
Clock_tickStop() function 103, 105
clocks 32, 102
  creating dynamically 103, 104
  disabling 136
  starting 103
  stopping 104
  ticks for, manipulating 103, 105
  ticks for, tracking 105
  when to use 33
code size, reducing 136
configuration size
  basic size benchmark configuration 137
counting semaphores 83
Create hook function
  for hardware interrupts 41, 42
  for software interrupts 53, 54
  for tasks 64, 65
create() function
  Clock module 103
  Hwi module 40
  Mailbox module 97
  Semaphore module 84
  Swi module 47
  Task module 61, 151
critical regions, protecting (see gates)

D

data size, reducing 135
dec() function, Swi module 48, 49, 51
Delete hook function
  for hardware interrupts 41, 42
  for software interrupts 53, 54
  for tasks 65
delete() function
  Mailbox module 97
  Semaphore module 84
  Swi module 53
  Task module 61, 152
dequeue() function, Queue module 99
deterministic performance 119
device-specific modules 126
disable() function
  Hwi module 36, 128, 149
  Swi module 36, 53, 150
  Task module 36, 151
disableInterrupt() function, Hwi module 36

dispatcher 130
  optimization 135
documents (resources) 19
download 19
dynamic configuration 11

E

enable() function
  Hwi module 40, 128, 149
  Swi module 150
  Task module 151
End hook function
  for hardware interrupts 41, 42
  for software interrupts 53, 54
enqueue() function, Queue module 99
enter() function, Gate module 94
Error module 113
Error_check() function 113
Error_init() function 113
Error_print() function 113
Error_raise() function 113
errors
  finding in configuration 27
Event module 12, 88
  used with Mailbox 98
Event_create() function 89, 93
Event_pend() function 88, 94, 98
Event_post() function 88, 93
events 88
  associating with mailboxes 98
  creating dynamically 89
  posting 88, 89
  posting implicitly 91
  waiting on 88, 89
execution states of tasks 61
  Task_Mode_BLOCKED 61, 63
  Task_Mode_INACTIVE 62
  Task_Mode_READY 61, 63
  Task_Mode_RUNNING 61, 62
  Task_Mode_TERMINATED 61, 63
execution states of threads 34
Exit hook function, for tasks 65, 66
exit() function, Task module 63

F

forum, E2E community 19
fragmentation, memory 119
free() function 117
  C++ 17
  Memory module 117
function names 17
functions
  (see also hook functions)

G

Gate_enter() function 94
Gate_leave() function 94
GateHwi module 95
GateHwi_create() function 95
GateMutex module 96
GateMutexPri module 96
gates 94
  preemption-based implementations of 95
  priority inheritance with 96
  priority inversion, resolving 96
  semaphore-based implementations of 96
GateSwi module 95
GateTask module 95
get() function, Queue module 100
getHookContext() function
  Hwi module 39
getHookContext() function, Swi module 54
getTicks() function, Clock module 103
getTrigger() function, Swi module 49

H

hal package 126
hardware interrupts 32, 40
  compared to other types of threads 34
  creating 40
  disabling 128
  enabled at startup 31
  enabling 128
  hook functions for 41, 42
  interrupt dispatcher for 130, 135
  priority of 35
  registers saved and restored by 131
  timing benchmarks for 149
  when to use 33
head() function, Queue module 100
Heap implementation
  used by Memory module 118
HeapBuf module 12, 118, 120, 134
HeapMem module 12, 118, 119, 134
  system heap 116
HeapMin module 119
HeapMultiBuf module 12, 118, 121, 134
heaps 118
  HeapBuf implementation 120
  HeapMem implementation 119
  HeapMultiBuf implementation 121
  implementations of 118
  optimizing 134
  system 116
HeapTrack module 118, 123
hook context pointer 39
hook functions 34, 38
  for hardware interrupts 41, 42
  for software interrupts 53
  for tasks 64, 66
hook sets 38
Hwi dispatcher 130
Hwi module 12, 40, 41
Hwi threads (see hardware interrupts)
Hwi_create() function 40
Hwi_delete() function
  hook function 41
Hwi_disable() function 36, 105

Hwi_disableInterrupt() function 36
Hwi_enable() function 40
Hwi_getHookContext() function 39, 41, 44
Hwi_plug() function 40
Hwi_restore() function 105
Hwi_setHookContext() function 41, 43

I

Idle Loop 32, 77
  compared to other types of threads 34
  priority of 36
  when to use 33
Idle module 12
  manager 77
inactive state 62
inc() function, Swi module 48, 49, 50
insert() function, Queue module 100
interrupt keyword 131
Interrupt Latency benchmark 149
INTERRUPT pragma 131
Interrupt Service Routines (ISRs) (see hardware interrupts)
interrupts
  zero latency 140
interrupts (see hardware interrupts, software interrupts)
inter-task synchronization (see semaphores)
ISR stack (see system stack)
ISRs (Interrupt Service Routines) (see hardware interrupts)

L

latency 140
latency benchmark 149
leave() function, Gate module 94
Load module 12, 133
  configuration 133
Load_calculateLoad() function 134
Load_getCPULoad() function 134
Load_getGlobalHwiLoad() function 134
Load_getGlobalSwiLoad() function 134
Load_getTaskLoad() function 134

M

Mailbox module 12, 97
Mailbox_create() function 93, 97
Mailbox_delete() function 97
Mailbox_pend() function 94, 97
Mailbox_post() function 93, 97
mailboxes 97
  associating events with 98
  creating 97
  deleting 97
  posting buffers to 98
  posting implicitly 91
  reading buffers from 98
main() function 30
  calling BIOS_start() 31
  functions called before 41, 53
malloc heap, reducing size of 135
malloc() function 117
  C++ 17
memory
  allocation of (see heaps)
  fragmentation 119
  leaks, detecting 123
  requirements for, minimizing 135
Memory module 117, 118
Memory_alloc() function 86, 117
Memory_free() function 117
modules
  list of 12
multithreading (see threads)
mutex 96
mutual exclusion (see semaphores)

N

name mangling 17, 18
name overloading 18
naming conventions 17
next() function, Queue module 100

O

optimization 134
or() function, Swi module 48, 49, 52

P

packages
  SYS/BIOS list 12
pend() function
  Event module 88, 89, 98
  Mailbox module 98
  Semaphore module 84, 154
performance 134
plug() function, Hwi module 40
POSIX threads 32
post() function
  Event module 88, 89
  Mailbox module 98
  Semaphore module 84, 153
  Swi module 48, 49, 150
posting Swis 46
preemption 36
preemption-based gate implementations 95
prev() function, Queue module 100
printf() function, removing 136
priority inheritance 96
priority inheritance, with gates 96
priority inversion 96
priority inversion problem, with gates 96
priority levels of threads 34
project
  building 28
put() function, Queue module 100

Q

Queue module 99
Queue_create() function 80
Queue_dequeue() function 99
Queue_empty() function 99
Queue_enqueue() function 99
Queue_get() function 87
Queue_head() function 100
Queue_insert() function 100
Queue_next() function 78, 100
Queue_prev() function 100
Queue_put() function 80, 86, 100
Queue_remove() function 100
queues 99
  atomic operation of 100
  FIFO operation on 99
  inserting elements in 100
  iterating over 100
  removing elements from 100

R

Ready hook function
  for software interrupts 53, 54
  for tasks 64, 66
ready state 61, 63
Register hook function
  for hardware interrupts 41
  for software interrupts 53, 54
  for tasks 64, 65
remove() function, Queue module 100
reset function 30
resources 19
restore() function
  Hwi module 128
  Swi module 53
ROV tool 133
RTS thread protection, disabling 137
running state 61, 62

S

semaphore application size 158, 160
Semaphore module 12, 83
Semaphore_create() function 80, 84
Semaphore_delete() function 84
Semaphore_pend() function 79, 84, 87
Semaphore_post() function 74, 79, 84, 87
semaphore-based gate implementations 96
semaphores 83
  binary semaphores 83
  configuring type of 83
  counting semaphores 83
  creating 84
  deleting 84
  example of 84
  posting implicitly 91
  signaling 84
  timing benchmarks for 153
  waiting on 84
setHookContext() function, Swi module 54
setPri() function, Task module 152
size benchmarks 157
  static module application sizes 158
software interrupts 32, 46
  compared to other types of threads 34
  creating dynamically 47
  deleting 53
  disabling 136
  enabled at startup 31
  enabling and disabling 53
  hook functions for 53, 55
  posting, functions for 46, 48
  posting multiple times 49
  posting with Swi_andn() function 51
  posting with Swi_dec() function 51
  posting with Swi_inc() function 50
  posting with Swi_or() function 52
  preemption of 49, 53
  priorities for 35, 48
  priority levels, number of 48
  timing benchmarks for 150
  trigger variable for 49
  vs. hardware interrupts 52
  when to use 33, 52
stacks used by threads 34, 115
  optimization 64, 135
  tasks 63
standardization 11
start() function
  Clock module 103
Startup functions 30, 31
Startup module 112
startup sequence for SYS/BIOS 30
stat() function, Task module 64
static configuration 11
static module application sizes 158
stop() function
  Clock module 104
Swi module 12, 46
Swi threads (see software interrupts)
Swi_andn() function 46, 48, 49, 51
Swi_create() function 47
  hook function 53
Swi_dec() function 46, 48, 49, 51
Swi_delete() function 53
Swi_disable() function 36, 53
Swi_getHookContext() function 54, 55
Swi_getTrigger() function 49
Swi_inc() function 46, 48, 49, 50
Swi_or() function 46, 48, 49, 52
Swi_post() function 38, 46, 48, 49, 78
Swi_restore() function 53
Swi_setHookContext() function 54, 55
Switch hook function, for tasks 65, 66
synchronization
  see also events, semaphores 88
SYS/BIOS 11
  benefits of 11
  other documentation 19
  packages in 12
  startup sequence for 30
SysCallback module 112

SysConfig 22
  saving 25
SysMin module 111
system heap 116
System module 110
system stack 36
  configuring size 115
  reducing size of 136
  threads using 34
System_abort() function 59, 110
System_atexit() function 111
System_exit() function 110
System_flush() function 110
System_printf() function 43, 110

T

target-specific modules 126
task application size 158, 160
Task module 12, 60
task stack
  configuring size 115
  determining size used by 63
  overflow checking for 64
  threads using 34
task synchronization (see semaphores)
Task_create() function 61, 73
Task_delete() function 61
Task_disable() function 36
Task_exit() function 63, 66
Task_getHookContext() function 65
Task_idle task 62
Task_Mode_BLOCKED state 61, 63
Task_Mode_INACTIVE state 62
Task_Mode_READY state 61, 63
Task_Mode_RUNNING state 61, 62
Task_Mode_TERMINATED state 61, 63
Task_setHookContext() function 65
Task_setPri() function 62, 63
Task_stat() function 62, 64
Task_yield() function 63, 69, 74
tasks 32, 60
  begun at startup 31
  blocked 36, 63
  compared to other types of threads 34
  deleting 61
  disabling 136
  execution states of 61
  hook functions for 64, 66
  idle 62
  priority level 35, 62
  scheduling 62
  terminating 63
  timing benchmarks for 151
  when to use 33
  yielding 71
terminated state 61, 63
text strings, not storing on target 136
thread scheduler, disabling 34
threads 31
  creating dynamically 35
  execution states of 34
  hook functions in 34, 38
  pending, ability to 34
  posting mechanism of 34
  preemption of 36
  priorities of 35
  priorities of, changing dynamically 35
  priority levels, number of 34
  sharing data with 34
  stacks used by 34
  synchronizing with 34
  types of 32, 33, 34
  yielding of 34, 36
ti.sysbios.family.* packages 12
ti.sysbios.gates package 12
ti.sysbios.hal package 12
ti.sysbios.heaps package 12
ti.sysbios.knl package 12
ti.sysbios.utils package 12
tick() function, Clock module 102
tickReconfig() function, Clock module 103
tickSource parameter 102
tickStart() function, Clock module 103
tickStop() function, Clock module 103
Timer module 12, 105
Timer_create() function 80
timers 102, 105
  clocks using 102
  when to use 33
time-slice scheduling 71
Timestamp module 106
Timestamp_get32() function 43, 107
timestamps 102, 106
timing benchmarks 149
  hardware interrupt benchmarks 149
  Interrupt Latency benchmark 149
  semaphore benchmarks 153
  software interrupt benchmarks 150
  task benchmarks 151
timing services (see clocks; timers; timestamps)
trigger variable for software interrupts 49

V

validation of configuration 27

W

waiting thread 36
wrapper function 18

Y

yield() function, Task module 153
yielding 36, 63

Z

zero latency interrupts 140

IMPORTANT NOTICE AND DISCLAIMER

TTI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (https://www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products

Mailing Address: Texas Instruments, Post Office Box 655303 Dallas, Texas 75265
Copyright © 2020, Texas Instruments Incorporated