TI-RTOS 2.20

User's Guide

Literature Number: SPRUHD4M
June 2016

Contents

Preface 6

1 About TI-RTOS 7
- 1.1 What is TI-RTOS? 7
- 1.2 What are the TI-RTOS Components? 8
- 1.3 SYS/BIOS: The TI-RTOS Kernel 9
- 1.4 UIA: TI-RTOS Instrumentation 9
- 1.5 NDK: TI-RTOS Networking 10
- 1.6 TI-RTOS Network Services 10
- 1.7 FatFS Module: TI-RTOS File System 10
- 1.8 TI-RTOS Drivers and Board Initialization 11
  - 1.8.1 Drivers 11
  - 1.8.2 MWare 11
  - 1.8.3 MSPWare 12
  - 1.8.4 TivaWare 12
  - 1.8.5 C26xxWare and the CC3200 Driverlib 12
- 1.9 XDCtools 13
- 1.10 wolfSSL Embedded SSL Library 13

2 Instrumentation with TI-RTOS 14
- 2.1 Overview 14
- 2.2 Adding Logging to a Project 15
- 2.3 Modifying an Example to Upload Logging Data at Runtime 17
  - 2.3.1 Project Changes 17
  - 2.3.2 Code Changes 19
  - 2.3.3 Configuration Changes 20
- 2.4 Using Log Events 22
  - 2.4.1 Adding Log Events to your Code 22
  - 2.4.2 Using Instrumented or Non-Instrumented Libraries 22
- 2.5 Viewing the Logs 23
  - 2.5.1 Using RTOS Analyzer and System Analyzer 23
  - 2.5.2 Viewing Log Records in ROV 24

3 Debugging TI-RTOS Applications 25
- 3.1 Using CCS Debugging Tools 25
  - 3.1.1 Stepping Through TI-RTOS Code 27
- 3.2 Using IAR Debugging Tools 27
- 3.3 Generating printf Output 28
  - 3.3.1 Output with printf() 28
  - 3.3.2 Output with System_printf() 28
- 3.4 Controlling Software Versions for Use with TI-RTOS 31
- 3.5 Understanding the Build Flow 32

4 Board-Specific Files 33
- 4.1 Overview 33

4.2 Board-Specific Code Files . . . 34
4.3 Linker Command Files . . . 35
4.4 Target Configuration Files . . . 35

5 TI-RTOS Drivers . . . 36
5.1 Overview . . . 37
5.2 Driver Framework . . . 38
5.2.1 Static Configuration . . . 38
5.2.2 Driver Object Declarations . . . 39
5.2.3 Dynamic Configuration and Common APIs . . . 42
5.2.4 TI-RTOS Driver Implementations for Concerto Devices . . . 42
5.2.5 TI-RTOS Driver Implementations for TivaC Devices . . . 43
5.2.6 TI-RTOS Driver Implementations for CC26xx Devices . . . 43
5.2.7 TI-RTOS Driver Implementations for CC3200 Devices . . . 44
5.2.8 TI-RTOS Driver Hwis for MSP43x Devices . . . 45
5.3 Camera Driver . . . 48
5.3.1 Static Configuration . . . 48
5.3.2 Runtime Configuration . . . 48
5.3.3 Camera Modes . . . 48
5.3.4 APIs . . . 49
5.3.5 Examples . . . 49
5.4 EMAC Driver . . . 50
5.4.1 Static Configuration . . . 50
5.4.2 Runtime Configuration . . . 50
5.4.3 APIs . . . 50
5.4.4 Usage . . . 50
5.4.5 Instrumentation . . . 51
5.5 GPIO Driver . . . 52
5.5.1 Static Configuration . . . 52
5.5.2 Runtime Configuration . . . 54
5.5.3 APIs . . . 54
5.5.4 Usage . . . 54
5.5.5 Instrumentation . . . 55
5.5.6 Examples . . . 55
5.6 I2C Driver . . . 56
5.6.1 Static Configuration . . . 56
5.6.2 Runtime Configuration . . . 56
5.6.3 APIs . . . 56
5.6.4 Usage . . . 57
5.6.5 I2C Modes . . . 58
5.6.6 I2C Transactions . . . 59
5.6.7 Instrumentation . . . 62
5.7 I2S Driver . . . 63
5.7.1 Static Configuration . . . 63
5.7.2 Runtime Configuration . . . 63
5.7.3 I2S Modes . . . 63
5.7.4 APIs . . . 64
5.8 LCD Driver . . . 66
5.8.1 Static Configuration . . . 66
5.8.2 Runtime Configuration . . . 66

5.8.3 APIs . . . 67
5.8.4 Usage . . . 67
5.8.5 Instrumentation . . . 67
5.9 PWM Driver . . . 68
5.9.1 Static Configuration . . . 68
5.9.2 Runtime Configuration . . . 68
5.9.3 APIs . . . 68
5.9.4 Usage . . . 69
5.9.5 PWM Modes . . . 69
5.9.6 Instrumentation . . . 70
5.10 SDSPI Driver . . . 71
5.10.1 Static Configuration . . . 71
5.10.2 Runtime Configuration . . . 71
5.10.3 APIs . . . 71
5.10.4 Usage . . . 72
5.10.5 Instrumentation . . . 72
5.11 SPI Driver . . . 73
5.11.1 Static Configuration . . . 73
5.11.2 Runtime Configuration . . . 73
5.11.3 APIs . . . 73
5.11.4 Usage . . . 74
5.11.5 Callback and Blocking Modes . . . 75
5.11.6 SPI Transactions . . . 76
5.11.7 Master/Slave Modes . . . 77
5.11.8 Instrumentation . . . 78
5.12 UART Driver . . . 79
5.12.1 Static Configuration . . . 79
5.12.2 Runtime Configuration . . . 79
5.12.3 APIs . . . 79
5.12.4 Usage . . . 80
5.12.5 UART DMA Driver for TivaC Devices . . . 81
5.12.6 UART DMA Driver for SimpleLink CC32xx Devices . . . 81
5.12.7 Instrumentation . . . 82
5.13 USBMSCHFatFs Driver . . . 83
5.13.1 Static Configuration . . . 83
5.13.2 Runtime Configuration . . . 83
5.13.3 APIs . . . 83
5.13.4 Usage . . . 84
5.13.5 Instrumentation . . . 85
5.14 USB Reference Modules . . . 86
5.14.1 USB Reference Modules in TI-RTOS . . . 87
5.14.2 USB Reference Module Design Guidelines . . . 88
5.15 USB Device and Host Modules . . . 89
5.16 Watchdog Driver . . . 91
5.16.1 Static Configuration . . . 91
5.16.2 Runtime Configuration . . . 91
5.16.3 APIs . . . 91
5.16.4 Usage . . . 92
5.16.5 Instrumentation . . . 92

5.17 WiFi Driver . . . 93
5.17.1 Static Configuration . . . 93
5.17.2 Runtime Configuration . . . 94
5.17.3 APIs . . . 94
5.17.4 Usage . . . 95
5.17.5 Instrumentation . . . 95

6 TI-RTOS Utilities . . . 96
6.1 Overview . . . 96
6.2 UARTMon Module . . . 96
6.2.1 UARTMon with CCS Tools . . . 98
6.2.2 GUI Composer . . . 102
6.3 UART Example Implementation . . . 102

7 Using the FatFs File System Drivers . . . 103
7.1 Overview . . . 103
7.2 FatFs and TI-RTOS . . . 104
7.3 Using FatFs . . . 105
7.3.1 Static FatFS Module Configuration . . . 105
7.3.2 Defining Drive Numbers . . . 106
7.3.3 Preparing FatFs Drivers . . . 106
7.3.4 Opening Files Using FatFs APIs . . . 107
7.3.5 Opening Files Using C I/O APIs . . . 107
7.4 Cautionary Notes . . . 107

8 Rebuilding TI-RTOS . . . 108
8.1 Rebuilding TI-RTOS . . . 109
8.1.1 Building TI-RTOS for CCS . . . 109
8.1.2 Building TI-RTOS for IAR . . . 110
8.1.3 Building TI-RTOS for GCC . . . 110
8.1.4 Rebuilding the TI-RTOS Drivers with the Debug Profile . . . 111
8.2 Rebuilding MSPWare's driverlib for TI-RTOS and Its Drivers . . . 111
8.3 Rebuilding Individual Components . . . 112

9 Memory Usage with TI-RTOS . . . 113
9.1 Memory Footprint Reduction . . . 113
9.2 Networking Stack Memory Usage . . . 125

A Revision History . . . 126

Index . . . 130

Read This First

About This Manual

This document describes TI-RTOS and contains information related to all supported device families. The version number as of the publication of this manual is v2.20.

Notational Conventions

This document uses the following conventions:

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

  Here is a sample program listing:

```
#include <xdc/runtime/System.h>
int main(void){
    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.

Trademarks

Registered trademarks of Texas Instruments include Stellaris, and StellarisWare.

Trademarks of Texas Instruments include: the Texas Instruments logo, Texas Instruments, TI, TI.COM, BoosterPack, C2000, C5000, C6000, Code Composer, Code Composer Studio, Concerto, controlSUITE, DSP/BIOS, E2E, MSP430, MSP430Ware, MSP432, OMAP, SimpleLink, SPOX, Sitara, TI-RTOS, Tiva, TivaWare, TMS320, TMS320C5000, TMS320C6000, and TMS320C2000.

ARM is a registered trademark, and Cortex is a trademark of ARM Limited.

Windows is a registered trademark of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

IAR Systems and IAR Embedded Workbench are registered trademarks of IAR Systems AB:

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

June 9, 2016

About TI-RTOS

This chapter provides an overview of TI-RTOS and its components.

Topic **Page**

1.1	What is TI-RTOS?	7
1.2	What are the TI-RTOS Components?	8
1.3	SYS/BIOS: The TI-RTOS Kernel	9
1.4	UIA: TI-RTOS Instrumentation	9
1.5	NDK: TI-RTOS Networking	10
1.6	TI-RTOS Network Services	10
1.7	FatFS Module: TI-RTOS File System	10
1.8	TI-RTOS Drivers and Board Initialization	11
1.9	XDCtools	13
1.10	wolfSSL Embedded SSL Library	13

1.1 What is TI-RTOS?

TI-RTOS is a scalable, one-stop embedded tools ecosystem for TI devices. It scales from a real-time multitasking kernel (SYS/BIOS) to a complete RTOS solution including additional middleware components and device drivers. By providing essential system software components that are pre-tested and preintegrated, TI-RTOS enables you to focus on differentiating your application.

If you use Code Composer Studio (CCS), you can install TI-RTOS in the **CCS App Center**. There are several different versions of TI-RTOS to support different device families, including TI-RTOS for MSP43x, TI-RTOS for TivaC, and TI-RTOS for C2000. You can also use TI-RTOS outside of CCS, including with IAR Embedded Workbench and GNU compilers.

For information about installing TI-RTOS and creating and configuring examples that use TI-RTOS, see the TI-RTOS Getting Started Guide for your device family:

- *TI-RTOS for C2000 Getting Started Guide* -- SPRUHU3
- *TI-RTOS for MSP43x Getting Started Guide* -- SPRUHU4
- *TI-RTOS for TivaC Getting Started Guide* -- SPRUHU5
- *TI-RTOS for CC13xx/CC26xx SimpleLink™ Wireless MCUs Getting Started Guide* -- SPRUHU7
- *TI-RTOS for CC32xx SimpleLink™ Wireless MCUs Getting Started Guide* -- SPRUHU8

1.2 What are the TI-RTOS Components?

TI-RTOS contains its own source files, pre-compiled libraries (both instrumented and non-instrumented), and examples. Additionally, TI-RTOS contains a number of components within its "`products`" subdirectory. The components of TI-RTOS are as follows. Some components are not available for all device families.

Table 1–1. TI-RTOS Components

TI-RTOS Component	Name	PDF Documentation Location
TI-RTOS	TI-RTOS examples	Chapter 3 of the *TI-RTOS Getting Started Guide* for your device
TI-RTOS Kernel	SYS/BIOS	*SYS/BIOS (TI-RTOS Kernel) User's Guide* -- SPRUEX3
TI-RTOS Instrumentation	UIA	*System Analyzer User's Guide* -- SPRUH43
TI-RTOS Networking	NDK	*TI Network Developer's Kit (NDK) Guide* -- SPRU523 *TI Network Developer's Kit (NDK) API Reference* -- SPRU524
TI-RTOS Network Services	Network Services	Section 1.6 of this User's Guide
TI-RTOS File System	FatFS	Chapter 7 of this User's Guide
TI-RTOS USB	USB stack	Section 5.14 and Section 5.15 of this User's Guide
TI-RTOS Drivers and Board Initialization	Drivers and TivaWare, MSPWare, Mware, CC26xxWare, or the CC3200 SDK's driverlib	Section 1.8 and Chapter 5 of this User's Guide

- **TI-RTOS Kernel — SYS/BIOS.** SYS/BIOS is a scalable real-time kernel. It is designed to be used by applications that require real-time scheduling and synchronization or real-time instrumentation. It provides preemptive multi-threading, hardware abstraction, real-time analysis, and configuration tools. SYS/BIOS is designed to minimize memory and CPU requirements on the target.
- **TI-RTOS Instrumentation — UIA.** The Unified Instrumentation Architecture (UIA) provides target content that aids in the creation and gathering of instrumentation data (for example, Log data).
- **TI-RTOS Networking — NDK.** The Network Developer's Kit (NDK) is a platform for development and demonstration of network enabled applications on TI embedded processors.
- **TI-RTOS Network Services.** The Network Services component provides application layer network protocols, such as an HTTP Client and an SNTP Client.
- **MSPWare, MWare, TivaWare, CC26xxWare, and the CC3200 SDK's driverlib.** These provide software designed to simplify and speed development of applications on the corresponding device family. These components are rebuilt to include only the portions required by TI-RTOS
- **XDCtools.** This core component provides the underlying tooling for configuring and building TI-RTOS and its components. XDCtools is installed as part of CCS v6.x. If you install TI-RTOS outside CCS, a compatible version of XDCtools is installed automatically.

1.3 SYS/BIOS: The TI-RTOS Kernel

SYS/BIOS is an advanced real-time operating system from Texas Instruments for use in a wide range of DSPs, microprocessors, and microcontrollers. It is designed for use in embedded applications that need real-time scheduling, synchronization, and instrumentation. SYS/BIOS is designed to minimize memory and CPU requirements on the target. SYS/BIOS provides a wide range of services, such as:

- Preemptive, deterministic multi-threading
- Hardware abstraction
- Memory management
- Configuration tools
- Real-time analysis

For more information about SYS/BIOS, see the following:

SYS/BIOS User's Guide (SPRUEX3)

SYS/BIOS API and configuration reference. In the top-level TI-RTOS installation directory, open the Release Notes and follow the links to **Documentation** and then to **Documentation Overview**. In the Documentation Overview page, choose the **TI-RTOS Kernel Runtime APIs and Configuration (cdoc)** item.

SYS/BIOS on Texas Instruments Wiki

TI-RTOS forum on TI's E2E Community

1.4 UIA: TI-RTOS Instrumentation

The Unified Instrumentation Architecture (UIA) provides target content that aids in the creation and gathering of instrumentation data (for example, Log data).

The System Analyzer tool suite, which is part of CCS, provides a consistent and portable way to instrument software. It includes the views that can be opened from the **Tools > RTOS Analyzer** and T**ools > System Analyzer** menus in CCS. It enables software to be re-used with a variety of silicon devices, software applications, and product contexts. It works together with UIA to provide visibility into the real-time performance and behavior of software running on TI's embedded single-core and multicore devices.

For more information about UIA and System Analyzer, see the following:

System Analyzer User's Guide (SPRUH43)

UIA API and configuration reference. In the top-level TI-RTOS installation directory, open the Release Notes and follow the links to **Documentation** and then to **Documentation Overview**. In the Documentation Overview page, choose the **TI-RTOS Instrumentation Runtime APIs and Configuration (cdoc)** item.

System Analyzer on Texas Instruments Wiki

1.5 NDK: TI-RTOS Networking

The Network Developer's Kit (NDK) is a platform for development and demonstration of network enabled applications on TI embedded processors, currently limited to the TMS320C6000 family and ARM processors. The NDK stack serves as a rapid prototyping platform for the development of network and packet processing applications. It can be used to add network connectivity to existing applications for communications, configuration, and control. Using the components provided in the NDK, developers can quickly move from development concepts to working implementations attached to the network.

The NDK is a networking stack that operates on top of SYS/BIOS.

For more information about NDK, see the following:

NDK User's Guide (SPRU523)

NDK Programmer's Reference Guide (SPRU524)

NDK Support Package Ethernet Driver Design Guide (SPRUFP2)

NDK configuration reference. In the top-level TI-RTOS installation directory, open the Release Notes and follow the links to **Documentation** and then to **Documentation Overview**. In the Documentation Overview page, open the **TI-RTOS Networking Configuration (cdoc)**.

NDK API reference.
Run <*tirtos_install*>/products/ndk_#_##_##_##/docs/doxygen/html/index.html.

NDK on Texas Instruments Wiki

TI-RTOS forum on TI's E2E Community

1.6 TI-RTOS Network Services

The Network Services component provides application layer network protocols, such as an HTTP Client and an SNTP Client. These are designed for TI embedded processors. Documentation is provided in `<tirtos_install>/products/ns_<version>/docs/NS_Users_Guide.html`.

1.7 FatFS Module: TI-RTOS File System

FatFS is an open-source FAT file system module intended for use in embedded systems. The API used by your applications is generic to all FatFS implementations, and is described and documented at http://elm-chan.org/fsw/ff/00index_e.html. In order to use FatFS in TI-RTOS applications, use the ti.mw.fatfs.FatFS module.

For more information about FatFS, see the following:

Chapter 7, "Using the FatFs File System Drivers"

1.8 TI-RTOS Drivers and Board Initialization

TI-RTOS provides drivers for device families for which a *Ware package is supported by TI-RTOS. This *Ware packages include TivaWare, MSPWare, MWare, CC26xxWare, and the CC3200SDK Driverlib. The *Ware libraries distributed with TI-RTOS have been reduced in size to include only the necessary portions of the libraries.

1.8.1 Drivers

TI-RTOS includes drivers for a number of peripherals. See Chapter 5 for a list of the specific drivers and details about each one.

The drivers are in the `<tirtos_install>/products/tidrivers_<version>/packages/ti/drivers` directory. TI-RTOS examples are provided to show how to use these drivers.

Note that all of these drivers are built on top of MWare, MSPWare, and TivaWare. These drivers provide the following advantages over those provided by MWare, MSPWare, and TivaWare:

- The TI-RTOS drivers are thread-safe for use with SYS/BIOS threads.
- The TI-RTOS drivers are provided in both instrumented and non-instrumented versions. The instrumented versions support logging and asserts.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details about the driver APIs.

1.8.2 MWare

MWare is the M3 portion of controlSUITE, a software package that provides support for F28M3x (Concerto) devices. It includes low-level drivers and examples.

- The version of MWare provided with TI-RTOS differs from the version in controlSUITE in that it has been rebuilt. See the TI-RTOS.README file in the `<tirtos_install>\products\MWare_v###a` directory for more specific details. To indicate that the version has been modified, the name of the MWare folder has an added letter (beginning with "a" and to be incremented in subsequent versions). For example `<tirtos_install>\products\MWare_v202a`.

Note that the MWare drivers are not thread-safe. You can use synchronization mechanisms provided by SYS/BIOS to protect multiple threads that access the same MWare APIs.

For more information about MWare and controlSUITE, see the following:

Documents in <*tirtos_install*>/products/MWare_##

controlSUITE on Texas Instruments Wiki

controlSUITE Product Folder

1.8.3 MSPWare

MSPWare is an extensive suite of drivers, code examples, and design resources designed to simplify and speed development of MSP430 and MSP432 microcontroller applications. Currently, TI-RTOS uses MSPWare to support MSP430F5xx, MSP430F6xx, and MSP432 devices. TI-RTOS utilizes MSPWare's driverlib, usblib430, and grlib components.

The version of MSPWare provided with TI-RTOS differs from the full version in several ways. See the TI-RTOS.README file in the `<tirtos_install>\products\MSPWare_2_##_##_##a` directory for more specific details. To indicate that the version has been modified, the name of the MSPWare folder has an added letter (beginning with "a" and to be incremented in subsequent versions). For example `<tirtos_install>\products\MSPWare_2_00_00_40c`.

Note that the MSPWare drivers are not thread-safe. You can use synchronization mechanisms provided by SYS/BIOS to protect multiple threads that access the same MSPWare APIs.

For more information about MSPWare, see the following:

Release notes in <*tirtos_install*>/products/MSPWare_#_##_##_##a

Documents in <*tirtos_install*>/products/MSPWare_#_##_##_##a/driverlib/doc

Documents in <*tirtos_install*>/products/MSPWare_#_##_##_##a/usblib430/MSP430_USB_Software/Documentation

MSPWare Product Folder

1.8.4 TivaWare

This software is an extensive suite of software designed to simplify and speed development of Tiva-based (ARM Cortex-M) microcontroller applications. (TivaWare was previously called StellarisWare.)

The version of TivaWare provided with TI-RTOS differs from the standard release in that it has been rebuilt. See the TI-RTOS.README file in the `<tirtos_install>\products\TivaWare_C_Series-1.#` directory for more specific details.

Note that the TivaWare drivers are not thread-safe. You can use synchronization mechanisms provided by SYS/BIOS to protect multiple threads that access the same TivaWare APIs.

For more information about TivaWare, see the following:

Documents in <*tirtos_install*>/products/TivaWare_####/docs

TivaWare Product Folder

Online StellarisWare Workshop

1.8.5 C26xxWare and the CC3200 Driverlib

The CC3200 Driverlib provides driver source code and libraries for SimpleLink Wireless MCUs. This Driverlib is a subset of the CC3200 SDK. It provides register-level access to CC3200 peripherals. The version of the Driverlib provided with TI-RTOS differs from the standard release in that it has been rebuilt with an Operating System Abstraction Library (OSAL) for TI-RTOS.

CC13xxWare and CC26xxWare are software suites that provide register-level access to CC13xx and CC26xx peripherals.

Note that these drivers are not thread-safe. You can use synchronization mechanisms provided by SYS/BIOS to protect multiple threads that access the same CC3200 Driverlib and CC26xxWare APIs.

For more information, see the following:

SimpleLink WiFi Radio Tool

1.9 XDCtools

XDCtools is a separate software component provided by Texas Instruments that provides the underlying tooling needed for configuring and building SYS/BIOS, NDK, and UIA.

TI-RTOS installs XDCtools only if the version needed by TI-RTOS has not already been installed as part of a CCS or SYS/BIOS installation. If TI-RTOS installs XDCtools, it places it in the top-level CCS directory (for example, c:\ti), not the TI-RTOS products directory.

- XDCtools provides the XGCONF Configuration Editor and the scripting language used in the *.cfg files. This is used to configure modules in a number of the components that make up TI-RTOS.
- XDCtools provides the tools used to build the configuration file. These tools are used automatically by CCS if your project contains a *.cfg file. This build step generates source code files that are then compiled and linked with your application code.
- XDCtools provides a number of modules and runtime APIs that TI-RTOS and its components leverage for memory allocation, logging, system control, and more.

XDCtools is sometimes referred to as "RTSC" (pronounced "rit-see"—Real Time Software Components), which is the name for the open-source project within the Eclipse.org ecosystem for providing reusable software components (called "packages") for use in embedded systems. For more about how XDCtools and SYS/BIOS are related, see the SYS/BIOS User's Guide (SPRUEX3).

For more information about XDCtools, see the following:

XDCtools API and configuration reference. In the top-level TI-RTOS installation directory, open the Release Notes and follow the links to **Documentation** and then to **Documentation Overview**. In the Documentation Overview page, choose the **TI-RTOS Kernel Runtime APIs and Configuration (cdoc)** item and see help for the xdc.runtime modules.

RTSC-Pedia Wiki

TI-RTOS forum on TI's E2E Community

1.10 wolfSSL Embedded SSL Library

The wolfSSL embedded SSL library (formerly called CyaSSL) is a lightweight, portable, C-language-based SSL/TLS library. It is targeted at applications for IoT, embedded, and RTOS primarily because of its size, speed, and feature set. The wolfSSL library supports industry standards up to the current TLS 1.2 and DTLS 1.2. It can be as small as 1/20th the size of OpenSSL.

The wolfSSL library is not installed as part of TI-RTOS. You can download and install it separately. In order to use wolfSSL in TI-RTOS applications, follow the steps in the Using wolfSSL with TI-RTOS wiki page. The wolfSSL library must be used with the TI-RTOS NDK stack.

For more information about wolfSSL, see the following:

wolfSSL website

wolfSSL manual

Instrumentation with TI-RTOS

This chapter describes how to instrument your application with log calls and view the data with System Analyzer (SA).

Topic **Page**

2.1 Overview 14
2.2 Adding Logging to a Project 15
2.3 Modifying an Example to Upload Logging Data at Runtime 17
2.4 Using Log Events 22
2.5 Viewing the Logs 23

2.1 Overview

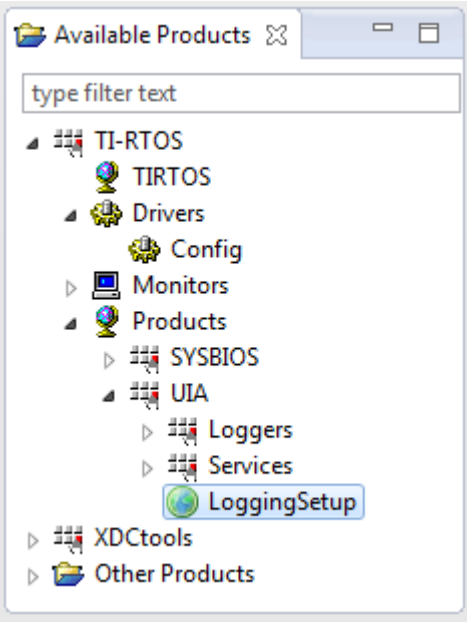
TI-RTOS uses the Unified Instrumentation Architecture (UIA) to instrument your application with log calls. The data can be viewed and visualized with System Analyzer (SA) to create execution graphs, load graphs and more. For detailed information on using UIA and SA refer to the Getting Started Guide in the <*tirtos_install*>/products/uia_#_##_##_##/docs directory and the System Analyzer User's Guide (SPRUH43).

Note that System Analyzer includes the views that can be opened from both the **Tools > RTOS Analyzer** and T**ools > System Analyzer** menus in CCS. That is, the RTOS Analyzer tools in CCS are part of System Analyzer.

2.2 Adding Logging to a Project

To add SYS/BIOS logging to a project, follow these steps:

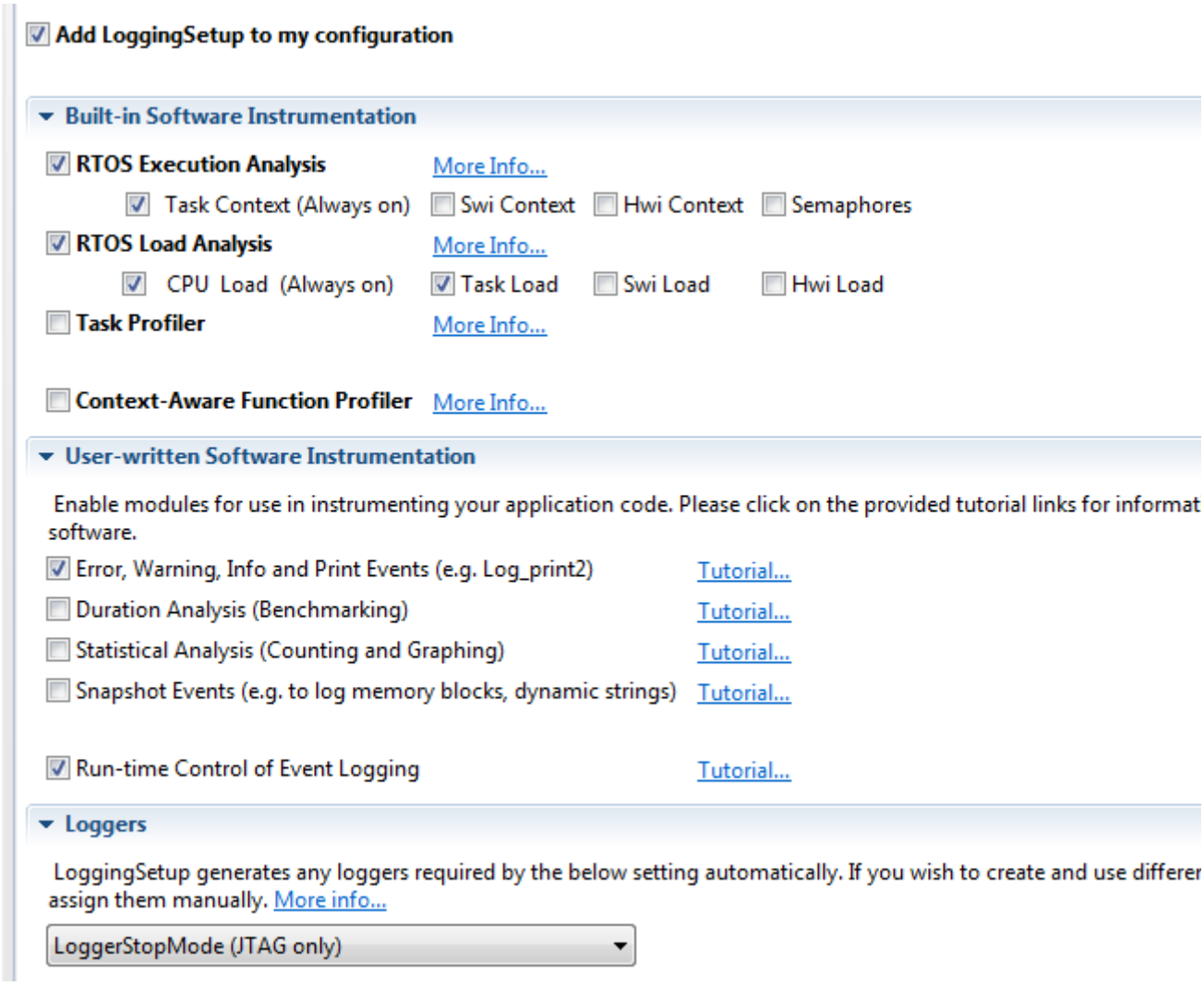
1. Double-click on the configuration file (.cfg) for your project to open it with the XGCONF Configuration Editor.
2. If LoggingSetup is already listed in your Outline pane, skip to Step 5.
3. In the "Available Products" area, expand the list as shown here to find the **LoggingSetup** module in the UIA product.



4. Right-click on the LoggingSetup module, and select **Use LoggingSetup**. This adds the LoggingSetup module to your project and opens the configuration page for the module.

5. Use the configuration page for the LoggingSetup module as follows:

   a) In the **Built-in Software Instrumentation** area, use the check boxes to select what types of threads you want to be logged for execution analysis, including tasks, software interrupts (Swi), and hardware interrupts (Hwi). If you check the **Run-time control of Event Logging** box, you can turn that type of logging on or off at runtime.



   b) Also in the **Built-in Software Instrumentation** area, you can check boxes if you want the CPU load to be logged for various types of activity.

   c) In the **User-written Software Instrumentation** area, you can enable logging of any additional instrumentation you have added with application code.

   d) In the **Loggers** area, you configure the logger to use in your main application. Calls to Log_info(), Log_warning(), and Log_error() in your main application as well as any instrumented driver logs will be sent to this logger. By default, LoggingSetup creates a logger that sends events over JTAG when the target is halted (that is, in Stop Mode).

The examples provided with TI-RTOS include and configure the LoggingSetup module. For more information on using LoggingSetup refer to Section 5.3.1 in the System Analyzer User's Guide (SPRUH43).

2.3 Modifying an Example to Upload Logging Data at Runtime

The UART Console example uses UIA to upload logging data at runtime to RTOS Analyzer and System Analyzer views in CCS. All other TI-RTOS examples, including UART Echo, use Stop Mode uploading of such data. This section provides the steps to modify the UART Echo example to use the USB for the same type of runtime data uploading performed by the UART Console example. These steps can be adapted to other TI-RTOS examples.

In order to change from stop mode to runtime uploading, you need to make changes to the UART Echo project, code, and configuration as described in the following pages.

2.3.1 Project Changes

Add the following two files to your UART Echo project:

- USBCDCD_LoggerIdle.c
- USBCDCD_LoggerIdle.h

These two files are included in the UART Console example. You can choose **Project > Add Files** in CCS and copy them to your project from the `<tirtos_install>/examples/<target>/uartconsole` directory.

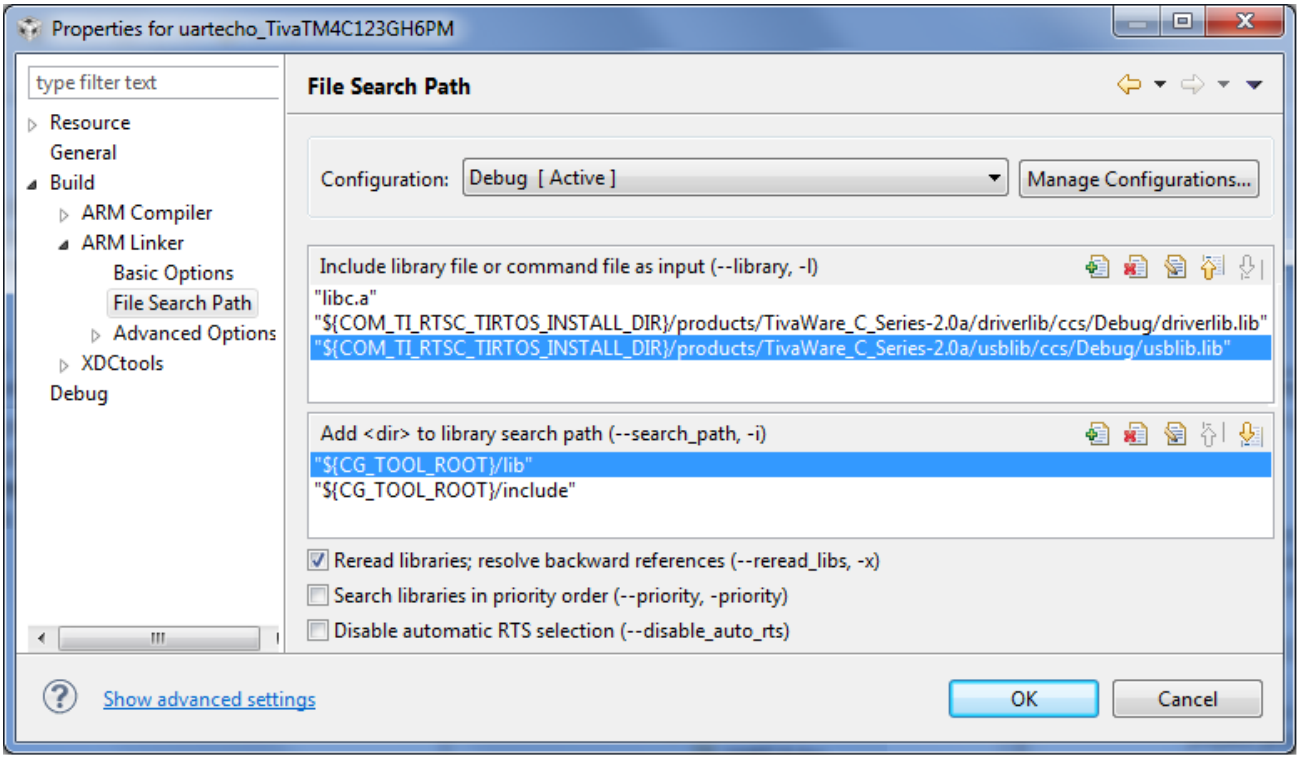
The UART Echo examples already include the appropriate USB library. This library is provided by MWare, TivaWare, and MSPWare. If you are modifying an example other than UART Echo, add the appropriate library from the following list to your project:

- `<tirtos_install>\products\MWare_v20#a\MWare\usblib\ccs\Debug\usblib.lib`
- `<tirtos_install>\products\TivaWare_C_Series-2.#a\usblib\ccs\Debug\usblib.lib`
- `<tirtos_install>\products\MSPWare_2_##_##_##a\driverlib\ccs-MSP430F5529\ccs-MSP430F5529.lib`

To add a library to a CCS project, follow these steps:

1. Right-click on the project name in the Project Explorer pane of CCS and select **Properties** from the context menu.
2. Expand the **Build > Linker** category and select the **File Search Path** category.
3. Click the **+** button over the **Include library file or command file as input** field.
4. Click **File System** in the Add file path dialog.
5. Browse to the location of the appropriate usblib library, and select the library file. Click **Open**.
6. Click **OK** in the Add file path dialog.

7. Click **OK** in the Properties dialog.



2.3.2 Code Changes

Open the uartecho.c file in CCS and add the following code:

- Include the USBCDCD_LoggerIdle.h header file:

```
#include "USBCDCD_LoggerIdle.h"
```

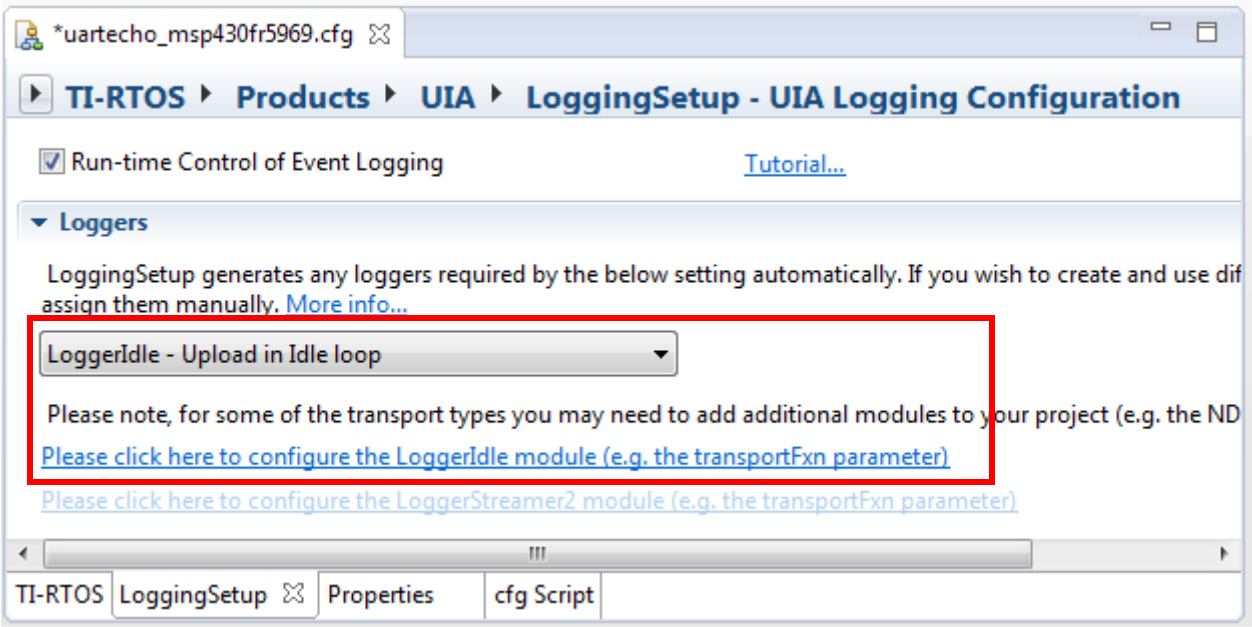
- Add the calls to Board_initUSB() and USBCDCD_init() to the main() function as shown in green below:

```
Int main(Void)
{
    Error_Block eb;
    Task_Params taskParams;

    /* Call board init functions. */
    Board_initGeneral();
    Board_initGPIO();
    Board_initUART();

    Board_initUSB(Board_USBDEVICE);

    System_printf("Starting the example\nSystem provider is set to SysMin,"
                  "halt the target and use ROV to view output.\n");
    /* SysMin will only print to the console when you call flush or exit */
    System_flush();

    /* Turn on user LED */
    GPIO_write(Board_LED, Board_LED_ON);

    /* Initialize the USB CDC device for logging transport */
    USBCDCD_init();

    /* Create the task */
    Error_init(&eb);
    Task_Params_init(&taskParams);
    taskParams.instance->name = "echo";
    echo = Task_create(echoFxn, &taskParams, &eb);
    if (echo == NULL) {
        System_printf("Task was not created\n");
        System_abort("Aborting...\n");
    }

    /* Enable interrupts and start SYS/BIOS */
    BIOS_start();

    return (0);
}
```

2.3.3 Configuration Changes

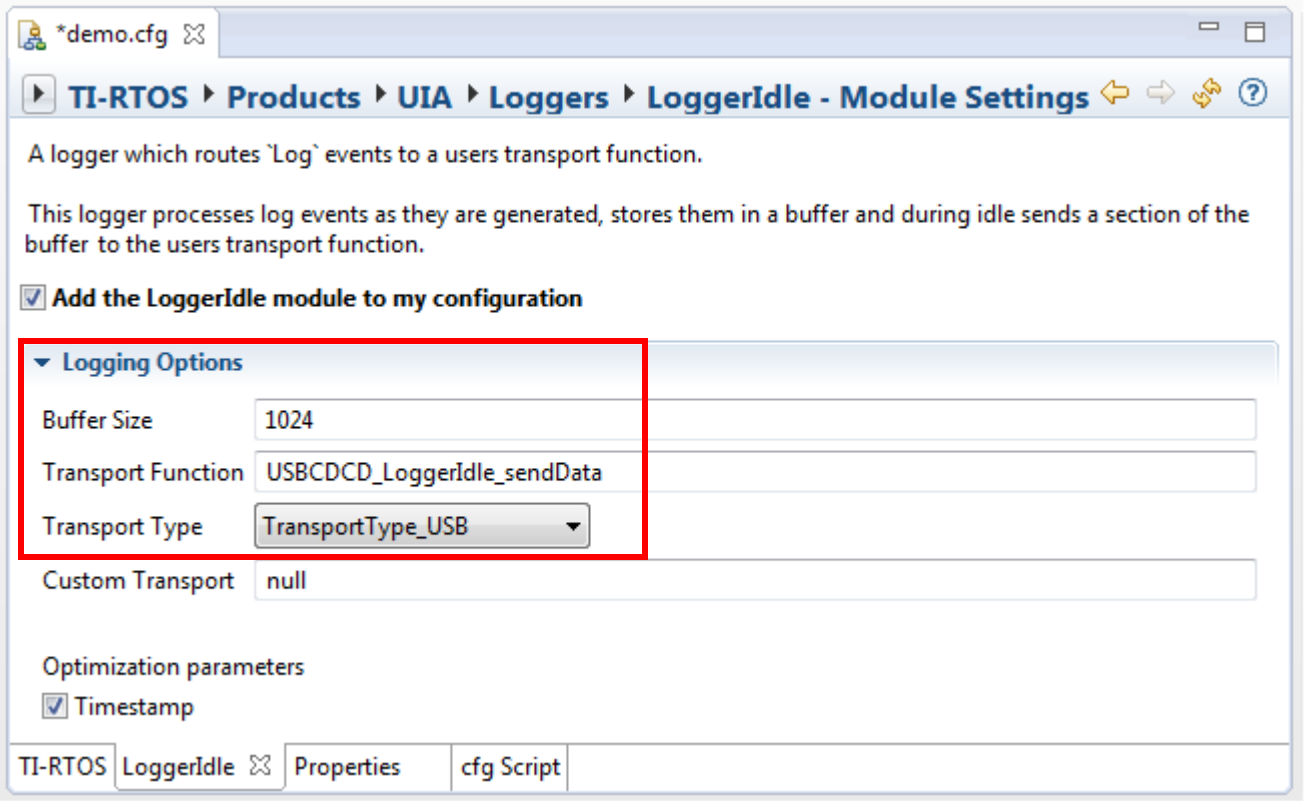
You can modify the project's configuration with the XGCONF Configuration Editor or with a text editor. Here are the steps for both of these methods:

1. Using the XGCONF Configuration Editor, open the UART Echo project's uartecho.cfg file.
2. Select the **LoggingSetup** module in the Outline pane.
3. In the LoggingSetup configuration page, move to the **Loggers** section and change the logger type to **LoggerIdle - Upload in Idle loop**.



4. Follow the **Please click here to configure the LoggerIdle module** link.

5. In the Logger Idle configuration page, check the **Add the LoggerIdle module to my configuration** box.
6. Set the **Buffer Size** to 1024.
   Set the **Transport Function** to USBCDCD_LoggerIdle_sendData,
   Set the **Transport Type** to TransportType_USB.



7. Save the configuration file.

To modify the configuration with a text editor, add the following statements at the end of the uartecho.cfg file:

```
LoggingSetup.loggerType = LoggingSetup.LoggerType_IDLE;
LoggerIdle.transportType = LoggerIdle.TransportType_USB;
LoggerIdle.bufferSize = 1024;
LoggerIdle.transportFxn = "&USBCDCD_LoggerIdle_sendData";
```

Note: The configuration file should already contain the following statement:

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

2.4 Using Log Events

You can add Log events to your application and control whether Log events are processed by drivers as described in the following sub-sections.

2.4.1 Adding Log Events to your Code

Your application can send messages to a Log using the standard Log module APIs (xdc.runtime.Log).

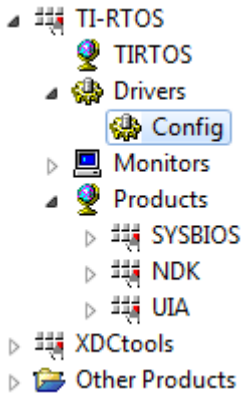
Log calls are of the format `Log_typeN(String, arg1, arg2… argN)`. Valid types are print, info, warning and error. N is the number of arguments between 0 and 5. For example:

```
Log_info2("tsk1 Entering. arg0,1 = %d %d", arg0, arg1)
```

See the SYS/BIOS Log example project for more use cases.

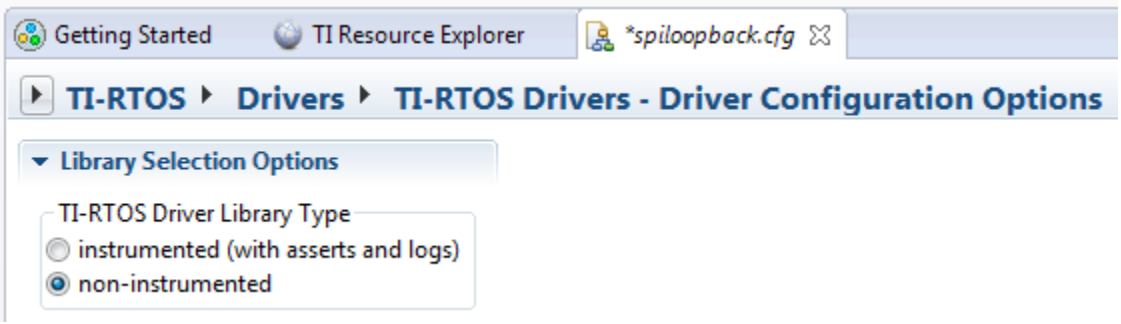
2.4.2 Using Instrumented or Non-Instrumented Libraries

TI-RTOS allow you to control whether or not Log events are handled by choosing to build with the instrumented or non-instrumented libraries. The instrumented libraries process Log events and Asserts, while the non-instrumented libraries do not.



To select the type of library to build with, follow these steps:

1. Double-click on the configuration file (.cfg) for your project to open it with the XGCONF Configuration Editor.
2. In the “Available Products” area, expand the **Drivers** item and choose the **Config** module.
3. On the configuration page, choose whether to use the instrumented or non-instrumented libraries.



4. On the same page, check the boxes for any drivers your application will use. The WiFi driver must be configured as a separate module. Use the Drivers folder in the Available Products pane.

See Section 5.2.1 for more about configuring instrumented or non-instrumented libraries. Refer to the individual drivers in Chapter 5 for details about what is logged and which Diags masks are used.

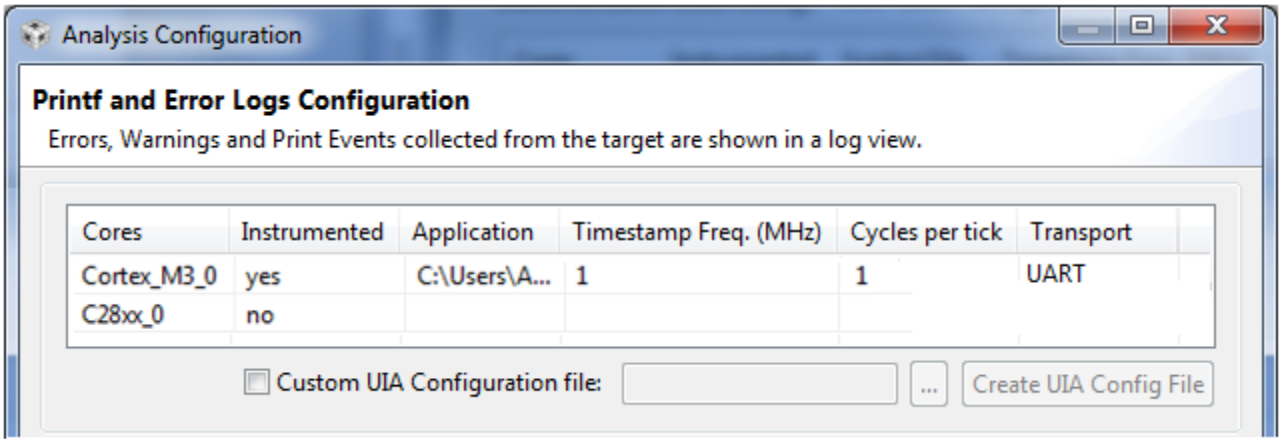
2.5 Viewing the Logs

You can use CCS to view Log messages using the RTOS Analyzer, System Analyzer, and/or ROV tools.

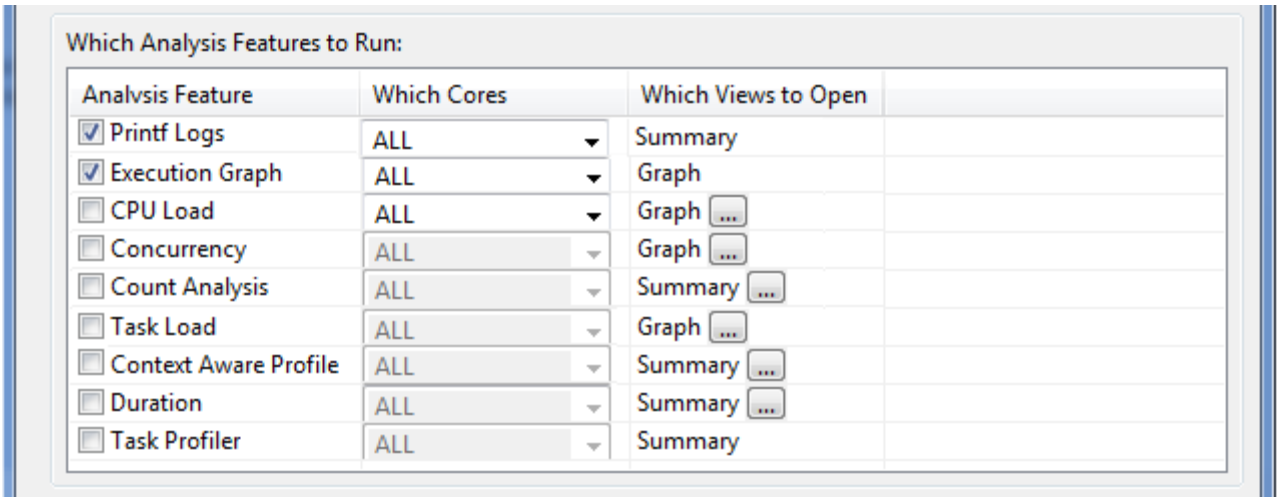
2.5.1 Using RTOS Analyzer and System Analyzer

After you have built and run your application, follow these steps in the CCS Debug view to see Log messages from your application with RTOS Analyzer:

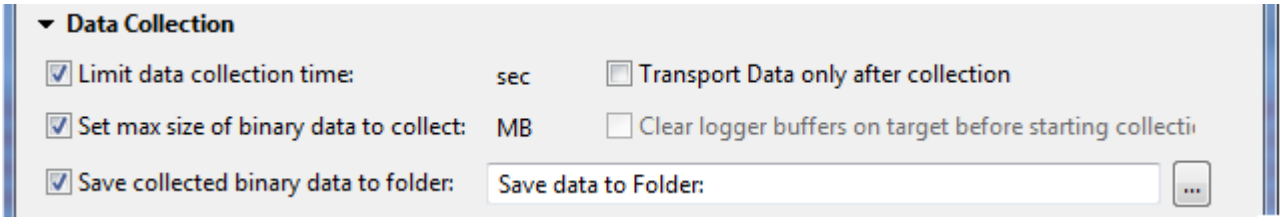
1. Open an analyzer by selecting **Tools > RTOS Analyzer > Printf and Error Logs**.
2. The Analysis Configuration detects the type of transport you are using.



3. Select additional analyzer views you would like to run.



4. Configure the analyzer to run for a set time or forever (that is, until you manually pause the data transfer). You can also choose when to process the data (Transport Data only after collection), whether to clear existing data and save the data to a file which can be imported back into SA.



If you save data to a file, you can analyze it later by selecting **Tools > RTOS Analyzer > Open File > Open Binary File**.

See Section 4.2 ("Starting an RTOS Analyzer or System Analyzer Session") in the System Analyzer User's Guide (SPRUH43) for more about using this dialog.

2.5.2 Viewing Log Records in ROV

The RTOS Object View (ROV) can be used to view log events stored on the target.

After you have built and run your application, you can open the ROV tool in the CCS Debug view by selecting **Tools > RTOS Object View (ROV)** and then navigating to the logging module you want to view (for example, LoggerStopMode or LoggerIdle). When the target is halted, ROV repopulates the data. Select the **Records** tab to view log events still stored in the buffer. For loggers configured to use JTAG, the records shown here are also uploaded to System Analyzer. If you are using the LoggerIdle module, these are the records that have not yet been sent.

See the http://rtsc.eclipse.org/docs-tip/RTSC_Object_Viewer web page for more about using the RTOS Object View (ROV) tool.

Debugging TI-RTOS Applications

This chapter provides information about ways to debug your TI-RTOS applications.

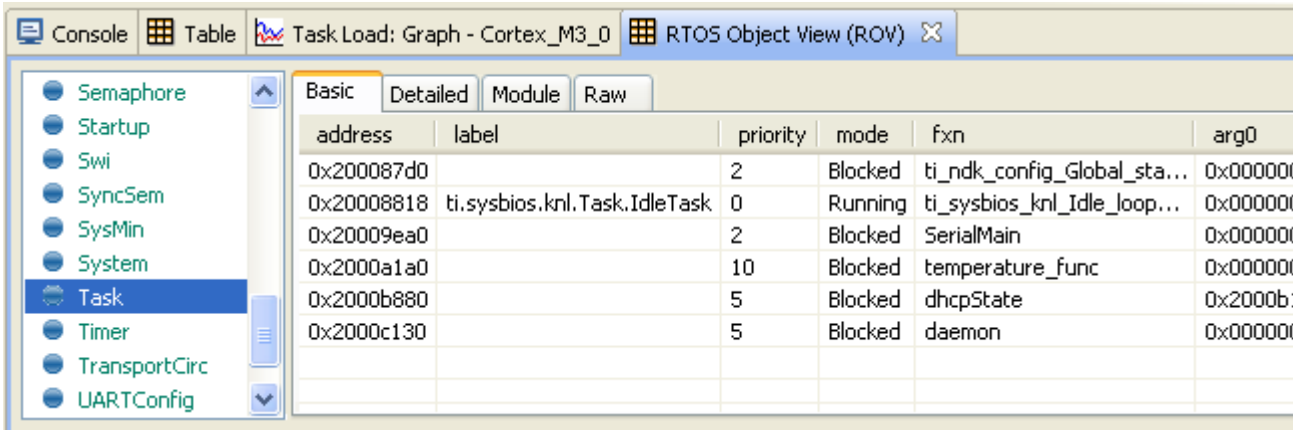
Topic **Page**

3.1 Using CCS Debugging Tools 25
3.2 Using IAR Debugging Tools 27
3.3 Generating printf Output 28
3.4 Controlling Software Versions for Use with TI-RTOS 31
3.5 Understanding the Build Flow 32

3.1 Using CCS Debugging Tools

Within Code Composer Studio (CCS), there are several tools you can use to debug your TI-RTOS applications:

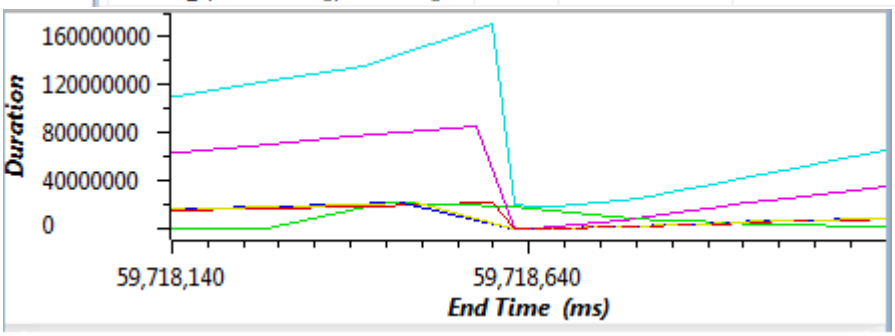
- **RTOS Object View (ROV)** is a stop-mode debugging tool, which means it can receive data about an application only when the target is halted, not when it is running. ROV is a tool provided by the XDCtools component. ROV gets information from many of the modules your applications are likely to use.



The ROV tool is also available for use with TI-RTOS examples within IAR Embedded Workbench. See the *TI-RTOS Getting Started Guide* for your device family for details.

- **System Analyzer** includes analysis features for viewing the CPU and thread loads, the execution sequence, thread durations, and context profiling. The features include graphs, detailed logs, and summary logs. These views gather data from the UIA component. For information, see the System Analyzer User's Guide (SPRUH43).

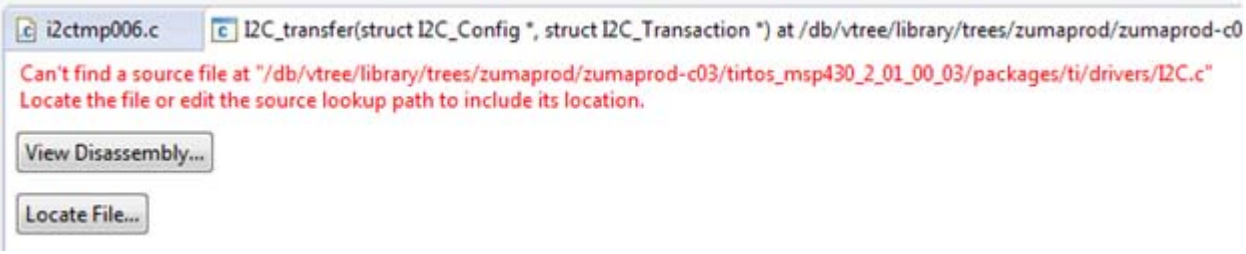
Name	Count	Incl Count Min	Incl Count Max	Incl Count Average
C64XP_0, serverFxn(), doLoad().0	14	2000203	2051632	2,003,924.71
C64XP_1, serverFxn(), doLoad().0	15	2000194	2000640	2,000,245.80
C64XP_2, serverFxn(), doLoad().0	16	2000195	2000622	2,000,244.00



- **Printf-style output** lets you use the tried-and-true debugging mechanism of sending execution information to the console. For information, see "Generating printf Output" on page 28.
- **Standard CCS IDE features** provide many tools for debugging your applications. In CCS, choose **Help > Help Contents** and open the **Code Composer Help > Views and Editors** category for a list of debugging tools and more information. These debugging features include:
  - Source-level debugger
  - Assembly-level debugger
  - Breakpoints (software and hardware) See Section 3.1.1 for information about stepping through driver code.
  - Register, memory, cache, variable, and expression views
  - Pin and port connect views
  - Trace Analyzer view
- **Exception Handling** is provided by SYS/BIOS. If this module is enabled, the execution state is saved into a buffer that can be viewed with the ROV tool when an exception occurs. Details of the behavior of this module are target-specific. In the CCS online help, see the SYS/BIOS API Reference help on the ti.sysbios.family.c64p.Exception module or the ti.sysbios.family.arm.exc.Exception module for details.
- **Assert Handling** is provided by XDCtools. It provides configurable diagnostics similar to the standard C assert() macro. In the CCS online help, see the XDCtools API Reference help on the xdc.runtime.Assert module for details.

3.1.1 Stepping Through TI-RTOS Code

Stepping through code is vital when debugging an application. When using CCS there are instances where stepping into a TI-RTOS Kernel or Driver API will produce an output message on the code editor window similar to the following. A similar message is shown in IAR Embedded Workbench.

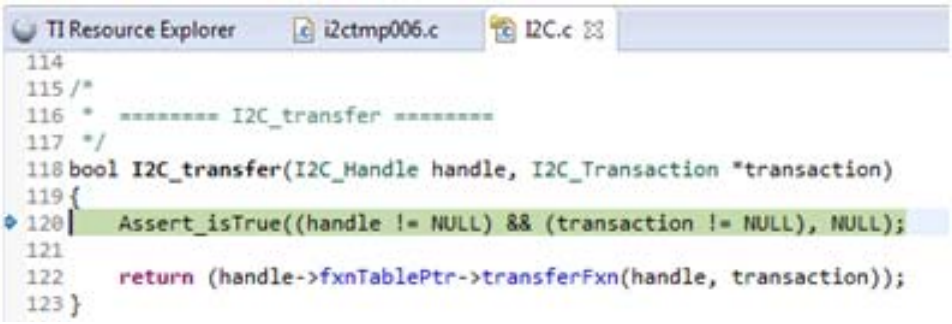


Since TI-RTOS provides pre-compiled libraries, the debug information in the library is based on the file locations when the libraries were built. Because these locations differ on your system, when the debugger attempts to access TI-RTOS library debug information, it cannot find the source files. There are two ways to correct this issue: rebuild TI-RTOS or locate the source files.

Rebuilding TI-RTOS in your development environment regenerates the debug information for all libraries. This process is only done once per TI-RTOS installation. (You will need to repeat these steps if you use a different TI-RTOS product or if install a newer TI-RTOS version.) To rebuild TI-RTOS, see Section 8.1 and Section 8.1.4.

The other method is to locate the source files within the file system. This process is faster than rebuilding TI-RTOS, but will need to be repeated for every TI-RTOS driver or kernel module being debugged. To location the source files for CCS, follow these steps:

1. Click the **Locate File** button when the message shown above appears.
2. Navigate to the directory that contains the source file mentioned in the message. For TI-RTOS drivers, this is likely `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers`.
3. Click **OK**. The editor window will search for the file and show the source code.



IAR Embedded Workbench provides similar tools for locating the source files within the file system.

3.2 Using IAR Debugging Tools

For information about using TI-RTOS examples in IAR, see the wiki page on Creating TI-RTOS Applications in IAR Embedded Workbench on the Texas Instruments wiki.

3.3 Generating printf Output

Along with many advanced GUI debugging features described in “Using CCS Debugging Tools” on page 25, TI-RTOS provides flexibility with the tried-and-true printf method of debugging. TI-RTOS supports both the standard printf() and a more flexible replacement called System_printf().

3.3.1 Output with printf()

By default, the printf() function outputs data to a CIO buffer on the target. When CCS is attached to the target (for example, via JTAG or USB), the printf() output is displayed in the Console window. It is important to realize that when the CIO buffer is full or a ‘\n’ is output, a CIO breakpoint is hit on the target. This allows CCS to read the data and output the characters to the console. Once the data is read, CCS resumes running the target. This interruption of the target can have significant impact on a real-time system. Because of this interruption and the associated performance overhead, use of the printf() API is discouraged.

The UART Console example shows how to route the printf() output to a UART via the add_device() API.

3.3.2 Output with System_printf()

The xdc.runtime.System module provided by the XDCtools component offers a more flexible and potentially better-performing replacement to printf() called System_printf().

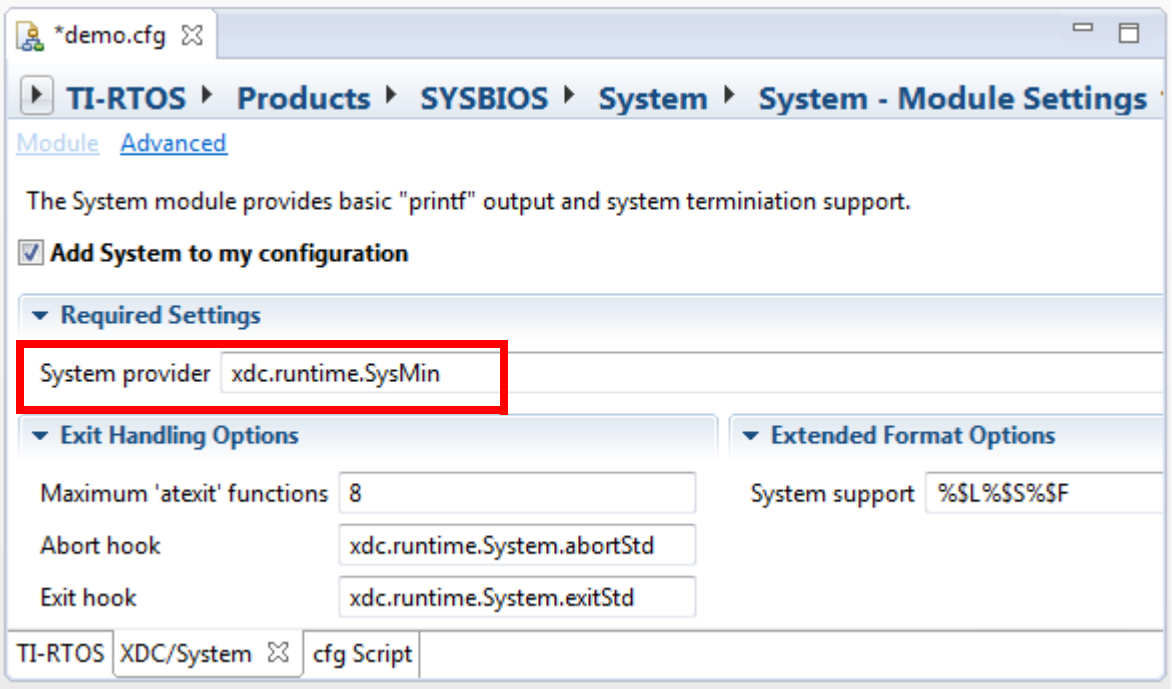
The System module allows different low-level implementations (System Support implementations) to be plugged in based on your needs. You can plug in the System Support implementation you want to use via the application configuration. Your choice does not require any changes to the runtime code.

Currently the following System Support implementations are available:

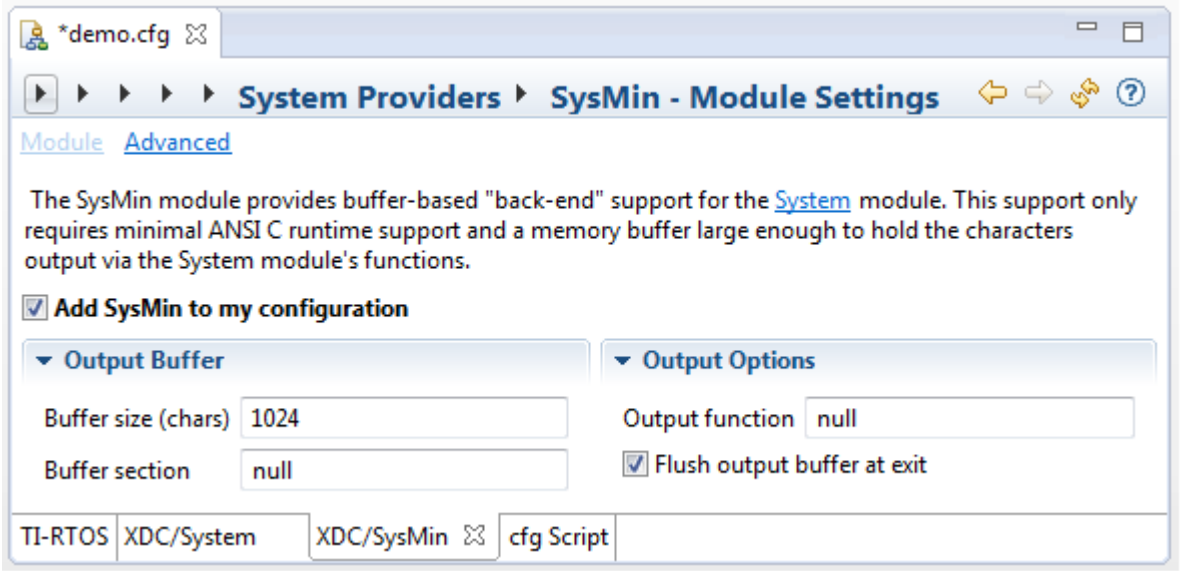
- **SysMin:** Stores output to an internal buffer. The buffer is flushed to stdout (which goes to the CCS Console view) when System_flush() is called or when an application terminates (for example, when BIOS_exit() or exit() is called). When the buffer is full, the oldest characters are over-written. Characters that have not been sent to stdout can be viewed via the RTOS Object View (ROV) tool. The SysMin module is part of the XDCtools component. Its full module path is xdc.runtime.SysMin.
- **SysCallback:** Simply calls user-defined functions that implement the System module's functionality. The UART Console example provides a set of functions that use the UART. The SysCallback module is part of the XDCtools component. Its full module path is xdc.runtime.SysCallback.
- **SysStd:** Sends the characters to the standard printf() function. The SysStd module is part of the XDCtools component. Its full module path is xdc.runtime.SysStd.

Most TI-RTOS examples use either the SysMin or SysStd module. The UART Console example uses SysCallback and routes the output to a UART.

To configure the SysMin module, open the application's *.cfg file with the XGCONF Configuration Editor. In the Outline area, select the System module. Configure the System Provider to use SysMin as follows:



Then, find the SysMin module in the Outline pane, and configure the output buffer and options as needed. For example, here are the settings used by most examples provided with TI-RTOS:



The following statements create the same configuration as the graphical settings shown for the System and SysMin modules:

```
var System = xdc.useModule('xdc.runtime.System');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
System.SupportProxy = SysMin;
```

The following table shows the pros and cons of the various System provider modules:

Table 3-1 System providers shipped with TI-RTOS

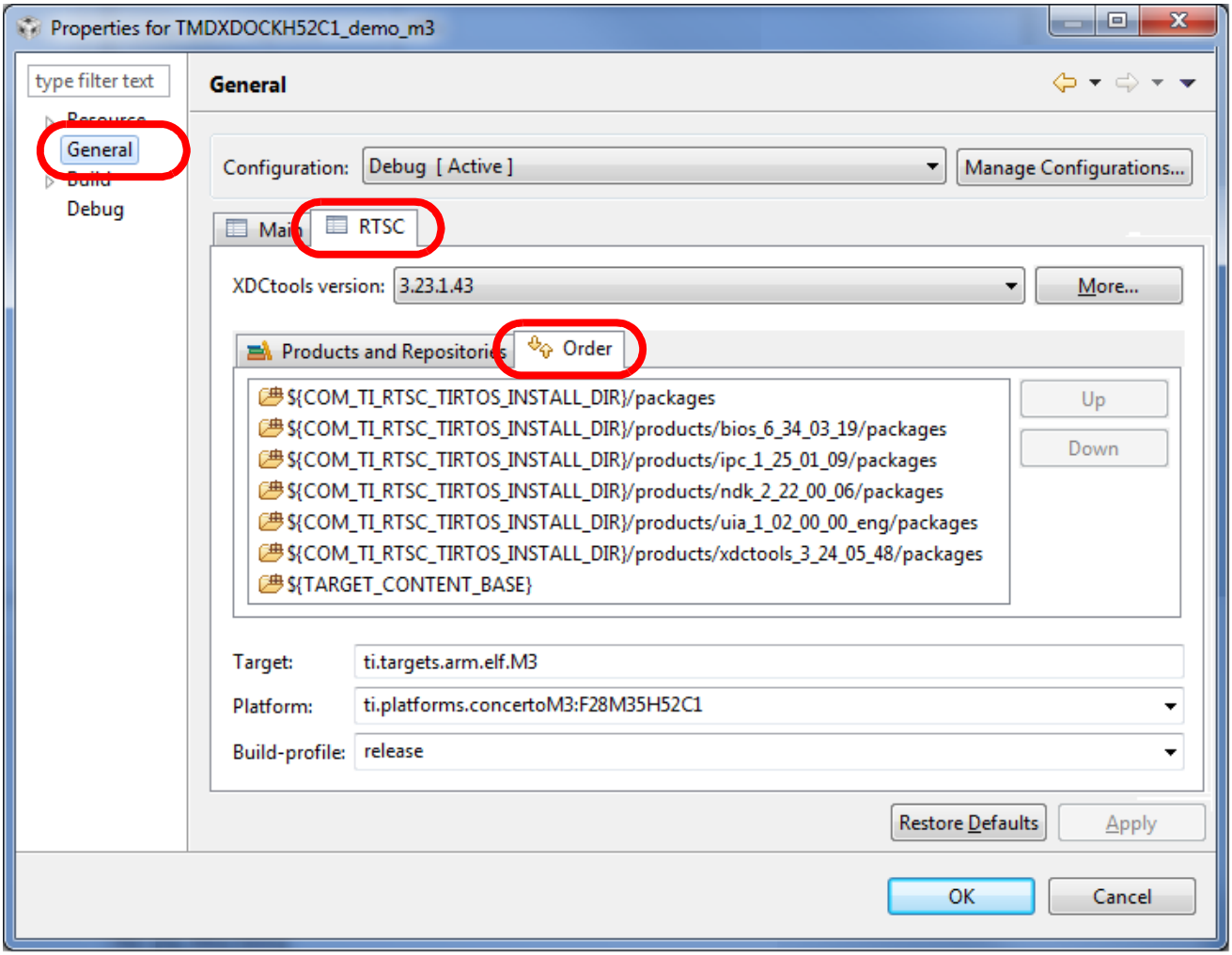
System Provider	Pros	Cons
SysMin	• Good performance	• Requires RAM (but size is configurable) • Potentially lose data • Out-of-box experience • To view in CCS console, you must add System_flush() or have the application terminate • Can use ROV to view output, but requires you halt the target
SysStd	• Easy to use (just like printf)	• Bad to use (just like printf). CCS halts target when CIO buffer is full or a '\n' is written • Cannot be called from a SYS/BIOS Hwi or Swi thread
SysCallback	• Can be used for many custom purposes	• Requires that you provide your own callback functions

Please note, the System module also provides the additional APIs that can be used instead of standard 'C' functions: System_abort(), System_atexit(), System_exit(), System_putch(), and System_flush().

3.4 Controlling Software Versions for Use with TI-RTOS

You do not need to add the "products" subdirectory to the RTSC (also called XDCtools) discovery path. Once CCS has found the main TI-RTOS directory, it will also find the additional components provided in that tree.

In addition, the components installed with TI-RTOS will be used as needed by examples you import with the TI Resource Explorer. When you choose **Project > Properties** for a project that uses TI-RTOS, the sub-components are not checked in the **RTSC** tab of the **General** category. However, the version installed with TI-RTOS is automatically used for sub-components that are needed by the example. You can see these components and which versions are used by going to the **Order** tab.



If, at a later time, you install newer software versions that you want to use instead of the versions installed with TI-RTOS, you can use the **Products and Repositories** tab to add those versions to your project and the **Up** and **Down** buttons in the **Orders** tab to make your newer versions take precedence over the versions installed with TI-RTOS. However, you should be aware that is it possible that newer component versions may not be completely compatible with your version of TI-RTOS.

Note that in the **RTSC** tab, the XDCtools version in the drop-down list is the version that controls UI behavior in CCS, such as the XGCONF editor and various RTSC dialog layouts. The XDCtools version in the list of products is the version used for APIs and configuration, such as the xdc.runtime modules.

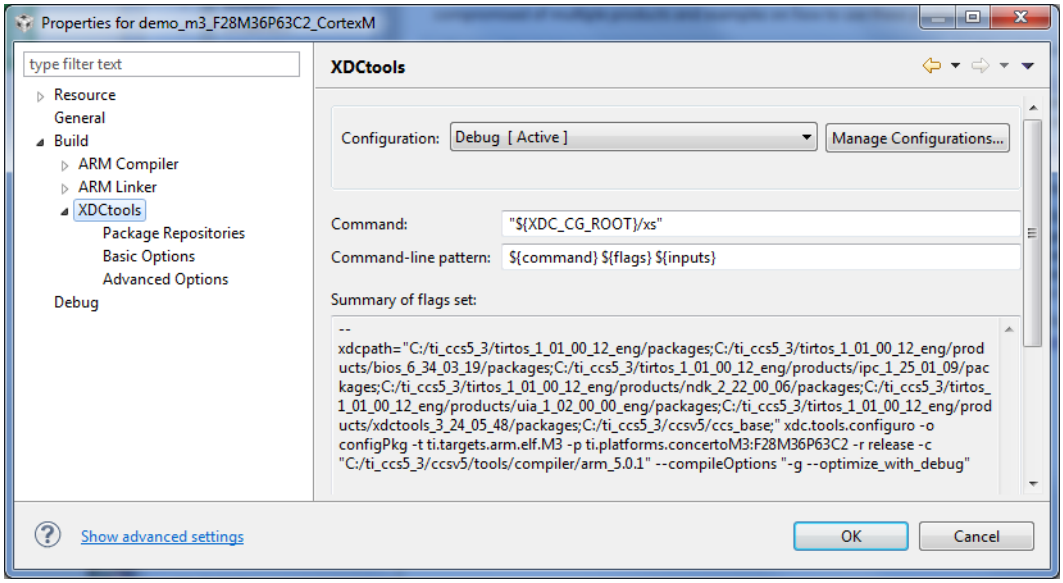
3.5 Understanding the Build Flow

The build flow for TI-RTOS applications begins with an extra step to process the configuration file (*.cfg) in the project. The configuration file is a script file with syntax similar to JavaScript. You can edit it graphically in CCS using the XGCONF Configuration Editor. The configuration determines which modules in TI-RTOS components are used, sets global behavior parameters for modules, and statically creates objects managed by the modules. Static configuration has several advantages, including reducing code memory use by the application. Components that can be configured using this file include XDCtools, SYS/BIOS, TI-RTOS, NDK, and UIA.

The configuration file is processed by the XDCtools component. If you look at the messages printed during the build, you will see a command line that runs the “xs” executable in the XDCtools component with the “xdc.tools.configuro” tool specified. For example:

```
'Invoking: XDCtools'

"<>/xs" --xdcpath="<tirtos_install>/packages;
<bios_install>/packages;<uia_install>/packages;" xdc.tools.configuro -o configPkg
-t ti.targets.arm.elf.M3 -p ti.platforms.concertoM3:F28M35H52C1 -r release
-c "C:/ccs/ccsv6/tools/compiler/tms470" "../<project>.cfg"
```

In CCS, you can control the command-line options used with XDCtools by choosing **Project > Properties** from the menus and selecting the **Build > XDCtools** category.



Target settings for processing your individual project are in the **RTSC** tab of the **CCS General** category. (RTSC is the name for the Eclipse specification implemented by XDCtools.)

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

For more information about the build flow, see Chapter 2 of the SYS/BIOS User’s Guide (SPRUEX3). For command-line details about xdc.tools.configuro, see the RTSC-pedia reference topic.

Board-Specific Files

This chapter provides information that is specific to targets for which you can use TI-RTOS.

Topic **Page**

4.1 Overview 33
4.2 Board-Specific Code Files 34
4.3 Linker Command Files 35
4.4 Target Configuration Files 35

4.1 Overview

Currently, TI-RTOS provides examples for the following boards:

Family	Device on Board	Board
Concerto (ARM M3 + DSP 28x)	F28M35H52C1	TMDXDOCKH52C1 Experimenter Kit
Concerto (ARM M3 + DSP 28x)	F28M36P63C2	TMDXDOCK28M36 Experimenter Kit
ARM (Tiva)	TM4C123GH6PM	EK-TM4C123GXL LaunchPad
ARM (Stellaris)	LM4F120H5QR	EK-LM4F120XL LaunchPad (earlier version of EK-TM4C123GXL LaunchPad)
ARM (Tiva) Cortex-M4F	TM4C129XNCZAD	DK-TM4C129X Evaluation Kit
ARM (Tiva)	TM4C1294NCPDT	EK-TM4C1294XL Evaluation Kit
ARM (Tiva)	TM4C129ENCPDT	EK-TM4C129EXL Evaluation Kit
MSP430F5xx	MSP430F5529	MSP-EXP430F5529LP LaunchPad
MSP430FR5xx	MSP430FR5969	MSP-EXP430FR5969LP LaunchPad
MSP430FR5xx	MSP430FR5994	MSP-EXP430FR5994LP LaunchPad
MSP430FR6xx	MSP430FR6989	MSP-EXP430FR6989LP LaunchPad
MSP432	MSP432P401R	MSP-EXP432P401RLP LaunchPad
CC1310	CC1310F128	CC1310 LaunchPad
CC1310	CC1310F128	CC1310 Development Kit
CC1350	CC1350F128	CC1350 LaunchPad
CC1350	CC1350F128	CC1350 SensorTag
CC2650	CC2650F128	CC2650 LaunchPad

CC2650	CC2650F128	CC2650 SensorTag
CC2650	CC2650F128	CC2650 Development Kit
CC3200	CC3200	CC3200-LAUNCHXL

F28M3x devices contain both M3 and 28x subsystems.

TI-RTOS can also be used on other boards. Examples are provided specifically for the supported boards, but libraries are provided for each of these device families, so that you can port the examples to similar boards. The Texas Instruments Wiki contains a TI-RTOS Porting Guide and a topic on Creating TI-RTOS Projects for Other MSP430 Devices.

4.2 Board-Specific Code Files

TI-RTOS examples contain a board-specific C file (and its companion header file). The filenames are <*board*>.c and <*board*>.h, where <*board*> is the name of the board, such as TMDXDOCKH52C1. Notice that an underscore is used in place of a hyphen in file and folder names for board names that contain a hyphen, such as EKS-LM4F232.

All the examples for a specific board have identical <*board*> files. These files are considered part of the application, and you can modify them as needed.

The board-specific code files do not perform any dynamic memory allocation.

The <*board*> files perform board-specific configuration of the drivers provided by TI-RTOS. For example, they perform the following:

- GPIO port and pin configuration
- LED configuration

In addition, the board-specific files provide the following functions that you can use in your applications, These are typically called from main(). Files are provided only for boards on which the driver is supported.

- <*board*>_initDMA() function
- <*board*>_initEMAC() function
- <*board*>_initGeneral() function
- <*board*>_initGPIO() function
- <*board*>_initI2C() function
- <*board*>_initSDSPI() function
- <*board*>_initSPI() function
- <*board*>_initUART() function
- <*board*>_initUSB() function
- <*board*>_initUSBMSCHFatFs() function
- <*board*>_initWatchdog() function
- <*board*>_initWiFi() function

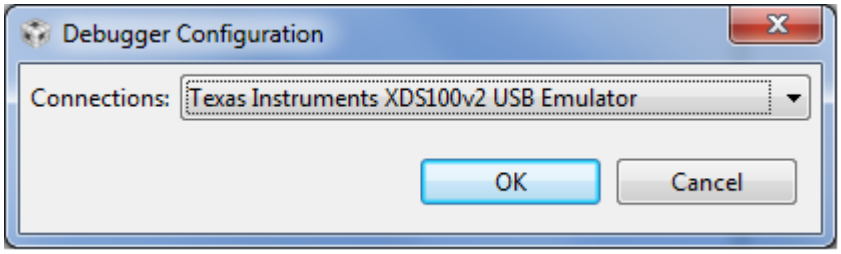
4.3 Linker Command Files

All of TI-RTOS examples contain a <*board*>.cmd linker command file. A different file is provided for each supported board. These files define memory segments and memory sections used by the application.

4.4 Target Configuration Files

To create a target configuration for an example provided with TI-RTOS, use Step 3 (Debugger Configuration) in the TI Resource Explorer. (To create TI-RTOS example projects using the TI Resource Explorer, see Chapter 3 of the TI-RTOS Getting Started Guide.)

When you click the link for Step 3, you see the Debugger Configuration dialog. Choose an emulator from the list. For F28M3x devices, choose the **Texas Instruments XDS 100v2 USB Emulator**. For Tiva devices, choose the **Stellaris In-Circuit Debug Interface**. For MSP430 devices, choose the **TI MSP430 USB1**. For MSP432 devices, use the **Texas Instruments XDS 110 USB Emulator**.



The Debugger Configuration step creates a CCS Target Configuration File (*.ccxml). This file specifies the connection and device for the project for use in a debugging session. You can choose **View > Target Configurations** in CCS to see and edit these files.

Note: If you want to use a simulator instead of a hardware connection, select any emulator in the Debugger Configuration dialog and click **OK**. Then choose **View > Target Configurations.** Expand the **Projects** list and double-click on the *.ccxml file for your example project to open the target configuration editor. Select **Texas Instruments Simulator** in the Connection field, and the simulator for your device in the Device list. Then click **Save**.

For the F28M3x Demo example, you should not use a C28 target configuration. Instead, use the target configuration for the M3 and connect to the C28 and load that application manually as described in the example's readme file.

TI-RTOS Drivers

This chapter provides information about the drivers provided with TI-RTOS.

Topic **Page**

5.1 Overview 37
5.2 Driver Framework 38
5.3 Camera Driver 48
5.4 EMAC Driver 50
5.5 GPIO Driver 52
5.6 I2C Driver 56
5.7 I2S Driver 63
5.8 LCD Driver 66
5.9 PWM Driver 68
5.10 SDSPI Driver 71
5.11 SPI Driver 73
5.12 UART Driver 79
5.13 USBMSCHFatFs Driver 83
5.14 USB Reference Modules 86
5.15 USB Device and Host Modules 89
5.16 Watchdog Driver 91
5.17 WiFi Driver 93

5.1 Overview

TI-RTOS includes drivers for a number of peripherals. These drivers are in the `<tirtos_install>/products/tidrivers_<version>/packages/ti/drivers` directory. TI-RTOS examples show how to use these drivers. Note that all of these drivers are built on top of MWare, MSPWare, TivaWare, CC26xxWare, and CCWare. This chapter contains a section for each driver.

- **Camera.** Driver for CC3200 Camera BoosterPack.
- **EMAC.** Ethernet driver used by the networking stack (NDK) and not intended to be called directly.
- **GPIO.** API set intended to be used directly by the application or middleware to manage the GPIO interrupts, pins, and ports (and therefore the LEDs).
- **I^2C.** (Inter-Integrated Circuit) API set intended to be used for attaching low-speed peripherals to embedded system boards. The APIs are used directly by the application or middleware.
- **I^2S.** (Inter-IC Sound) API set intended to be used for connecting digital audio devices so that audio signals can be communicated between devices. The APIs are used directly by the application or middleware.
- **LCD.** Driver for CC26xx LCD display.
- **PIN.** Driver for CC26xx Pin interrupts.
- **PWM.** API set intended to be used directly by the application or middleware to generate Pulse Width Modulated signals.
- **SDSPI.** SPI-based SD driver used by FatFs and not intended to be interfaced directly.
- **SPI.** API set intended to be used directly by the application or middleware to communicate with the Serial Peripheral Interface (SPI) bus. This driver has been designed to operate in an RTOS environment such as SYS/BIOS. It protects SPI transactions with OS primitives supplied by SYS/BIOS. SPI is sometimes called SSI (Synchronous Serial Interface).
- **UART.** API set intended to be used directly by the application to communicate with the UART.
- **USBMSCHFatFs.** USB MSC Host under FatFs (for flash drives). This driver is used by FatFS and is not intended to be called directly.
- **Other USB functionality.** See the USB examples for reference modules that provide support for the Human Interface Device (HID) class (mouse and keyboard) and Communications Device Class (CDC). This code is provided as part of the examples, not as a separate driver.
- **Watchdog.** API set for use directly by the application or middleware to manage the Watchdog timer.
- **WiFi.** Driver used by a Wi-Fi device's host driver to exchange commands, data, and events between the host MCU and the wireless network processor. Not intended to be interfaced directly.

5.2 Driver Framework

TI-RTOS drivers have a common framework for static configuration and for a set of APIs that all drivers implement. This section describes that common framework. The driver-specific sections after the framework description provide details about individual implementations.

5.2.1 Static Configuration

The following line in the *.cfg file for a TI-RTOS application causes all TI-RTOS drivers to be available to the application build.

```
var driversConfig = xdc.useModule('ti.drivers.Config');
```

Note that this does not mean that all the drivers will be compiled into the application. To minimize the memory footprint of the application, only driver library code called by the application will be included in the compiled and linked executable.

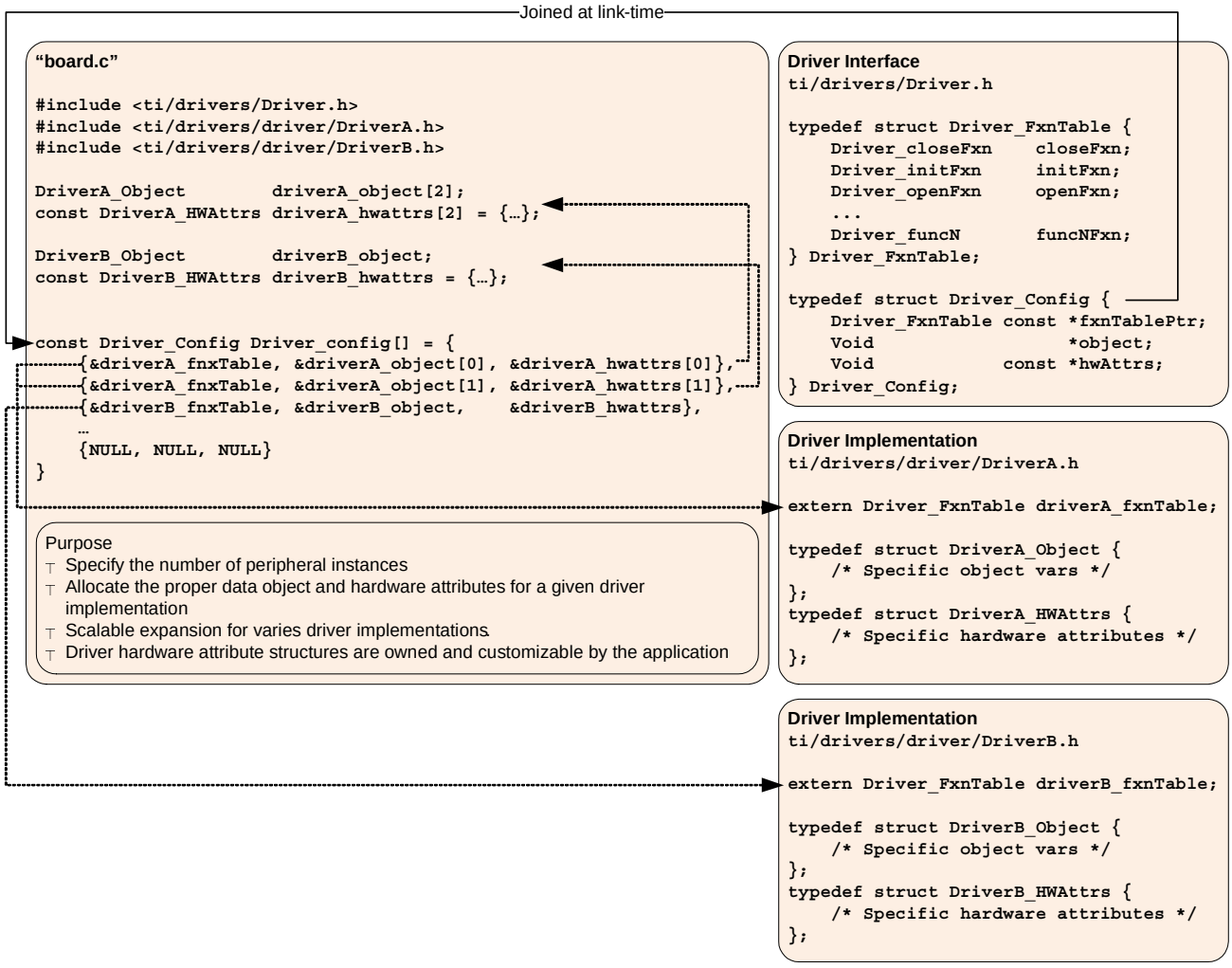
By default, the application is configured to use non-instrumented libraries, which do not process Log events and Asserts. You can select the instrumented libraries by using XGCONF as shown in Section 2.4.2 or by adding the following statement to your application's *.cfg file:

```
driversConfig.libType = driversConfig.LibType_Instrumented;
```

Refer to the individual drivers in this chapter for details about what is logged and which Diags masks are used when instrumentation is enabled.

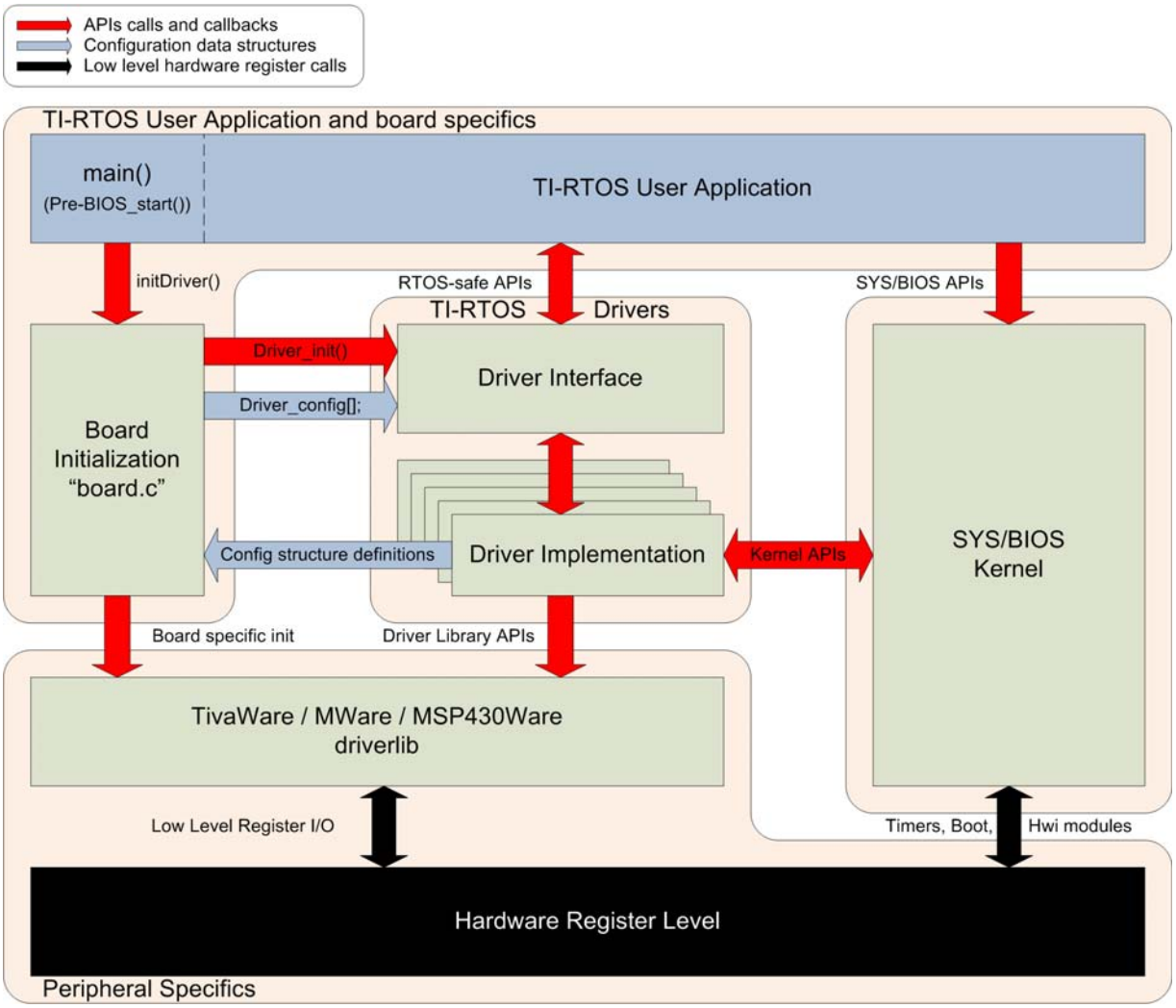
5.2.2 Driver Object Declarations

All TI-RTOS drivers require the application to allocate data storage and define a set of data structures with specific hardware attributes. Drivers are designed in a two-tier hierarchy to facilitate scalable driver additions and enhancements while providing a consistent application programming interface.



This diagram shows the relationship between a driver interface and two driver implementations. The driver interface named "Driver" is configured to operate on two driver implementations: "DriverA" and "DriverB". The driver's Driver_config[] structure contains three instances. The first two instances are of type "DriverA" and the third is of type "DriverB".

Applications interface with a TI-RTOS driver using a top-level *driver interface*. This interface is configured via a set of data structures that specify one or more specific lower-level *driver implementations*. Driver files are located in the `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers` directory. The interfaces that define data structures are in `Driver.h`, while driver implementations are defined in an additional subdirectory, named after the driver interface. For example, the UART driver interface resides in the `UART.h` file, and its driver implementations exist in the `\uart` subdirectory.



5.2.2.1 Driver Interface

Each driver's interface defines a configuration data structure as:

```
typedef struct Driver_Config {
    Driver_FxnTable const *fxnTablePtr;
    Void                  *object;
    Void            const *hwAttrs;
} Driver_Config;
```

(The GPIO driver is an exception. See Section 5.5.1.1.)

The application must declare a NULL-terminated array of Driver_Config elements as `Driver_config[]`.The index argument in a driver's _open() call is used to select the array element of this `Driver_config[]` array where each element corresponds to a peripheral instance. There is no correlation between the index and the peripheral designation (such as UART0 or UART1). For example, it is possible to use UART_config[0] for UART1.

Each individual Driver_Config element must be populated by pointers to a specific driver implementation's Driver_FxnTable, Driver_Object, and Driver_HWAttrs data structures. While the function table is defined by the driver implementation, the implementation-specific data object and hardware attribute structures need to be defined by the application. With this Driver_config[] table, it is possible to use any number of permutations of driver implementations per driver interface; assuming that the device has the same number of peripherals available.

5.2.2.2 Driver Implementations

The application needs to create instances of both the object and hardware attribute structures for every peripheral used with a given driver implementation. Instances of data objects are used to store driver variables on a per peripheral basis and should be accessed exclusively by the driver. Hardware attribute structures are used to specify implementation-specific constants such as peripheral base addresses, interrupt vectors, GPIO pins, and more. Field definitions for these hardware attributes are available in the driver's Doxygen documentation.

All TI-RTOS examples use a <*board*>.c file that contains necessary data object and hardware structure instances, similar to the following:

```
static DriverA_Object driverAObject;

const DriverA_HWAttrs driverAHWAttrs = {
    type field0;
    type field1;
    ...
    type fieldn;
};
```

These structures should be used as a reference when moving from a development board to a custom printed circuit board. The following is an example that integrates a UART driver implementation into the UART driver interface:

```
/* UART objects */
UARTTiva_Object uartTivaObjects[EKS_LM4F232_UARTCOUNT];

/* UART configuration structure */
const UARTTiva_HWAttrs uartTivaHWAttrs [EKS_LM4F232_UARTCOUNT] = {
    {UART0_BASE, INT_UART0},    /* UART0 */
};

const UART_Config UART_config[] = {
    {&UARTTiva_fxnTable, &uartTivaObjects[0], &uartTivaHWAttrs[0]},
    {NULL, NULL, NULL}
};
```

5.2.3 Dynamic Configuration and Common APIs

TI-RTOS drivers all implement the following APIs (with the exception of the GPIO driver*).

- `Void Driver_init(Void)`
  - Initializes the driver. Must be called only once and before any calls to the other driver APIs. Generally, this is done before SYS/BIOS is started.
  - The board files in the examples call this function for you.
- `Void Driver_Params_init(Driver_Params *params)`
  - Initializes the driver's parameter structure to default values. All drivers, with the exception of GPIO, implement the Params structure. The Params structure is empty for some drivers.
- `Driver_Handle Driver_open(UInt index, Driver_Params *params)`
  - Opens the driver instance specified by the index with the params provided.
  - If the params field is NULL, the driver uses default values. See specific drivers for their defaults.
  - Returns a handle that will be used by other driver APIs and should be saved.
  - If there is an error opening the driver or the driver has already been opened, *Driver_open*() returns NULL.
- `Void Driver_close(Driver_Handle handle)`
  - Closes the driver instance that was opened, specified by the driver handle returned during open.
  - Closes the driver immediately, without checking if the driver is currently in use. It is up to the application to determine when to call *Driver*_close() and to ensure it doesn't disrupt on-going driver activity.
  - The Watchdog driver does not have a close() function, because the watchdog timer cannot be disabled once it has been enabled.

* The GPIO driver implements only GPIO_init() to avoid complicating the driver. See Section 5.5 for information on using the GPIO driver.

5.2.4 TI-RTOS Driver Implementations for Concerto Devices

If you are modifying the <*board*>.c file for an application, you will see types and data structures that are defined by the lower-level driver implementations. These implementations are provided in the driver directories. For example, the lower-level implementation for the I2C driver is in the `I2CTiva.c` and `I2CTiva.h` files in the `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers\i2c` directory.

The lower-level driver implementations for the TI-RTOS drivers on the M3 portion of Concerto devices are as follows. (The *.c file is listed, but the *.h file contains important type definitions.)

- **EMAC:** EMACTiva.c
- **GPIO:** GPIOTiva.c
- **I2C:** I2CTiva.c
- **SDSPI:** SDSPITiva.c
- **SPI:** SPITivaDMA.c
- **UART:** UARTTiva.c
- **USBMSCHFatFs:** USBMSCHFatFsTiva.c
- **Watchdog:** WatchdogTiva.c

5.2.5 TI-RTOS Driver Implementations for TivaC Devices

If you are modifying the <*board*>.c file for an application, you will see types and data structures that are defined by the lower-level driver implementations. These implementations are provided in the driver directories. For example, the lower-level implementation for the I2C driver is in the `I2CTiva.c` and `I2CTiva.h` files in the `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers\i2c` directory.

The lower-level driver implementations for the TI-RTOS drivers on Tiva devices are as follows. (The *.c file is listed, but the *.h file contains important type definitions.)

- **EMAC:** EMACTiva.c
- **GPIO:** GPIOTiva.c
- **I2C:** I2CTiva.c
- **PWM:** PWMTiva.c
- **SDSPI:** SDSPITiva.c
- **SPI:** SPITivaDMA.c
- **UART:** UARTTiva.c
- **UART DMA:** UARTTivaDMA.c
- **USBMSCHFatFs:** USBMSCHFatFsTiva.c
- **Watchdog:** WatchdogTiva.c
- **WiFi:** WiFiCC3100.c

5.2.6 TI-RTOS Driver Implementations for CC26xx Devices

For CC26xx devices, the <board.c> and <board.h> files are replaced with literal "Board.c" and "Board.h" files. The purpose of these files is to #include a common device-specific "Board.c" and "Board.h" located in <ti/board>. The board files located in `<tirtos_install>\products\tidrivers_<version>\packages\ti\boards` are shared with other CC2650 examples, so use caution if you choose to modify these files.

The data structures in these board files are defined by lower-level driver implementations. These implementations are provided in the driver directories. For example, the lower-level implementation for the I2C driver is in the I2CCC26XX.c and I2CCC26XX.h files in the `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers\i2c` directory.

The lower-level driver implementations for the TI-RTOS drivers on CC26xx devices are as follows. (The *.c file is listed, but the *.h file contains important type definitions.)

- **Crypto:** CryptoCC26XX.c
- **I2C:** I2CCC26XX.c
- **PIN:** PINCC26XX.c
- **SPI:** SPICC26XXDMA.c
- **UART:** UARTCC26XX.c
- **Watchdog:** WatchdogCC26XX.c

5.2.7 TI-RTOS Driver Implementations for CC3200 Devices

If you are modifying the <board>.c file for an application, you will see types and data structures that are defined by the lower-level driver implementations. These implementations are provided in the driver directories. For example, the lower-level implementation for the I^2C driver is in the I2CCC3200.c and I2CCC3200.h files in the `<tirtos_install>\products\tidrivers_<version>\packages\ti\drivers\i2c` directory.

The lower-level driver implementations for the TI-RTOS drivers on CC3200 devices are as follows. (The *.c file is listed, but the *.h file contains important type definitions.)

- **Camera:** CameraCC3200DMA.c.
- **GPIO:** GPIOCC3200.c.
- **I2C:** I2CCC3200.c
- **I2S:** I2SCC3200DMA.c
- **Power:** PowerCC3200.c
- **PWM:** PWMTimerCC3200.c
- **SDSPI:** SDSPICC3200.c
- **SPI:** SPICC3200DMA.c
- **UART:** UARTCC3200.c
- **UART DMA:** UARTCC3200DMA.c
- **Watchdog:** WatchdogCC3200.c

5.2.8 TI-RTOS Driver Hwis for MSP43x Devices

MSP432 devices use the Hwi dispatcher provided by the TI-RTOS Kernel. The dispatcher creates the necessary interrupts at run-time, so no special steps are required to support MSP432 interrupts.

However, for MSP430 devices, the TI-RTOS Kernel does not use a Hwi dispatcher to allow for run-time creation of interrupts. For this reason, MSP430 users must create Hwis statically in the application's *.cfg file. Follow the steps below to configure the appropriate Hwis for applications that use TI-RTOS drivers.

1. Identify the TI-RTOS drivers and implementations that you want to add into your application. TI-RTOS has a set of MSP43x driver implementations to support the USCI and EUSCI peripherals. Some TI-RTOS drivers (for example, WiFi) have dependencies on other TI-RTOS drivers.
2. Use Table 5-1 for MSP430F5xxx devices and Table 5-2 for MSP430FR5xxx devices to determine whether these drivers require any Hwi interrupts to be created.

Table 5-1 Hwi functions required for TI-RTOS driver ISRs (USCI on MSP430F5xxx)

TI-RTOS Driver	MSP430 Driver Implementations	Interrupt Service Routine Hwi Function Name
GPIO	GPIOMSP430	GPIO_hwiIntFxn
I^2C	I2CUSCIB	I2CUSCIB_hwiIntFxn
SDSPI	SDSPIUSCIA, SDSPIUSCIB	N/A. This driver is polling based
SPI	SPIUSCIADMA, SPIUSCIBDMA	A DMA interrupt function defined by the user must call the SPI driver's SPI_serviceISR function.
UART	UARTUSCIA	UARTUSCIA_hwiIntFxn
Watchdog	WatchdogMSP430	N/A. This driver only generates a reset signal
WiFi	WiFiCC3100	The WiFiCC3100_hostIntHandler function is tied to a GPIO interrupt. A user DMA interrupt function calls SPI_serviceISR. (WiFi driver uses SPI driver.)

Table 5-2 Hwi functions required for TI-RTOS driver ISRs (EUSCI on MSP430FR5xxx/FR6xxx)

TI-RTOS Driver	MSP430 Driver Implementations	Interrupt Service Routine Hwi Function Name
GPIO	GPIOMSP430	GPIO_hwiIntFxn
I^2C	I2CEUSCIB	I2CEUSCIB_hwiIntFxn
SDSPI	SDSPIEUSCIA, SDSPIEUSCIB	N/A. This driver is polling based
SPI	SPIUSCIADMA, SPIUSCIBDMA	A DMA interrupt function defined by the user must call the SPI driver's SPI_serviceISR function.
UART	UARTEUSCIA	UARTEUSCIA_hwiIntFxn
Watchdog	WatchdogMSP430	N/A. This driver only generates a reset signal
WiFi	WiFiCC3100	The WiFiCC3100_hostIntHandler function is tied to a GPIO interrupt. A user DMA interrupt function calls SPI_serviceISR. (WiFi driver uses SPI driver.)

3. If the TI-RTOS driver is interrupt driven, find the peripheral's base address for every driver implementation entry in the HWAttrs data structure of the driver's *Driver*_config[] array.

   For example, I2C_Config[0] has its HWAttrs data structure configured to USCI_B0_BASE. Similarly, I2C_Config[1] has its HWAttrs data structure configured to use USBI_B1_BASE.

```
/* I2C objects */
I2CUSCIB_Object i2cUSCIBObjects[MSP_EXP430F5529LP_I2CCOUNT];

/* I2C configuration structure */
const I2CUSCIB_HWAttrs i2cUSCIBHWAttrs[MSP_EXP430F5529LP_I2CCOUNT] = {
    {
        USCI_B0_BASE,
        USCI_B_I2C_CLOCKSOURCE_SMCLK
    },
    {
        USCI_B1_BASE,
        USCI_B_I2C_CLOCKSOURCE_SMCLK
    }
};

const I2C_Config I2C_config[] = {
    {
        &I2CUSCIB_fxnTable,
        &i2cUSCIBObjects[0],
        &i2cUSCIBHWAttrs[0]
    },
    {
        &I2CUSCIB_fxnTable,
        &i2cUSCIBObjects[1],
        &i2cUSCIBHWAttrs[1]
    },
    {NULL, NULL, NULL}
};
```

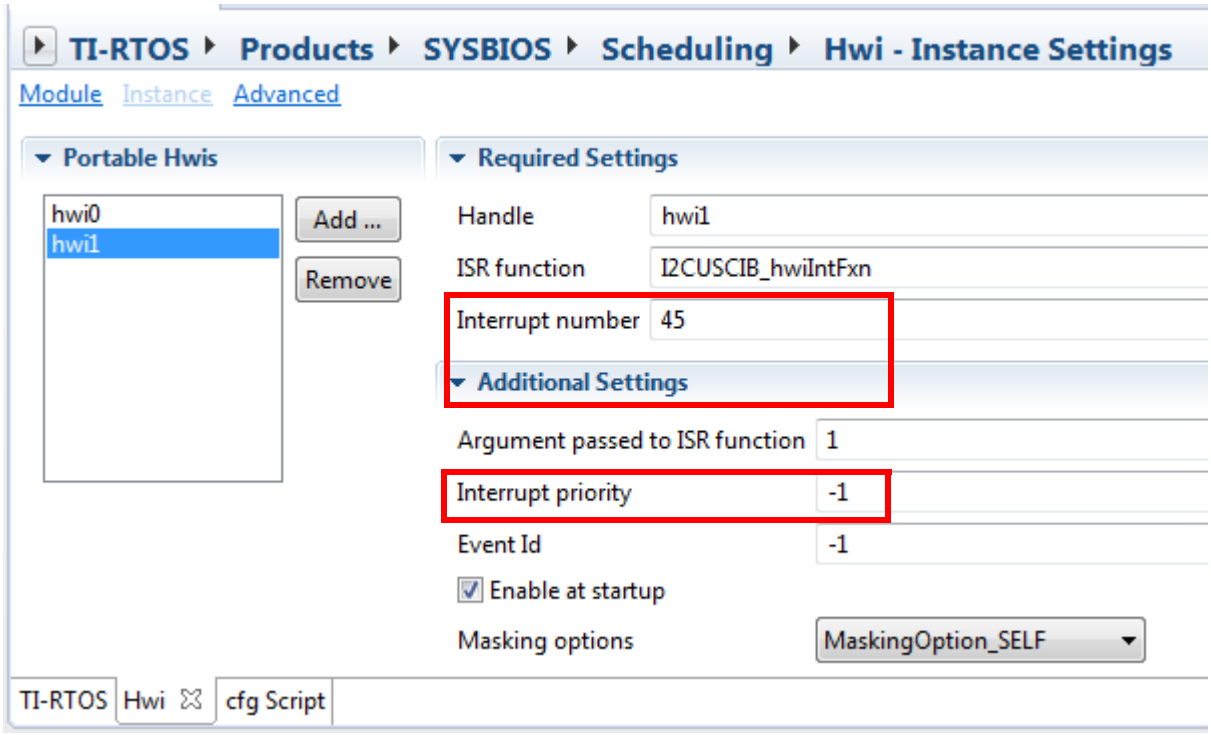
4. Find the associated interrupt vector number for each peripheral at the specified base address. For example, on the MSP430F5529, the USBI_B0 interrupt vector is 55 and the USCI_B1 interrupt vector is 45. The interrupt vector number is set in the device's main *.h header file. For MSP430F5529, the file would be msp430f5529.h.
   - In CCS, this file is found in: <*CCS_install*>/ccsv6/ccs_base/msp430/include.
   - In IAR, this file is found in: <*IAR_install*>/430/inc.
5. Create Hwi objects for each entry in the *Driver*_config[] array using the information obtained in the previous steps. Map the information to the Hwi in the following manner:
   - **Hwi (ISR) function.** Use the Hwi function name from Table 5-1.
   - **Interrupt vector number.** Use the vector number from the device's main *.h header file.
   - **Argument passed to ISR function.** For the GPIO driver, the argument must be the GPIO port number corresponding to the interrupt. For all other drivers, use the index number into the *Driver*_config[] array.

Hwi objects can be created using the graphical user interface or by manually adding it to the project's *.cfg file. Both of the following examples configure two Hwi objects that run the I2CUSCIB_hwiIntFxn function required by the I^2C driver with the interrupt vectors for USBI_B0 and USBI_B1.

This example shows the statements in the *.cfg file:

```
/* ================ Hwi configuration ================ */
/*
 * All Hwis for MSP430 must be created statically; including Hwis for TI-RTOS
 * drivers
 */
var hwi0Params = new Hwi.Params();
hwi0Params.instance.name = "hwi0";
hwi0Params.arg = 0;
Program.global.hwi0 = Hwi.create(55, "&I2CUSCIB_hwiIntFxn", hwi0Params);
var hwi1Params = new Hwi.Params();
hwi1Params.instance.name = "hwi1";
hwi1Params.arg = 1;
Program.global.hwi1 = Hwi.create(45, "&I2CUSCIB_hwiIntFxn", hwi1Params);
```

This example shows how to configure the Hwi objects graphically with the XGCONF Configuration Editor:



5.3 Camera Driver

The Camera driver is used to retrieve the data being transferred by the Camera sensor. This driver provides an API for capturing the image from the Camera sensor. The camera sensor control and implementation are the responsibility of the application using the interface.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.3.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.3.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the Camera driver requires the application to initialize board-specific settings and provide the Camera driver with the Camera_config structure.

5.3.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initCamera() function that initializes the board-specific Camera peripheral settings. This function also calls Camera_init() to initialize the Camera driver.

5.3.2.2 Camera_Params Structure

The Camera_Params structure may be used to override the default settings for an Camera instance you are creating. The params in the structure must be set before calling Camera_open(). The structure has the following fields:

```
typedef struct Camera_Params {
    Camera_CaptureMode         captureMode;     /* blocking or callback mode */
    uint32_t                   outputClock;     /* to set divider */
    Camera_HSyncPolarity       hsyncPolarity;   /* polarity of horizontal Sync */
    Camera_VSyncPolarity       vsyncPolarity;   /* polarity of vertical Sync */
    Camera_PixelClkConfig      pixelClkConfig;  /* rising edge or falling edge */
    Camera_ByteOrder           byteOrder;       /* order of bytes captured */
    Camera_IfSynchoronisation  interfaceSync;   /* camera-sensor synchronisation */
    Camera_StopCaptureConfig   stopConfig;      /* action when capture stops */
    Camera_StartCaptureConfig  startConfig;     /* action when capture starts */
    uint32_t                   captureTimeout;  /* timeout length for capture */
    void                       *custom;         /* custom target-specific option */
```

5.3.3 Camera Modes

The Camera operation mode determines whether transmit and/or receive modes are enabled. The mode is specified with one of the following constants:

- *Camera_MODE_BLOCKING:* Uses a semaphore to block while data is being sent. Context of the call must be a Task.
- *Camera_MODE_CALLBACK:* Non-blocking call, which will return immediately. When the capture by the interrupt, is finished the configured callback function is called.

Other enumerated types are available for other Camera driver parameters.

5.3.4 APIs

In order to use the Camera module APIs, the Camera.h header file should be included in an application as follows:

```
#include <ti/drivers/Camera.h>
```

The following are the Camera APIs:

- **Camera_init()** initializes the Camera module.
- **Camera_Params_init()** initializes an Camera_Params data structure.
- **Camera_open()** initializes a given Camera instance.
- **Camera_close()** deinitializes a given Camera instance.
- **Camera_control()** performs implementation-specific features on a given Camera peripheral.
- **Camera_capture()** handles the capture of a frame.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.3.4.1 Opening the Camera driver

To open a Camera driver instance, initialize a Camera_Params object and specify the desired parameters.

```
Camera_Handle      handle;
Camera_Params      params;

Camera_Params_init(&params);
params.captureMode       =  Camera_MODE_BLOCKING;
/* Change any other params as needed */

handle = Camera_open(someCamera_configIndexValue, &params);
if (!handle) {
    System_printf("Camera did not open");
}
```

5.3.4.2 Writing Data

The following example calls Camera_capture() to cause a picture to be taken by the camera and the photo to be placed in a buffer.

```
unsigned char captureBuffer[1920];

ret = Camera_capture(handle, &captureBuffer, sizeof(captureBuffer));
```

5.3.5 Examples

See the SimpleLink Wi-Fi CC3200 Software Development Kit (SDK) for examples that use this driver.

5.4 EMAC Driver

This is the Ethernet driver used by the networking stack (NDK).

5.4.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.4.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the EMAC driver requires the application to initialize board-specific portions of the EMAC and provide the EMAC driver with the EMAC_config structure.

5.4.2.1 Board-Specific Configuration

The <board>.c files contain a <*board*>_initEMAC() function that must be called to initialize the board-specific EMAC peripheral settings. This function also calls the EMAC_init() to initialize the EMAC driver.

5.4.2.2 EMAC_config Structure

The <*board*>.c file also declare the EMAC_config structure. This structure must be provided to the EMAC driver. It must be initialized before the EMAC_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.4.3 APIs

To use the EMAC module APIs, the EMAC header file should be included in an application as follows:

```
#include <ti/drivers/EMAC.h>
```

The following EMAC API is provided:

- **EMAC_init()** sets up the EMAC driver. This function must be called before the NDK stack thread is started.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

See the NDK documentation for information about NDK APIs that can be used if the EMAC driver is enabled and initialized.

5.4.4 Usage

The EMAC driver is designed to be used by the NDK. The only function that must be called is the EMAC_init() function. This function must be called before BIOS_start() is called to ensure that the driver is initialized before the NDK starts.

5.4.5 Instrumentation

The EMAC driver logs the following actions using the Log_print() APIs provided by SYS/BIOS.

- EMAC driver setup success or failure.
- EMAC started or stopped.
- EMAC failed to receive or transmit a packet.
- EMAC successfully sent or received a packet.
- No packet could be allocated.
- Packet is too small for the received buffer.

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information.

5.5 GPIO Driver

The GPIO module allows you to manage General Purpose I/O pins via simple and portable APIs. GPIO pin behavior is usually configured statically, but can also be configured or reconfigured at runtime.

The application is required to supply a device specific GPIO*xxx*_Config structure to the module, where *xxx* is the name of the target family. This structure communicates to the GPIO module which GPIO pins are used by the application and how they are to be configured. (See the GPIO_PinConfig array description in Section 5.5.1.2.)

The application is required to call GPIO_init(). This function initializes all the GPIO pins defined in the GPIO_PinConfig table to the configurations specified. Once initialization is complete, the other APIs can be used to access the pins.

Because of its simplicity, the GPIO driver does not follow the model of other TI-RTOS drivers in which a driver application interface has separate device-specific implementations. This difference is most apparent in the GPIO*xxx*_Config structure (described in more detail in Section 5.5.1.1), which does not require you to specify a particular function table or object.

5.5.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.5.1.1 GPIO_Config Structure

The `<board>.c` file declares a board-specific GPIO*xxx*_Config structure. This structure is used internally by the GPIO Driver and must be provided by the user. Currently the GPIO*xxx*_Config structure usually consists of pointers to two arrays—an array of GPIO_PinConfig elements and an array of GPIO_Callback elements—their respective number of elements, and an interrupt priority field used to configure the interrupts that will be used for input pins with callbacks.

Below is an example of a typical GPIO*xxx*_Config structure, in this case specific to Tiva boards:

```
typedef struct GPIOTiva_Config {
    /*! Pointer to the board's GPIO_PinConfig array */
    GPIO_PinConfig  *pinConfigs;

    /*! Pointer to the board's GPIO_CallbackFxn array */
    GPIO_CallbackFxn  *callbacks;

    /*! number of GPIO_PinConfigs defined */
    uint32_t numberOfPinConfigs;

    /*! number of GPIO_Callbacks defined */
    uint32_t numberOfCallbacks;

    /*! GPIO interrupt priority. Setting (~0) configures lowest priority */
    uint32_t intPriority;
} GPIOTiva_Config;
```

A brief discussion of several fields in this structure follows. See the Doxygen help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.5.1.2 GPIO_pinConfig Array

The elements in this array define the configuration and device-specific identities for each of the physical GPIO pins used by the application. A pin is referenced in the application by its corresponding index in this array.

The pin type (that is, INPUT/OUTPUT), its initial state (that is OUTPUT_HIGH or LOW), and interrupt behavior (RISING/FALLING edge, etc.) are configured in each element of this array.

For example, this GPIO_PinConfig array for Tiva is provided in the EK_TM4C1294XL.h file.

```
GPIO_PinConfig gpioPinConfigs[] = {
    /* Input pins */
    /* EK_TM4C1294XL_USR_SW1 */
    GPIOTiva_PJ_0 | GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_RISING,
    /* EK_TM4C1294XL_USR_SW2 */
    GPIOTiva_PJ_1 | GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_RISING,

    /* Output pins */
    /* EK_TM4C1294XL_USR_D1 */
    GPIOTiva_PN_1 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_HIGH | GPIO_CFG_OUT_LOW,
    /* EK_TM4C1294XL_USR_D2 */
    GPIOTiva_PN_0 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_HIGH | GPIO_CFG_OUT_LOW,
};
```

5.5.1.3 GPIO_callbackFxn Array

Each element in this array is a callback function pointer for each of the GPIO pins configured to interrupt the device. The indexes for these array elements correspond to the pins defined in the GPIO_pinConfig array. These function pointers can be defined statically by referencing the callback function name in the array element, or dynamically, by setting the array element to NULL and using GPIO_setCallback() at runtime to plug the callback entry.

For example, this GPIO_callbackFxn array for Tiva is provided in the EK_TM4C1294XL.h file.

```
GPIO_CallbackFxn gpioCallbackFunctions[] = {
    NULL,  /* EK_TM4C1294XL_USR_SW1 */
    NULL   /* EK_TM4C1294XL_USR_SW2 */
};
```

Pins not used for interrupts can be omitted from the callback array to reduce memory usage (if they are placed at end of GPIO_pinConfig array).

The callback function syntax should match the following:

```
void (*GPIO_CallbackFxn)(unsigned int index);
```

The index parameter is the same index that was passed to GPIO_setCallback(). This allows the same callback function to be used for multiple GPIO interrupts, by using the index to identify the GPIO that caused the interrupt.

5.5.1.4 intPriority

(Not used for MSP430.)

This parameter defines the priority of the interrupt associated with the pins. Values for this parameter are device-specific. You should be well-acquainted with the interrupt controller used in your device before setting this parameter to a non-default value. The sentinel value of ~(0) (the default value) is used to indicate that the lowest possible priority should be used.

5.5.1.5 MSP430 GPIO Interrupts

As with other MSP430 drivers, the GPIO driver requires port interrupts be statically created in the configuration file. However the GPIO driver also requires the port number to be set as a Hwi argument (see Section 5.2.8).

5.5.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the GPIO driver requires the application to initialize board-specific portions of the GPIO and provide the GPIO driver with a board-specific GPIO*xxx*_config structure.

5.5.2.1 Board-Specific Configuration

The `<board>.c` files contain a <*board*>_initGPIO() function that must be called at runtime—usually within main()—to initialize board-specific GPIO peripheral settings. Unlike other drivers, there in no board-specific initialization performed by this function. It simply calls GPIO_init(), which initializes the GPIO driver and configures all the pins as prescribed by the GPIO*xxx*_Config structure.

5.5.3 APIs

In order to use the GPIO module APIs, the GPIO header file should be included in an application as follows:

```
#include <ti/drivers/GPIO.h>
```

The following are the GPIO APIs:

- **GPIO_init()** sets up the configured GPIO pins.
- **GPIO_read()** gets the current state of the specified GPIO input pin.
- **GPIO_write()** sets the state of the specified GPIO pin to on or off.
- **GPIO_toggle()** toggles the state of the specified GPIO pin.
- **GPIO_setCallback()** dynamically binds a callback function to the specified GPIO input pin.
- **GPIO_setConfig()** dynamically configures the specified GPIO input pin.
- **GPIO_clearInt()** clears the interrupt flag for the specified GPIO pin.
- **GPIO_disableInt()** disables interrupts on the specified GPIO pin.
- **GPIO_enableInt()** enables interrupts on the specified GPIO pin.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.5.4 Usage

Once the GPIO_init() function has been called, the other GPIO APIs functions can be called. For example, LEDs can be switched on as follows:

```
GPIO_write(Board_LED0, Board_LED_ON);
GPIO_write(Board_LED1, Board_LED_ON);
GPIO_write(Board_LED2, Board_LED_ON);
```

For GPIO interrupts, once the GPIO_setCallback() function has been called to install a callback for a pin that pin's interrupt can be enabled as shown below:

```
/* Install callback and enable interrupts */
GPIO_setCallback(Board_BUTTON0, gpioButtonFxn0);
GPIO_setCallback(Board_BUTTON1, gpioButtonFxn1);

GPIO_enableInt(Board_BUTTON0);
GPIO_enableInt(Board_BUTTON1);
```

5.5.5 Instrumentation

The GPIO driver logs the following actions using the Log_print() APIs provided by SYS/BIOS:

- GPIO pin read.
- GPIO pin toggled.
- GPIO pin written to.
- GPIO hardware interrupt creation failure.
- GPIO interrupt flag cleared.
- GPIO interrupt enabled.
- GPIO interrupt disabled.

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information.

5.5.6 Examples

All the TI-RTOS examples use the GPIO driver. The GPIO Interrupt example demonstrates interrupt usage. The GPIO_init() function is called in the board-specific file (for example, CC3200_LAUNCHXL.c). A filled in GPIO_Config structure is provided in the same file.

5.6 I²C Driver

This section assumes that you have background knowledge and understanding about how the I²C protocol operates. For the full I²C specifications and user manual (UM10204), see the NXP Semiconductors website.

The I²C driver has been designed to operate as a single I²C master by performing I²C transactions between the target and I²C slave peripherals. The I²C driver does not support I²C slave mode at this time. I²C is a communication protocol—the specifications define how data transactions are to occur via the I²C bus. The specifications do not define how data is to be formatted or handled, allowing for flexible implementations across different peripheral vendors. As a result, the I²C handles only the exchange of data (or transactions) between master and slaves. It is the left to the application to interpret and manipulate the contents of each specific I²C peripheral.

The I²C driver has been designed to operate in a RTOS environment such as SYS/BIOS. It protects its transactions with OS primitives supplied by SYS/BIOS.

5.6.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.6.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the I²C driver requires the application to initialize board-specific portions of the I²C and provide the I²C driver with the I2C_config structure.

5.6.2.1 Board-Specific Configuration

The <board>.c files contain a <*board*>_initI2C() function that must be called to initialize the board-specific I²C peripheral settings. This function also calls the I2C_init() to initialize the I²C driver.

5.6.2.2 I2C_config Structure

The <*board*>.c file also declare the I2C_config structure. This structure must be provided to the I²C driver. It must be initialized before the I2C_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.6.3 APIs

In order to use the I²C module APIs, the I2C.h header file should be included in an application as follows:

```
#include <ti/drivers/I2C.h>
```

The following are the I²C APIs:

- **I2C_init()** initializes the I²C module.
- **I2C_Params_init()** initializes an I2C_Params data structure. It defaults to Blocking mode.
- **I2C_open()** initializes a given I²C peripheral.
- **I2C_close()** deinitializes a given I²C peripheral.

- **I2C_transfer()** handles the I²C transfer for SYS/BIOS.

The I2C_transfer() API can be called only from a Task context. It requires an I2C_Tramsaction structure that specifies the location of the write and read buffer, the number of bytes to be processed, and the I²C slave address of the device.

See the help in <*tirtos_install*>\products\tidrivers_<*version*>\docs\html\index.html for details.

5.6.4 Usage

The application needs to supply the following structures in order to set up the framework for the driver:

- **I2C_Params** specifies the transfer mode and any callback function to be used. See Section 5.6.4.1.
- **I2C_Transaction** specifies details about a transfer to be performed. See Section 5.6.4.2.
- **I2C_Callback** specifies a function to be used if you are using callback mode. See Section 5.6.4.3.

5.6.4.1 I²C Parameters

The I2C_Params structure is used with the I2C_open() function call. If the transferMode is set to I2C_MODE_BLOCKING, the transferCallback argument is ignored. If transferMode is set to I2C_MODE_CALLBACK, a user-defined callback function must be supplied.

```
typedef struct I2C_Params {
    I2C_TransferMode transferMode;          /* Blocking or Callback mode */
    I2C_CallbackFxn  transferCallbackFxn;   /* Callback function pointer */
} I2C_Params;
```

5.6.4.2 I²C Transaction

The I2C_Transaction structure is used to specify what type of I2C_transfer needs to take place.

```
typedef struct I2C_Transaction {
    UChar *writeBuf;        /* Pointer to a buffer to be written */
    UInt   writeCount;      /* Number of bytes to be written */

    UChar *readBuf;         /* Pointer to a buffer to be read */
    UInt   readCount;       /* Number of bytes to be read */

    UChar  slaveAddress;    /* Address of the I2C slave device */

    UArg   arg;             /* User definable argument to the callback function */
    Ptr    nextPtr;         /* Driver uses this for queuing in I2C_MODE_CALLBACK */
} I2C_Transaction;
```

slaveAddress specifies the I²C slave address the I²C will communicate with. If writeCount is nonzero, I2C_transfer writes writeCount bytes from the buffer pointed by writeBuf. If readCount is nonzero, I2C_transfer reads readCount bytes into the buffer pointed by readBuf. If both writeCount and readCount are non-zero, the write operation always runs before the read operation.

The optional arg variable can only be used when the I²C driver has been opened in Callback mode. This variable is used to pass a user-defined value into the user-defined callback function.

nextPtr is used to maintain a linked-list of I2C_Transactions when the I²C driver has been opened in Callback mode. It must never be modified by the user application.

5.6.4.3 I²C Callback Function Prototype

This typedef defines the function prototype for the I²C driver's callback function for Callback mode. When the I²C driver calls this function, it supplies the associated I2C_Handle, a pointer to the I2C_Transaction that just completed, and a Boolean value indicating the transfer result. The transfer result is the same as from the I2C_transfer() when operating in Blocking mode.

```
typedef Void (*I2C_Callback)(I2C_Handle, I2C_Transaction *, Bool);
```

5.6.5 I²C Modes

The I²C driver supports two modes of operation, *blocking* and *callback* modes. The mode is determined when the I²C driver is opened using the I2C_Params data structure. If no I2C_Params structure is specified, the I²C driver defaults to blocking mode. Once opened, the only way to change the operation mode is to close and re-open the I²C instance with the new mode.

5.6.5.1 Opening in Blocking Mode

By default, the I²C driver operates in blocking mode. In blocking mode, a Task's code execution is blocked until an I²C transaction has completed. This ensures that only one I²C transaction operates at a given time. Other tasks requesting I²C transactions while a transaction is currently taking place are also placed into a blocked state and are executed in the order in which they were received.

```
I2C_Handle i2c;
UInt peripheralNum = 0;     /* Such as I2C0 */
I2C_Params i2cParams;

I2C_Params_init(&i2cParams);
i2cParams.transferMode = I2C_MODE_BLOCKING;
i2cParams.transferCallbackFxn = NULL;

i2c = I2C_open(peripheralNum, &i2cParams);
if (i2c == NULL) {
    /* Error opening I2C */
}
```

If no I2C_Params structure is passed to I2C_open(), default values are used. If the open call is successful, it returns a non-NULL value.

5.6.5.2 Opening in Callback Mode

In callback mode, an I²C transaction functions asynchronously, which means that it does not block a Task's code execution. After an I²C transaction has been completed, the I²C driver calls a user-provided hook function. If an I²C transaction is requested while a transaction is currently taking place, the new transaction is placed onto a queue to be processed in the order in which it was received.

```
I2C_Handle i2c;
UInt peripheralNum = 0;     /* Such as I2C0 */
I2C_Params i2cParams;

I2C_Params_init(&i2cParams);
i2cParams.transferMode = I2C_MODE_CALLBACK;
i2cParams.transferCallbackFxn = UserCallbackFxn;

i2c = I2C_open(peripheralNum, &i2cParams);
if (i2c == NULL) {
    /* Error opening I2C */
}
```

5.6.5.3 Specifying an I²C Bus Frequency

The I²C controller's bus frequency is determined as part the I2C_Params data structure and is set when the application calls I2C_open(). The standard I²C bus frequencies are 100 kHz and 400 kHz, with 100 kHz being the default.

```
I2C_Handle i2c;
UInt peripheralNum = 0; /* Such as I2C0 */
I2C_Params i2cParams;

I2C_Params_init(&i2cParams); /* Default is I2C_100kHz */
i2cParams.bitRate = I2C_400kHz;
i2c = I2C_open(peripheralNum, &i2cParams);
if (i2c == NULL) {
    /* Error Initializing I2C */
}
```

5.6.6 I²C Transactions

I²C can perform three types of transactions: Write, Read, and Write/Read. All I²C transactions are atomic operations with the slave peripheral. The I2C_transfer() function determines how many bytes need to be written and/or read to the designated I²C peripheral by reading the contents of an I2C_Transaction data structure.

The basic I2C_Transaction arguments include the slave peripheral's I²C address, pointers to write and read buffers, and their associated byte counters. The I²C driver always writes the contents from the write buffer before it starts reading the specified number of bytes into the read buffer. If no data needs to be written or read, simply set the corresponding counter(s) to 0.

5.6.6.1 Write Transaction (Blocking Mode)

As the name implies, an I²C write transaction writes data to a specified I²C slave peripheral. The following code writes three bytes of data to a peripheral with a 7-bit slave address of 0x50.

```
I2C_Transaction   i2cTransaction;
UChar             writeBuffer[3];
UChar             readBuffer[2];
Bool              transferOK;

i2cTransaction.slaveAddress = 0x50;     /* 7-bit peripheral slave address */
i2cTransaction.writeBuf = writeBuffer;  /* Buffer to be written */
i2cTransaction.writeCount = 3;          /* Number of bytes to be written */
i2cTransaction.readBuf = NULL;          /* Buffer to be read */
i2cTransaction.readCount = 0;           /* Number of bytes to be read */

transferOK = I2C_transfer(i2c, &i2cTransaction); /* Perform I2C transfer */
if (!transferOK) {
    /* I2C bus fault */
}
```

5.6.6.2 Read Transaction (Blocking Mode)

A read transaction reads data from a specified I^{2}C slave peripheral. The following code reads two bytes of data from a peripheral with a 7-bit slave address of 0x50.

```
I2C_Transaction   i2cTransaction;
UChar             writeBuffer[3];
UChar             readBuffer[2];
Bool              transferOK;

i2cTransaction.slaveAddress = 0x50;     /* 7-bit peripheral slave address */
i2cTransaction.writeBuf = NULL;         /* Buffer to be written */
i2cTransaction.writeCount = 0;          /* Number of bytes to be written */
i2cTransaction.readBuf = readBuffer;    /* Buffer to be read */
i2cTransaction.readCount = 2;           /* Number of bytes to be read */

transferOK = I2C_transfer(i2c, &i2cTransaction); /* Perform I2C transfer */
if (!transferOK) {
    /* I2C bus fault */
}
```

5.6.6.3 Write/Read Transaction (Blocking Mode)

A write/read transaction first writes data to the specified peripheral. It then writes an I^{2}C restart bit, which starts a read operation from the peripheral. This transaction is useful if the I^{2}C peripheral has a pointer register that needs to be adjusted prior to reading from referenced data registers. The following code writes three bytes of data, sends a restart bit, and reads two bytes of data from a peripheral with the slave address of 0x50.

```
I2C_Transaction   i2cTransaction;
UChar             writeBuffer[3];
UChar             readBuffer[2];
Bool              transferOK;

i2cTransaction.slaveAddress = 0x50;     /* 7-bit peripheral slave address */
i2cTransaction.writeBuf = writeBuffer;  /* Buffer to be written */
i2cTransaction.writeCount = 3;          /* Number of bytes to be written */
i2cTransaction.readBuf = readBuffer;    /* Buffer to be read */
i2cTransaction.readCount = 2;           /* Number of bytes to be read */

transferOK = I2C_transfer(i2c, &i2cTransaction); /* Perform I2C transfer */
if (!transferOK) {
    /* I2C bus fault */
}
```

5.6.6.4 Write/Read Transaction (Callback Mode)

In callback mode, I^{2}C transfers are non-blocking transactions. After an I^{2}C transaction has completed, the I^{2}C interrupt routine calls the user-provided callback function, which was passed in when the I^{2}C driver was opened.

In addition to the standard I2C_Transaction arguments, an additional user-definable argument can be passed through to the callback function.

```
I2C_Transaction   i2cTransaction;
UChar             writeBuffer[3];
UChar             readBuffer[2];
Bool              transferOK;

i2cTransaction.slaveAddress = 0x50;     /* 7-bit peripheral slave address */
i2cTransaction.writeBuf = writeBuffer;  /* Buffer to be written */
i2cTransaction.writeCount = 3;          /* Number of bytes to be written */
i2cTransaction.readBuf = readBuffer;    /* Buffer to be read */
i2cTransaction.readCount = 2;           /* Number of bytes to be read */
i2cTransaction.arg = someOptionalArgument;

/* I2C_transfers will always return successful */
I2C_transfer(i2c, &i2cTransaction);     /* Perform I2C transfer */
```

5.6.6.5 Queuing Multiple I^2C Transactions

Using the callback mode, you can queue up multiple I^2C transactions. However, each I^2C transfer must use a unique instance of an I2C_Transaction data structure. In other words, it is not possible to reschedule an I2C_Transaction structure more than once. This also implies that the application must make sure the I2C_Transaction isn't reused until it knows that the I2C_Transaction is available again.

The following code posts a Semaphore after the last I2C_Transaction has completed. This is done by passing the Semaphore's handle through the I2C_Transaction data structure and evaluating it in the UserCallbackFxn.

```
Void UserCallbackFxn(I2C_Handle handle, I2C_Transaction *msg, Bool transfer) {
    if (msg->arg != NULL) {
        Semaphore_post((Semaphore_Handle)(msg->arg));
    }
}

Void taskfxn(arg0, arg1) {
    I2C_Transaction   i2cTransaction0;
    I2C_Transaction   i2cTransaction1;
    I2C_Transaction   i2cTransaction2;

    /* Set up i2cTransaction0/1/2 here */
        ...
    i2cTransaction0.arg = NULL;
    i2cTransaction1.arg = NULL;
    i2cTransaction2.arg = semaphoreHandle;

    /* Start and queue up the I2C transactions */
    I2C_transfer(i2c, &i2cTransaction0);
    I2C_transfer(i2c, &i2cTransaction1);
    I2C_transfer(i2c, &i2cTransaction2);

    /* Do other optional code here */
        ...

    /* Pend on the I2C transactions to have completed */
    Semaphore_pend(semaphoreHandle);
}
```

5.6.7 Instrumentation

The instrumented I^2C library contains Log_print() statements that help to debug I^2C transfers. The I^2C driver logs the following actions using the Log_print() APIs provided by SYS/BIOS:

- I^2C object opened or closed.
- Data written or read in the interrupt handler.
- Transfer results.

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information. Diags_USER2 provides detailed logs intended to help determine where a problem may lie in the I^2C transaction. This level of diagnostics will generate a significant amount of Log entries. Use this mask when granular transfer details are needed.

5.7 I²S Driver

The I²S driver facilitates the use of Inter-IC Sound (I²S), which is used to connect digital audio devices so that audio signals can be communicated between devices. The I²S driver simplifies reading and writing to any of the Multichannel Audio Serial Port (McASP) peripherals on the board with Receive and Transmit support. These include blocking, non-blocking, read and write characters on the McASP peripheral.

5.7.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.7.2 Runtime Configuration

The board's I²S peripheral and pins must be configured before initializing an I²S instance.

As the overview in Section 5.2.2 indicates, the I²S driver requires the application to initialize board-specific settings and provide the I²S driver with the I2S_config structure.

5.7.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initI2S() function that initializes the board-specific I2S peripheral settings. This function also calls I2S_init() to initialize the I²S driver.

5.7.2.2 I2S_Params Structure

The I2S_Params structure may be used to override the default settings for an I²S instance you are creating. The params in the structure must be set before calling I2S_open().

The structure has the following fields:

```
typedef struct I2S_Params  {
    I2S_OpMode       operationMode;
    uint32_t         samplingFrequency; /* in samples/second, default = 16000 */
    unsigned char    slotLength;        /* default = 16 */
    unsigned char    bitsPerSample;     /* default = 16 */
    unsigned char    numChannels;       /* Mono/Stereo */

    I2S_DataMode     readMode;          /* mode for all read calls */
    I2S_Callback     readCallback;      /* pointer to read callback */
    uint32_t         readTimeout;

    I2S_DataMode     writeMode;         /* mode for all write calls */
    I2S_Callback     writeCallback;     /* pointer to write callback */
    uint32_t         writeTimeout;
    void *           customParams;
} I2S_Params;
```

5.7.3 I²S Modes

The I²S operation mode determines whether transmit and/or receive modes are enabled. The mode is specified with one of the following constants:

- *I2S_OPMODE_TX_ONLY:* Enable transmit only.
- *I2S_OPMODE_RX_ONLY:* Enable receive only.

- *I2S_OPMODE_TX_RX_SYNC:* Enable both transmit and receive.

A separate data mode may be specified for read calls and write calls. The available modes are:

- *I2S_MODE_CALLBACK:* This mode is non-blocking. Calls to read or write return immediately. When the transfer is finished, the configured callback function is called.
- *I2S_MODE_ISSUERECLAIM:* Call I2S_readIssue() and I2S_writeIssue() to queue buffers to the I^2S. I2S_readReclaim() blocks until a buffer of data is available. I2S_writeReclaim() blocks until a buffer of data has been issued and the descriptor can be returned back to the caller.

5.7.4 APIs

In order to use the I^2S module APIs, the I2S.h header file should be included in an application as follows:

```
#include <ti/drivers/I2S.h>
```

The following are the I^2S APIs:

- **I2S_init()** initializes the I^2S module.
- **I2S_Params_init()** initializes an I2S_Params data structure.
- **I2S_open()** initializes a given I^2S instance.
- **I2S_close()** deinitializes a given I^2S instance.
- **I2S_control()** performs implementation-specific features on a given I^2S peripheral.
- **I2S_read()** queues a buffer for reading from the peripheral.
- **I2S_readIssueFxn()** queues a buffer for reading from the peripheral.
- **I2S_readReclaimFxn()** retrieves a received buffer of data from the peripheral.
- **I2S_write()** queues a buffer for writing from the peripheral.
- **I2S_writeIssueFxn()** queues a buffer for writing from the peripheral.
- **I2S_writeReclaimFxn()** retrieves a sent buffer of data from the peripheral.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.7.4.1 Opening the I^2S driver

To open a I^2S driver instance, initialize a I2S_Params object and specify the desired parameters.

```
I2S_Handle      handle;
I2S_Params      params;
I2SCC3200DMA_SerialPinParams customParams;

I2S_Params_init(&params);
params.operationMode   = I2S_MODE_TX_RX_SYNC;
/* Change other params as required */

handle = I2S_open(someI2S_configIndexValue, &params);
if (!handle) {
    System_printf("I2S did not open");
}
```

5.7.4.2 Writing Data

The following example calls I2S_write() to write to an I^2S driver instance that has been opened. It first queues up two buffers of text. Within an infinite loop, it then calls I2S_writeReclaim to retrieve a buffer., prints the size of the buffer retrieved, and re-queues the buffer.

```
const unsigned char hello[] = "Hello World\n";
const unsigned char hello1[] = "Hello World1\n";
I2S_BufDesc writeBuffer1;
I2S_BufDesc writeBuffer2;
I2S_BufDesc *pDesc = NULL;

writeBuffer1.bufPtr  = &hello;
writeBuffer1.bufSize = sizeof(hello);
writeBuffer2.bufPtr  = &hello1;
writeBuffer2.bufSize = sizeof(hello1);

ret = I2S_write(handle, &writeBuffer1);
ret = I2S_write(handle, &writeBuffer2);

while(1)
{
    ret = I2S_writeReclaim(handle, &pDesc);
    System_printf("The I2S wrote %d bytes\n", ret);
    pDesc->bufPtr  = &hello;
    pDesc->bufSize = sizeof(hello);
    ret = I2S_write(handle, pDesc);
}
```

5.7.4.3 Reading Data

The following example calls I2S_read() to queue a buffer for reading from an I^2S driver instance. It first queues up two buffers of text. Within an infinite loop, it then calls I2S_readReclaim to queue a buffer and reads the buffer.

```
unsigned char rxBuffer[20];
unsigned char rxBuffer1[20];
I2S_BufDesc readBuffer1;
I2S_BufDesc readBuffer2;
I2S_BufDesc *pDesc = NULL;

readBuffer1.bufPtr = &rxBuffer;
readBuffer1.bufSize = 20;
readBuffer2.bufPtr = &rxBuffer1;
readBuffer2.bufSize = 20;

ret = I2S_read(handle, &readBuffer1);
ret = I2S_read(handle, &readBuffer2);

while(1)
{
    ret = I2S_readReclaim(handle, &pDesc);
    System_printf("The I2S read %d bytes\n", ret);
    pDesc->bufPtr = &rxBuffer;
    pDesc->bufSize = 20;
    ret = I2S_read(handle, pDesc);
}
```

5.8 LCD Driver

The LCD driver manages the DOGM128W-6 LCD display, which is the one used on the SmartRF06 EB (Evaluation Board). This LCD display can be written to by first writing the data to display to an internal buffer, and then sending the buffer content to the LCD display. The LCD driver uses the SPI driver for updating the display. The driver contains different functions for both modifying the internal buffer and for updating the display with the buffer content.

The DOGM128W-6 LCD display contains 128 pixel columns and 64 pixel rows. The origin (0,0) location of the display is in the upper-left corner. This LCD driver uses a 5x7 font, thus each character takes up 5 pixels width and 7 pixels height. See the LCDDogm1286.h help topic in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

This driver is located in `<tirtos_install>\products\tidrivers_<version>\packages\ti\mw\lcd`. Note that this location is different from the location of most other TI-RTOS drivers, because this is a middleware driver.

In order to use this driver, C files use the following #include statement. This statement is already included in the board files for the targets that support this driver.

```
#include <ti/mw/lcd/LCDDogm1286.h>
```

5.8.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

To enable this driver, add the following statement to your application's *.cfg file.

```
var mwConfig = xdc.useModule('ti.mw.Config');
```

Note that this package location is different from the location of most other drivers, because this is a middleware driver.

5.8.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the LCD driver uses the LCD_config structure.

5.8.2.1 LCD_config Structure

The <*board*>.c file declares the LCD_config structure. This structure must be provided to the LCD driver. It must be initialized before the LCD_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.8.3 APIs

In order to use the LCD module APIs, include the SDSPI header file in an application as follows:

```
#include <ti/mw/lcd/LCDDogm1286.h>
```

The following are some of the LCD APIs:

- **LCD_init()** sets up the LCD for operation.
- **LCD_open()** initializes the SPI parameters and other objects used by the driver. Sends an initialization command to the LCD.
- **LCD_close()** closes the objects and resources used by the LCD driver.
- **LCD_writeLine()** writes one line to a buffer and sends it to the display.
- **LCD_bufferClear()** empties the specified LCD buffer.
- **LCD_bufferPrintString()** writes a string to the specified buffer.
- **LCD_bufferInvert()** inverts the pixels in a given region of the specified buffer.
- **LCD_update()** writes the specified buffer to the display.
- **LCD_setContrast()** sets the LCD contrast level.
- **LCD_Params_init()** initializes an LCD_Params structure to its defaults.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.8.4 Usage

The LCD driver uses an array of LCD_Buffer structures named lcdBuffers. lcdBuffers is implemented in the application with each entry being an instance of a LCD_Buffer. The number of buffers available must be passed as an argument to LCD_open().

Once the LCD has been initialized and opened, LCD APIs can be used to write to the buffers and send all or part of a buffer to the display.

When the application has finished using the LCD, it should call LCD_close().

5.8.5 Instrumentation

The instrumented LCD library contains Log_print() statements that help debug LCD activity.

5.9 PWM Driver

The PWM module facilitates the generation of Pulse Width Modulated signals via simple and portable APIs. The PWM driver is designed such that a driver instance generates a single waveform. This section assumes that you have an understanding of Pulse Width Modulation techniques.

5.9.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.9.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the PWM driver requires the application to initialize board-specific settings and provide the PWM driver with the PWM_config structure.

5.9.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initPWM() function that initializes the board-specific PWM peripheral settings. This function also calls PWM_init() to initialize the PWM driver.

5.9.2.2 PWM_config Structure

The <*board*>.c file also declares the PWM_config structure. This structure must be provided to the PWM driver. It must be initialized before the PWM_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.9.3 APIs

To use the PWM module APIs, the PWM.h header file should be included in an application as follows:

```
#include <ti/drivers/PWM.h>
```

The following are the PWM APIs:

- **PWM_init()** initializes the PWM module.
- **PWM_Params_init()** initializes an PWM_Params data structure.
- **PWM_open()** initializes a given PWM instance.
- **PWM_close()** deinitializes a given PWM instance.
- **PWM_control()** performs implementation-specific features to a given PWM peripheral.
- **PWM_getPeriodCounts()** returns the PWM period in timer ticks.
- **PWM_getPeriodMicroSecs()** returns the PWM period in microseconds.
- **PWM_setDuty()** sets a PWM instances duty cycle.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.9.4 Usage

The application needs to supply the following structures in order to set up the framework for the driver:

- **PWM_Params** specifies the period, units in which the duty is specified and the PWM output polarity. See Section 5.9.4.1.

5.9.4.1 PWM Parameters

The PWM_Params structure is used to initialize a PWM driver instance with the PWM_open() function call. Before opening the driver, the desired PWM period should be specified set in the PWM_Params. The period must be specified in microseconds. Additionally, the PWM output polarity and the duty mode should also be configured as desired.

```
typedef struct PWM_Params {
    uint32_t     period;         /* PWM period in microseconds */
    PWM_DutyMode dutyMode;       /* Units which duty is specified */
    PWM_Polarity polarity;       /* Set duty active high or active low */
} PWM_Params;
```

5.9.5 PWM Modes

The PWM operating mode determines the units in which the duty specified when calling PWM_setDuty(). The PWM driver supports three modes of operation:

- *PWM_DUTY_COUNTS:* The duty is specified in PWM timer counts.
- *PWM_DUTY_TIME:* The duty is specified in microseconds.
- *PWM_DUTY_SCALAR:* The duty is an integer scaled to the period, where 0 corresponds to a duty of 0% and 65535 corresponds to 100% duty.

The mode is determined by the PWM_DutyMode field within PWM_Params data structure. The PWM_Params default for this field is PWM_DUTY_TIME mode. Once opened, the only way to change the operating mode is to close and re-open the PWM instance with a new mode.

5.9.5.1 Opening the PWM driver

To open a PWM driver instance, initialize a PWM_Params object and specify the desired PWM period. Additionally, if a duty mode other than PWM_DUTY_TIME (default) is desired, specify it in the PWM_Params before opening the driver instance.

```
PWM_Handle handle;
PWM_Params params;
uint8_t    pwmOutputNumber = 0;

PWM_Params_init(&params);
params.period = 20000;             // Period in microseconds
params.dutyMode = PWM_DUTY_COUNTS; // Set PWM duty mode

handle = PWM_open(Board_PWM0, &params);
if (handle == NULL) {
    /* Error opening PWM */
}

PWM_setDuty(handle, 3000);         // Set a duty cycle of 3000 PWM timer counts
```

5.9.6 Instrumentation

The instrumented PWM library contains Log_print() statements that help to debug PWM driver calls. The PWM driver logs the following actions using the Log_print() APIs provided by SYS/BIOS:

- PWM object opened or closed.
- The duty cycle of a PWM output has been changed.

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information. Diags_USER2 provides detailed logs intended to help determine if a problem has occurred while changing a duty cycle. This level of diagnostics generates a significant number of Log entries. Use this mask when granular details are needed.

5.10 SDSPI Driver

The SDSPI FatFs driver is used to communicate with SD (Secure Digital) cards via SPI (Serial Peripheral Interface).

The SDSPI driver is a FatFs driver module for the FatFs middleware module. With the exception of the standard TI-RTOS driver APIs—SDSPI_open(), SDSPI_close(), and SDSPI_init()—the SDSPI driver is exclusively used by FatFs module to handle the low-level hardware communications. See Chapter 7, "Using the FatFs File System Drivers" for usage guidelines.

The SDSPI driver only supports one SSI (SPI) peripheral at a given time. It does not utilize interrupts.

The SDSPI driver is polling based for performance reasons and due the relatively high SPI bus bit rate. This means it does not utilize the SPI's peripheral interrupts, and it consumes the entire CPU time when communicating with the SPI bus. Data transfers to or from the SD card are typically 512 bytes, which could take a significant amount of time to complete. During this time, only higher priority Tasks, Swis, and Hwis can preempt Tasks making calls that use the FatFs.

5.10.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

To enable this driver, add the following statements to your application's *.cfg file.

```
var FatFs = xdc.useModule('ti.mw.fatfs.FatFS');
```

5.10.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the SDSPI driver requires the application to initialize board-specific portions of the SDSPI and provide the SDSPI driver with the SDSPI_config structure.

5.10.2.1 Board-Specific Configuration

The <board>.c files contain a <*board*>_initSDSPI() function that must be called to initialize the board-specific SDSPI peripheral settings. This function also calls the SDSPI_init() to initialize the SDSPI driver.

5.10.2.2 SDSPI_config Structure

The <*board*>.c file also declare the SDSPI_config structure. This structure must be provided to the SDSPI driver. It must be initialized before the SDSPI_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.10.3 APIs

In order to use the SDSPI module APIs, include the SDSPI header file in an application as follows:

```
#include <ti/drivers/SDSPI.h>
```

The following are the SDSPI APIs:

- **SDSPI_init()** sets up the specified SPI and GPIO pins for operation.
- **SDSPI_open()** registers the SDSPI driver with FatFs and mounts the FatFs file system.

- **SDSPI_close()** unmounts the file system and unregisters the SDSPI driver from FatFs.
- **SDSPI_Params_init()** initializes a SDSPI_Params structure to its defaults.

See the help in <*tirtos_install*>\products\tidrivers_<*version*>\docs\html\index.html for details.

5.10.4 Usage

Before any FatFs or C I/O APIs can be used, the application needs to open the SDSPI driver. The SDSPI_open() function ensures that the SDSPI disk functions get registered with the FatFs module that subsequently mounts the FatFs volume to that particular drive.

```
SDSPI_Handle sdspiHandle;
SDSPI_Params sdspiParams;
UInt peripheralNum = 0;     /* Such as SPI0 */
UInt FatFsDriveNum = 0;

SDSPI_Params_init(&sdspiParams);
sdspiHandle = SDSPI_open(peripheralNum, FatFsDriveNum, &sdspiParams);
if (sdspiHandle == NULL) {
    System_abort("Error opening SDSPI\n");
}
```

Similarly, the SDSPI_close() function unmounts the FatFs volume and unregisters SDSPI disk functions.

```
SDSPI_close(sdspiHandle);
```

Note that it is up to the application to ensure the no FatFs or C I/O APIs are called before the SDSPI driver has been opened or after the SDSPI driver has been closed.

5.10.5 Instrumentation

The SDSPI driver does not make any Log calls.

5.11 SPI Driver

The Serial Peripheral Interface (SPI) driver is a generic, full-duplex driver that transmits and receives data on a SPI bus. SPI is sometimes called SSI (Synchronous Serial Interface).

The SPI protocol defines the format of a data transfer over the SPI bus, but it leaves flow control, data formatting, and handshaking mechanisms to higher-level software layers.

5.11.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.11.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the SPI driver requires the application to initialize board-specific portions of the SPI and to provide the SPI driver with the SPI_config structure.

5.11.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initSPI() function that must be called to initialize the board-specific SPI peripheral settings. This function also calls the SPI_init() to initialize the SPI driver.

5.11.2.2 SPI_config Structure

The <*board*>.c file also declares the SPI_config structure. This structure must be provided to the SPI driver. It must be initialized before the SPI_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.11.3 APIs

In order to use the SPI module APIs, the SPI.h header file should be included in an application as follows:

```
#include <ti/drivers/SPI.h>
```

The following are the SPI APIs:

- **SPI_init()** initializes the SPI module.
- **SPI_Params_init()** initializes a SPI_Params data structure to default values.
- **SPI_open()** initializes a given SPI peripheral.
- **SPI_close()** deinitializes a given SPI peripheral.
- **SPI_transfer()** handles the SPI transfers for SYS/BIOS.

The SPI_transfer() API can be called only from a Task context when used in SPI_MODE_BLOCKING. It requires a SPI_Transaction structure that specifies the location of the write and read buffer and the number of SPI frames to be transmitted/received. In SPI frame formats, data is sent in full-duplex mode.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.11.4 Usage

The application needs to supply the following structures in order to set up the framework for the driver:

- **SPI_Params** specifies the transfer mode and any callback function to be used. See Section 5.11.4.1.
- **SPI_Transaction** specifies details about a transfer to be performed. See Section 5.11.4.2.
- **SPI_Callback** specifies a function to be used if you are using callback mode. See Section 5.11.4.3.

5.11.4.1 SPI Parameters

The SPI_Params structure is used with the SPI_open() function call.

If the transferMode is set to SPI_MODE_BLOCKING, the transferCallback argument is ignored. If transferMode is set to SPI_MODE_CALLBACK, a user-defined callback function must be supplied. The mode parameter determines whether the SPI operates in master or slave mode. The desired SPI bit transfer rate, frame data size, and frame format are specified with bitRate, dataSize and frameFormat respectively.

```
typedef struct SPI_Params {
    SPI_TransferMode transferMode;        /* Blocking or Callback mode */
    SPI_CallbackFxn  transferCallbackFxn; /* Callback function pointer */
    SPI_Mode         mode;                /* Master or Slave mode */
    UInt             bitRate;             /* SPI bit rate in Hz */
    UInt             dataSize;            /* SPI data frame size in bits */
    SPI_FrameFormat  frameFormat;         /* SPI frame format */
} SPI_Params;
```

5.11.4.2 SPI Frame Formats, Transactions, and Data Sizes

The SPI driver can configure the device's SPI peripheral with various SPI frameFormat options: SPI (with various polarity and phase settings), TI, and Micro-wire.

The smallest single unit of data transmitted onto the SPI bus is called a SPI frame and is of size dataSize. A series of SPI frames transmitted/received on a SPI bus is known as a SPI transaction. A SPI_transfer() of a SPI transaction is performed atomically.

```
typedef struct SPI_Transaction {
    UInt    count;      /* Number of frames for this transaction */

    Ptr     txBuf;      /* Ptr to a buffer with data to be transmitted */
    Ptr     rxBuf;      /* Ptr to a buffer to receive data */

    UArg    arg;        /* Argument to be passed to the callback function */
} SPI_Transaction;
```

The txBuf and rxBuf parameters are both pointers to data buffers. If txBuf is NULL, the driver sends SPI frames with all data bits set to 0. If rxBuf is NULL, the driver discards all SPI frames received.

When the SPI is opened, the dataSize value determines the element types of txBuf and rxBuf. If the dataSize is from 4 to 8 bits, the driver assumes the data buffers are of type UChar (unsigned char). If the dataSize is larger than 8 bits, the driver assumes the data buffers are of type UShort (unsigned short).

The optional arg variable can only be used when the SPI driver has been opened in callback mode. This variable is used to pass a user-defined value into the user-defined callback function.

Specifics about SPI frame formatting and data sizes are provided in device-specific data sheets and technical reference manuals.

5.11.4.3 SPI Callback Function Prototype

This typedef defines the function prototype for the SPI driver's callback function for callback mode:

```
typedef Void (*SPI_Callback)(SPI_Handle, SPI_Transaction *);
```

When the SPI driver calls this function, it supplies the associated SPI_Handle and a pointer to the SPI_Transaction that just completed. There is no formal definition for what constitutes a successful SPI transaction, so every callback is considered a successful transaction. The application or middleware should examine the data to determine if the transaction met application-specific requirements.

5.11.5 Callback and Blocking Modes

The SPI driver supports two modes of operation: blocking and callback modes. The mode is determined by the mode parameter in the SPI_Params data structure used when the SPI driver is opened. If no SPI_Params structure is specified, the SPI driver defaults to blocking mode. Once a SPI driver is opened, the only way to change the operation mode is to close and re-open the SPI instance with the new mode.

5.11.5.1 Opening a SPI Driver in Blocking Mode

By default, the SPI driver operates in blocking mode. In blocking mode, a Task's code execution is blocked until a SPI transaction has completed. This ensures that only one SPI transaction operates at a given time. Other tasks requesting SPI transactions while a transaction is currently taking place are also placed into a blocked state and are executed in the order in which they were received.

```
SPI_Handle  spi;
UInt        peripheralNum = 0; /* Such as SPI0 */
SPI_Params  spiParams;

SPI_Params_init(&spiParams);
spiParams.transferMode = SPI_MODE_BLOCKING;
spiParams.transferCallbackFxn = NULL;

spi = SPI_open(peripheralNum, &spiParams);
if (spi == NULL) {
    /* Error opening SPI */
}
```

Blocking mode is not supported in the execution context of a Swi or Hwi.

If no SPI_Params structure is passed to SPI_open(), default values are used. If the open call is successful, it returns a non-NULL value.

5.11.5.2 Opening a SPI Driver in Callback Mode

In callback mode, a SPI transaction functions asynchronously, which means that it does not block code execution. After a SPI transaction has been completed, the SPI driver calls a user-provided hook function.

```
SPI_Handle  spi;
UInt        peripheralNum = 0; /* Such as SPI0 */
SPI_Params  spiParams;

SPI_Params_init(&spiParams);
spiParams.transferMode = SPI_MODE_CALLBACK;
spiParams.transferCallbackFxn = UserCallbackFxn;

spi = SPI_open(peripheralNum, &spiParams);
if (spi == NULL) {
    /* Error opening SPI */
}
```

Callback mode is supported in the execution context of Tasks, Swis and Hwis. However, if a SPI transaction is requested while a transaction is taking place, the SPI_transfer() returns FALSE.

5.11.6 SPI Transactions

SPI_transfer() always performs full-duplex SPI transactions. This means the SPI simultaneously receives data as it transmits data. The application is responsible for formatting the data to be transmitted as well as determining whether the data received is meaningful. The following code snippets perform SPI transactions.

Transferring n 4-8 bit SPI frames:

```
SPI_Transaction  spiTransaction;
UChar            transmitBuffer[n];
UChar            receiveBuffer[n];
Bool             transferOK;

SPI_Params_init(&spiParams);
spiParams.dataSize = 6;  /* dataSize can range from 4 to 8 bits */
spi = SPI_open(peripheralNum, &spiParams);

...

spiTransaction.count = n;
spiTransaction.txBuf = transmitBuffer;
spiTransaction.rxBuf = receiveBuffer;

transferOK =  SPI_transfer(spi, &spiTransaction);
if (!transferOK) {
   /* Error in SPI transfer or transfer is already in progress */
}
```

Transferring n 9-16 bit SPI frames:

```
SPI_Transaction  spiTransaction;
UShort           transmitBuffer[n];
UShort           receiveBuffer[n];
Bool             transferOK;

SPI_Params_init(&spiParams);
spiParams.dataSize = 12;  /* dataSize can range from 9 to 16 bits */
spi = SPI_open(peripheralNum, &spiParams);

...

spiTransaction.count = n;
spiTransaction.txBuf = transmitBuffer;
spiTransaction.rxBuf = receiveBuffer;

transferOK =  SPI_transfer(spi, &spiTransaction);
if (!transferOK) {
   /* Error in SPI transfer or transfer is already in progress */
}
```

5.11.7 Master/Slave Modes

This SPI driver functions in both SPI master and SPI slave modes. Logically, the implementation is identical; however the difference between these two modes is driven by hardware. As a SPI master, the peripheral is in control of the clock signal and therefore will commence communications to the SPI slave immediately. As a SPI slave, the SPI driver prepares the peripheral to transmit and receive data in a way such that the peripheral is ready to transfer data when the SPI master initiates a transaction.

Asserting on Chip Select

The SPI protocol requires that the SPI master asserts a SPI slave's chip select pin prior starting a SPI transaction. While this protocol is generally followed, various types of SPI peripherals have different timing requirements as to when and for how long the chip select pin must remain asserted for a SPI transaction.

Commonly, the SPI master uses a hardware chip select to assert and de-assert the SPI slave for every data frame. In other cases, a SPI slave imposes the requirement of asserting the chip select over several SPI data frames. This is generally accomplished by using a regular, general-purpose output pin. Due to the complexity of such SPI peripheral implementations, the SPI driver provided with TI-RTOS has been designed to operate transparently to the SPI chip select. When the hardware chip select is used, the peripheral automatically selects/enables the peripheral. When using a software chip select, the application needs to handle the proper chip select and pin configuration.

- **Hardware chip select.** No additional action by the application is required.
- **Software chip select.** The application needs to handle the chip select assertion and de-assertion for the proper SPI peripheral.

Note that the implementation of hardware chip select is device-dependent. MSP43x does not support the hardware chip select feature. Tiva devices performs hardware chip select only when pin-muxed out.

5.11.8 Instrumentation

The instrumented SPI library contains Log_print() and Log_error() statements that help debug SPI transfers. The SPI driver logs the following actions:

- SPI object opened or closed
- DMA transfer configurations enabled
- SPI interrupt occurred
- Initialization error occurred
- Semaphore pend or post

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information. Diags_USER2 provides detailed logs intended to help determine where a problem may lie in the SPI transactions. This level of diagnostics will generate a significant amount of Log entries. Use this mask when granular transfer details are needed.

5.12 UART Driver

A UART is used to translate data between the chip and a serial port. The UART driver simplifies reading and writing to any of the UART peripherals on the board with multiple modes of operation and performance. These include blocking, non-blocking, and polling as well as text/binary mode, echo and return characters.

5.12.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.12.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the UART driver requires the application to initialize board-specific portions of the UART and provide the UART driver with the UART_config structure.

5.12.2.1 Board-Specific Configuration

The <board>.c files contain a <*board*>_initUART() function that must be called to initialize the board-specific UART peripheral settings. This function also calls the UART_init() to initialize the UART driver.

5.12.2.2 UART_config Structure

The <*board*>.c file also declare the UART_config structure. This structure must be provided to the UART driver. It must be initialized before the UART_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.12.3 APIs

In order to use the UART module APIs, the UART header file should be included in an application as follows:

```
#include <ti/drivers/UART.h>
```

The following are the UART APIs:

- **UART_init()** initializes the UART module.
- **UART_Params_init ()** initializes the UART_Params struct to its defaults for use in calls to UART_open().
- **UART_open()** opens a UART instance.
- **UART_close()** closes a UART instance.
- **UART_write()** writes a buffer of characters to the UART.
- **UART_writePolling()** writes a buffer to the UART in the context of the call and returns when finished.
- **UART_writeCancel()** cancels the current write action and unblocks or make the callback.
- **UART_read()** reads a buffer of characters to the UART.
- **UART_readPolling()** reads a buffer to the UART in the context of the call and returns when finished.

- **UART_readCancel()** cancels the current read action and unblocks or make the callback.

See the help in <*tirtos_install*>\products\tidrivers_<*version*>\docs\html\index.html for details.

5.12.4 Usage

The UART driver does not configure any board peripherals or pins; this must be completed before any calls to the UART driver. The examples call Board_initUART(), which is mapped to a specific initUART() function for the board. The board-specific initUART() functions are provided in the board .c and .h files. For example, a sample UART setup is provided in the TMDXDOCKH52C1_initUART() function in the TMDXDOCKH52C1.c file. This function sets up the peripheral and pins used by UART0 for operation through the JTAG emulation connection (no extra hardware needed). The examples that use the UART driver call the Board_initUART() function from within main().

Once the peripherals are set up, the application must initialize the UART driver by calling UART_init(). If you add the provided board setup files to your project, you can call the Board_initUART() function within main().

Once the UART has been initialized, you can open UART instances. Only one UART index can be used at a time. If the index is already in use, the driver returns NULL and logs a warning. Opening a UART requires four steps:

1. Create and initialize a UART_Params structure.
2. Fill in the desired parameters.
3. Call UART_open() passing in the index of the UART from the configuration structure and Params.
4. Save the UART handle that is returned by UART_open(). This handle will be used to read and write to the UART you just created.

For example:

```
UART_Handle uart;
UART_Params uartParams;

Board_initUART();          // Calls UART_init for you

/* Create a UART with data processing off. */
UART_Params_init(&uartParams);
uartParams.writeDataMode = UART_DATA_BINARY;
uartParams.readDataMode = UART_DATA_BINARY;
uartParams.readReturnMode = UART_RETURN_FULL;
uartParams.readEcho = UART_ECHO_OFF;

uart = UART_open(Board_UART, &uartParams);
```

Options for the writeMode and readMode parameters are UART_MODE_BLOCKING and UART_MODE_CALLBACK.

- UART_MODE_BLOCKING uses a semaphore to block while data is being sent. The context of the call must be a SYS/BIOS Task.
- UART_MODE_CALLBACK is non-blocking and will return while data is being sent in the context of a Hwi. The UART driver will call the callback function whenever a write or read finishes. In some cases, the action might have been canceled or received a newline, so the number of bytes sent/received are passed in. Your implementation of the callback function can use this information as needed.

Options for the writeDataMode and readDataMode parameters are UART_MODE_BINARY and UART_MODE_TEXT. If the data mode is UART_MODE_BINARY, the data is passed as is, without processing. If the data mode is UART_MODE_TEXT, write actions add a return before a newline character, and read actions replace a return with a newline. This effectively treats all device line endings as LF and all host PC line endings as CRLF.

Options for the readReturnMode parameter are UART_RETURN_FULL and UART_RETURN_NEWLINE. These determine when a read action unblocks or returns. If the return mode is UART_RETURN_FULL, the read action unblocks or returns when the buffer is full. If the return mode is UART_RETURN_NEWLINE, the read action unblocks or returns when a newline character is read.

Options for the readEcho parameter are UART_ECHO_OFF and UART_ECHO_ON. This parameter determines whether the driver echoes data back to the UART. When echo is turned on, each character that is read by the target is written back independent of any write operations. If data is received in the middle of a write and echo is turned on, the characters echoed back will be mixed in with the write data.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.12.5 UART DMA Driver for TivaC Devices

For TivaC devices, the UART driver can be configured to use DMA, if desired. The <*board*>.c file contains configuration for both the DMA-based UART driver and the non-DMA-based UART driver. To use the DMA-based UART driver, compile <*board*>.c with the preprocessor symbol TI_DRIVERS_UARTDMA set to 1. This can be set either in <*board*>.c by adding:

```
#define TI_DRIVERS_UART_DMA 1
```

or, in the CCS project settings, under the compiler flags:

```
--define=TI_DRIVERS_UART_DMA=1
```

Of the TI-RTOS UART examples, only the UART Echo example is suitable for using UART DMA.

The UART Console example calls scanf(), requiring the UART driver to inspect the data and return from a UART_read() call when a newline character is received. The UART DMA driver does not examine input or output data, so using UART DMA with the UART Console example causes the call to scanf(), which calls UART_read(), to hang waiting for input.

The other UART example, UART Logging, calls UART_writePolling(), which does not use DMA; only UART_write() and UART_read() use DMA. Although the UART Echo example can be built to use UART DMA, it is not an interesting use case, as it reads and writes only one character at a time.

5.12.6 UART DMA Driver for SimpleLink CC32xx Devices

For CC32xx devices, the UART driver can be configured to use DMA, if desired. The <*board*>.c file contains configuration for both the DMA-based UART driver, and the non-DMA-based UART driver. To use the DMA-based UART driver, compile <*board*>.c with the preprocessor symbol TI_DRIVERS_UARTDMA set to 1. This can be set either in <*board*>.c by adding:

```
#define TI_DRIVERS_UART_DMA 1
```

or, in the CCS project settings, under the compiler flags:

```
--define=TI_DRIVERS_UART_DMA=1
```

Of the TI-RTOS UART examples, only the UART Echo example is suitable for using UART DMA.

The other UART example, UART Logging, calls UART_writePolling(), which does not use DMA; only UART_write() and UART_read() use DMA. Although the UART Echo example can be built to use UART DMA, it is not an interesting use case, as it reads and writes only one character at a time.

5.12.7 Instrumentation

The UART module provides instrumentation data by making log calls.

5.12.7.1 Logging

The UART driver is instrumented with Log events that can be viewed with UIA and RTOS Analyzer. Diags masks can be turned on and off to provide granularity to the information that is logged.

Use Diags_USER1 to see general Log events such as success opening a UART, number of bytes read or written, and warnings/errors during operation.

Use Diags_USER2 to see more granularity when debugging. Each character read or written will be logged as well as several other key events.

The UART driver makes log calls when the following actions occur:

- UART_open() success or failure
- UART_close() success
- UART interrupt triggered
- UART_write() finished
- Byte was written
- UART_read() finished
- Byte was read
- UART_write() finished, canceled or timed out
- UART_read() finished, canceled or timed out

5.13 USBMSCHFatFs Driver

The USBMSCHFatFs driver is a FatFs driver module that has been designed to be used by the FatFs module. With the exception of the standard TI-RTOS driver APIs—_open(), _close(), and _init()—the USBMSCHFatFs driver is exclusively used by FatFs module to handle communications to a USB flash drive. See Chapter 7 for usage guidelines.

The USBMSCHFatFs driver is uses the USB Library, which is provided with TivaWare and MWare to communicate with USB flash drives as a USB Mass Storage Class (MSC) host controller. Only one USB flash drive connected directly to the USB controller at a time is supported.

Tasks that make FatFs calls can be preempted only by higher priority tasks, Swis, and Hwis.

5.13.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

To enable messages about this driver's activity, add the following statement to your application's *.cfg file.

```
var FatFs = xdc.useModule('ti.mw.fatfs.FatFS');
```

5.13.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the USBMSCHFatFs driver requires the application to initialize board-specific portions of the USBMSCHFatFs and provide the USBMSCHFatFs driver with the USBMSCHFatFs_config structure.

5.13.2.1 Board-Specific Configuration

The <board>.c files contain a <*board*>_initUSBMSCHFatFs() function that must be called to initialize the board-specific USBMSCHFatFs peripheral settings. This function also calls the USBMSCHFatFs_init() to initialize the USBMSCHFatFs driver.

5.13.2.2 USBMSCHFatFs_config Structure

The <*board*>.c file also declare the USBMSCHFatFs_config structure. This structure must be provided to the USBMSCHFatFs driver. It must be initialized before the USBMSCHFatFs_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.13.3 APIs

In order to use the USBMSCHFatFs module APIs, the USBMSCHFatFs header file should be included in an application as follows:

```
#include <ti/drivers/USBMSCHFatFs.h>
```

The following are the USBMSCHFatFs APIs:

- **USBMSCHFatFs_init()** initializes the USBMSCHFatFs data objects pointed by the driver's config structure.
- **USBMSCHFatFs_open()** registers the USBMSCHFatFs driver with FatFs and mounts the FatFs file system.
- **USBMSCHFatFs_close()** unmounts the file system and unregisters the USBMSCHFatFs driver from FatFs.
- **USBMSCHFatFs_Params_init()** initializes a USBMSCHFatFs_Params structure to its defaults.
- **USBMSCHFatFs_waitForConnect()** blocks a task's execution until a USB flash drive was detected.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.13.4 Usage

Before the FatFs APIs can be used, the application needs to open the USBMSCHFatFs driver. The USBMSCHFatFs_open() function ensures that the USBMSCHFatFs disk functions get registered with the FatFs module. The FatFs module then mounts the FatFs volume to that particular drive.

Internally, opening the USBMSCHFatFs driver creates a high-priority Task to service the USB library. The default priority for this task is 15 and runs every 10 SYS/BIOS system ticks. You can change the priority of this task using the USBMSCHFatFs_Params structure.

```
USBMSCHFatFs_Handle usbmschfatfsHandle;
USBMSCHFatFs_Params usbmschfatfsParams;
UInt peripheralNum = 0; /* Such as USB0 */
UInt FatFsDriveNum = 0;

USBMSCHFatFs_Params_init(&usbmschfatfsParams);
usbmschfatfsHandle =
              USBMSCHFatFs_open(peripheralNum, FatFsDriveNum, &usbmschfatfsParams);
if (usbmschfatfsHandle == NULL) {
    System_abort("Error opening USBMSCHFatFs\n");
}
```

Similarly, the close() function unmounts the FatFs volume and unregisters the USBMSCHFatFs disk functions.

```
USBMSCHFatFs_close(usbmschfatfsHandle);
```

The application must ensure the no FatFs or C I/O APIs are called before the USBMSCHFatFs driver has been opened or after the USBMSCHFatFs driver has been closed.

Although the USBMSCHFatFs driver may have been opened, there is a possibility that a USB flash drive may not be present. To ensure that a Task will wait for a USB drive to be present, the USBMSCHFatFs driver provides the USBMSCHFatFs_waitForConnect() function to block the Task's execution until a USB flash drive is detected.

5.13.5 Instrumentation

The USBMSCHFatFs driver logs the following actions using the Log_print() APIs provided by SYS/BIOS:

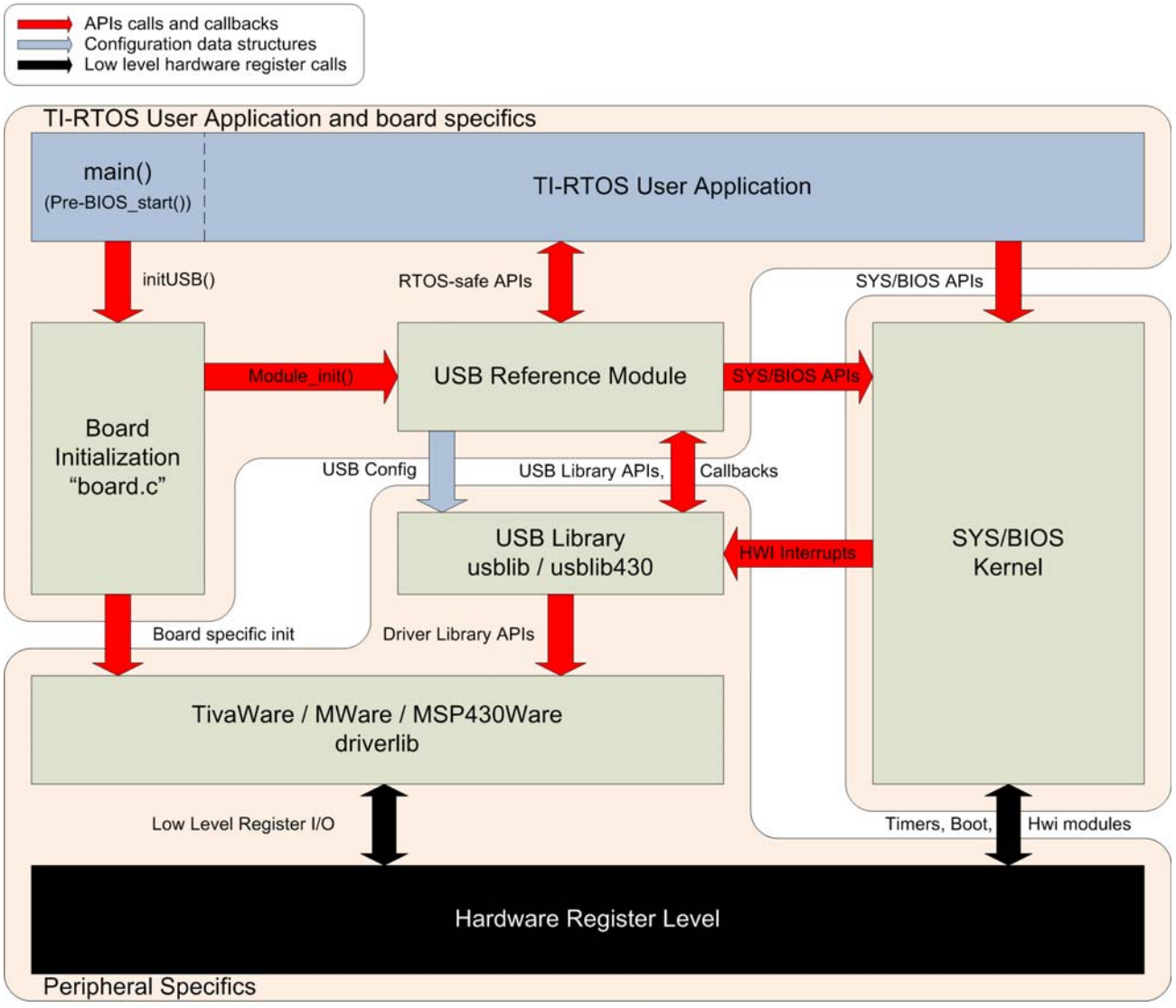
- USB MSC device connected or disconnected.
- USB drive initialized.
- USB drive read or failed to read.
- USB drive written to or failed to write.
- USB status OK or error.

Logging is controlled by the Diags_USER1 and Diags_USER2 masks. Diags_USER1 is for general information and Diags_USER2 is for more detailed information.

5.14 USB Reference Modules

This section provides general guidelines for integrating TI's USB Library into an RTOS environment such as SYS/BIOS. The USB Library incorporated with TI-RTOS is a released version of TivaWare's, MWare's, or MSPWare's USB library. This document does not explain each Ware's USB Library in detail. Instead, it points out important design considerations to consider in application development.

The USB library is highly customizable, and it uses its associated driverlib software to access physical registers on the device, in particular those of the USB controller. To avoid limiting its capabilities by providing a driver that uses the library in a particular way, the TI-RTOS USB examples are structured as reference modules with the expectation that the developer makes the necessary changes for production.



Reference modules are examples that give developers full access, so they can make changes and modifications as needed. The goal of these modules is to provide a starting point for integrating the USB library into a SYS/BIOS application.

5.14.1 USB Reference Modules in TI-RTOS

Each module handles the following items:

- Initializes the USB library and provides the necessary memory allocation, data structures, and callback functions.
- Installs the associated USB interrupt service routine provided with the USB library as a SYS/BIOS HWI object.
  - For MSP43x devices, interrupts are installed via the configuration file (*.cfg). The interrupt service routine was generated using the MSP43x USB Descriptor Tool.
- Provides a set of thread-safe APIs that can be used by one or more SYS/BIOS Tasks.
- Creates the necessary RTOS primitives to protect critical regions and allows Tasks to block when possible.
- For USB Host examples, it also creates separate Task that services the USB stack.

5.14.1.1 Reference module APIs

All of the reference modules include the following APIs. Each module also includes specific APIs unique to that particular module.

- `Module_init()` – This function initializes the USB library, creates RTOS primitives, and installs the proper interrupt handler. For the host examples, it also creates a Task to service the USB controller.
- `Module_waitForConnect()` – This function causes a Task to block when the USB controller is not connected.

5.14.1.2 USB Examples

TI-RTOS has six USB reference examples and one USB FatFs (MSC host) driver. (On-the-go (OTG) examples are not available with TI-RTOS.) The reference examples and driver are as follows:

- **HID Host Keyboard** – Allows a USB keyboard to be connected to the target. Keys pressed on the keyboard are registered on the target.
- **HID Host Mouse** – Allows a USB mouse to be connected to the target. The target registers the overall mouse movements and button presses.
- **HID Device Keyboard** – Causes the target to emulate a USB keyboard. When connected to a workstation, the target functions as another USB keyboard.
- **HID Device Mouse** – Causes the target to emulate a USB mouse when connected to a workstation.
- **CDC Device (Serial)** – The target enumerates a virtual serial COM port on a workstation. This method of communication is commonly used to replace older RS-232 serial adapters.
- **HID Mouse and CDC composite device** – This example enumerates two different USB devices—a HID mouse and a CDC serial virtual COM port.
- **MSC Host** (Mass Storage) – This example uses an actual driver instead of a USB reference module. This driver is modeled after the FatFs driver APIs. This driver allows external mass storage devices such a USB flash drives to be used with FatFs.

5.14.1.3 USB Reference Modules for MSP43x

The USB reference modules for MSP43x devices closely follow the USB examples available in MSPWare. Here are a few items to note:

- Since USB reference modules for MSP43x are imported via TI Resource Explorer, a full copy of the MSPWare's usblib430 USB stack and a set of pre-generated USB descriptor files are copied into the CCS project.
- The generated USB descriptor files are considered user code. These descriptor files have been tested to work with this version of TI-RTOS. Refer to the MSPWare USB documentation if you are generating custom USB descriptors using the USB Descriptor Tool.
- The UsbIsr.c file, which is generated by the USB Descriptor Tool, contains the interrupt service routine needed by MSPWare's usblib430 library. The TI-RTOS USB reference module examples use this interrupt service routine through configuration in the project's *.cfg file.

5.14.2 USB Reference Module Design Guidelines

This section discusses the structure of the USB reference examples.

Design considerations involved in creating these examples included:

- **USB Device Specifics.** Each module contains memory, data structures, and a callback function needed to function properly with the USB library. In device mode, the reference module also includes device descriptors that need to be sent to the USB host controller upon request.
- **OS Primitives.** OS primitives that implement gates, mutexes, and semaphores are used to guard data against race-conditions and reduce unwanted processing time by blocking Tasks when needed.
- **Memory Allocation.** The USB library is designed so that the user application performs all required memory allocation. In a multi-tasked / preempted environment such as SYS/BIOS, it is necessary to protect this memory from other threads. In the reference examples, this is done using the GateMutex module.
- **Callback Functions.** The USB library requires user-provided callback functions to notify the application of events. The USB reference modules provide a set of callback functions to notify the module of status updates. The callback functions update an internal state variable and in some cases post Semaphores to unblock pending Tasks.
- **Interrupts.** Some of the events that trigger callback functions are hardware notifications about the device being connected or disconnected from a USB host controller.

5.14.2.1 Device Mode

USB Device mode examples are rather straightforward. In device mode, the job of the USB library is to respond to the USB host controller with its current state/status. By making USB library API calls in device mode, the example updates information stored in the USB controller's endpoints. This information can be queried by the USB host controller.

5.14.2.2 Host Mode

All USB Host mode examples install a high-priority Task to service the USB controller. This Task calls the USB library's HCDMain() function, which maintains the USB library's internal state machine. This state machine performs actions that include enumerating devices and performing callbacks as described in the Tiva USB library documentation.

To protect the USB library from race conditions between the service Task and other Tasks making calls to the module's APIs, a GateMutex is used.

5.14.2.3 On-The-Go Mode

OTG is not currently used by a USB reference module.

5.15 USB Device and Host Modules

See the USB examples for reference modules that provide support for the Human Interface Device (HID) class (mouse and keyboard) and the Communications Device Class (CDC). This code is provided as part of the examples, not as a separate driver.

The code for the HID keyboard device is in USBKBD.c in the USB Keyboard Device example. This file provides the following functions:

- USBKBD_init()
- USBKBD_waitForConnect()
- USBKBD_getState()
- USBKBD_putChar()
- USBKBD_putString()

The code for the HID keyboard host is in USBKBH.c in the USB Keyboard Host example. This file provides the following functions:

- USBKBH_init()
- USBKBH_waitForConnect()
- USBKBH_getState()
- USBKBH_setState()
- USBKBH_putChar()
- USBKBH_putString()

The code for the HID mouse device is in USBMD.c in the USB Mouse Device example. This file provides the following functions:

- USBMD_init()
- USBMD_waitForConnect()
- USBMD_setState()

The code for the HID mouse host is in USBMH.c in the USB Mouse Host example. This file provides the following functions:

- USBMH_init()
- USBMH_waitForConnect()
- USBMH_getState()

The code for the CDC device is in USBCDCD.c in the F28M3x Demo example, the USB Serial Device example, and the UART Console example. This file provides the following functions:

- USBCDCD_init()
- USBCDCD_waitForConnect()
- USBCDCD_sendData()
- USBCDCD_receiveData()

The code for the CDC mouse is in USBCDCMOUSE.c in the USB CDC Mouse Device example. This file provides the following functions:

- USBCDCMOUSE_init()
- USBCDCMOUSE_receiveData()
- USBCDCMOUSE_sendData()
- USBCDCMOUSE_waitForConnect()

5.16 Watchdog Driver

A watchdog timer can be used to generate a reset signal if a system has become unresponsive. The Watchdog driver simplifies configuring and starting the watchdog peripherals. The watchdog peripheral can be configured with resets either on or off and a user-specified timeout period.

When the watchdog peripheral is configured not to generate a reset, it can be used to cause a hardware interrupt at a programmable interval. The driver provides the ability to specify a user-provided callback function that is called when the watchdog causes an interrupt.

5.16.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

5.16.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the Watchdog driver requires the application to initialize board-specific portions of the watchdog and to provide the Watchdog driver with the Watchdog_config structure.

5.16.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initWatchdog() function that must be called to initialize the board-specific watchdog peripheral settings. This function also calls the Watchdog_init() to initialize the Watchdog driver.

5.16.2.2 Watchdog_config Structure

The <*board*>.c file also declares the Watchdog_config structure. This structure must be provided to the Watchdog driver. It must be initialized before the Watchdog_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.16.3 APIs

In order to use the Watchdog module APIs, the Watchdog header file should be included in an application as follows:

```
#include <ti/drivers/Watchdog.h>
```

The following are the Watchdog APIs:

- **Watchdog_init()** initializes the Watchdog module.
- **Watchdog_Params_init()** initializes the Watchdog_Params struct to its defaults for use in calls to Watchdog_open().
- **Watchdog_open()** opens a Watchdog instance.
- **Watchdog_clear()** clears the Watchdog interrupt flag.
- **Watchdog_setReload()** sets the Watchdog reload value.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.16.4 Usage

The Watchdog driver does not configure board peripherals. This must be done before any calls to the Watchdog driver. The examples include board-specific initWatchdog() functions in the board .c and .h files. Once the watchdog is initialized, a Watchdog object can be created through the following steps:

1. Create and initialize the Watchdog_Params structure.
2. Assign desired values to parameters.
3. Call Watchdog_open().
4. Save the Watchdog_Handle returned by Watchdog_open(). This will be used to interact with the Watchdog object just created.

To have a user-defined function run at the hardware interrupt caused by a watchdog timer timeout, define a Void-type function that takes an argument of type Watchdog_Handle cast as a UArg as follows:

```
typedef Void (*Watchdog_Callback)(UArg);
```

An example of the Watchdog creation process that uses a callback function:

```
Watchdog_Params params;
Watchdog_Handle watchdog;

Board_initWatchdog();

/* Create and enable a Watchdog with resets enabled */
Watchdog_Params_init(&params);
params.resetMode = Watchdog_RESET_ON;
params.callbackFxn = UserCallbackFxn;

watchdog = Watchdog_open(Board_WATCHDOG, &params);
if (watchdog == NULL) {
    /* Error opening watchdog */
}
```

If no Watchdog_Params structure is passed to Watchdog_open(), the default values are used. By default, the Watchdog driver has resets turned on, no callback function specified, and stalls the timer at breakpoints during debugging.

Options for the resetMode parameter are Watchdog_RESET_ON and Watchdog_RESET_OFF. The latter allows the watchdog to be used like another timer interrupt. When resetMode is Watchdog_RESET_ON, it is up to the application to call Watchdog_clear() to clear the Watchdog interrupt flag to prevent a reset. Watchdog_clear() can be called at any time.

5.16.5 Instrumentation

The Watchdog module provides instrumentation data by both making log calls. The Watchdog driver logs the following actions using the Log_print() APIs provided by SYS/BIOS.

- Watchdog_open() success or failure
- Reload value changed

5.17 WiFi Driver

The TI-RTOS WiFi driver implements many elements needed to communicate with a TI Wi-Fi device such as the SimpleLink Wi-Fi CC3100. The WiFi driver uses the TI-RTOS SPI module and implements a state machine to send and receive commands, data, and events to and from a Wi-Fi device.

This driver's APIs let you open a WiFi driver instance to communicate with the Wi-Fi device's host driver without further direct calls to the WiFi driver from the application. TI-RTOS provides host drivers for its supported Wi-Fi devices in `<tirtos_install>\products\tidrivers_<version>\packages\ti\mw\wifi`. Note that this location is different from the location for most other drivers, because this is a middleware driver. In order to use this driver, C files should use the following #include statement:

```
#include <ti/mw/wifi/cc3x00/simplelink/include/simplelink.h>
```

You can configure the driver to allow calling the WiFi driver from a single thread or to be safe to call from multiple threads. The multi-threaded version of host driver consumes more resources than the single-thread version. The WiFi driver supports only one instance of the driver.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.17.1 Static Configuration

See Section 2.4.2 and Section 5.2.1 for information about configuring your application to use the instrumented driver libraries, which are helpful in debugging.

To enable this driver, add the following statement to your application's *.cfg file.

```
var mwConfig = xdc.useModule('ti.mw.Config');
```

Note that this package location is different from the location of most other drivers, because this is a middleware driver.

By default, the WiFi library linked into the project is prebuilt with a version of the WiFi device's host driver that is only safe to call from a single task. You can choose to allow calling the WiFi driver from multiple threads. The multi-threaded version of host driver consumes more resources than the single-thread version.

If you choose the multi-threaded version, internal calls by the WiFi host driver are run from within a Task thread called SelectThread in order to allow other host driver API calls to run while the WiFi driver is waiting for a response. You can configure the priority of the SelectThread task; the default priority is 1, which is just above the priority of the Idle thread.

In addition to library type, the WiFi driver requires the maximum TX and RX data payload sizes to be configured statically. These payload sizes are used by the WiFi module to create appropriately-sized buffers for use by the WiFi driver and Wi-Fi device's host driver. They can be specified graphically as shown in the previous image or textually as follows:

```
WiFi.txPayloadSize = 1468;
WiFi.rxPayloadSize = 1468;
```

In order to use the WiFi driver, your configuration must also include the SPI module. See Section 5.11, *SPI Driver* for details.

5.17.2 Runtime Configuration

As the overview in Section 5.2.2 indicates, the WiFi driver requires the application to initialize board-specific portions of the WiFi driver and provide the WiFi driver with the WiFi _config structure. A SPI_config structure is also required by the WiFi driver.

5.17.2.1 Board-Specific Configuration

The <*board*>.c files contain a <*board*>_initWiFi() function that must be called to initialize the board-specific WiFi peripheral settings. This function also calls WiFi_init() and SPI_init() to initialize the WiFi driver and its resources.

5.17.2.2 WiFi_config Structure

The <*board*>.c file also declares the WiFi_config structure. This structure must be provided to the WiFi driver. It must be initialized before the WiFi_init() function is called and cannot be changed afterwards.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

Note that the SPI_config structure must also be present and initialized before the WiFi driver may be used. See Section 5.11, *SPI Driver* for details.

5.17.3 APIs

In order to use the WiFi module APIs, the WiFi header file should be included in an application as follows:

```
#include <ti/mw/wifi/cc3x00/simplelink/include/simplelink.h>
```

The following are the WiFi APIs:

- **WiFi_init()** initializes the WiFi module.
- **WiFi_Params_init()** initializes the WiFi_Params struct to its defaults for use in calls to WiFi_open().
- **WiFi_open()** opens a WiFi instance.
- **WiFi_close()** closes a WiFi instance.

See the help in `<tirtos_install>\products\tidrivers_<version>\docs\html\index.html` for details.

5.17.4 Usage

Before any APIs from the Wi-Fi device's host driver can be used, the application must open the WiFi driver. The WiFi_open() function configures the SPI driver, creates necessary interrupts, and registers a callback to inform the application of events that may occur on the Wi-Fi device. Once WiFi_open() has returned, host driver APIs may be used to start sending commands and data to the Wi-Fi device.

```
WiFi_Params params;
WiFi_Handle handle;

/* Open WiFi */
WiFi_Params_init(&params);
params.bitRate = 5000000;   /* Set bit rate to 5 MHz */
handle = WiFi_open(Board_wifiIndex, Board_spiIndex, userCallback, &params);
if (handle == NULL) {
    System_abort("Error opening WiFi\n");
}

/* Host driver APIs such as socket() may now be called. */
```

The WiFi_close() function should be called when use of the host driver APIs is complete.

5.17.5 Instrumentation

The WiFi driver provides instrumentation data by making Log calls.

5.17.5.1 Logging

The WiFi driver is instrumented with Log events that can be viewed with UIA and RTOS Analyzer. Diags masks can be turned on and off to provide granularity to the information that is logged. Use Diags_USER1 to see general Log events. The WiFi driver logs the following actions using the Log_print() APIs provided by SYS/BIOS.

- WiFi device enabled or disabled
- Interrupts enabled or disabled
- WiFi_open() success or failure
- WiFi_close() success
- Send or receive buffer overrun
- Reads and writes to WiFi device completed
- SPI_transfer() failure

TI-RTOS Utilities

This chapter provides information about utilities provided by TI-RTOS.

Topic **Page**

6.1 Overview 96
6.2 UARTMon Module 96
6.3 UART Example Implementation 102

6.1 Overview

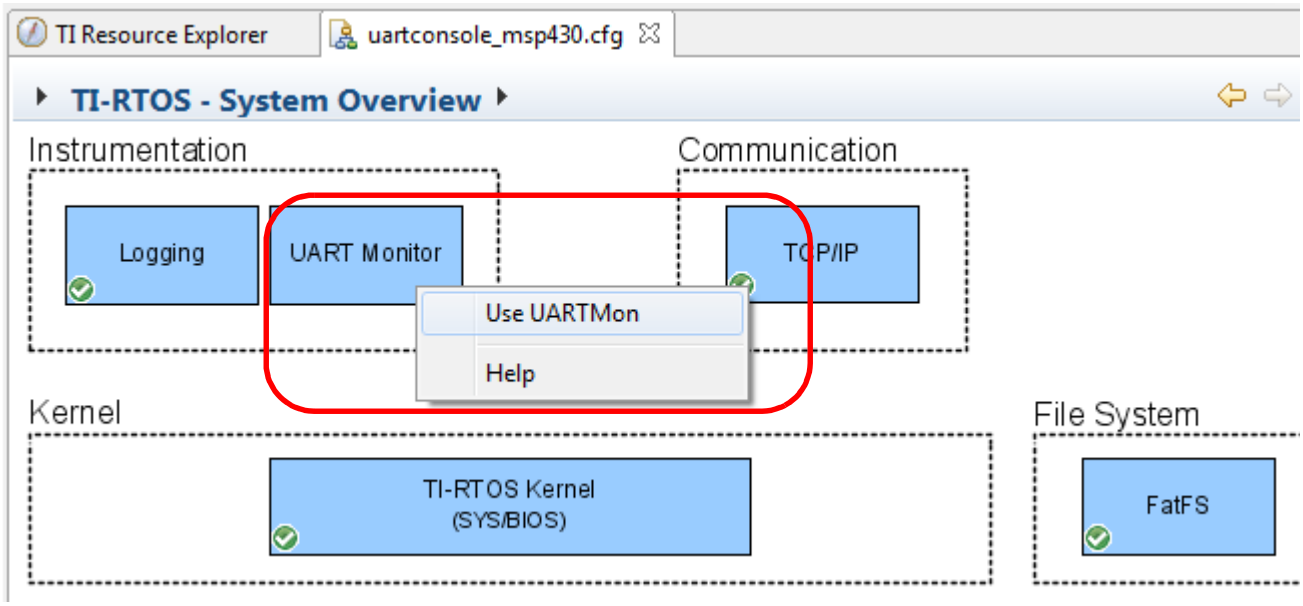
Utilities for use with TI-RTOS are provided in the `<tirtos_install>\packages\ti\tirtos\utils` directory. This chapter describes such modules.

6.2 UARTMon Module

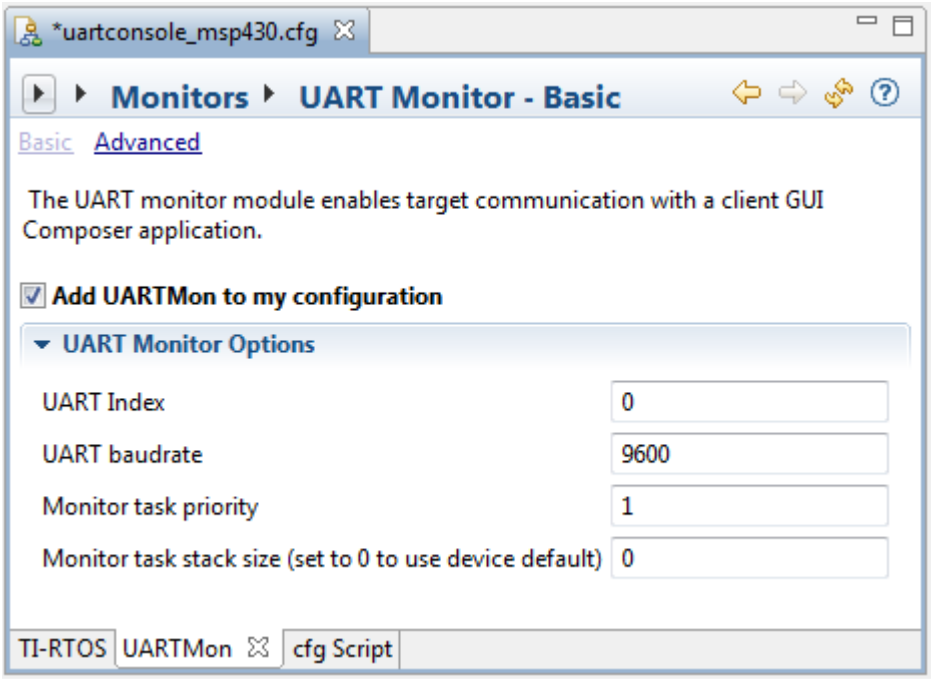
The UARTMon module (ti.tirtos.utils.UARTMon) enables host communication with a target device using the target's UART. The target device can respond to requests to read and write memory at specified addresses. CCS includes features that allow you to leverage this utility to monitor the target device with the Debug view or with GUI Composer.

The GPIO example enables the UARTMon module. See the readme file in the example project for information about the example.

To use UARTMon in your application, open the project's *.cfg file with the XGCONF Configuration Editor. Select the TIRTOS module, and choose the **System Overview** to see the diagram below. Right-click on the **UART Monitor** module and select **Use UARTMon** from the drop-down to add it to your application. No extra user code is needed on the target to use this utility.

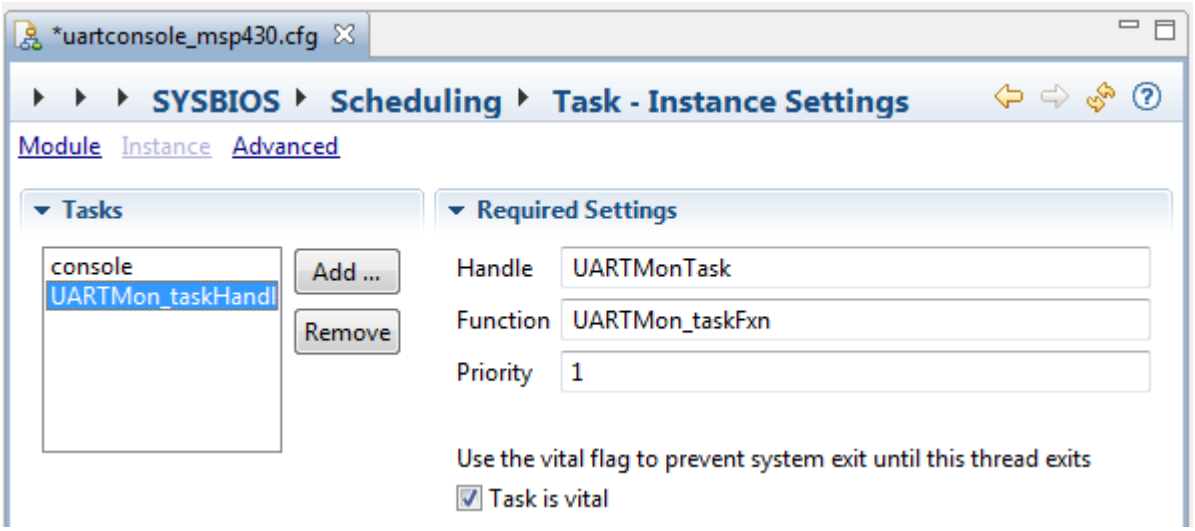


To configure this module, select **UARTMon** in the Outline pane to view its configuration page.



The UART Index property is the board index of the UART peripheral to be used as a monitor. In the Board.h file, Board_UART0 has an index of 0 and Board_UART1 has an index of 1. Other options that can be adjusted are the baud rate for the UART and the priority and stack size for the Task that performs the monitoring.

Once UARTMon is enabled in your configuration, a task called UARTMonTask is automatically created and can be seen among your task instances in XGCONF Configuration Editor as shown below. This task will also show up in ROV when you are debugging.



The UARTMon module has no C APIs.

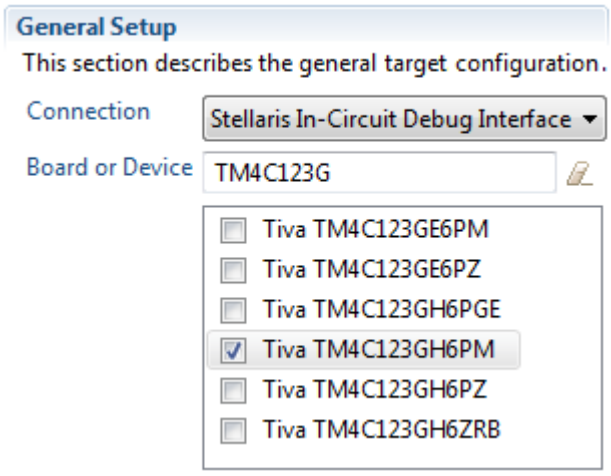
The GPIO Interrupt examples for the MSP-EXP430F5529LP and Tiva EK-TM4C123GXL LaunchPad boards have UARTMon enabled.

6.2.1 UARTMon with CCS Tools

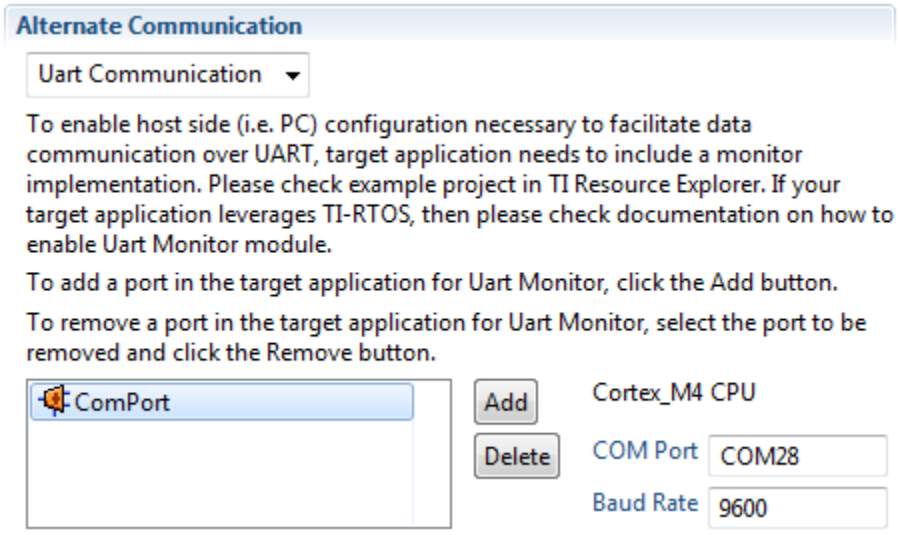
CCS supports UART communication alongside a JTAG connection. This section explains how to create the necessary target configuration and run the debug session.

Follow these steps to create a target configuration file that allows you to use a UART Monitor connection in addition to your existing JTAG connection:

1. Choose **File > New > Target Configuration File** from the CCS menus.
2. Type a filename for this configuration, and click **Finish**. The target configuration will be stored in a *.ccxml file.
3. In the Target Configuration window, select the main connection used to communicate with the device. For example, for Stellaris and Tiva boards, you might use the **Stellaris In-Circuit Debug Interface**.
4. To specify the device, begin typing the name of your device. The filter field shows only those devices that match what you type.
5. When you see your device, check the box next to it.
6. If the device you select has a UART on its board, you see the Alternate Communication area to the right of the device selection.



7. Make sure UART Communication is selected in the drop-down list. It is typically the only option.

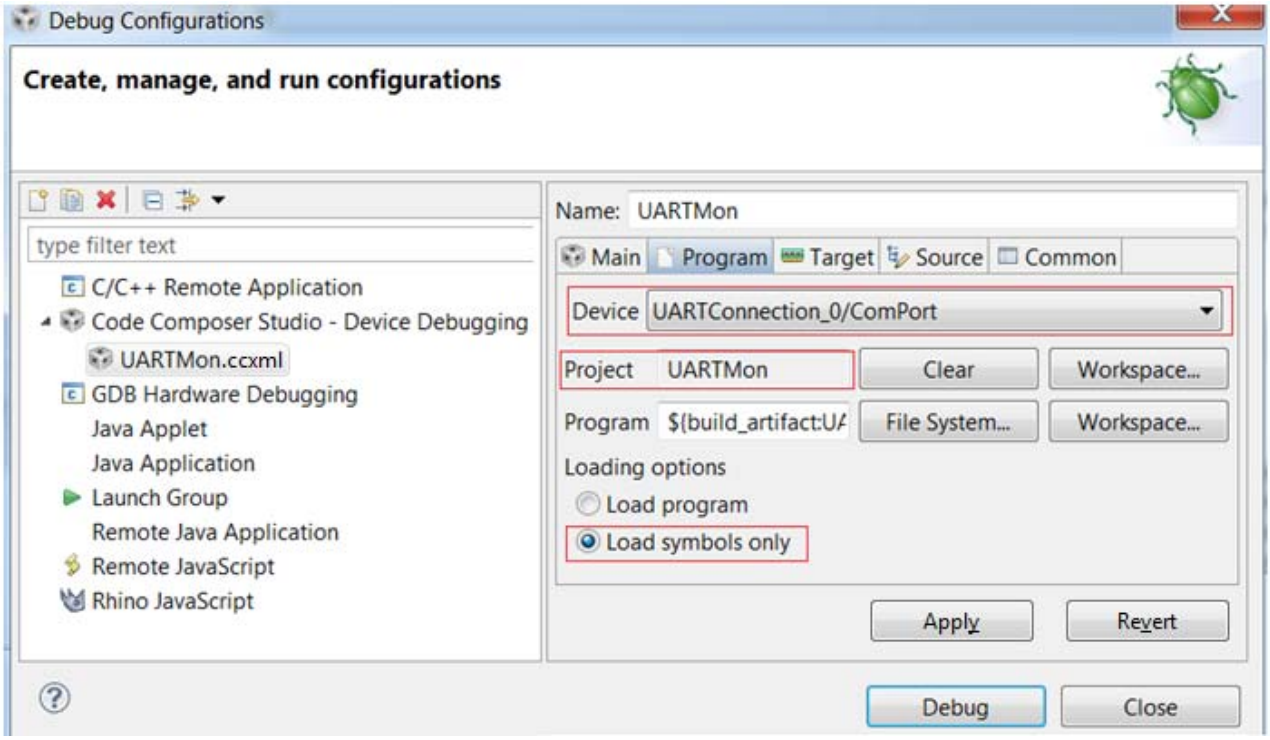


Note: When using a driver based on the Tiva In-Circuit Debug Interface (ICDI) or an MSP430 driver, the COM Port must have the same number as one identified in the Windows Device manager. When using a XDSv2 USB Emulator, there is no such limitation; the emulator can create a new COM port.

8. Click the **Add** button and select the **ComPort** that is created.
9. Modify the **COM Port** and **Baud Rate** as needed.
10. Click **Save** to save your target configuration file.

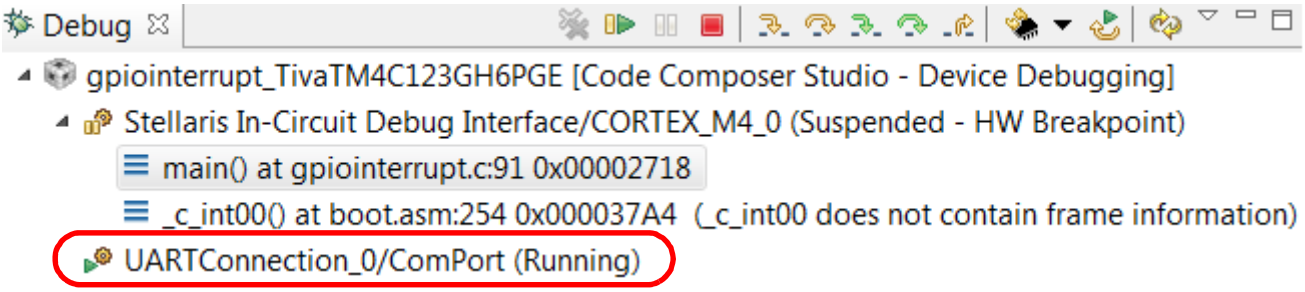
To run and debug a program that has the UARTMon module enabled, follow these steps:

1. Build your application if you have not already done so.
2. Choose **Run > Debug Configurations** from the CCS menus.

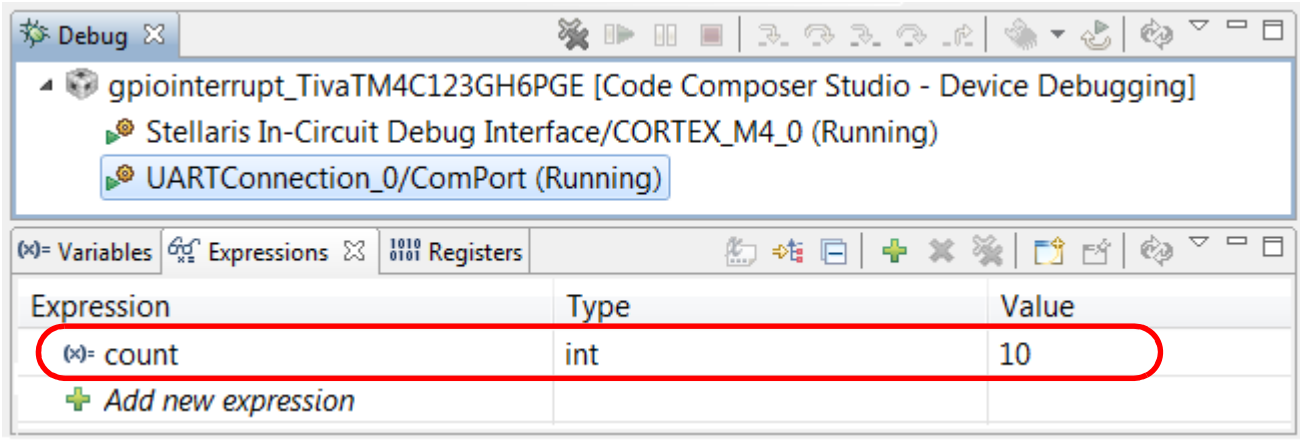


3. Expand the **Code Composer Studio - Device Debugging** category and select the target configuration you just created. (If your target configuration file is not listed under the Device Debugging category, close this dialog, launch the target configuration, and then re-open the Debug Configuration dialog to cause the new target configuration to be listed.)
4. In the right page of the dialog, choose the **Program** tab.
5. Make sure the interface or emulator used for non-UART communication is selected in the **Device** drop-down list. For example, for Tiva and Stellaris boards, you might be using the **Stellaris In-Circuit Debug Interface**.
6. If your project is not already selected for the non-UART interface or emulator, click **Workspace** in the **Project** row and select the project you want to debug. Click **OK**.
7. Select the **Load program** loading option for this device.
8. Move back up to the **Device** drop-down list. This time, select the **UARTConnection_0/ComPort** option in the **Device** drop-down list.
9. If your project is not already selected for the UART connection, click **Workspace** in the **Project** row and select the project you want to debug. Click **OK**.
10. Select the **Load symbols only** loading option for this device. If you skip this step, the debugger will attempt to program the device using the UART connection.
11. Click **Debug**.

12. In the Debug view, the UARTConnection is listed among the available connections and is marked as Running, meaning that the COM Port specified is now being listened to.



13. If you configured the project to enable UARTMon as described in the previous section, you can select the UART connection to watch variables and expressions the same way you would with an emulator.



In this example, the count variable in the Expressions window is being watched using the UART Connection when that item is selected. If the Stellaris In-Circuit Debug Interface is selected, that connection is used to watch the same variable.

6.2.2 GUI Composer

GUI Composer is a tool in CCS for creating custom graphical user interfaces that interact with your target application. You can use it to create interface widgets that are bound to variables in the running target and update themselves accordingly. With UARTMon included in your application, GUI Composer can use the UART connection to interact with the running target.

For example, the count variable shown in the previous section can be bound to a dial widget in GUI Composer. When the value of the count variable changes on the target, the UART connection is used to change the reading on the dial. In addition, you can use the dial to set the value of the count variable on the target. To learn more about GUI Composer, see the Texas Instruments Wiki.

6.3 UART Example Implementation

The `UARTUtils.c` file provides an example implementation using a UART. Three of the System functions are initialized (the others default to NULL) in the `uartconsole.cfg` file. The example uses the SysCallback module provided by XDCtools.

The configuration source is as follows. These statements create the same configuration as the graphical settings shown in Section 6.2:

```
var SysCallback = xdc.useModule('xdc.runtime.SysCallback');
SysCallback.abortFxn = "&UARTUtils_systemAbort";
SysCallback.putchFxn = "&UARTUtils_systemPutch";
SysCallback.readyFxn = "&UARTUtils_systemReady";
System.SupportProxy = SysCallback;
```

In `uartconsole.c`, main() does the following

1. Calls the board-specific setupUART() function to initialize the UART peripheral.
2. Calls UARTUtils_systemInit() as follows to initialize the UART 0 software. After the UARTUtils_systemInit function is called, any System_printf output will be directed to UART 0.

```
/* Send System_printf to the UART 0 also */
UARTUtils_systemInit(0);
```

Using the FatFs File System Drivers

This chapter provides an overview of FatFs and discusses how FatFs is interconnected and used with TI-RTOS.

Topic **Page**

7.1	Overview	103
7.2	FatFs and TI-RTOS	104
7.3	Using FatFs	105
7.4	Cautionary Notes	107

7.1 Overview

FatFs is a free, 3rd party, generic File Allocation Table (FAT) file system module designed for embedded systems. The module is available for download at http://elm-chan.org/fsw/ff/00index_e.html along with API documentation explaining how to use the module. Details about the FatFs API are not discussed here. Instead, this section gives a high-level explanation about how it is integrated with TI-RTOS.

The FatFs drivers provided by TI-RTOS enable you to store data on removable storage media such as Secure Digital (SD) cards and USB flash drives (Mass Storage Class). Such storage may be a convenient way to transfer data between embedded devices and conventional PC workstations.

7.2 FatFs and TI-RTOS

TI-RTOS provides a FatFS module and extends this module by supplying "FatFs" drivers that link into the FatFs implementation. The FatFS module is aware of the multi-threaded environment and protects itself with OS primitives supplied by SYS/BIOS.

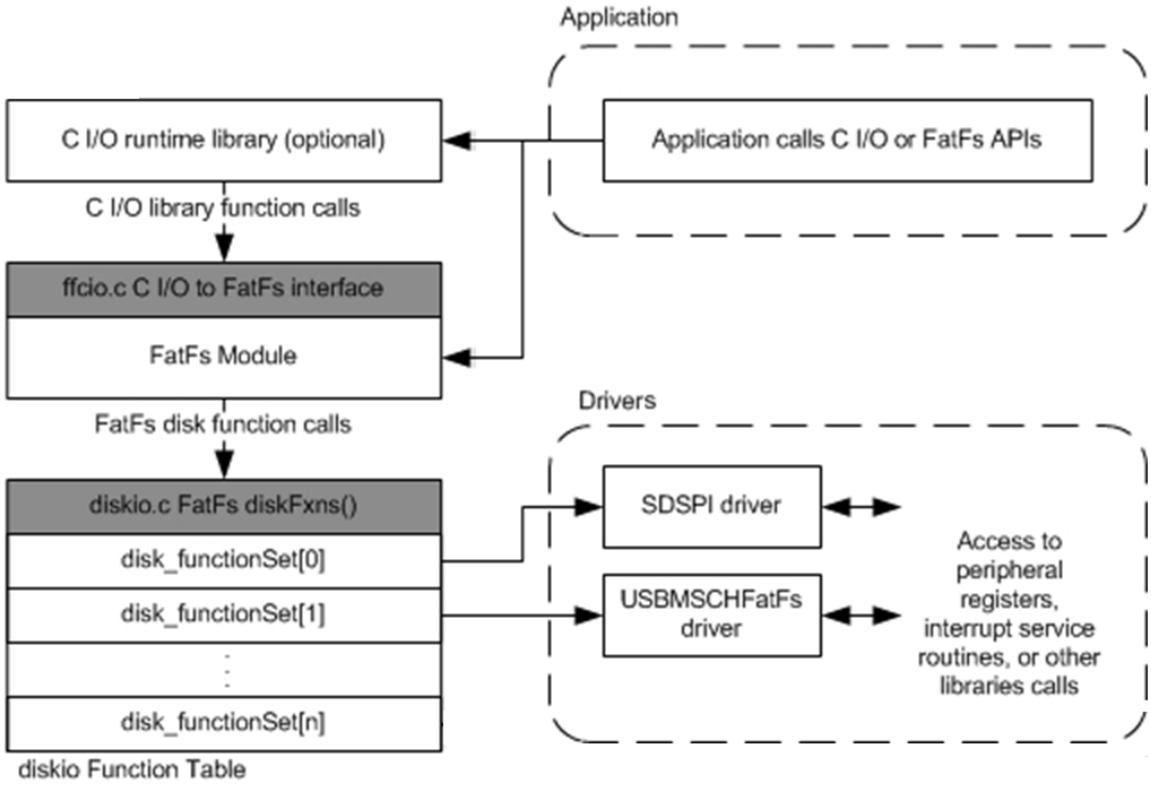


Figure 7-1 FatFs data flow

From the start of this data flow to the end, the components involved behave as follows:

- **Application.** The top application layer calls the basic open, close, read, and write functions. Users who are familiar with FatFs can easily use the FatFs API, which is documented at the module's download site. Alternatively, the application can also connect the C input/output (C I/O) runtime support library in TI's Code Generation Tools to FatFs. You can call familiar functions such as fopen(), fclose(), fread(), and fwrite(). Functionally, the C I/O interface and the FatFs APIs perform the same operations (with a few exceptions described in Section 7.3).
- **FatFS module.** The next layer, the ti.mw.fatfs.FatFS module, is provided as part of TI-RTOS. This module handles the details needed to manage and use the FAT file system, including the media's boot sector, FAT tables, root directories, and data regions. It also protects its functions in a multi-threaded environment. Internally, the FatFS module makes low-level data transfer requests to the Disk IO functions described on the FatFs product web page. Implementations of this set of functions are called "FatFs drivers" in this document.
- **diskIO Function table.** To allow products to provide multiple FatFs drivers, the FatFS module contains a simple driver table. You can use this to register multiple FatFs drivers at runtime. Based on the drive number passed by FatFs, the driver table routes FatFs calls to a particular FatFs driver.

- **FatFs drivers.** The last layer in Figure 7-1 is the FatFs drivers. TI-RTOS comes with pre-built FatFs drivers that plug into the FatFS module. A FatFs driver has no knowledge of the internal workings of FatFs. Its only task is to perform disk-specific operations such as initialization, reading, and writing. The FatFs driver performs read and write operations in data block units called sectors (commonly 512 bytes). Details about writing data to the device are left to the particular FatFs driver, which typically accesses a peripheral's hardware registers or uses a driver library.

7.3 Using FatFs

The subsections that follow show how to configure FatFs statically, how to prepare the FatFs drivers for use in your application, and how to open files. For details about performing other file-based actions once you have opened a file, see the FatFs APIs described on http://elm-chan.org/fsw/ff/00index_e.html in the "Application Interface" section or the standard C I/O functions.

The TI-RTOS F28M3x Demo example and all 3 FatFs File Copy examples use FatFs with the SDSPI driver. The FatSD USB Copy example uses the USBMSCHFatFs driver.

7.3.1 Static FatFS Module Configuration

To incorporate the FatFS module into an application, simply "use" this module in a configuration (.cfg) file. You can do this by searching the **Available Products** list in the XGCONF Configuration Editor for FatFS, selecting the FatFS module, and checking the **Enable FAT File System in My Application** box. Or, you can add the following statement to the .cfg file.

```
var FatFS = xdc.useModule('ti.mw.fatfs.FatFS');
```

Note: The name of the product and the drivers is "FatFs" with a lowercase "s". The name of the module is "FatFS" with an uppercase "S". If you are using a text editor to write configuration statements, be sure to use the uppercase "S". If you are using the XGCONF Configuration Editor to edit your configuration graphically, the correct capitalization is used automatically.

By default, the prefix string used in C I/O fopen() calls that uses this module is "fat" and no RAM disk is created. You can these defaults by modifying the FatFS module properties.

For example, you can change the C I/O prefix string used in fopen() calls by adding this line to the .cfg file:

```
FatFS.fatfsPrefix = "newPrefix";
```

The application would then need to use the prefix in C I/O fopen() calls as follows:

```
src = fopen("newPrefix:0:signal.dat", "w");
```

See the online help for the module for more details about FatFS configuration.

You will also need to configure the FatFs driver or drivers you want to use. See Section 5.10, *SDSPI Driver* and Section 5.13, *USBMSCHFatFs Driver* for details.

7.3.2 Defining Drive Numbers

Calls to the open() functions of individual FatFs drivers—for example, SDSPI_open()—require a drive number argument. Calls to the C I/O fopen() function and the FatFs APIs also use the drive number in the string that specifies the file path. The following C code defines driver numbers to be used in such functions:

```
/* Drive number used for FatFs */
#define SD_DRIVE_NUM           0
#define USB_DRIVE_NUM          1
```

Here are some statements from the FatSD USB Copy example that use these drive number definitions. Note that STR(SD_DRIVE_NUM) uses a MACRO that expands SD_DRIVE_NUM to 0.

```
SDSPI_Handle sdspiHandle;
SDSPI_Params sdspiParams;
FILE *src;
const Char  inputfilesd[]  = "fat:"STR(SD_DRIVE_NUM)":input.txt";

/* Mount and register the SD Card */
SDSPI_Params_init(&sdspiParams);
sdspiHandle = SDSPI_open(Board_SDSPI0, SD_DRIVE_NUM, &sdspiParams);

/* Open the source file */
src = fopen(inputfilesd, "r");
```

7.3.3 Preparing FatFs Drivers

In order to use a FatFs driver in an application, you must do the following:

- **Include the header file for the driver.** For example:

  ```
  #include <ti/drivers/SDSPI.h>
  ```

- **Run the initialization function for the driver.** All drivers have init() functions—for example, SDSPI_init()—that need to be run in order to set up the hardware used by the driver. Typically, these functions are run from main(). In the TI-RTOS examples, a board-specific initialization function for the driver is run instead of running the driver's initialization function directly. For example:

  ```
  Board_initSDSPI();
  ```

- **Open the driver.** The application must open the driver before the FatFs can access the drive and its FAT file system. Similarly, once the drive has been closed, no other FatFs calls shall be made. All drivers have open() functions—for example, SDSPI_open()—that require a drive number to be passed in as an argument. For example:

  ```
  sdspiHandle = SDSPI_open(Board_SDSPI0, SD_DRIVE_NUM, NULL);
  ```

See Section 5.10, *SDSPI Driver* and Section 5.13, *USBMSCHFatFs Driver* for details about the FatFs driver APIs.

7.3.4 Opening Files Using FatFs APIs

Details on the FatFs APIs can be found at http://elm-chan.org/fsw/ff/00index_e.html in the "Application Interface" section.

The drive number needs to be included as a prefix in the filename string when you call f_open() to open a file. The drive number used in this string needs to match the drive number used to open the FatFs driver. For example:

```
res = f_open(&fsrc, "SD_DRIVE_NUM:source.dat", FA_OPEN_EXISTING | FA_READ);

res = f_open(&fdst, "USB_DRIVE_NUM:destination.dat", FA_CREATE_ALWAYS | FA_WRITE);
```

A number of other FatFs APIs require a path string that should include the drive number. For example, f_opendir(), f_mkdir(), f_unlink(), and f_chmod().

Although FatFs supports up to 10 (0-9) drive numbers, the diskIO function table supports only up to 4 (0-3) drives. You can modify this default by changing the definition of _VOLUMES in the ffconf.h file in the FatFS module. You will then need to rebuild TI-RTOS as described in Section 8.1.

It is important to use either the FatFs APIs or the C I/O APIs for file operations. Mixing the APIs in the same application can have unforeseen consequences.

7.3.5 Opening Files Using C I/O APIs

The C input/output runtime implementation for FatFs works similarly to the FatFs API. However, you must add the file name prefix configured for the FatFS module ("fat" by default) and the logical drive number as prefixes to the filename. The file name prefix is extracted from the filename before it gets passed to the FatFs API.

In this example, the default file name prefix is used and the drive number is 0:

```
fopen("fat:0:input.txt", "r");
```

It is important to use either the FatFs APIs or the C I/O APIs for file operations. Mixing the APIs in the same application can have unforeseen consequences.

7.4 Cautionary Notes

FatFs drivers perform data block transfers to and from physical media. Depending on the FatFs driver, writing to and reading from the disk could prevent lower-priority tasks from running during that time. If the FatFs driver blocks for the entire transfer time, only higher-priority SYS/BIOS Tasks, Swis or Hwis can interrupt the Task making FatFs calls. In such cases, the application developer should consider how often and how much data needs to be read from or written to the media.

By default the FatFS module keeps a complete sector buffered for each opened file. While this requires additional RAM, it helps mitigate frequent disk operations when operating on more than one file simultaneously.

The FatFS implementation allows up to four unique volumes (or drives) to be registered and mounted.

Rebuilding TI-RTOS

This chapter describes how and when to rebuild TI-RTOS and components of TI-RTOS.

Topic **Page**

8.1 Rebuilding TI-RTOS 109
8.2 Rebuilding MSPWare's driverlib for TI-RTOS and Its Drivers 111
8.3 Rebuilding Individual Components 112

8.1 Rebuilding TI-RTOS

In most cases, you will not need to rebuild the TI-RTOS libraries. Pre-built libraries for CCS, IAR, and GCC are provided when you install TI-RTOS. However, if you want to change the compiler or linker options, you may need to rebuild the libraries.

8.1.1 Building TI-RTOS for CCS

By default, TI-RTOS is ready to be rebuilt for use with CCS from a top-level make file called `tirtos.mak`.

If TI-RTOS is installed in `c:\ti`, you can print a list of available make rules by running the following command from a command shell window:

```
% cd <tirtos_install>
% ../<xdctools>/gmake -f tirtos.mak
```

To rebuild the TI-RTOS drivers and several of its included components (SYS/BIOS, NDK, and UIA), for example, you can run the following:

```
% ../<xdctools>/gmake -f tirtos.mak all
```

If you installed CCS and TI-RTOS in a location other than `c:\ti`, you can edit the definition of DEFAULT_INSTALL_DIR in `tirtos.mak` to point to this location. Note that all other product installation locations are defined relative to the DEFAULT_INSTALL_DIR, but you can adjust them as necessary. You can also pass in installation locations as necessary. For example to use a different location for XDCtools, do the following:

```
% ../<xdctools>/gmake -f tirtos.mak XDCTOOLS_INSTALL_DIR=c:/ti/xdctools_version
```

The following list (from TI-RTOS for MSP43x for example) shows items you can change and sample values. The `tirtos.mak` file differs for each device family. The version numbers in your copy of the `tirtos.mak` file will match the versions of the components installed with TI-RTOS.

```
CCS_BUILD ?= true
DEFAULT_INSTALL_DIR := c:/ti
ti.targets.msp430.elf.MSP430X ?=$(DEFAULT_INSTALL_DIR)/ccsv6/tools/compiler/msp430_4.4.3

XDCTOOLS_INSTALL_DIR ?= $(DEFAULT_INSTALL_DIR)/xdctools_3_31_01_33_core
export XDCTOOLS_JAVA_HOME   ?= $(DEFAULT_INSTALL_DIR)/ccsv6/eclipse/jre

TIRTOS_INSTALL_DIR    ?= $(DEFAULT_INSTALL_DIR)/tirtos_msp43x_2_20_00_21
TIDRIVERS_INSTALL_DIR ?= $(TIRTOS_INSTALL_DIR)/products/tidrivers_msp43x_2_15_00_15
BIOS_INSTALL_DIR      ?= $(TIRTOS_INSTALL_DIR)/products/bios_6_45_00_15
UIA_INSTALL_DIR       ?= $(TIRTOS_INSTALL_DIR)/products/uia_2_00_02_27
MSPWARE_INSTALL_DIR   ?= $(TIRTOS_INSTALL_DIR)/products/MSPWare_2_00_00_40c
MSP430HEADERS         ?= $(DEFAULT_INSTALL_DIR)/ccsv6/ccs_base/msp430/include
MSP432HEADERS         ?= $(DEFAULT_INSTALL_DIR)/ccsv6/ccs_base/arm/include
```

If you are rebuilding on Linux, change all of the Windows paths in the `tirtos.mak` file to Linux paths.

The `CCS_BUILD?=true` flag in the `tirtos.mak` file causes TI-RTOS to be rebuilt for CCS by default. Other supported tool-chains (such as IAR) also have flags that can be turned on to build for them as well. If these are not needed, keep them turned off for a faster build.

8.1.2 Building TI-RTOS for IAR

By default, TI-RTOS is not rebuilt for use with IAR when you run the top-level `tirtos.mak` make file. To rebuild TI-RTOS for IAR Embedded Workbench, follow these steps:

1. Edit the `tirtos.mak` file and find the following lines:

```
IAR_BUILD ?= false
IAR_MSP430_INSTALL_DIR ?= c:/iar
IAR_ARM_INSTALL_DIR    ?= c:/iar
```

2. Set the IAR_BUILD flag to true in `tirtos.mak`. Alternately, you can pass a different value on the make command line as follows:

```
% ../<xdctools>/gmake -f tirtos.mak all IAR_BUILD=true
```

3. Change the IAR compiler installation directory to match the location where you installed IAR. Alternately, you can pass a different value on the make command line as follows:

```
% ../<xdctools>/gmake -f tirtos.mak all IAR_MSP430_INSTALL_DIR=YOUR_PATH
```

4. Modify the installation locations as needed for the components of TI-RTOS (SYS/BIOS, NDK and UIA) that you want to rebuild for IAR.

For a faster build, you can turn off TI-RTOS building for CCS by setting the CCS_BUILD flag to false.

8.1.3 Building TI-RTOS for GCC

By default, TI-RTOS is not rebuilt for GCC when you run the top-level `tirtos.mak` make file. The GCC code generator used is the Linaro distribution gcc-arm-none-eabi-4_7-2012q4 version that ships with CCS. To rebuild TI-RTOS with GCC, follow these steps:

1. Edit the `tirtos.mak` file and find the following lines:

```
GCC_BUILD ?= false
GCC_INSTALL_DIR := $(CCS_COMPILERS_DIR)
```

2. Set the above GCC_BUILD flag to true in `tirtos.mak`. Alternately, you can pass a value on the make command line as follows:

```
% ../<xdctools>/gmake -f tirtos.mak all GCC_BUILD=true
```

3. If you installed CCS in a location other than c:\ti, change the path for GCC_INSTALL_DIR to specify the correct location. Alternately, you can pass a different value on the make command line as follows:

```
% ../<xdctools>/gmake -f tirtos.mak all GCC_INSTALL_DIR=YOUR_PATH
```

4. Modify the installation locations as needed for the components of TI-RTOS (SYS/BIOS, NDK and UIA) that you want to rebuild for GCC.

For a faster build, you can turn off TI-RTOS building for CCS and IAR by setting the CCS_BUILD and IAR_BUILD flags to false.

8.1.4 Rebuilding the TI-RTOS Drivers with the Debug Profile

By default, the TI-RTOS driver libraries are rebuilt with the **release** profile. The release profile sets compiler flags to optimize libraries for performance. During the compilation process, this causes the compiler to perform several operations to achieve better performance, one of which is to reorganize code. Reorganized code is difficult to debug when stepping through code.

If you would like to step through driver library code during debugging, you can rebuild the driver libraries without optimization by following these steps:

1. Open the `tirtos.mak` file and find lines similar to the following. (This example is from TI-RTOS for SimpleLink.)

```
XDCARGS= \
    profile='release' \
    CCWareDir='$(CCWARE_INSTALL_DIR)' \
```

2. Change the profile parameter to `'debug'`:

```
XDCARGS= \
    profile='debug' \
    CCWareDir='$(CCWARE_INSTALL_DIR)' \
```

3. Rebuild the TI-RTOS drivers as follows:

```
% cd <tirtos_install>
% ../<xdctools>/gmake -f tirtos.mak clean-drivers drivers
```

Your applications must be rebuilt to use the non-optimized TI-RTOS driver library. Once debugging is complete, repeat the steps above setting `profile='release'` to return to the optimized library.

8.2 Rebuilding MSPWare's driverlib for TI-RTOS and Its Drivers

The TI-RTOS drivers for MSP43x depend on MSPWare's driverlib as an abstraction layer to access peripheral registers. This level of abstraction promotes code reusability and scales well for TI-RTOS drivers, because device specifics are stored in driverlib.

To reduce the build time of CCS projects and to be consistent with other TI driverlib components, MSPWare's driverlib source files have been compiled into a library in the TI-RTOS installation.

TI-RTOS allows you to build for either MSP430, MSP432, or both. Set one of the following definitions in the `tirtos.mak` file to false if you do not want to build for that target.

```
MSP430_BUILD ?= true
MSP432_BUILD ?= true
```

TI-RTOS provides prebuilt TI-RTOS drivers and prebuilt MSPWare driverlib libraries only for the MSP430F5529, MSP430FR5969, MSP430FR5994, MSP430FR6989, and MSP432P401R. Libraries for other MSP43x devices can be added by editing the `tirtos.mak` file. To build TI-RTOS drivers for other

devices, add those devices to the MSP430DEVLIST or MSP432DEVLIST variable (with spaces between the devices in the list). For example, the following modification to the `tirtos.mak` file causes MSPWare's driverlib and TI-RTOS drivers to be built for the MSP430F5529, MSP430F6779, and MSP432P401R.

```
# To build TI-RTOS driver libraries for other MSP430 devices; simply append the
# device names to MSP430DEVLIST (separated by whitepsaces)
# MSP430DEVLIST := \
#     MSP430F5529 \
#     MSP430F5527 \
#     MSP430F6459 \
#     MSP430F6989 \
#     etc...

MSP430DEVLIST := MSP430F5529 MSP430F6779
...
MSP432DEVLIST := MSP432P401R
```

After updating the necessary variables, rebuild the TI-RTOS drivers as follows:

```
% cd <tirtos_install>
% ../<xdctools>/gmake -f tirtos.mak drivers
```

8.3 Rebuilding Individual Components

The MWare and TivaWare rebuilding mechanism is substantially different from the TI-RTOS rebuilding mechanism. See the documentation for these products for details.

Driver libraries in the versions of MWare and TivaWare distributed with TI-RTOS have been rebuilt. For details, see the TI-RTOS.README file in the top-level folder of the MWare and TivaWare components within the TI-RTOS installation.

Memory Usage with TI-RTOS

This chapter provides links to information about memory usage.

Topic		Page
9.1	Memory Footprint Reduction	113
9.2	Networking Stack Memory Usage	125

9.1 Memory Footprint Reduction

Many configuration parameters impact the size (both code and data) of a TI-RTOS application. This section discusses the approaches TI-RTOS takes to minimize the size of its examples. For a more detailed discussion on how to reduce the size of the kernel, please refer to the *SYS/BIOS User's Guide* (SPRUEX3) appendix on "Minimizing the Application Footprint."

The TI-RTOS examples are divided into the following types of examples:

- **Peripheral Examples.** These examples are designed to demonstrate the usage of a peripheral or feature. These examples are designed to have a small footprint and make use of many of the strategies described in this section.
- **Demo Examples.** The demo examples are "kitchen-sink" examples; that is, they use a wide variety of features. There is no overall design of the memory use strategy for these demos. Decisions were made to allow the program to fit into the available memory of the target device, while still showcasing multiple peripherals and features.
- **Empty Examples.** Two different "Empty" examples can be created with the New Project Wizard if you are using CCS. These examples are intended as a starting point for new development. The two different types of "Empty" projects are:
  - **Empty (Minimal) Project:** Disables kernel features and debug capabilities to minimize the footprint.
  - **Empty Project:** Enables more kernel features and debug capabilities at the cost of a larger footprint.

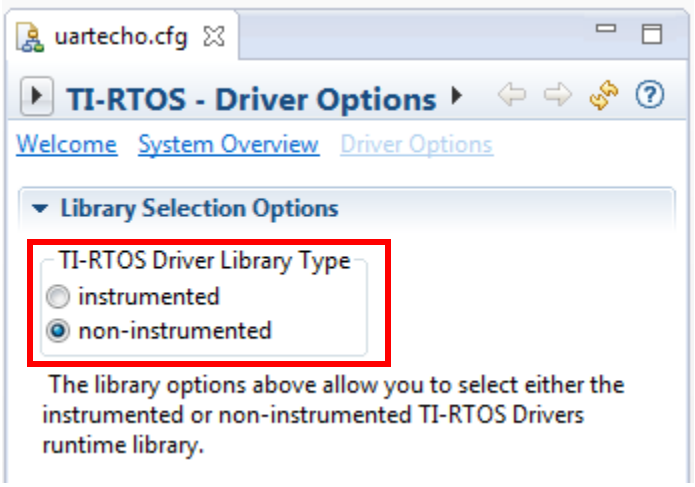
The footprint reduction approaches described in this section are generally not used in the "Empty" examples, but are used in the "Empty (Minimal)" examples. The exceptions are that the SysMin module and the custom BIOS library (with asserts and logging enabled) are used in the Empty examples. Any other configuration changes described here can be made to the Empty examples if needed.

The peripheral examples (available in TI Resource Explorer in CCS) are designed to have a small footprint. The UART examples are exceptions to this rule, because along with UART functionality they are intended to show various approaches for debugging an application.

The following configuration changes help reduce both the data and code footprint in the TI-RTOS peripheral examples. You may want to use these strategies in your own applications.

For most configuration changes, both the graphical (XGCONF) and script-based methods of modifying the configuration are shown. Use whichever method you prefer.

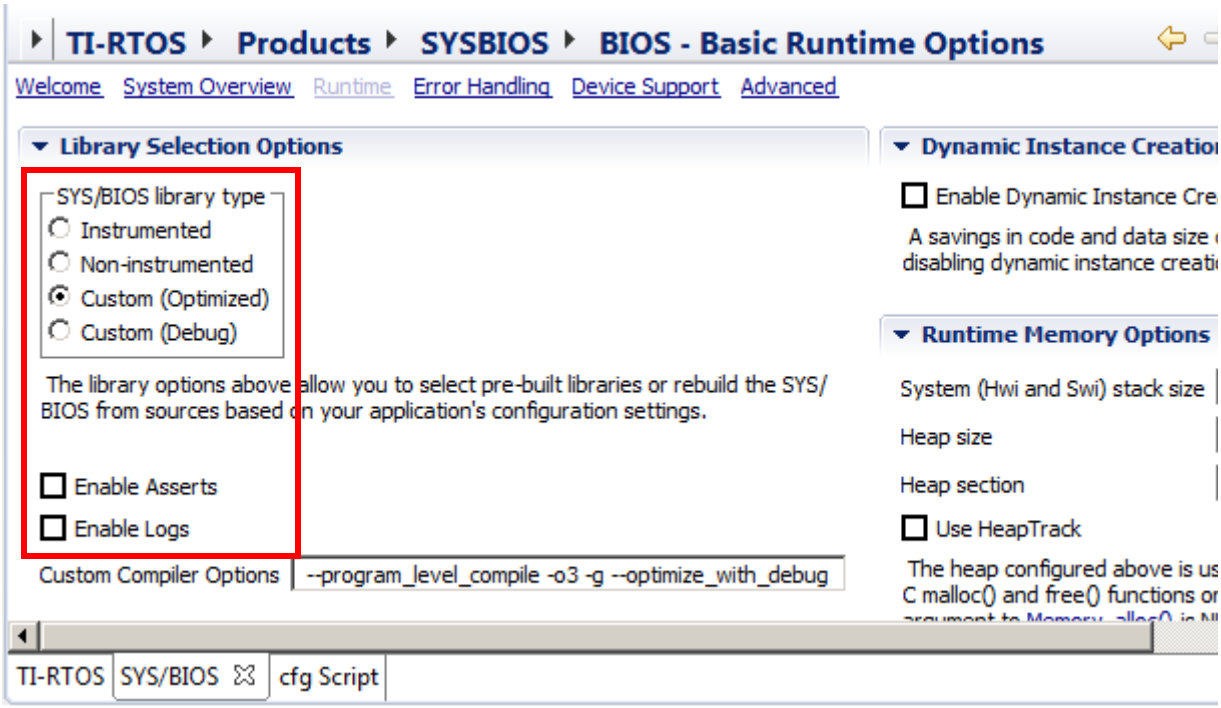
Non-Instrumented TI-RTOS drivers: You can use either an instrumented or non-instrumented TI-RTOS driver library. The instrumented library contains trace statements (Log_print*N*() calls) and assert checking (Assert_isTrue() calls). The non-instrumented library does not contain these statements. All the peripheral examples use the non-instrumented TI-RTOS driver library.



An example's *.cfg file contains statements like the following for the driver library:

```
var TIRTOS = xdc.useModule('ti.tirtos.TIRTOS');
TIRTOS.libType = TIRTOS.LibType_Instrumented;
```

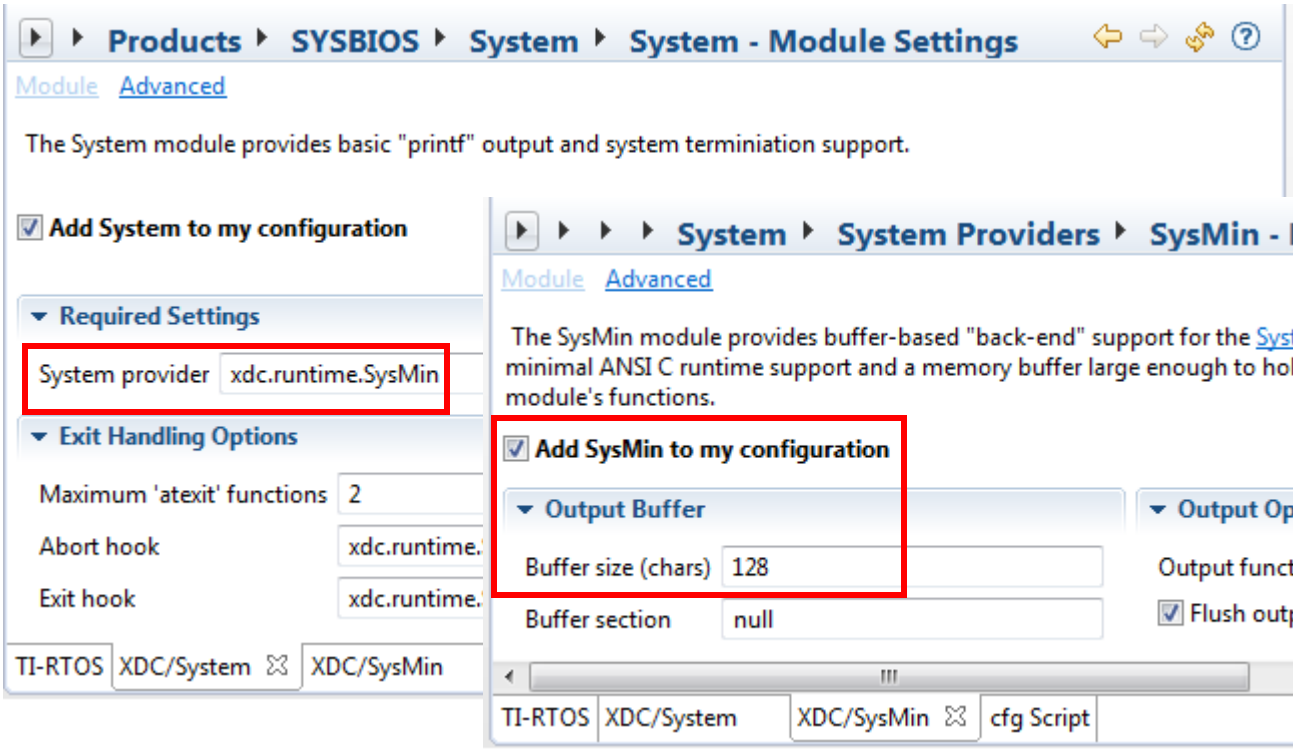
BIOS Custom Library: The kernel comes with both instrumented and non-instrumented libraries. In addition, it can perform a custom build to include only functionality required by the application. The TI-RTOS peripheral examples use the custom build. They also disable the kernel's logging and assert checking. See the "Compiler and Linker Optimization" section of the *SYS/BIOS User's Guide* for details.



If you edit configuration scripts directly, these statements have the same effect as the XGCONF settings above:

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
BIOS.logsEnabled = false;
BIOS.assertsEnabled = false;
```

Minimal System Provider: The System module allows users to plug in different System Support Proxies. Each proxy has pros and cons See Section 3.3.1, *Output with printf()*, page 3-28 for details about the available System module proxies. Most TI-RTOS peripheral examples use the smaller SysMin proxy, which uses an internal buffer to store System output. The size of the buffer is also reduced.



If you edit configuration scripts directly, these statements have the same effect as the XGCONF settings above:

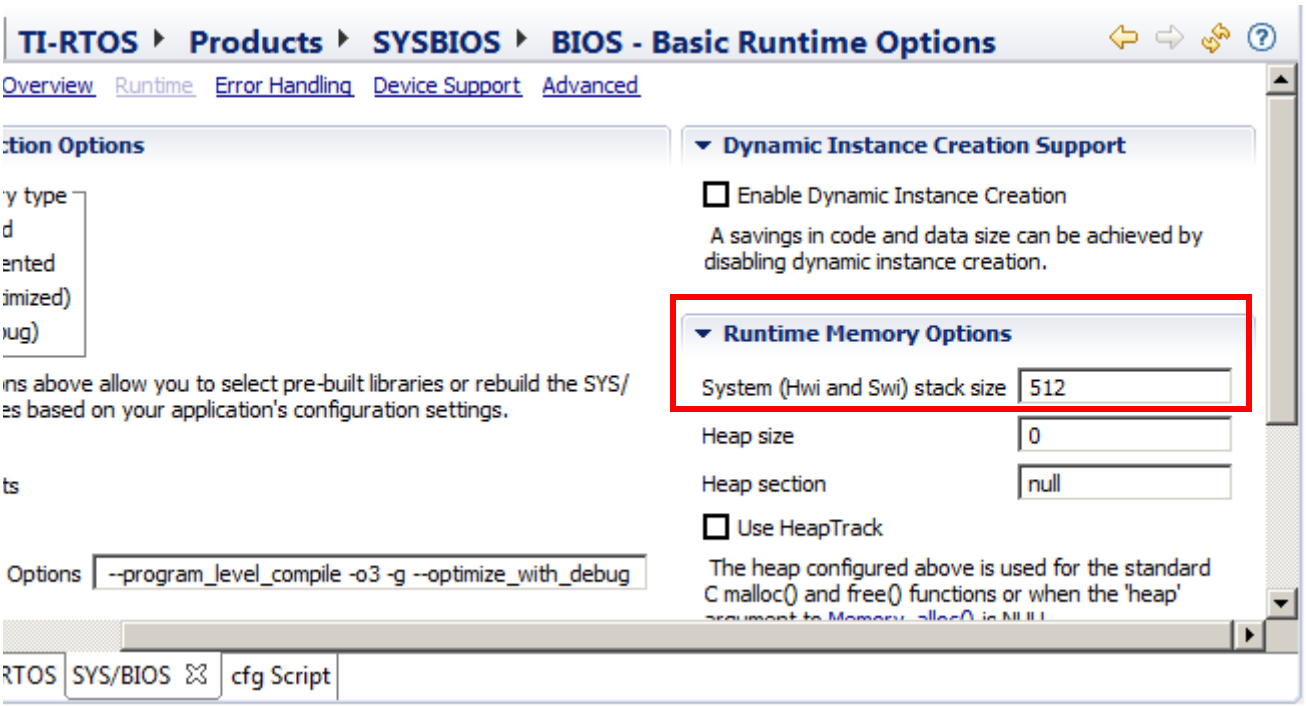
```
var System = xdc.useModule('xdc.runtime.System');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
System.SupportProxy = SysMin;
SysMin.bufSize = 128;
```

Note: The System output can be viewed in the RTOS Object Viewer (ROV) in CCS.

System Stack Size: The Hwi and Swi threads share a single System stack. Each device has a default System stack size, which is set by the Program.stack property. Several TI-RTOS examples (especially the MSP43x examples) do not use the default value. Instead, the Program.stack property is set in the example's .cfg file.

Note: The non-MSP43x examples do not reduce stack size as aggressively as the MSP43x examples. This is because the non-MSP43x example's source code (*.c and *.cfg) are generic and must run on several different devices.

To determine the best value for this property, each example was run with the default Program.stack setting. After an example ran under all conditions, the ROV in CCS was used to examine Hwi usage. The "Module" tab for Hwi objects shows the stack's peak usage. The example's Program.stack was set to a size higher than the peak but lower than the default. For example:

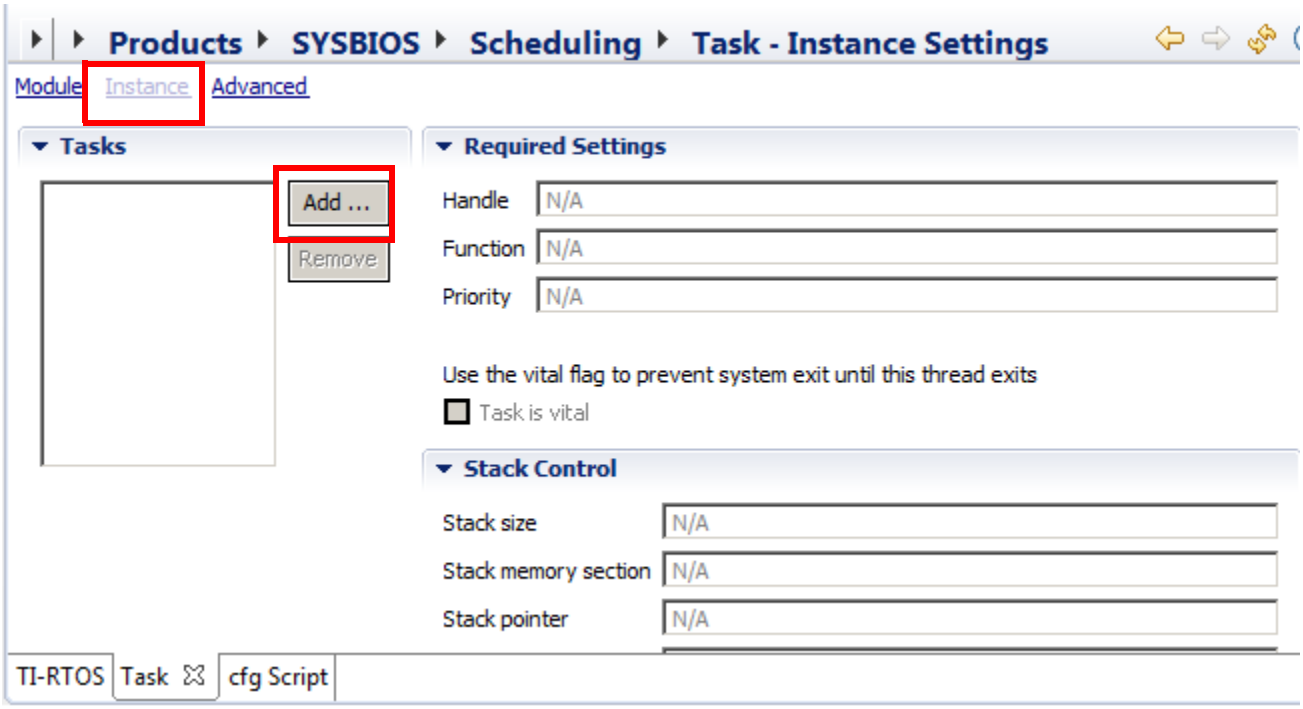


If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
Program.stack = 0x200;
```

Static Tasks: The majority of the examples statically create their Tasks in their *.cfg files. This reduces the code footprint because code is not needed for functions such Task_create().

To statically create a task, go to the **Instance** panel for configuring the Task module and click **Add**.



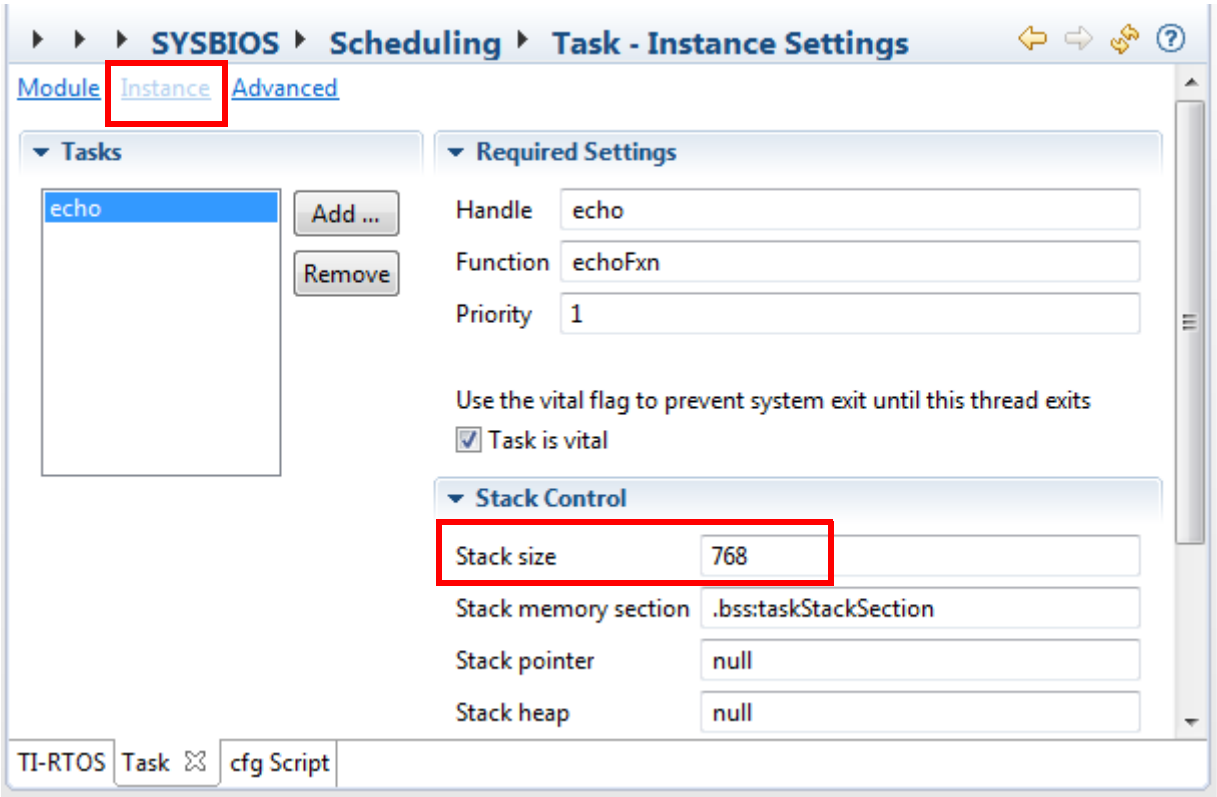
An example's *.cfg file contains statements like the following to statically create an object used by the example:

```
var taskParams = new Task.Params();
taskParams.instance.name = "taskFxn";
taskParams.stackSize = 0x300;
Program.global.task = Task.create("&taskFxn", task0Params);
```

Task Stack Size: Each Task thread in the application has its own stack. Each device has a default Task stack size. Many examples (especially the MSP43x examples) do not use the default value. Instead, the Task.stackSize property is set in the example's *.cfg file.

Note: The non-MSP43x examples do not reduce stack size as aggressively as the MSP43x examples. This is because the non-MSP43x example's source code (*.c and *.cfg) are generic and must run on several different devices.

To determine the best value for this property, each example was run with the default stackSize. After letting a example run under all conditions, the ROV in CCS was used to examine Task usage. The "Detailed" tab for Task objects shows the stack peak usage. The stackSize for each Task was set to a size higher than the peak but lower than the default. For example:

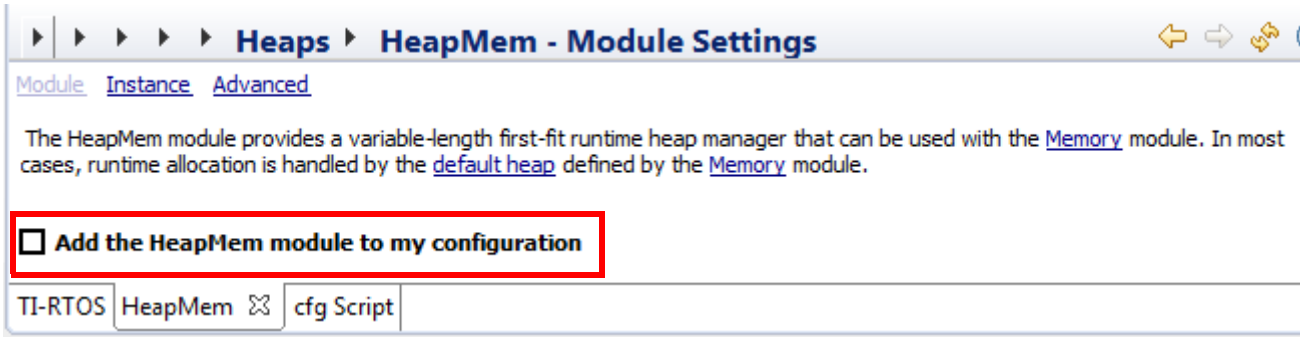


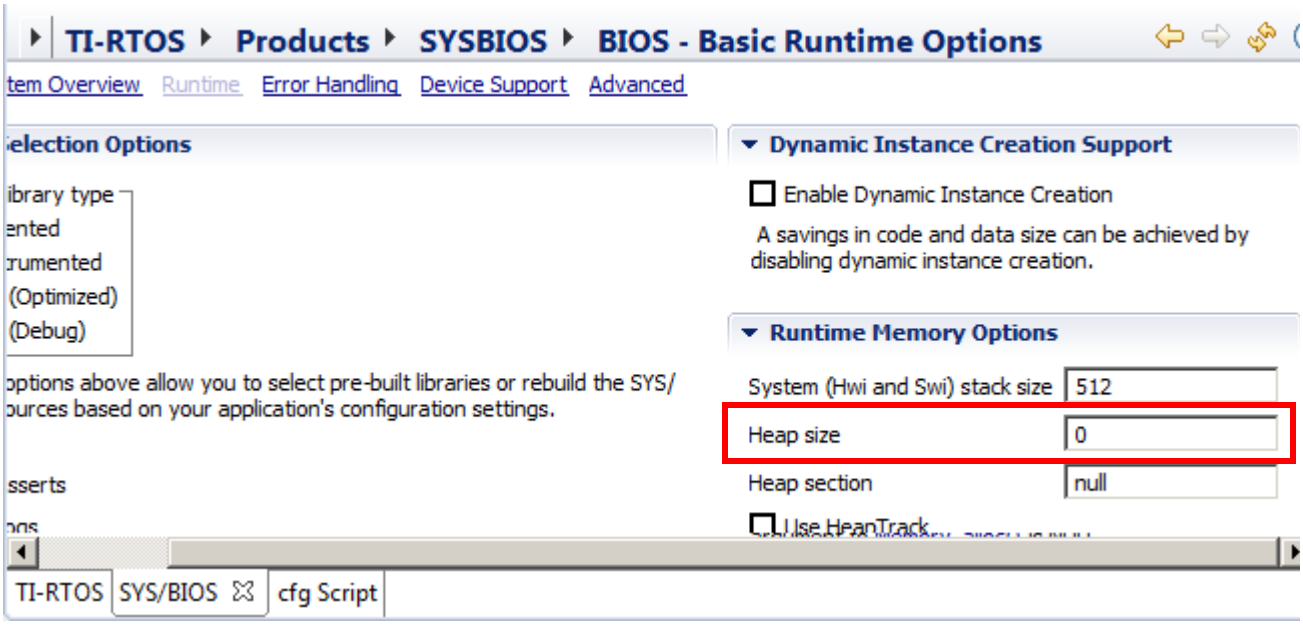
If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
taskParams.stackSize = 0x300;
```

Memory allocation: None of the TI-RTOS drivers allocate memory, except for the EMAC and USBMSCHFatFs drivers. The examples do not allocate memory either, except for the networking (wired and wireless) and USB examples. The examples statically create all kernel objects (such as Tasks and Semaphores) in the *.cfg file. This is done because run-time creation of kernel object allocates memory dynamically. Of course, for real applications, run-time object creation might be required.

Note that the networking stack allocates memory from a heap, so this approach cannot be used if the networking stack is used.





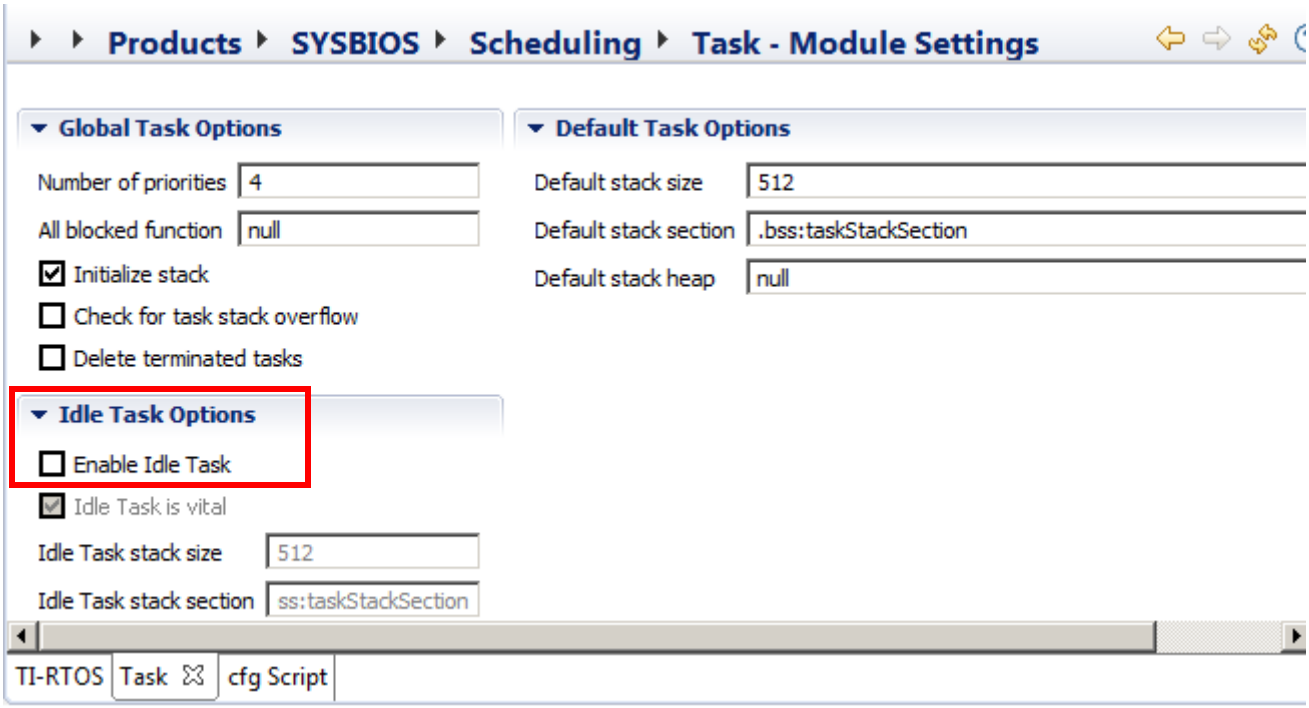
If you edit configuration scripts directly, disable dynamic memory allocation by removing the following statement from an example's *.cfg file:

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

In addition, add the following statement to the example's *.cfg file:

```
BIOS.heapSize = 0;
```

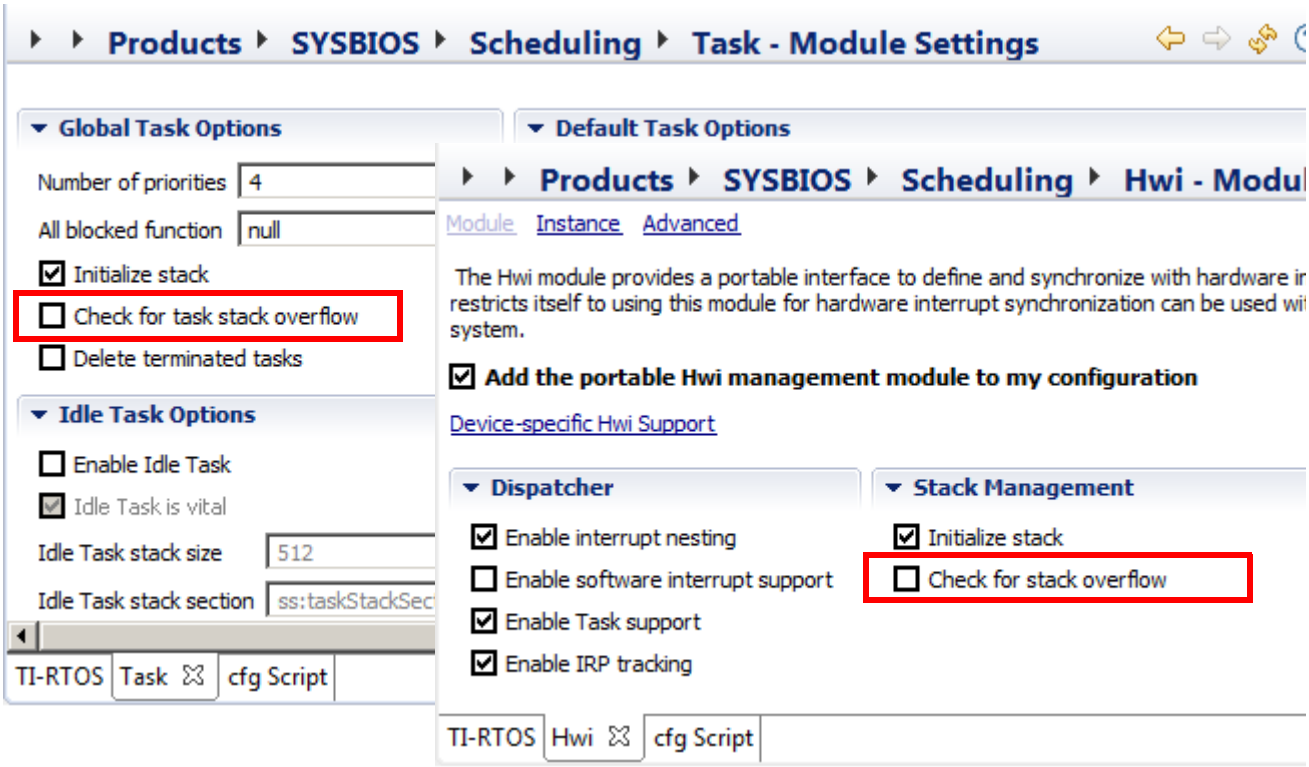
No Idle Task: The kernel, by default, has an Idle task that runs if no other thread is running. The Idle task runs low-priority functions (for example, to check for stack overflows). For the MSP43x examples, the Idle task is not enabled. This allows the MSP43x to be placed in a power-saving mode.



If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
Task.enableIdleTask = false;
```

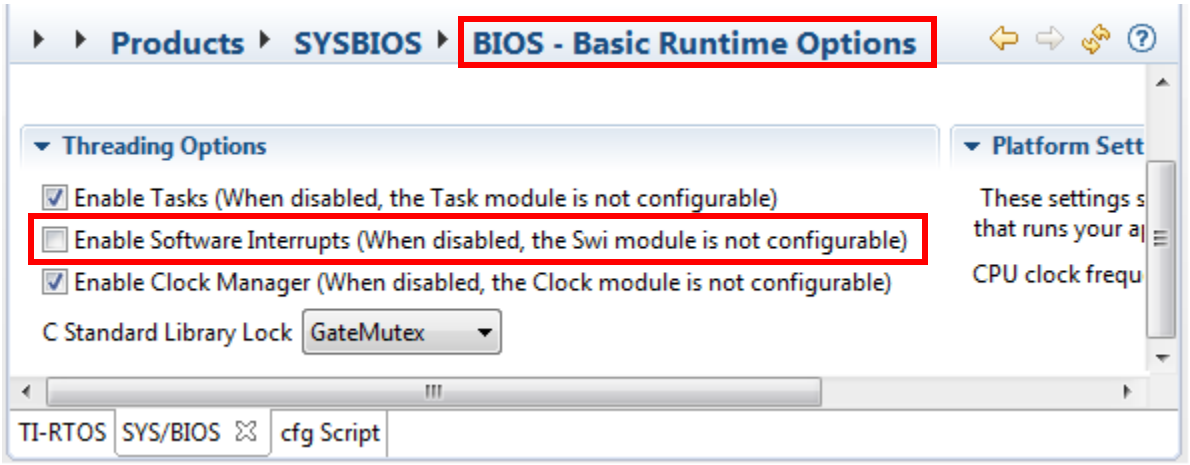
Stack Checking: The kernel, by default, verifies that the System stack and Tasks stacks have not overflowed. The System stack checks are performed in the Idle Task. The Task stack checks are performed at every context switch. The top of the stack is examined to make sure it has the correct "magic" value. Since a overflowed Task or System stack is show in ROV, the Task stack check was removed from the MSP43x examples to reduce the code footprint. See the *SYS/BIOS User's Guide* for details about these properties.



If you edit configuration scripts directly, these statements have the same effect as the XGCONF settings above:

```
Task.checkStackFlag = false;
Hwi.checkStackFlag = false;
```

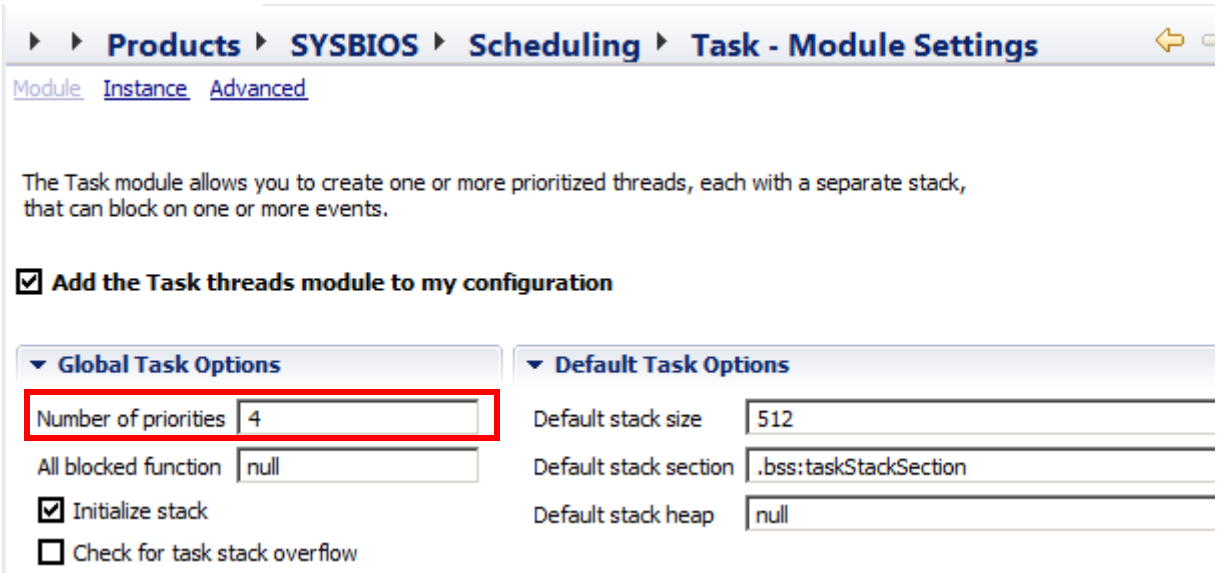
Software interrupts (Swis) disabled: The kernel, by default, enables software interrupts. For examples that do not use Swis, this type of thread is disabled. See the *SYS/BIOS User's Guide* for details about Swis. Note the EMAC driver uses a Swi, so networking examples cannot use this trick.



If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
BIOS.swiEnabled = false;
```

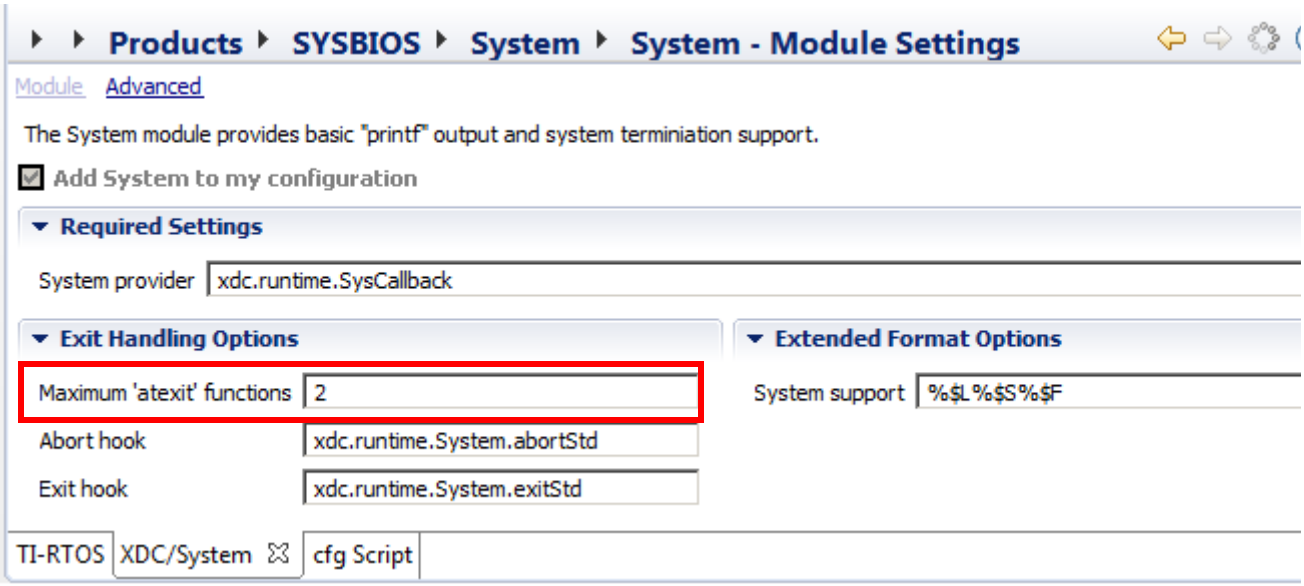
Number of Task Priorities: The kernel allows Tasks to have different priorities. See the *SYS/BIOS User's Guide* for details about Task priorities. The TI-RTOS examples lower the maximum number of Task priorities to 4.



If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
Task.numPriorities = 4;
```

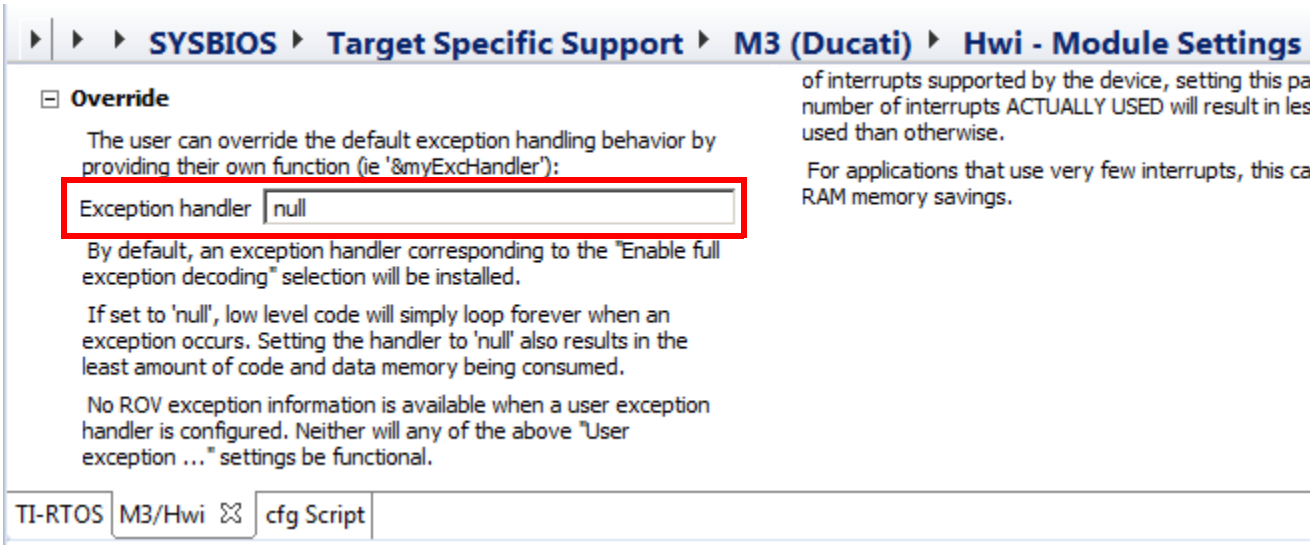
Number of atexit functions: The kernel allows System atexit() functions to be registered. See the Kernel Runtime APIs and Configuration (cdoc) online help for more about the xdc.runtime.System module's atexit() functions. The TI-RTOS examples lower the maximum number of System atexit functions to 2.



If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
System.maxAtexitHandlers = 2;
```

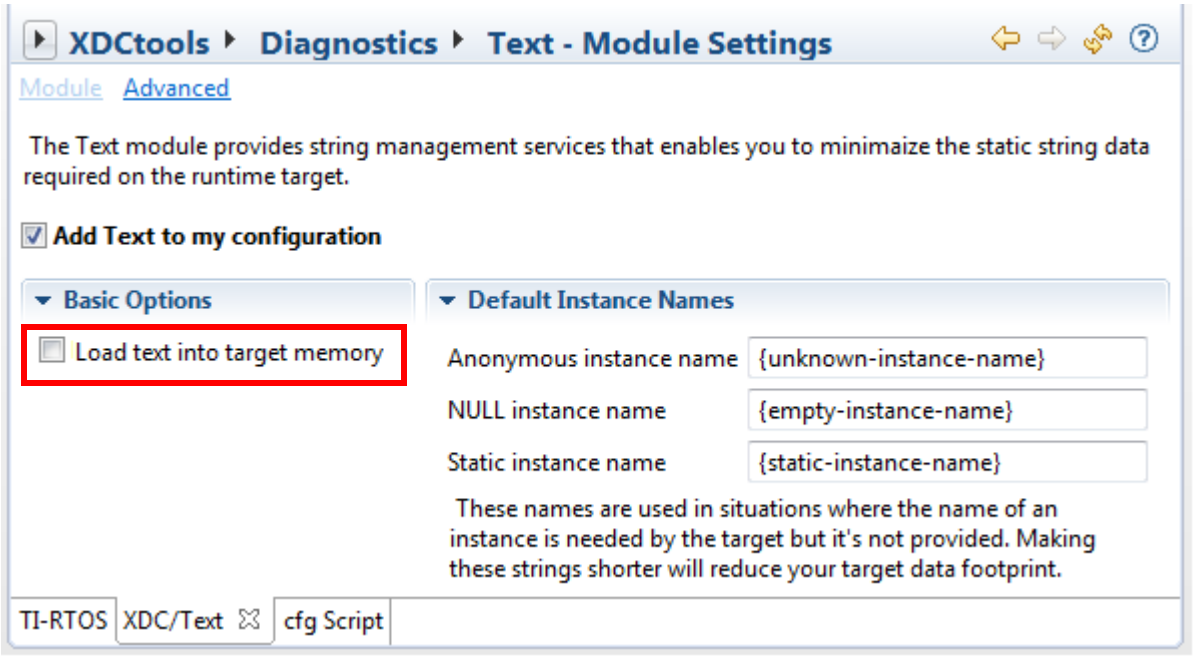
Kernel exception handling (for ARM): By default the kernel plugs in an exception handler to make debugging an exception easier. The exception handler can be removed to reduce code footprint. It is recommended that you leave the exception handler in place during development.



If you edit configuration scripts directly, these statements have the same effect as the XGCONF setting:

```
var m3Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
m3Hwi.excHandlerFunc = null;
```

Strings: Since no logging or asserts are enabled for the TI-RTOS examples, the strings associated with those facilities can be omitted. However, removing the strings for logging and asserts also removes additional strings. See the Kernel Runtime APIs and Configuration (cdoc) online help for more about the xdc.runtime.Text module's isLoaded property.



If you edit configuration scripts directly, this statement has the same effect as the XGCONF setting above:

```
Text.isLoaded = false;
```

9.2 Networking Stack Memory Usage

See TI-RTOS Networking Stack Memory Usage on the Texas Instruments Wiki for details about to adjusting memory usage of the networking stack (NDK).

Revision History

Table A–1 lists the significant changes made in recent versions of this document.

Table A–1. Revision History

Revision	Chapter	Location	Additions/Modifications/Deletions
SPRUHD4M	Preface		Current software version number is v2.20.
	Instrumentation	Section 2.3.1	The directory name containing the examples has been simplified.
	Board-Specific	Section 4.1	Support for the MSP430FR5994 Launchpad has been added.
SPRUHD4L	Preface		Current software version number is v2.16.
	Overview	Section 1.6	The chapter on the Network Services API has been removed. See the documentation in the following location instead: `<tirtos_install>/products/ns_<version>/docs/NS_Users_Guide.html`.
	Board-Specific	Section 4.1	Support for several CC13xx/CC26xx boards has been added.
SPRUHD4K	Preface		Current software version number is v2.15.
	Overview	Chapter 1	The IPC component has been removed from TI-RTOS.
	Overview	Section 1.8.1	The Doxygen-based API help has been moved to `<tirtos_install>/products/tidrivers_<version>/docs/html`.
	Overview and Drivers	Section 1.8 and Section 5.1	The location of the driver files has changed from the `<tirtos_install>/packages/ti/drivers` directory to `<tirtos_install>/products/tidrivers_<version>/packages/ti/drivers`. In addition, board files have been moved to `<tirtos_install>/products/tidrivers_<version>/packages/ti/boards`.
	Drivers	Section 5.2.1	A new ti.drivers.Config module has been added. This is the only module that needs to be configured. The configuration statements of the form xdc,useModule("ti.drivers.UART) and the TIRTOS.useDRIVER properties are no longer needed and have been deprecated.
	Drivers		TI-RTOS drivers no longer provide data to ROV.
	Drivers	Section 5.5.1.3	The GPIO callback function takes an integer parameter that identifies the GPIO that caused the interrupt.
	Drivers	Section 5.8	The LCD driver is now documented. This is a middleware driver.

Table A–1. Revision History

Revision	Chapter	Location	Additions/Modifications/Deletions
	Drivers	Section 5.17	The WiFi driver has been moved to the middleware directory.
	FatFS	Chapter 7	The FatFS module files have been moved from the SYS/BIOS installation to the driver product. The module path is now ti.mw.fatfs.FatFS.
SPRUHD4J	Preface		Current software version number is v2.14.
	Overview	Section 1.10	Added a description of the wolfSSL embedded SSL library.
	Board-Specific	Section 4.1	Removed the MSP-EXP430F5529 Experimenter Board, DK-TM4C123G Evaluation Kit, and EKS-LM4F232 Evaluation Kit from the supported board list. Added MSP-EXP430FR6989LP to the supported board list.
	Drivers	Section 5.2.8 and Section 5.5.1.5	Added information about MSP430 driver implementation for GPIO.
	Rebuilding	Section 8.2	Added MSP-EXP430FR6989LP to the supported board list.
SPRUHD4I	Preface		Current software version number is v2.12.
	Board-Specific	Section 4.1	Support for MSP432 has been added.
	Drivers	Section 5.3 and Section 5.7	Added Camera driver and I^2S driver.
		Section 5.5	GPIO driver has been modified significantly.
SPRUHD4H	Preface		Current software version number is v2.11.
		Section 1.1 and Section 1.8.5	CCWare support is provided in TI-RTOS for SimpleLink Wireless MCUs.
	Instrumentation and Drivers	Section 2.4.2 and Section 5.2.1	Configuring instrumented or non-instrumented drivers has been moved from individual driver modules to the TIRTOS module.
	Debugging	Section 3.1.1	A section on debugging applications by stepping through TI-RTOS code has been added.
	Board-Specific	Section 4.1	Support for SimpleLink boards has been added.
	Drivers	Section 5.2.6 and Section 5.2.7	Information about driver implementations for SimpleLink devices has been added.
		Section 5.9	The PWM driver has been added.
		Section 5.12.5 and Section 5.12.6	Added information about configuring the UART driver to use DMA.
	Rebuilding	Section 8.1.4	Information about rebuilding the drivers with the debug profile has been added.

Table A–1. Revision History

Revision	Chapter	Location	Additions/Modifications/Deletions
SPRUHD4F	Preface		Current software version number is v2.00.
	About	Section 1.1	TI-RTOS now has separate installers for various device families. There is a separate *TI-RTOS Getting Started Guide* for each installer.
		Section 1.2	TI-RTOS has several components with TI-RTOS component names. For example, SYS/BIOS is also called the TI-RTOS Kernel.
	Instrumentation	Section 2.1	System Analyzer also includes the views available from the **Tools > RTOS Analyzer** menu.
		Section 2.2 and Section 2.3.3	The configuration properties for LoggingSetup have changed.
		Section 2.5	The menu commands and dialogs used to open System Analyzer views have changed.
	Board-Specific	Section 4.1	TI-RTOS examples have been added for the MSP-EXP430FR5969LP LaunchPad and EK-TM4C1294XL Evaluation Kit.
	Drivers	Section 5.2.8	EUSCI versions of the MSP430 driver have been added.
	Memory	Section 9.1	Pictures of configuration settings made in XGCONF are included to supplement the script-based statements. Also, instructions for removing the kernel exception handler have been added.
SPRUHD4E	Preface		Current software version number is v1.21.
	About	Section 1.1 and Section 4.1	Added MSP-EXP430F5529 Experimenter Board.
	Debugging	Section 3.1	ROV also available in IAR Embedded Workbench.
	Rebuilding	Section 8.1.3	Added section on building TI-RTOS for the GCC code generator.
	Memory Usage	Section 9.1	Two versions of the Empty example are now provided. The new one uses minimal memory. Also added removal of HeapMem enabling statement to description of how to disable dynamic memory allocation.
SPRUHD4D	Preface		Current software version number is v1.20.
	About	Section 1.1	Several new boards added to the table.
		Section 1.8.3	New section added for MSP430Ware.
		Section 1.11	Links added for MSP430Ware, MSP430 boards, and BoosterPacks.

Table A–1. Revision History

Revision	Chapter	Location	Additions/Modifications/Deletions
	Examples	Chapter 2	Details about individual examples moved to the readme files within the example projects. Other information previously in Chapter 2 moved to the *TI-RTOS Getting Started Guide*.
	Instrumentation	Section 2.3 to Section 2.3.3	New section added on converting an example to perform run-time uploading of instrumentation data.
	Boards	Section 4.1	Several new boards added to the table.
	Drivers	Section 5.2.8	New section added on Hwi objects and ISRs for MSP430 devices.
		Section 5.14.1.3	New section added on USB reference modules for MSP430.
		Section 5.17 to Section 5.17.1	WiFi driver can now be configured to support calling it from multiple threads.
	Utilities	--	The SysFlex module has been deprecated.
		Section 6.2 to Section 6.2.2	The UARTMon module has been added.
	Rebuilding	Section 8.1.1	The contents of tirtos.mak have changed.
		Section 8.1.2	New section added for building TI-RTOS for IAR.
		Section 8.2	New section added for rebuilding MSP430Ware libraries.
	Memory	Section 9.1	New section added to discuss ways to reduce the memory footprint.

Index

A

APIs
  common 42
  EMAC driver 50
  GPIO driver 54
  I2C driver 56
  LCD driver 67
  SDSPI driver 71
  UART driver 79
  USB device and host modules 89
  USBMSCHFatFs driver 83
  Watchdog driver 91
assert handling 26
Available Products list 15

B

board.c files 34
build flow 32

C

C28x
  support 34
Camera driver 37, 48
CC1310 Development Kit 33
CC1310 LaunchPad 33
CC1350 LaunchPad 33
CC1350 SensorTag 33
CC2650 Development Kit 34
CC2650 LaunchPad 33
CC2650 SensorTag 34
CC3200-LAUNCHXL 34
ccxml file 35
CDC device 37
COM Port 99
components 8
Concerto 34
configuration
  build flow 32
configuro tool 32
controlSUITE 11
  other documentation 11
CyaSSL 13

D

debugging 25
Demo examples 113
DK-TM4C129X 33
drivers 11, 37

E

EK-LM4F120XL 33
EK-TM4C123GXL 33
EK-TM4C1294XL 33
EK-TM4C129EXL 33
EMAC driver 37, 50
Empty example 113
Ethernet driver 37, 50
exception handling 26

F

F28M35H52C1 33
F28M36P63C2 33
FatFs driver 71, 83
flash drives 37, 83

G

GPIO driver 37, 52
GPIO pin
  configuration 34
GUI Composer 102

H

HID device 37

I

I2C driver 37, 56
I2S driver 37, 63
instrumentation 14
instrumented libraries 22

K

keyboard
  device 89
  host 89

L

LCD driver 37
LEDs
  configuration 34
linker command file 35
LM4F120H5QR 33
Load logging 16
Log module 22
  EMAC driver 51
  GPIO driver 55
  I2C driver 62
  UART driver 82
  USBMSCHFatFs driver 85
  viewing messages 23
  Watchdog driver 92, 95
logging 16
LoggingSetup module 15

M

M3 microcontroller 34
memory reduction 113
mouse
  device 89, 90
  host 89
MSC device 37
MSC host 83
MSP430F5529 33
MSP430FR5969 33
MSP430FR5994 33
MSP430FR6989 33
MSP432P401RLP 33
MSP-EXP430F5529LP 33
MSP-EXP430FR5969LP 33
MSP-EXP430FR5994LP 33
MSP-EXP430FR6989LP 33
MSP-EXP432P401RLP 33
MSPWare 8, 12
MWare 8, 9, 11
  other documentation 11, 12

N

NDK 8, 10, 50
  other documentation 10
non-instrumented libraries 22

P

Peripheral examples 113
PIN driver 37
printf() function 28
Printf-style output 26, 28
products directory 8
PWM driver 37

R

rebuilding
  TI-RTOS 109
ROV tool 24, 25, 28
RTOS Analyzer
  debugging with 23
RTOS Object View (ROV) 25

S

SD cards 71
SDSPI driver 37, 71
serial devices 90
simulator, debugging with 35
SPI (SSI) bus 71
SPI driver 37
SSL library 13
static configuration 32
StellarisWare 8
SYS/BIOS 8, 9
  logging 16
  other documentation 9
SysCallback module 28
SysMin module 28
  configuration 29
SysStd module 28
System Analyzer 9, 14, 26
System module 28
  configuration 29
System_printf() function 28

T

Target Configuration File 35
TI-RTOS 7
TivaWare 12
TLS library 13
TM4C123GH6PM 33
TM4C1294NCPDT 33
TM4C129ENCPDT 33
TM4C129XNCZAD 33
TMDXDOCK28M36 33
TMDXDOCKH52C1 33
TMDXDOCKH52C1.c file 34

U

UART driver 37, 79
UARTMon module 96
UIA 8, 9, 14
  other documentation 9
USB controller 83

USB Descriptor Tool 87, 88
USB driver 89
USBMSCHFatFs driver 37, 83

W

Watchdog driver 37, 91
  APIs 91
WiFi driver 37
wolfSSL 13

X

XDCtools 8, 13
  build settings 32
  other documentation 13

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303 Dallas, Texas 75265
Copyright © 2016, Texas Instruments Incorporated