CC2640 and CC2650 SimpleLink™ *Bluetooth*® low energy Software Stack 2.1.0/2.1.1

Developer's Guide

Literature Number: SWRU393B
February 2015–Revised January 2016

References

NOTE: You can find the latest version of this guide at *SimpleLink™ Bluetooth® low energy CC2640 Wireless MCU Software Developer's Reference Guide* (SWRU393).

1. *TI Bluetooth low energy Vendor-Specific HCI Reference Guide v2.0*, C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx\TI_BLE_Vendor_Specific_HCI_Guide.pdf
2. *TI CC26xx Technical Reference Manual*, (SWCU117)
3. *Measuring Bluetooth Smart Power Consumption Application Report*, (SWRA478)
4. *TI-RTOS Documentation Overview*, C:\ti\tirtos_simplelink_2_13_00_06\docs\docs_overview.html
5. *TI-RTOS Getting Started Guide*, C:\ti\tirtos_simplelink_2_13_00_06\docs\tirtos_Getting_Started_Guide.pdf
6. *TI-RTOS User's Guide*, C:\ti\tirtos_simplelink_2_13_00_06\docs\tirtos_User_Guide.pdf
7. *TI-RTOS SYS/BIOS Kernel User's Guide*, C:\ti\tirtos_simplelink_2_13_00_06\products\bios_6_42_00_08\docs\Bios_User_Guide.pdf
8. *TI-RTOS Power Management for CC26xx*, C:\ti\tirtos_simplelink_2_13_00_06\docs\Power_Management_CC26xx.pdf
9. *TI SYS/BIOS API Guide*, C:\ti\tirtos_simplelink_2_13_00_06\products\bios_6_42_00_08\docs\Bios_APIs.html
10. *CC26xxware DriverLib API*, C:\ti\tirtos_simplelink_2_13_00_06\products\cc26xxware_2_21_01_15600\doc\doc_overview.html
11. *Sensor Controller Studio*,http://www.ti.com/tool/sensor-controller-studio
12. *TI-RTOS API Reference*, C:/ti/tirtos_simplelink_2_13_00_06/docs/doxygen/html/index.html
13. *CC2640 Simple Network Processor API Guide*, C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx\ CC2640 Simple Network Processer API Guide.pdf
14. *ARM Cortex-M3 Devices Generic User's Guide*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf

    Available for download from the *Bluetooth* Special Interest Group (SIG) website:
15. *Specification of the Bluetooth System, Covered Core Package, Version: 4.1 (03-Dec-2013)*, https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=282159
16. *Device Information Service (Bluetooth Specification), Version 1.0 (24-May-2011)*, https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=238689

    Links
17. *TI Bluetooth low energy Wiki*, www.ti.com/ble-wiki
18. *Latest Bluetooth low energy Stack Download and Examples*, www.ti.com/ble-stack
19. *TI E2E Support Forum*, www.ti.com/ble-forum
20. *TI Designs Reference Library*, http://www.ti.com/general/docs/refdesignsearch.tsp

SimpleLink, Code Composer Studio, SmartRF are trademarks of Texas Instruments.
ARM, Cortex are registered trademarks of ARM Limited.
iBeacon is a trademark of Apple Inc.
Bluetooth is a registered trademark of Bluetooth SIG.
iOS is a registered trademark of Cisco.
Android is a trademark of Google Inc.
Intel is a registered trademark of Intel Corporation.
Windows 7 is a registered trademark of Microsoft Inc.
Python is a registered trademark of PSF.

Copyright © 2015–2016, Texas Instruments Incorporated

Overview

The purpose of this document is to give an overview of the TI SimpleLink™ *Bluetooth*® low energy CC2640 wireless MCU software development kit to begin creating a *Bluetooth* Smart custom application. This document also introduces the *Bluetooth* low energy specification. Do not use this document as a substitute for the complete specification. For more details, see the Specification of the Bluetooth System or some introductory material at the TI Bluetooth low energy Wiki.

1.1 Introduction

Version 4.1 of the *Bluetooth* specification allows for two systems of wireless technology: Basic Rate (BR: BR/EDR for Basic Rate/Enhanced Data Rate) and *Bluetooth* low energy. The *Bluetooth* low energy system was created to transmit small packets of data, while consuming significantly less power than BR/EDR devices.

Dual-mode devices can support BR and *Bluetooth* low energy and are branded as *Bluetooth* Smart Ready. In a *Bluetooth* wireless technology system, a mobile smart phone or laptop computer is a dual-mode device. Single-mode devices support only *Bluetooth* low energy and are branded as *Bluetooth* Smart. These single-mode devices are generally used for applications where low-power consumption is a primary concern, such as in applications that run on coin-cell batteries. For more information, see Figure 1-1.

Figure 1-1. *Bluetooth* **Smart and** *Bluetooth* **Smart Ready Branding Marks**

1.2 *Bluetooth* low energy Protocol Stack Basics

Figure 1-2 shows the *Bluetooth* low energy protocol stack architecture.

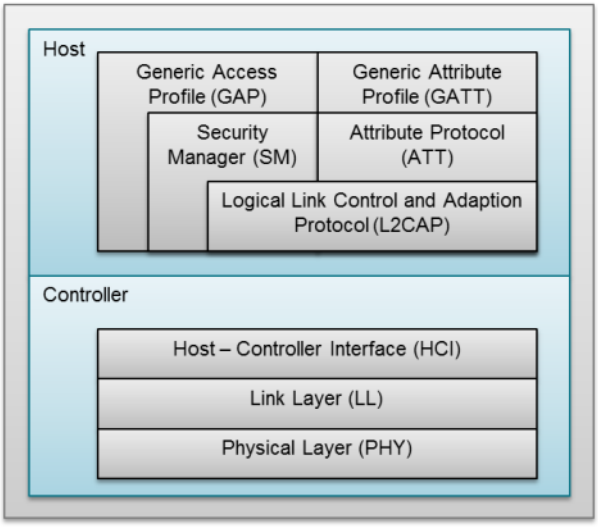


Figure 1-2. *Bluetooth* **low energy Protocol Stack**

Copyright © 2015–2016, Texas Instruments Incorporated

The *Bluetooth* low energy protocol stack (or protocol stack) consists of the controller and the host. This separation of controller and host derives from the implementation of standard *Bluetooth* BR/EDR devices, where the two sections are implemented separately. Any profiles and applications sit on top of the GAP and GATT layers of the protocol stack.

The physical layer (PHY) is a 1-Mbps adaptive frequency-hopping GFSK (Gaussian frequency-shift keying) radio operating in the unlicensed 2.4-GHz ISM (industrial, scientific, and medical) band.

The link layer (LL) controls the RF state of the device, with the device in one of five states:

- Standby
- Advertising
- Scanning
- Initiating
- Connected

Advertisers transmit data without connecting, while scanners scan for advertisers. An initiator is a device that responds to an advertiser with a request to connect. If the advertiser accepts the connection request, both the advertiser and initiator enter a connected state. When a device is connected, it connects as either master or slave. The device initiating the connection becomes the master and the device accepting the request becomes the slave.

The host control interface (HCI) layer provides communication between the host and controller through a standardized interface. This layer can be implemented either through a software API or by a hardware interface such as UART, SPI, or USB. Standard HCI commands and events are specified in the Specification of the Bluetooth System. TI's proprietary commands and events are specified in TI Bluetooth low energy Vendor-Specific HCI Reference Guide v2.0.

The link logical control and adaption protocol (L2CAP) layer provides data encapsulation services to the upper layers, allowing for logical end-to-end communication of data. The security manager (SM) layer defines the methods for pairing and key distribution, and provides functions for the other layers of the protocol stack to securely connect and exchange data with another device. See Section 5.3.5 for more information on TI's implementation of the SM layer. The generic access protocol (GAP) layer directly interfaces with the application and/or profiles, to handle device discovery and connection-related services for the device. GAP handles the initiation of security features. See Section 5.1 for more information on TI's implementation of the GAP layer.

The attribute protocol (ATT) layer protocol allows a device to expose certain pieces of data or *attributes*, to another device. The generic attribute protocol (GATT) layer is a service framework that defines the sub-procedures for using ATT. Data communications that occur between two devices in a *Bluetooth* low energy connection are handled through GATT sub-procedures. The application and/or profiles will directly use GATT. See Section 5.3 for more information on TI's implementation of the ATT and GATT layers.

Copyright © 2015–2016, Texas Instruments Incorporated

TI Bluetooth low energy Software Development Platform

The TI royalty-free *Bluetooth* low energy software development kit (SDK) is a complete software platform for developing single-mode *Bluetooth* low energy applications. This kit is based on the SimpleLink CC2640, complete System-on-Chip (SoC) *Bluetooth* Smart solution. The CC2640 combines a 2.4-GHz RF transceiver, 128-KB in-system programmable memory, 20KB of SRAM, and a full range of peripherals. The device is centered on an ARM® Cortex®-M3 series processor that handles the application layer and *Bluetooth* low energy protocol stack and an autonomous radio core centered on an ARM Cortex®-M0 processor that handles all the low-level radio control and processing associated with the physical layer and parts of the link layer. The sensor controller block provides additional flexibility by allowing autonomous data acquisition and control independent of the Cortex-M3 processor, further extending the low-power capabilities of the CC2640. Figure 2-1 shows the block diagram. For more information on the CC2640, see the CC26xx Technical Reference Manual (TRM).

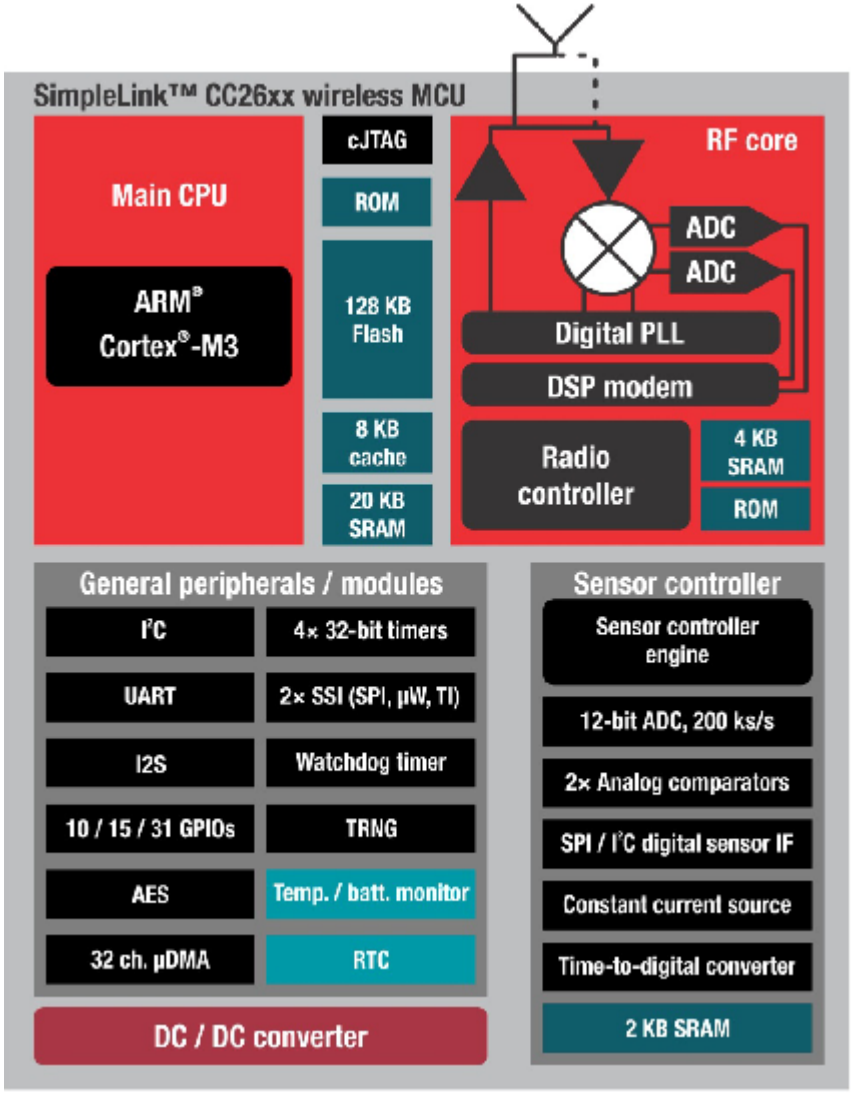


Figure 2-1. SimpleLink CC2640 Block Diagram

Copyright © 2015–2016, Texas Instruments Incorporated

2.1 Protocol Stack and Application Configurations

Figure 2-2 shows the platform that supports two different protocol stack and application configurations.

- **Single device:** The controller, host, profiles, and application are all implemented on the CC2640 as a true single-chip solution. This configuration is the simplest and most common when using the CC2640. This configuration is used by most of TI's sample projects. This configuration is the most cost-effective technique and provides the lowest-power performance.
- **Simple network processor:** The controller and host are implemented together on the CC2640, while the profiles and application are implemented on an external MCU. The application and profiles communicate with the CC2640 through the simple network processor (SNP) API that simplifies the management of the *Bluetooth* low energy network processor. The SNP API communicates with the *Bluetooth* low energy device using the Network Protocol Interface (NPI) over a serial (SPI or UART) connection. This configuration is useful for peripheral applications, which execute on either another device (such as an external microcontroller) or a PC without the requirement to implement complexities associated with an HCI-based protocol. In these cases, the application and profiles can be developed externally while running the *Bluetooth* low energy protocol stack on the CC2640. For a description of the SNP, see the CC2640 Simple Network Processor API Guide.

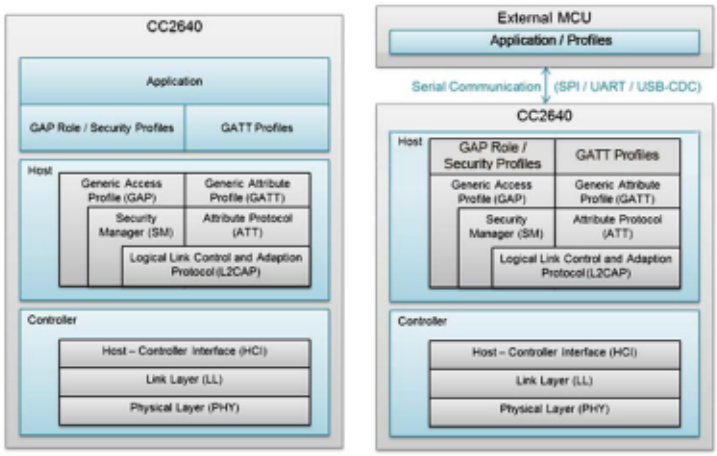


Figure 2-2. Single-Device and Simple Network Processor Configuration

Copyright © 2015–2016, Texas Instruments Incorporated

2.2 Solution Platform

This section describes the various components that are installed with the *Bluetooth* low energy stack SDK and the directory structure of the protocol stack and any tools required for development. Figure 2-3 shows the *Bluetooth* low energy stack development system.

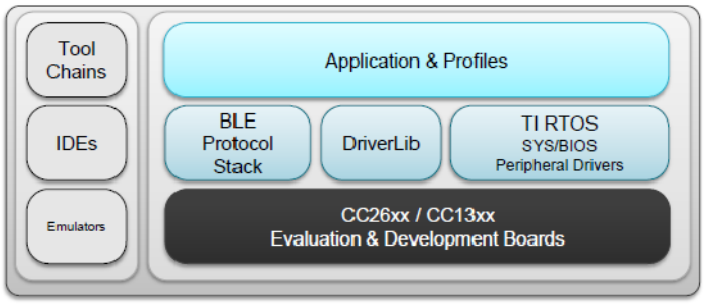


Figure 2-3. *Bluetooth* **low energy Stack Development System**

The solution platform includes the following components:

- **Real-time operating system (RTOS)** with the TI-RTOS SYS/BIOS kernel, optimized power management support, and peripheral drivers (SPI, UART, and so forth)
- **CC26xxware driverLib** provides a register abstraction layer and is used by software and drivers to control the CC2640 SoC.
- **The Bluetooth low energy protocol stack** is provided in library form with parts of the protocol stack in the CC2640 ROM.
- **Sample applications and profiles** make starting development using both proprietary and generic solutions easier. Certain applications and profiles (see Chapter 12 and https://developer.bluetooth.org/gatt/profiles/Pages/ProfilesHome.aspx) in the *Bluetooth* low energy SDK are fully qualified by *Bluetooth* SIG.

The following integrated development environments (IDEs) supported are the following:

- IAR Embedded Workbench for ARM
- Code Composer Studio™ (CCS)

2.3 Directory Structure

The *Bluetooth* low energy SDK installer includes the files to start evaluating and creating a custom application. The SDK has the following four parts:

- **Accessories** include the BTool windows application, boundary tool, and precompiled super hex files for common applications, such as SimpleBLEPeripheral and SensorTag. Except for the SensorTag, TI designed the prebuilt hex files for the SmartRF06 evaluation board and the CC2650EM-7ID evaluation module.
- **Components** include *Bluetooth* low energy protocol stack services.
- **Documents** include the *Software Developer's Guide*, *Vendor-Specific HCI API Guide*, and application notes.
- **Projects** include application examples, proprietary and certified *Bluetooth* SIG Profiles.

Copyright © 2015–2016, Texas Instruments Incorporated

2.4 Projects

The *Bluetooth* low energy stack SDK installer includes a large number of projects ranging from basic *Bluetooth* low energy functionality to use-case specific applications such as Heart Rate Sensor, Glucose Collector, and so forth. The following sections present the basic projects with which to begin. For more details on these and all other included projects, see Chapter 12.

- The **SimpleBLEPeripheral** project consists of sample code that demonstrates a simple *Bluetooth* low energy slave application in the single-device configuration. This project can be used as a reference for developing a slave and peripheral application.
- The **SimpleBLECentral** project demonstrates the other side of the connection. This project demonstrates a simple master and central application in the single-device configuration and can be a reference for developing master and central applications.
- The **SimpleBLEBroadcaster** project demonstrates the only role for implementing nonconnectable Beacon applications, such as Apple iBeacon™ and Google Eddystone. See *Bluetooth low energy Beacons Application Note* (SWRA475) for more information about *Bluetooth* Smart Beacons.
- The **SensorTag** project is a peripheral application that is configured to run on the CC2650 SensorTag reference hardware platform and communicate with the various peripheral devices of the SensorTag (for example, temperature sensor, gyro, magnetometer, and so forth).
- The **SimpleNP** project builds the SNP software for the CC2640. This project supports a configuration for the slave and peripheral roles. Refer to the CC2640 Simple Network Processor API Guide for APIs available in the SNP implementation.
- The **SimpleAP** project builds the simple application processor software for the CC2640. This project demonstrates an application (host) MCU communicating with the CC2640 running the SNP network processor application. See CC2640 Simple Network Processor API Guide in the Documents folder for APIs available for the SNP implementation.
- The **HostTest** project builds the *Bluetooth* low energy HCI-based network processor software for the CC2640. This project can be configured for master and slave roles and can be controlled by the BTool PC application. See the TI BLE Vendor-Specific HCI Reference Guide v2.0 in the Documents folder for APIs available for configuring and controlling the HostTest application.

2.5 Setting up the Integrated Development Environment

The IDE must be set up to browse through the relevant projects and view code. All embedded software for the CC2640 is developed using IAR Embedded Workbench for ARM (from IAR software) or CCS from TI on a Windows 7® or later PC. This section provides information on where to find this software and how to configure the workspace for each IDE.

All path and file references in this document assume that the *Bluetooth* low energy SDK is installed to the default path (**$BLE_INSTALL$**). TI recommends making a working copy of the *Bluetooth* low energy SDK before to making any changes. The *Bluetooth* low energy SDK uses relative paths and is designed to be portable, allowing the copying of the top-level directory to any valid path.

NOTE: If installing the *Bluetooth* low energy SDK to a nondefault path, do not exceed the maximum length of the file system namepath.

Copyright © 2015–2016, Texas Instruments Incorporated

2.5.1 Installing the SDK

To install the *Bluetooth* low energy stack SDK, run the ble_cc26xx_setupwin32_2_01_00_xxxxx.exe installer:

- xxxxx is the SDK build revision number.
- The default SDK install path is C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx.

This installer also installs the TI-RTOS bundle, XDC tools, and the BTool PC application (if not already installed). Table 2-1 lists the software and tools are supported and tested with this *Bluetooth* low energy stack SDK. Check the TI Bluetooth LE Wiki for the latest supported tool versions.

Table 2-1. Supported Tools and Software

Tool or Software	Version	Install Path
Bluetooth low energy Stack SDK Installer	2.1.0	C:\ti\simplelink\ble_cc26xx_2_01_00_xxxx x
IAR EW ARM IDE	7.40.2	Windows default
Code Composer Studio IDE	6.1.0	Windows default
TI-RTOS	2_13_00_06	C:\ti\tirtos_simplelink_2_13_00_06
XDC Tools	3_31_01_33_core	C:\ti\ xdctools_3_31_01_33_core
Sensor Controller Studio	1.0.1	Windows default
BTool PC Application	1.41.05	Windows default
SmartRF™ Flash Programmer 2	1.6.2	Windows default
SmartRF Studio 7	2.1.0	Windows default

2.5.2 IAR

IAR contains many features beyond the scope of this document. More information and documentation can be found at www.iar.com.

2.5.2.1 Configuring IAR Embedded Workbench for ARM

To configure the IAR embedded workbench for ARM do the following.

1. Download and install IAR EW ARM version 7.40.2. (This is the official version of IAR supported and tested for this release.) To get IAR choose one of the following methods:
   - Download the IAR Embedded Workbench 30-Day Evaluation Edition – This version of IAR is free, has full functionality, and includes all of the standard features. Download the IAR 30-day Evaluation Edition from http://supp.iar.com/Download/SW/?item=EWARM-EVAL.
   - Purchase the full-featured version of IAR Embedded Workbench – For complete BLE application development using the CC2640, TI recommends purchasing the complete version of IAR without any restrictions. You can find the information on purchasing the complete version of IAR at http://www.iar.com/en/Products/IAR-Embedded-Workbench/ARM/.
2. Run the ti_emupack_setup.exe file in the IAR installation, <iar_install>\arm\drivers\ti-xds.

   NOTE: IAR is usually installed to C:\Program Files (x86)\IAR Systems.
3. Select Run as Administrator when installing the emupack file.
4. Install to C:\ti\ccs_base (default).

   NOTE: For full verbosity during building, TI recommends showing all the build output messages.
5. Navigate to Set Tools→ Options→ Messages.

Copyright © 2015–2016, Texas Instruments Incorporated

6. Toggle Show Build Messages to All (see Figure 2-4).

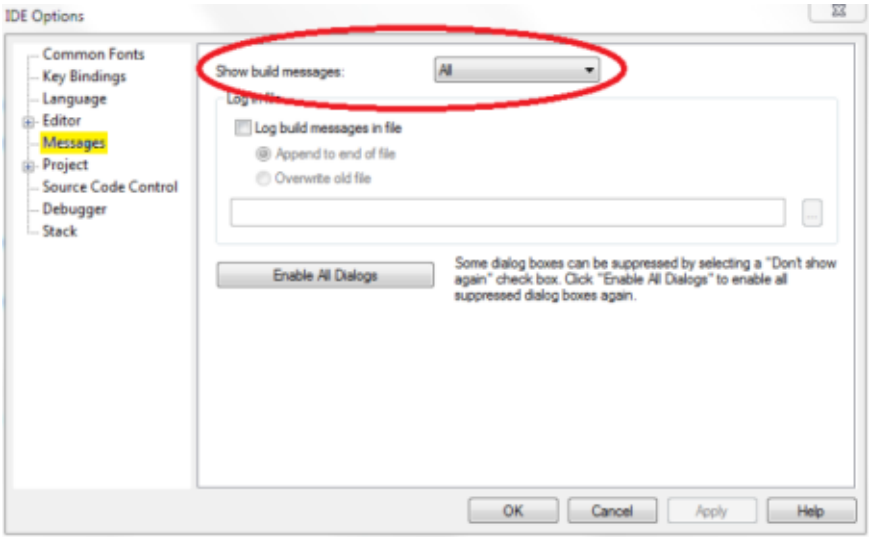


Figure 2-4. Full Verbosity

7. Navigate to Tools→ Custom Argument Variables.
8. Verify Custom Argument Variables points to the installed TI-RTOS and XDC tool paths set in the CC26xx TI-RTOS group (see Figure 2-5 for the TI-RTOS and XDC default tool paths).

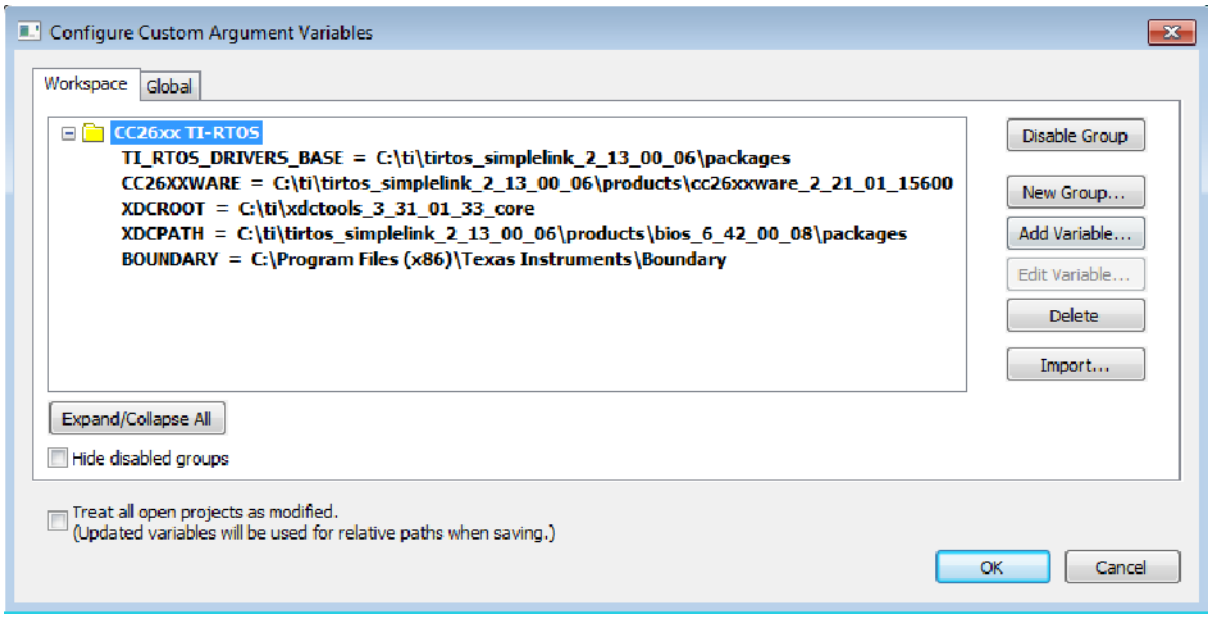


Figure 2-5. Custom Argument Variables

NOTE: If any additional argument groups on the Workspace or Global tabs are present that conflict with the CC26xx TI-RTOS group, disable the groups.

Copyright © 2015–2016, Texas Instruments Incorporated

2.5.2.2 Using IAR Embedded Workbench

This section describes how to open and build an existing project.

NOTE: The SimpleBLEPeripheral project is referenced throughout this document. All of the *Bluetooth* low energy projects included in the development kit have a similar structure.

2.5.2.2.1 Open an Existing Project

To open an existing project, do the following.

1. Open the IAR Embedded Workbench IDE from the Start Menu.
2. Navigate to File→ Open→ Workspace.
3. Select $BLE_INSTALL$\Projects\ble\SimpleBLEPeripheral\CC26xx\IAR\SimpleBLEPeripheral.eww.

NOTE: This workspace file is for the SimpleBLEPeripheral project. When selected, the files associated with the workspace become visible in the Workspace pane on the left side of the screen. See Figure 2-6.

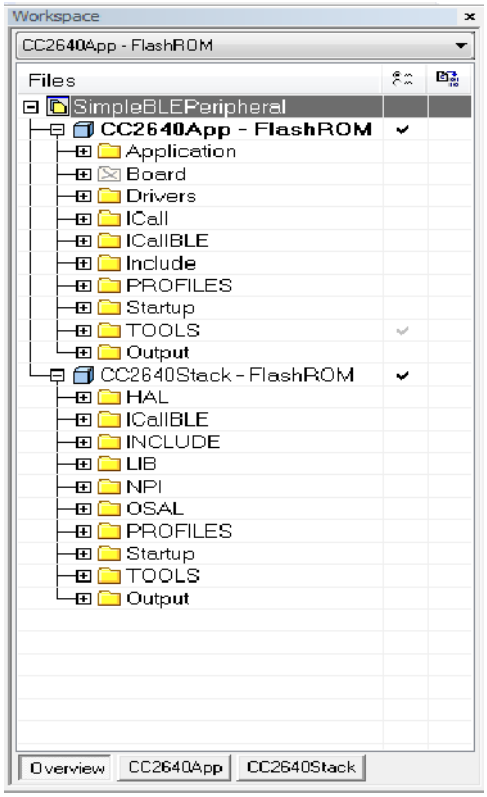


Figure 2-6. IAR Workspace Pane

This and all CC2640 project workspaces contain the following projects.

- The application project (CC2640App)
- The *Bluetooth* low energy protocol stack project (CC2640 stack)

Copyright © 2015–2016, Texas Instruments Incorporated

Select either project as the active project by clicking the respective tab at the bottom of the workspace pane. In Figure 2-6, the Overview tab is selected. This tab displays the file structure for both projects simultaneously. In this case, use the drop-down menu at the top of the workspace pane to select the active project. Each of these projects produces a separate downloadable object. TI chose this dual-image architecture so that the application can be updated independent of the stack. The SimpleBLEPeripheral sample project is the primary reference target for the description of a generic application in this guide. The SimpleBLEPeripheral project implements a basic *Bluetooth* low energy peripheral device including a GATT server with GATT services. This project can be a framework for developing peripheral-role applications.

2.5.2.2.2 Compile and Download

NOTE:

- Do not modify the CPU Variant in the project settings. All sample projects are configured with a CPU type, and changing this setting (that is, from CC2640 to CC2650) may result in build errors.
- All CC2640 and CC2650 code is binary-compatible and interchangeable for *Bluetooth* low energy-Stack software builds.
- The CPU type is the same for all silicon package types.

Because the workspace is split into two projects (application and stack), the following is the specific sequence for compilation and download.

1. Select Project→ Make to build the application project.
2. Select Project→ Make to build the stack project.
3. Select Project→ Download→ Download Active Application to download the application project.
4. Select Project→ Download→ Download Active Application to download the stack project.
5. Select the application project.

NOTE: For the inital download and whenever the stack project is modified, do the following.

- Choose Project→ Debug without Downloading to load the debug symbols in the debugger without programming the flash memory (it was updated using Download Active Application).

If the stack is not modified, do the following.

6. Select Project→ Make to build the application.
7. Select Project→ Download and Debug to download the application.

NOTE: When the application is downloaded (that is, flash memory programmed), you can use Project→ Debug without Downloading.

2.5.3 Code Composer Studio

CCS contains many features beyond the scope of this guide. For more information and documentation, see http://www.ti.com/tool/CCSTUDIO.

Check the *Bluetooth* low energy SDK release notes to see which CCS version to use and any required workarounds. Object code produced by CCS may differ in size and performance as compared to IAR produced object code.

2.5.3.1 Configure CCS

The following procedure describes installing and configuring the correct version of CCS and the necessary tools.

1. Download version 6.1.0 (or later) of CCS from http://processors.wiki.ti.com/index.php/Download_CCS.
2. Launch setup_ccs_win32.exe.

Copyright © 2015–2016, Texas Instruments Incorporated

3. Select Processor Support→ Simplelink Wireless MCUs→ CC26 Device Support TI ARM Complier.
4. Select Help→ Check for Updates.
5. Select View→ CCS App Center.
6. Select TI Arm Compiler.
7. Select Install Software (see Figure 2-7).

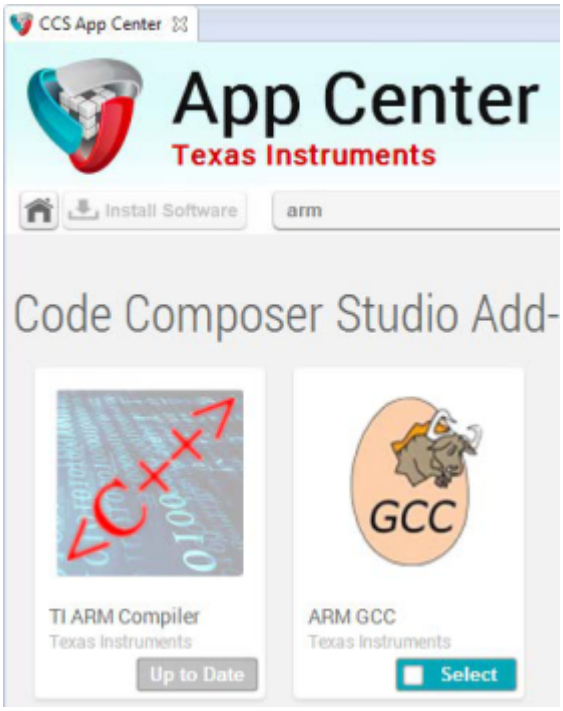


Figure 2-7. CCS App Center

Copyright © 2015–2016, Texas Instruments Incorporated

8. Select Help→ About Code to verify the installation details (see Figure 2-8).

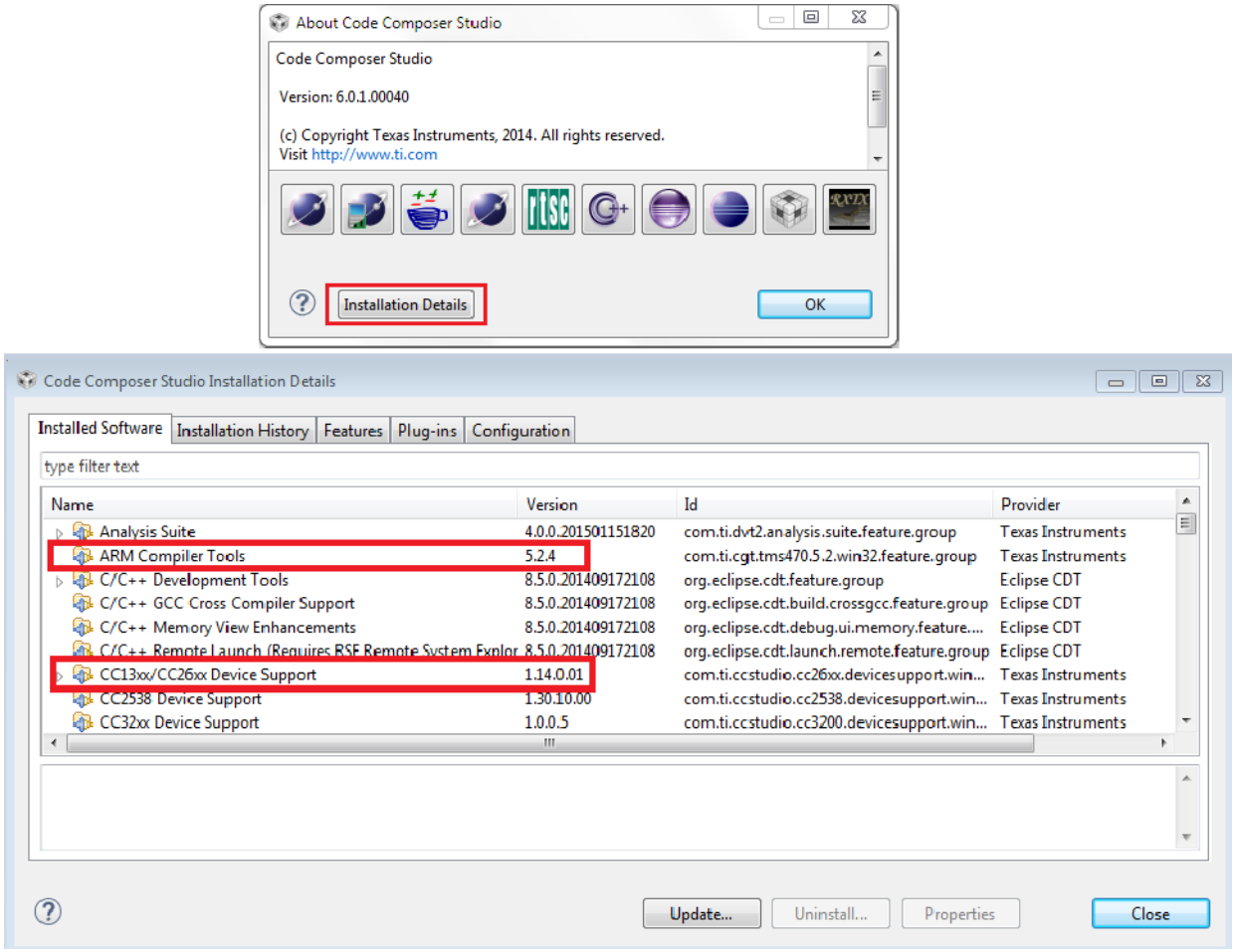


Figure 2-8. Installation Details

9. Verify that the ARM Compiler Tools is version 5.2.4 and that CC26xx Device Support is version 1.14.0.01 (or later).

2.5.3.2 Using CCS

This section describes how to open and build an existing project and references the SimpleBLEPeripheral project. All of the CCS *Bluetooth* low energy projects included in the development kit have a similar structure.

2.5.3.2.1 Import an Existing Project

To import an existing project, do the following.

1. Open the CCS IDE from the Start Menu.
2. Select Project→ Import CCS Project.
3. Select $BLE_INSTALL$\Projects\ble Projects\ble\SimpleBLEPeripheral\CC26xx\CCS.

NOTE: This is the CCS directory for the SimpleBLEPeripheral project. CCS discovers three projects (the application SensorTag, application SmartRF, and stack project).

4. Click the box next to an application project (depending on your development platform) and the stack project to select them.
5. Select Copy projects into workspace.

Copyright © 2015–2016, Texas Instruments Incorporated

6. Click Finish to import (see Figure 2-9).

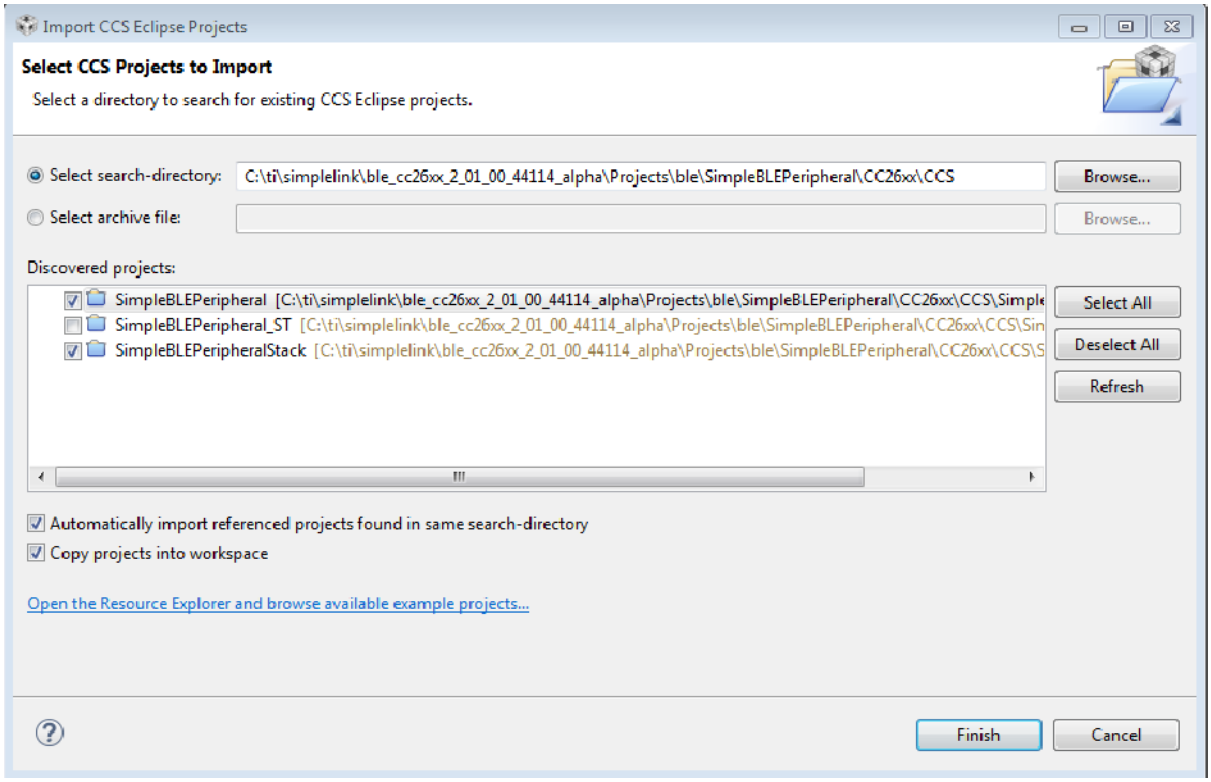


Figure 2-9. Import CSS Projects

Copyright © 2015–2016, Texas Instruments Incorporated

2.5.3.2.2 Workspace Overview

This workspace and all CC2640 project workspaces contain the following projects:

- The application project (SimpleBLEPeripheral)
- The stack project (SimpleBLEPeripheralStack)

Click the project name in the file explorer to select the project as the active project. In Figure 2-10, the application is selected as the active project. Each of these projects produces a separate, downloadable image. TI chose this dual-image architecture so that the application can be updated independent of the stack.

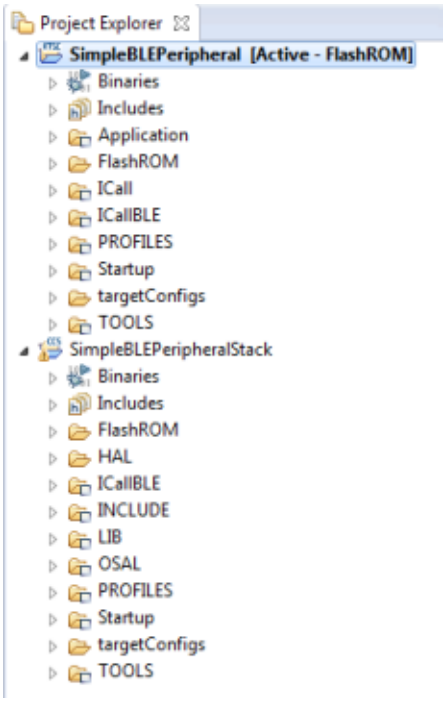


Figure 2-10. Project Explorer Structure

The SimpleBLEPeripheral sample project is the primary reference target for the description of a generic application in this guide. The SimpleBLEPeripheral project implements a basic *Bluetooth* low energy peripheral device including a GATT server with GATT services. This project can be used as a framework for developing peripheral-role applications.

Copyright © 2015–2016, Texas Instruments Incorporated

2.5.3.2.3 Compiling and Downloading

NOTE:

- Do not modify the CPU Variant in the project settings.
- All sample projects are configured with a CPU type and changing this setting (that is, from CC2640 to CC2650) may result in build errors.
- All CC2640/CC2650 code is binary compatible and interchangeable for *Bluetooth* low energy software stack builds.
- The CPU type is the same for all silicon package types.

Because the workspace is split into two projects (application and stack), the following is the specific sequence for compilation and download.

1. Set the application project as the active project.
2. Select Project→ Build All to build the project.
3. Set the stack project as the active project.
4. Select Project→ Build All to build the project.
5. Select the application project as the active project.
6. Select Run→ Debug to download the application.
7. Select the stack project as the active project.
8. Select Run→ Debug to download the stack.

NOTE: This procedure is required for the initial download and whenever the stack project is modified.

If the stack project is not modified, do the following.

1. Build the application.
2. Download the application.

2.6 Working With Hex Files

TI configured the application and stack projects to produce an Intel®-extended hex file in their respective output folders. These hex files lack overlapping memory regions and can be programmed individually with a flash programming tool, such as SmartRF Flash Programmer 2. You can combine the application and stack hex files into a super hex file manually or using free tools.

One example for creating the super hex file is with the IntelHex python script hex_merge.py, available at https://launchpad.net/intelhex/+download. To merge the hex files, install Python® 2.7.x and add it to your system path environment variables.

The following code is an example usage to create a merged SimpleBLEPeripheral_merged.hex file.

```
C:\Python27\Scripts>python hexmerge.py -o .\SimpleBLEPeripheral_merged.hex -r 0000:1FFFF
SimpleBLEPeripheralAppFlashROM.hex:0000:1FFFF SimpleBLEPeripheralStackFlashROM.hex --
overlap=error
```


Copyright © 2015–2016, Texas Instruments Incorporated

2.7 Accessing Preprocessor Symbols

Various C preprocessor symbols may need to be set or adjusted at the project level. The procedure to access the preprocessor symbols (predefined symbols) is based on the IDE being used. The following procedure describes how to access and modify preprocessor symbols using IAR and CCS.

NOTE: This procedure is for IAR.

1. Open the Project Options for either project in the C/C++ Compiler Category.
2. Open the Preprocessor tab.
3. View the Defined symbols box (see Figure 2-11).

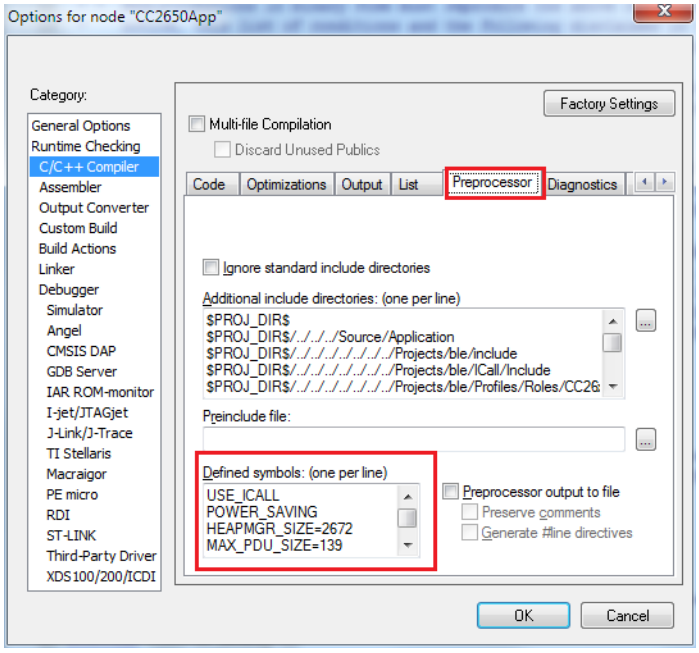


Figure 2-11. Defined Symbols Box

4. Add or edit the preprocessor symbols.

Copyright © 2015–2016, Texas Instruments Incorporated

In CSS, access preprocessor symbols by doing the following.

1. Open the Project Properties for either project.
2. Navigate to CSS Build→ ARM Compiler→ Advanced Options→ Predefined Symbols.
3. Use the buttons highlighted in Figure 2-12 to add, delete, or edit a preprocessor.

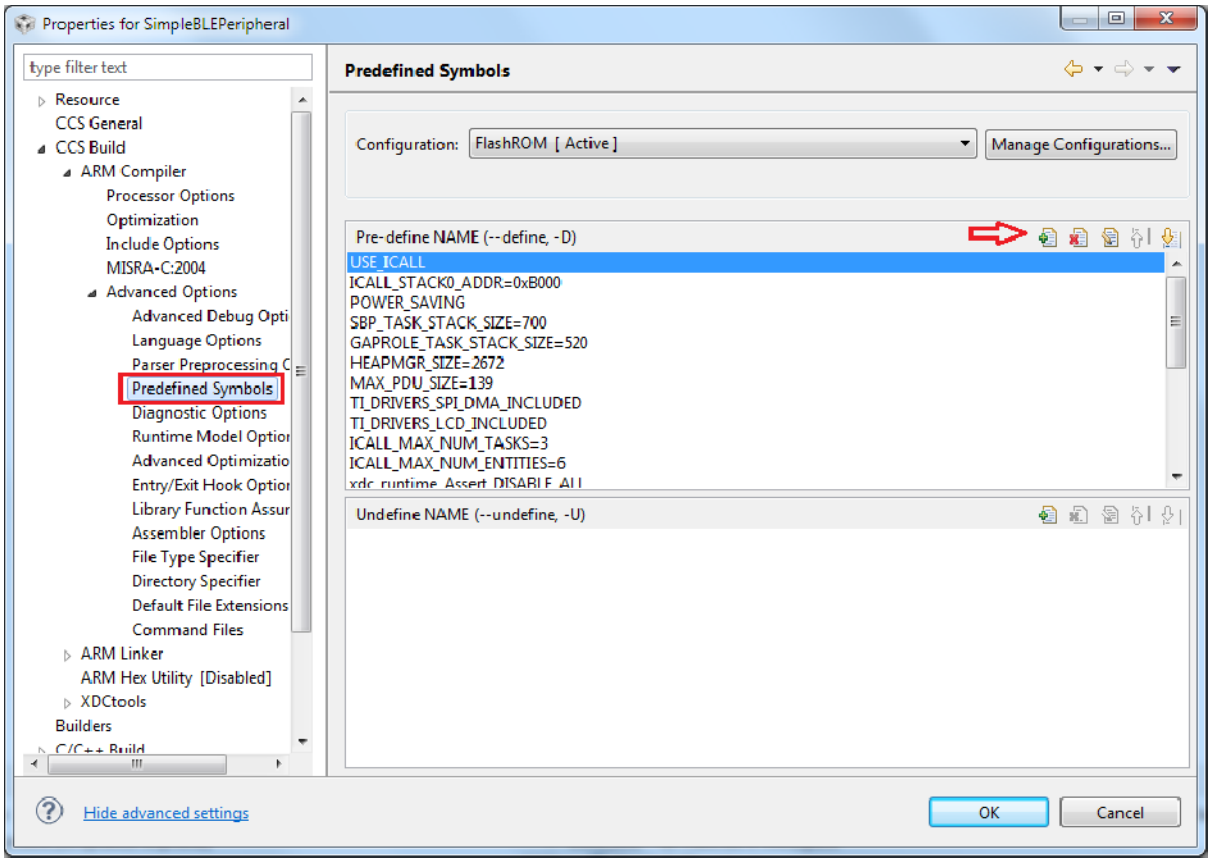


Figure 2-12. CSS Properties

Copyright © 2015–2016, Texas Instruments Incorporated

2.8 Top-Level Software Architecture

The CC2640 *Bluetooth* low energy software environment consists of the following parts:

- An RTOS
- An application image
- A stack image

The TI-RTOS is a real-time, pre-emptive, multithreaded operating system that runs the software solution with task synchronization. Both the application and *Bluetooth* low energy protocol stack exist as separate tasks within the RTOS. The *Bluetooth* low energy protocol stack has the highest priority. A messaging framework called indirect call (ICall) is used for thread-safe synchronization between the application and stack. Figure 2-13 shows the architecture.

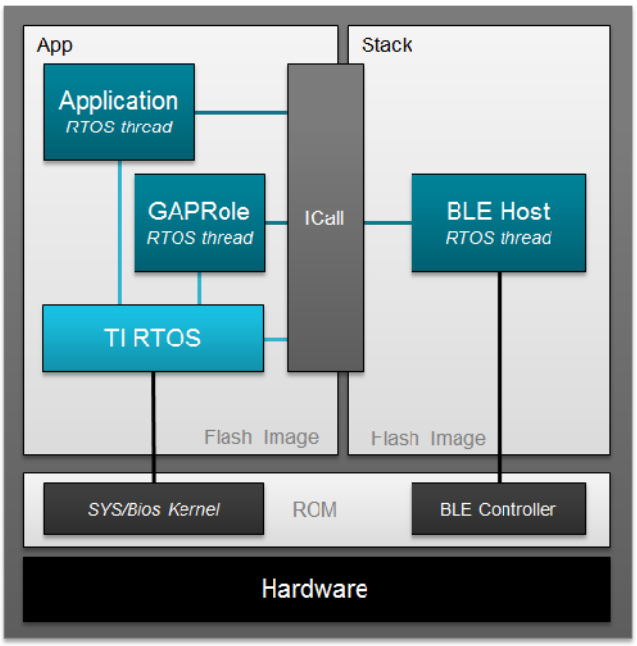


Figure 2-13. Top-Level Software Architecture

- The stack image includes the lower layers of the *Bluetooth* low energy protocol stack from the LL up to and including the GAP and GATT layers (see Figure 1-2). Most of the *Bluetooth* low energy protocol stack code is provided as a library.
- The application image includes the profiles, application code, drivers, and the ICall module.

2.8.1 Standard Project Task Hierarchy

Considering the SimpleBLEPeripheral project as an example, these tasks are listed by priority. A higher task number corresponds to a higher priority task:

- 5: *Bluetooth* low energy protocol stack task
- 3: GapRole task (peripheral role)
- 1: Application task (SimpleBLEPeripheral)

Section 3.3 introduces RTOS tasks. Chapter 5 describes interfacing with the *Bluetooth* low energy protocol stack. Section 5.2 describes the GapRole task. Section 4.2.1 describes the application task.

Copyright © 2015–2016, Texas Instruments Incorporated

RTOS Overview

TI-RTOS is the operating environment for *Bluetooth* low energy projects on CC2640 devices. The TI-RTOS kernel is a tailored version of the SYS/BIOS kernel and operates as a real-time, pre-emptive, multithreaded operating system with tools for synchronization and scheduling (XDCTools). The SYS/BIOS kernel manages four distinct levels of execution threads (see Figure 3-1).

- Hardware interrupt service routines (ISRs)
- Software interrupt routines
- Tasks
- Background idle functions

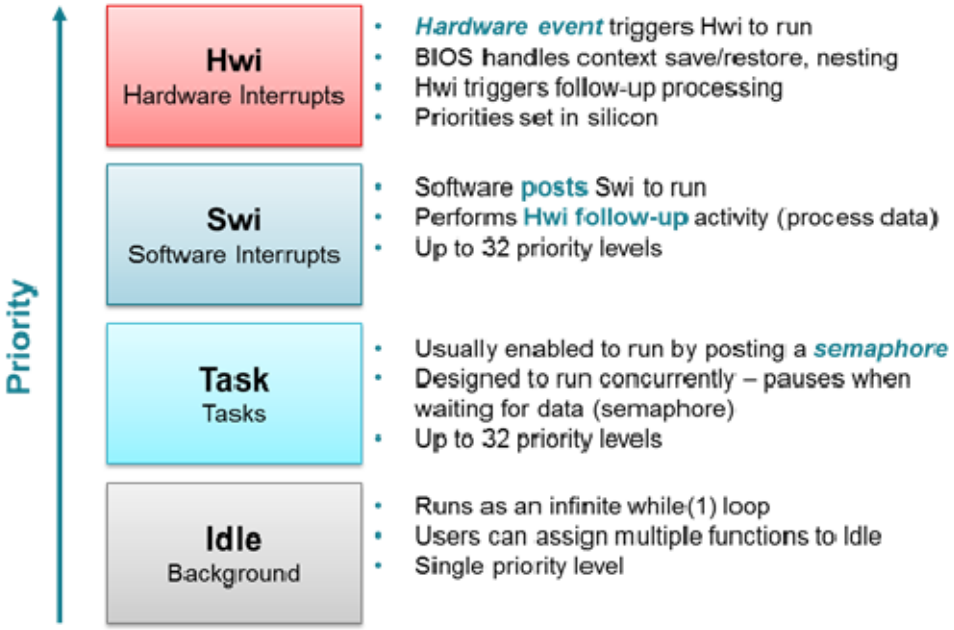


Figure 3-1. RTOS Execution Threads

This section describes these four execution threads and various structures used throughout the RTOS for messaging and synchronization. In most cases, the underlying RTOS functions have been abstracted to higher0level functions in the util.c file. The lower-level RTOS functions are described in the SYS/BIOS module section of the TI SYS/BIOS API Guide. This document also defines the packages and modules included with the TI-RTOS.

3.1 RTOS Configuration

The SYS/BIOS kernel provided with the installer can be modified using the RTOS configuration file (that is, appBLE.cfg for the SimpleBLEPeripheral project). In the IAR project, this file is in the application project workspace TOOLS folder. This file defines the various SYS/BIOS and XDCTools modules in the RTOS compilation, as well as system parameters such as exception handlers and timer-tick speed. The RTOS must then be recompiled for these changes to take effect by recompiling the project.

Copyright © 2015–2016, Texas Instruments Incorporated

The default project configuration is to use elements of the RTOS from the CC26xx ROM. In this case, some RTOS features are unavailable. If any ROM-unsupported features are added to the RTOS configuration file, use an RTOS in flash configuration. Using RTOS in flash consumes additional flash memory. The default RTOS configuration supports all features required by the respective example projects in the SDK.

See the TI-RTOS documentation for a full description of configuration options.

NOTE: If the RTOS configuration file is changed, do the following.

1. Delete the configPkg folder to force a rebuild of the RTOS. For example: $PROJ_DIR$\CC26xx\IAR\Application\CC2650\ConfigPkg
2. Select Project→ Rebuild All to rebuild the application project and build the RTOS. (The RTOS library is compiled as part of the Pre-Build phase of the Application Project.)

3.2 Semaphores

The kernel package provides several modules for synchronizing tasks such as the semaphore. Semaphores are the prime source of synchronization in the CC2640 software and are used to coordinate access to a shared resource among a set of competing tasks (that is, the application and *Bluetooth* low energy stack). Semaphores are used for task synchronization and mutual exclusion.

Figure 3-2 shows the semaphore functionality. Semaphores can be counting semaphores or binary semaphores. Counting semaphores keep track of the number of times the semaphore is posted with Semaphore_post(). When a group of resources are shared between tasks, this function is useful. Such tasks might call Semaphore_pend() to see if a resource is available before using one. Binary semaphores can have only two states: available (count = 1) and unavailable (count = 0). Binary semaphores can be used to share a single resource between tasks or for a basic-signaling mechanism where the semaphore can be posted multiple times. Binary semaphores do not keep track of the count; they track only whether the semaphore has been posted.

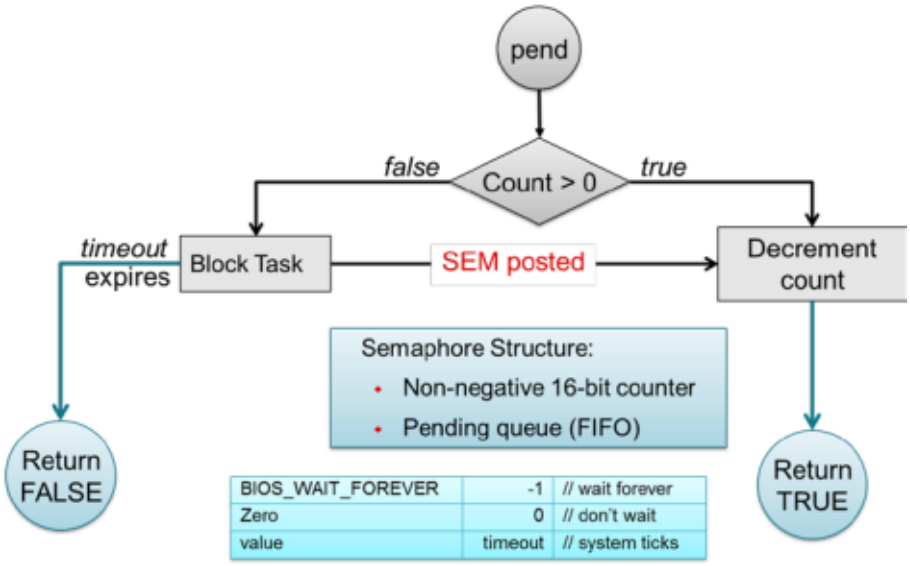


Figure 3-2. Semaphore Functionality

Copyright © 2015–2016, Texas Instruments Incorporated

3.2.1 Initializing a Semaphore

The following code depicts how a semaphore is initialized in RTOS. An example of this in the SimpleBLEPeripheral project is when a task is registered with the ICall module: ICall_registerApp() which eventually calls ICall_primRegisterApp(). These semaphores coordinate task processing. Section 4.2 describes this coordination further.

```
Semaphore_Handle sem;
sem = Semaphore_create(0, NULL, NULL);
```

3.2.2 Pending a Semaphore

Semaphore_pend() is a blocking call that allows another task to run while waiting for a semaphore. The time-out parameter allows the task to wait until a time-out, wait indefinitely, or not wait at all. The return value indicates if the semaphore was signaled successfully.

```
Semaphore_pend(sem, timeout);
```

3.2.3 Posting a Semaphore

Semaphore_post() signals a semaphore. If a task is waiting for the semaphore, this call removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, Semaphore_post() increments the semaphore count and returns. For a binary semaphore, the count is always set to 1.

```
Semaphore_post(sem);
```

3.3 RTOS Tasks

RTOS *tasks* are equivalent to independent threads that conceptually execute functions in parallel within a single C program. In reality, switching the processor from one task to another helps achieve concurrency.

Each task is always in one of the following modes of execution:

- Running: task is currently running
- Ready: task is scheduled for execution
- Blocked: task is suspended from execution
- Terminated: task is terminated from execution
- Inactive: task is on inactive list

One (and only one) task is always running, even if only the idle task (see Figure 3-1). The current running task can be blocked from execution by calling certain task module functions, as well as functions provided by other modules like semaphores. The current task can also terminate itself. In either case, the processor is switched to the highest priority task that is ready to run. See the task module in the package ti.sysbios.knl section of the TI SYS/BIOS API Guide for more information on these functions.

Numeric priorities are assigned to tasks, and multiple tasks can have the same priority. Tasks are readied to execute by highest to lowest priority level; tasks of the same priority are scheduled in order of arrival. The priority of the currently running task is never lower than the priority of any ready task. The running task is preempted and rescheduled to execute when there is a ready task of higher priority. In the SimpleBLEPeripheral application, the *Bluetooth* low energy protocol stack task is given the highest priority (5) and the application task is given the lowest priority (1).

Each RTOS task has an initialization function, an event processor, and one or more callback functions.

Copyright © 2015–2016, Texas Instruments Incorporated

3.3.1 Creating a Task

When a task is created, it has its own runtime stack for storing local variables as well as further nesting of function calls. All tasks executing within a single program share a common set of global variables, accessed according to the standard rules of scope for C functions. This set of memory is the context of the task. The following is an example of the application task being created in the SimpleBLEPeripheral project.

```
void SimpleBLEPeripheral_createTask(void)
{
  Task_Params taskParams;

  // Configure task
  Task_Params_init(&taskParams);
  taskParams.stack = sbpTaskStack;
  taskParams.stackSize = SBP_TASK_STACK_SIZE;
  taskParams.priority = SBP_TASK_PRIORITY;

  Task_construct(&sbpTask, SimpleBLEPeripheral_taskFxn, &taskParams, NULL);
}
```

The task creation is done in the main() function, before the SYS/BIOS scheduler is started by BIOS_start(). The task executes at its assigned priority level after the scheduler is started.

TI recommends using the existing application task for application-specific processing. When adding an additional task to the application project, the priority of the task must be assigned a priority within the RTOS priority-level range, defined in the appBLE.cfg RTOS configuration file.

```
/* Reduce number of Task priority levels to save RAM */
Task.numPriorities = 6;
```

Do not add a task with a priority equal to or higher than the *Bluetooth* low energy protocol stack task and related supporting tasks (for example, the GapRole task). See Section 2.8.1 for details on the system task hierarchy.

Ensure the task has a minimum task stack size of 512 bytes of predefined memory. At a minimum, each stack must be large enough to handle normal subroutine calls and one task preemption context. A task preemption context is the context that is saved when one task preempts another as a result of an interrupt thread readying a higher priority task. Using the TI-RTOS profiling tools of the IDE, the task can be analyzed to determine the peak task stack usage.

NOTE: The term *created* describes the instantiation of a task. The actual TI-RTOS method is to construct the task. See Section 3.11.6 for details on constructing RTOS objects.

Copyright © 2015–2016, Texas Instruments Incorporated

3.3.2 Creating the Task Function

When a task is constructed, a function pointer to a task function (for example, SimpleBLEPeripheral_taskFxn) is passed to the Task_Construct function. When the task first gets a chance to process, this is the function which the RTOS runs. Figure 3-3 shows the general topology of this task function.

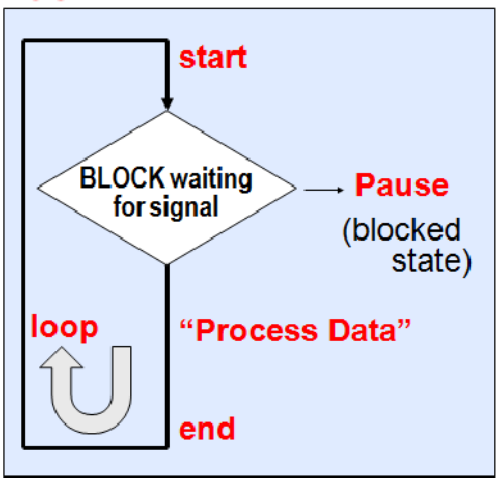


Figure 3-3. General Task Topology

In the SimpleBLEPeripheral task, the task spends most of its time in the blocked state, where it is pending a semaphore. When the semaphore of the task is posted to from an ISR, callback function, queue, and so forth, the task becomes ready, processes the data, and returns to this paused state. See Section 4.2.1 for more detail on the functionality of the SimpleBLEPeripheral task.

3.4 Clocks

Clock instances are functions that can be scheduled to run after a certain number of clock ticks. Clock instances are either one-shot or periodic. These instances start immediately upon creation, are configured to start after a delay, and can be stopped at any time. All clock instances are executed when they expire in the context of a software interrupt. The following example shows the minimum resolution is the RTOS clock tick period set in the RTOS configuration.

```
/* 10 us tick period */
Clock.tickPeriod = 10;
```

Each tick, which is derived from the RTC, launches a clock SWI that compares the running tick count with the period of each clock to determine if the associated function should run. For higher-resolution timers, TI recommends using a 16-bit hardware timer channel or the sensor controller.

Copyright © 2015–2016, Texas Instruments Incorporated

3.4.1 API

You can use the RTOS clock module functions directly (see the clock module in the SYS/BIOS API 0). For usability, these functions have been extracted to the following functions in util.c.

Clock_Handle Util_constructClock (Clock_Struct *pClock, Clock_FuncPtr clockCB, uint32_t clockDuration, uint32_t clockPeriod, uint8_t startFlag,UArg arg)

Initialize a TIRTOS Clock instance.

Parameters

- pClock – pointer to clock instance structure
- clockCB – function to be called upon clock expiration
- clockDuration – length of first expiration period
- clockPeriod – length of subsequent expiration periods. If set to 0, clock is a one-shot clock.
- startFlag – TRUE to start immediately, FALSE to wait. If FALSE, Util_startClock() must be called later.
- arg – argument passed to callback function

Returns Handle to the Clock instance

void Util_startClock(Clock_Struct *pClock)

Start an (already constructed) clock.

Parameters pClock – pointer to clock structure

bool Util_isActive(Clock_Struct *pClock)

Determine if a clock is currently running.

Parameters pClock – pointer to clock structure

Returns TRUE: clock is running.

FALSE: clock is not running.

void Util_stopClock(Clock_Struct *pClock)

Stop a clock.

Parameters: pClock – pointer to clock structure

Copyright © 2015–2016, Texas Instruments Incorporated

3.4.2 Functional Example

The following example from the SimpleBLEPeripheral project details the creation of a clock instance and how to handle the expiration of the instance.

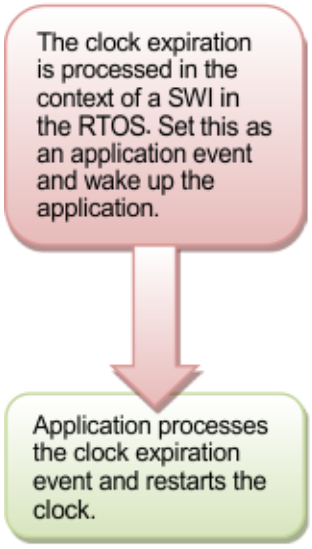
1. Define the clock handler function to service the clock expiration SWI.
   simpleBLEPeripheral.c:

```
//clock handler function
static void SimpleBLEPeripheral_clockHandler(UArg arg)
{
  // Store the event.
  events |= arg;

  // Wake up the application.
}
```

2. Create the clock instance.
   simpleBLEPeripheral.c:

```
// Clock instances for internal periodic events.
static Clock_Struct periodicClock;

// Create one-shot clocks for internal periodic events.
Util_constructClock(&periodicClock, SimpleBLEPeripheral_clockHandler,
                    SBP_PERIODIC_EVT_PERIOD, 0, false, SBP_PERIODIC_EVT);
```

3. Wait for the clock instance to expire and process in the application context (in the following flow diagram, green corresponds to the processor running in the application context and red corresponds to an SWI).



simpleBLEPeripheral.c:

```
//clock handler function
static void SimpleBLEPeripheral_clockHandler(UArg arg)
{
  // Store the event.
  events |= arg;

  // Wake up the application.
  Semaphore_post(sem);
}
```

simpleBLEPeripheral.c:

```
//handle event in application task handler
if (events & SBP_PERIODIC_EVT)
{
  events &= ~SBP_PERIODIC_EVT;

  Util_startClock(&periodicClock);

  // Perform periodic application task
  SimpleBLEPeripheral_performPeriodicTask();
}
```

3.5 Queues

Queues let applications process events in order by providing a FIFO ordering for event processing. A project may use a queue to manage internal events coming from application profiles or another task. Clocks must be used when an event must be processed in a time-critical manner. Queues are more useful for events that must be processed in a specific order.

Copyright © 2015–2016, Texas Instruments Incorporated

The Queue module provides a unidirectional method of message passing between tasks using a FIFO. In Figure 2-13, a queue is configured for unidirectional communication from task A to task B. Task A pushes messages onto the queue and task B pops messages from the queue in order. Figure 3-4 shows the queue messaging process.

Figure 3-4. Queue Messaging Process

3.5.1 API

The RTOS queue functions have been abstracted into functions in the util.c file. See the Queue module in the TI SYS/BIOS API Guide for the underlying functions. These utility functions combine the Queue module with the ability to notify the recipient task of an available message through semaphores. In CC2640 software, the semaphore used for this process is the same semaphore that the given task uses for task synchronization through ICall. For an example of this, see the SimpleBLECentral_enqueueMsg() function. Queues are commonly used to limit the processing time of application callbacks in the context of the higher level priority task. In this manner, the higher priority task pushes a message to the application queue for processing later instead of immediate processing in its own context. Section 3.5 further describes this process.

Queue_Handle Util_constructQueue(Queue_Struct *pQueue)

Initialize an RTOS queue.

Parameters:	pQueue – pointer to queue instance
Returns	Handle to the Queue instance

uint8_t Util_enqueueMsg(Queue_Handle msgQueue, Semaphore_Handle sem, uint8_t *pMsg)

Creates a queue node and puts the node in an RTOS queue.

Parameters	msgQueue – queue handle sem – event processing semaphore of the task with which the queue is associated pMsg – pointer to message to be queued
Returns	TRUE – Message was successfully queued. FALSE – Allocation failed and message was not queued.

uint8_t *Util_dequeueMsg(Queue_Handle msgQueue)

De-queues the message from an RTOS queue.

Parameters	msgQueue – queue handle
Returns	NULL: no message to dequeue Otherwise: pointer to dequeued message

Copyright © 2015–2016, Texas Instruments Incorporated

3.5.2 Functional Example

The following queue example from the simpleBLEPeripheral project follows the handling of a button press from HWI ISR to processing in the application context.

1. Define the enqueue function of the task so that it uses the semaphore to wake.

```
static uint8_t SimpleBLECentral_enqueueMsg(uint8_t event, uint8_t status,
                                           uint8_t *pData)
{
  sbcEvt_t *pMsg;

  // Create dynamic pointer to message.
  if (pMsg = ICall_malloc(sizeof(sbcEvt_t)))
  {
    pMsg->event = event;
    pMsg->status = status;
    pMsg->pData = pData;

    // Enqueue the message.
    return Util_enqueueMsg(appMsgQueue, sem, (uint8_t *)pMsg);
  }

  return FALSE;
}
```

2. Statically allocate and then construct queue.

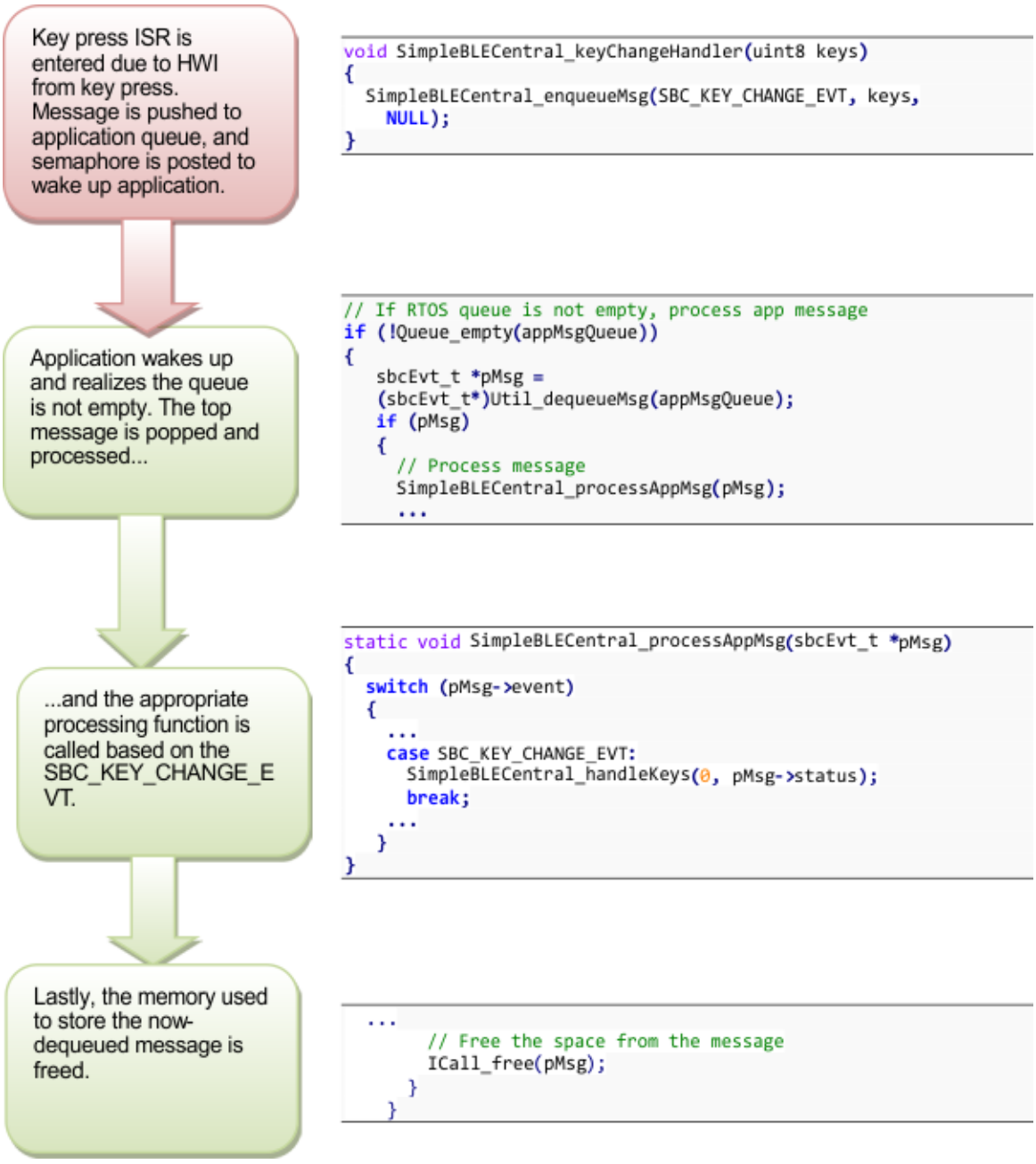
```
// Queue object used for app messages
static Queue_Struct appMsg;
static Queue_Handle appMsgQueue;
```

…

```
// Create an RTOS queue for messages to be sent to app.
appMsgQueue = Util_constructQueue(&appMsg);
```

3. Wait for button to be pressed and processed in application context (in the following diagram, green corresponds to the processor running in the application context and red corresponds to an HWI).

Copyright © 2015–2016, Texas Instruments Incorporated



3.6 Idle Task

The Idle module specifies a list of functions to be called when no other tasks are running in the system. In the CC2640 software, the idle task runs the Power Policy Manager.

3.7 Power Management

All power-management functionality is handled by the peripheral drivers and the *Bluetooth* low energy protocol stack. This feature can be enabled or disabled by including or excluding the POWER_SAVING preprocessor-defined symbol. When POWER_SAVING is enabled, the device enters and exits sleep as required for *Bluetooth* low energy events, peripheral events, application timers, and so forth. When POWER_SAVING is undefined, the device stays awake. See Section 9.2 for steps to modify preprocessor-defined symbols.

Copyright © 2015–2016, Texas Instruments Incorporated

More information on power-management functionality, including the API and a sample use case for a custom UART driver, can be found in the TI-RTOS Power Management for CC26xx included in the RTOS install. These APIs are required only when using a custom driver.

Also see *Measuring Bluetooth Smart Power Consumption* (SWRA478) for steps to analyze the system power consumption and battery life.

3.8 Hardware Interrupts

Hardware interrupts (HWIs) handle critical processing that the application must perform in response to external asynchronous events. The SYS/BIOS device-specific HWI modules are used to manage hardware interrupts. Specific information on the nesting, vectoring, and functionality of interrupts can be found in the *TI CC26xx Technical Reference Manual* (SWCU117). The SYS/BIOS User Guide details the HWI API and provides several software examples.

HWIs are abstracted through the peripheral driver to which they pertain to (see the relevant driver inChapter 6). Chapter 9 provides an example of using GPIOs as HWIs. Abstracting through the peripheral driver to which they pertain is the preferred method of using interrupts. Using the Hwi_plug() function, ISRs can be written which do not interact with SYS/BIOS. These ISRs must do their own context preservation to prevent breaking the time-critical *Bluetooth* low energy stack.

For the *Bluetooth* low energy protocol stack to meet RF time-critical requirements, all application-defined HWIs execute at the lowest priority. TI does not recommend modifying the default HWI priority when adding new HWIs to the system. No application-defined critical sections should exist to prevent breaking the RTOS or time-critical sections of the *Bluetooth* low energy protocol stack. Code executing in a critical section prevents processing of real-time interrupt-related events.

Copyright © 2015–2016, Texas Instruments Incorporated

3.9 Software Interrupts

See TI SYS/BIOS API Guide for detailed information about the SWI module. Software interrupts have priorities that are higher than tasks but lower than hardware interrupts (see Figure 3-5). The amount of processing in a SWI must be limited as this processing takes priority over the *Bluetooth* low energy protocol stack task. As described in Section 3.4, the clock module uses SWIs to preempt tasks. The only processing the clock handler SWI does is set an event and post a semaphore for the application to continue processing outside of the SWI. Whenever possible, the Clock module should be used to implement SWIs. A SWI can be implemented with the SWI module as described in TI SYS/BIOS API Guide.

NOTE: To preserve the RTOS heap, the amount of dynamically created SWIs must be limited as described in Section 3.11.6.

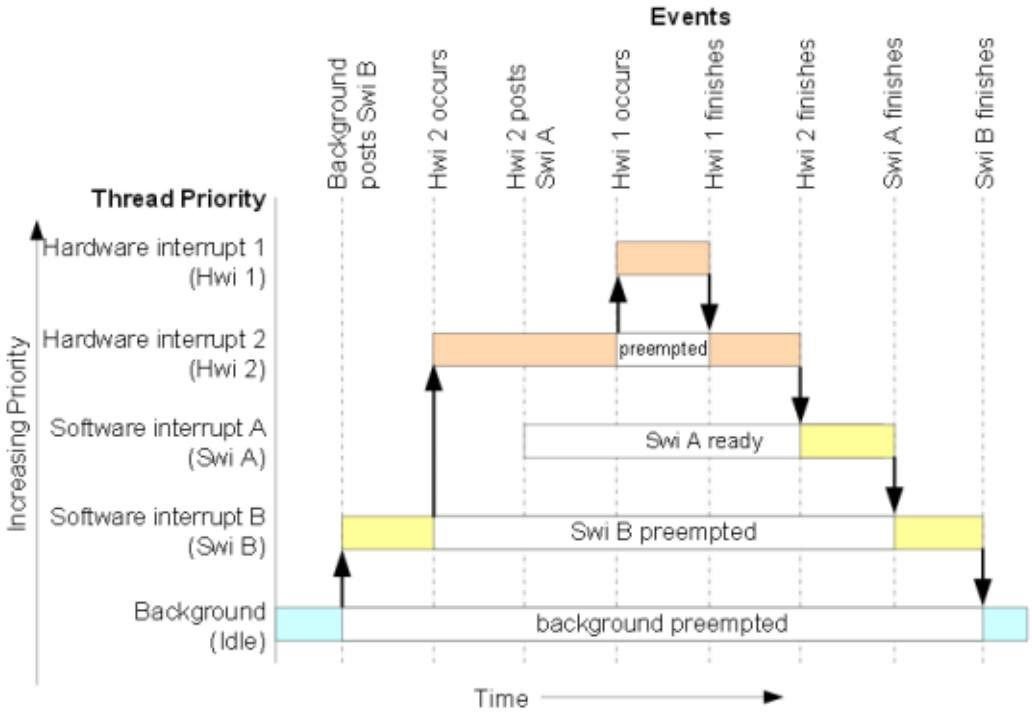


Figure 3-5. Preemption Scenario

3.10 Flash

The flash is split into erasable pages of 4kB. The application and stack projects must each start on a 4kB aligned flash address. The various sections of flash and their associate linker files are as follows.

- Application Image: code space for the application project. This image is configured in the linker configuration file of the application: cc26xx_ble_app.icf (IAR) and cc2650f128_tirtos_ccs.cmd (CCS).
- Stack Image: code space for the stack project. This image is configured in the linker configuration file of the stack: cc26xx_ble_stack.icf (IAR) and cc2650f128_tirtos_ccs_stack.cmd (CCS).
- Simple NV (SNV): area used for nonvolatile memory storage by the GAP Bond Manager and also available for use by the application. See Section 3.10.4 for configuring SNV. When configured, the SNV is part of the stack image.
- Customer Configuration Area (CCA): the last sector of flash used to store customer-specific chip configuration (CCFG) parameters. The unused space of the CCA sector is allocated to the application project. See Section 3.10.5.

Copyright © 2015–2016, Texas Instruments Incorporated

3.10.1 Flash Memory Map

This section describes the flash memory map at the system level. As Figure 3-6 shows, the application linker file point to symbols with a solid arrow and the stack linker file point to symbols with a dashed arrow.

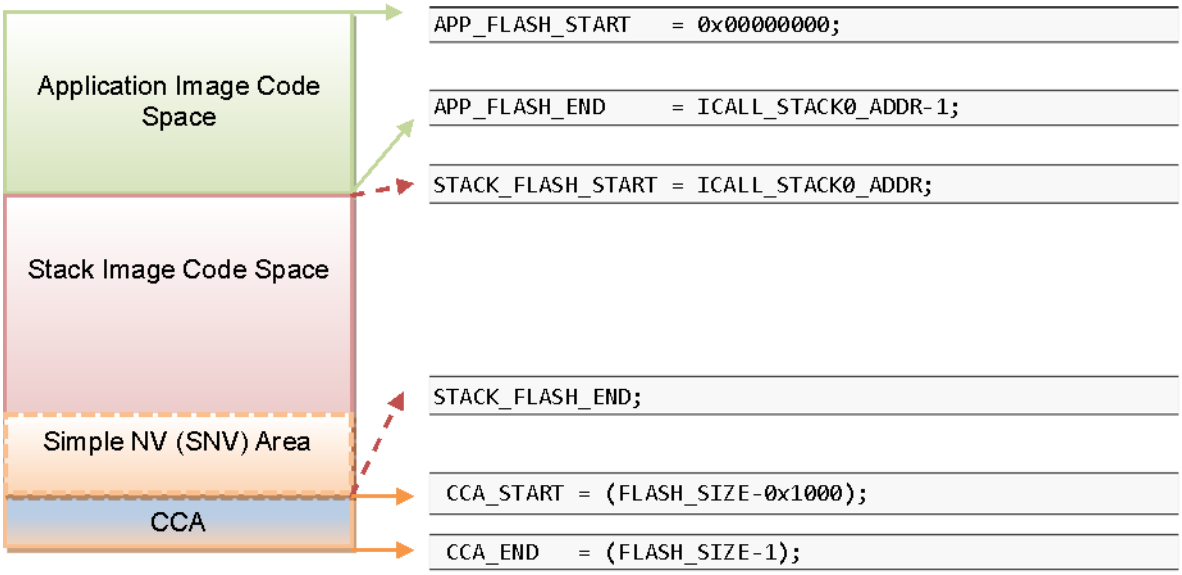


Figure 3-6. System Flash Map

Table 3-1 summarizes the Flash System Map definitions from Figure 3-6 and provides the associated linker definitions or symbols that can be found in the respective IDE linker files.

Table 3-1. Flash System Map Definitions

Symbol/Region	Meaning	Project	CCS Definition	IAR Definition
APP_FLASH_START	Start of flash/Start of App code image	App	APP_BASE	FLASH_START
APP_FLASH_END	End of App code image.	App	ICALL_STACK0_ADDR - APP_BASE - 1	FLASH_END
	(ICALL_STACK0_ADDR-1)			
STACK_FLASH_START	Start of Stack code image (ICALL_STACK0_ADDR)	Stack	ICALL_STACK0_ADDR	FLASH_START
STACK_FLASH_END	End of Stack flash code image, including SNV	Stack	FLASH_SIZE - RESERVED_SIZE - ICALL_STACK0_ADDR	FLASH_END
CCA sector	Last sector of flash. Contains the CCFG.	App	FLASH_LAST_PAGE	FLASH_LAST_PAGE
CCFG region	Location in CCA where Customer Configuration (CCFG) parameters are stored	App	Last 86 bytes of CCA	Last 86 bytes of CCA

Copyright © 2015–2016, Texas Instruments Incorporated

3.10.2 Application and Stack Flash Boundary

The application and stack code images are based on the common ICALL_STACK0_ADDR defined symbol. This value defines the hardcoded flash address (4kB aligned) of the entry function for the stack image: it is essentially the flash address of the application–stack project boundary. To ensure proper linking, both the application and stack projects must use the same ICALL_STACK0_ADDR defined symbol. By default, ICALL_STACK0_ADDR is configured to allocate unused flash to the application project but can be modified manually or automatically through the boundary tool. See Section 3.10.3 for information on manually modifying the flash boundary address. For information on using the boundary tool to configure the flash boundary address, see Section 3.12.

3.10.3 Manually Modifying Flash Boundary

The boundary tool is used to adjust the ICALL_STACK0_ADDR application-stack flash boundary so that the maximum amount of flash memory is allocated to the application project. The ICALL_STACK0_ADDR can be adjusted by performing the following steps.

For IAR:

1. Disable the boundary tool in the stack project (see Section 3.12 for configuring the boundary tool).
2. Adjust ICALL_STACK0_ADDR in TOOLS/IAR-Boundary.xcl:

```
--config_def ICALL_STACK0_ADDR=0x0000B000
```

NOTE: This file is shared with the application and stack projects.

3. Adjust the ICALL_STACK0_ADDR preprocessor-defined symbol in TOOLS/IAR-Boundary.bdef:

```
-D ICALL_STACK0_ADDR=0x0000B000
```

NOTE: This file is shared with the application and stack projects.

4. Rebuild the application and stack projects.
5. Verify there are no build errors in either project.

For CSS:

1. Disable the boundary tool in the stack project (see Section 3.12 for configuring the boundary tool).
2. Adjust the ICALL_STACK0_ADDR in TOOLS/ccsLinkerDefines.cmd:

```
--define=ICALL_STACK0_ADDR=0x0000B000
```

3. Adjust ICALL_STACK0_ADDR preprocessor symbol in the application project:

```
ICALL_STACK0_ADDR=0xB000
```

4. Rebuild the application and stack projects.
5. Verify there are no build errors in either project.

Some points to remember when modifying ICALL_STACK0_ADDR:

- The ICALL_STACK0_ADDR value must be a 4kB aligned address. Increasing the value has the net effect of allocating more flash memory to the application, while decreasing the value increases the allocation to the stack.
- The ICALL_STACK0_ADDR value must match in both the IAR-Boundary.xcl and IAR-Boundary.bdef files for IAR, or in the ccsLinkderDefines.cmd and preprocessor symbol for CCS.
- Both application and stack projects must be rebuilt when ICALL_STACK0_ADDR is modified. Always rebuild the stack project until it builds and links correctly, then rebuild the application.
- If a linker error occurs after manually adjusting ICALL_STACK0_ADDR, verify that adequate flash memory is allocated to each project.

Copyright © 2015–2016, Texas Instruments Incorporated

3.10.4 Using Simple NV

The SNV area of flash is used for storing persistent data, such as encryption keys from bonding or to store custom parameters. The protocol stack can be configured to reserve up to two 4kB flash pages for SNV. To minimize the number of erase cycles on the flash, the SNV manager performs compactions on the flash sector (or sectors) when the sector has 80% invalidated data. A compaction is the copying of valid data to a temporary area followed by an erase of the sector where the data was previously stored. Depending on the OSAL_SNV value as described in Table 3-2, this valid data is then either placed back in the newly erased sector or remains in a new sector. The number of flash sectors allocated to SNV can be configured by setting the OSAL_SNV preprocessor symbol in the stack project. Table 3-2 lists the valid values that can be configured as well as the corresponding trade-offs.

Table 3-2. OSAL_SNV Values

OSAL_SNV Value	Description
0	SNV is disabled. Storing of bonding keys in NV is not possible. Maximizes code space for the application and/or stack project. GAP Bond Manager must be disabled. In the Stack project, set pre-processor symbol NO_OSAL_SNV and disable GAP Bond Manager. See Section 10.4 for configuring *Bluetooth* low energy protocol stack features.
1	One flash sector is allocated to SNV. Bonding info is stored in NV. Flash compaction uses flash cache RAM for intermediate storage, thus a power-loss during compaction results in SNV data loss. Also, due to temporarily disabling the cache, a system performance degradation may occur during the compaction. Set preprocessor symbol OSAL_SNV=1 in the Stack project.
2	Default value. Two flash sectors are allocated to SNV. Bonding information is stored in NV. SNV data is protected against power-loss during compaction.

Other values for OSAL_SNV are invalid. Using less than the maximum value has the net effect of allocating more code space to the application or stack project. SNV can be read from or written to using the following APIs.

uint8 osal_snv_read(osalSnvId_t id, osalSnvLen_t len, void *pBuf)

Read data from NV.

Parameters id – valid NV item

len – length of data to read

pBuf – pointer to buffer to store data read

Returns SUCCESS: NV item read successfully

NV_OPER_FAILED: failure reading NV item

uint8 osal_snv_write(osalSnvId_t id, osalSnvLen_t len, void *pBuf)

Write data to NV

Parameters id – valid NV item

len – length of data to write

pBuf – pointer to buffer containing data to be written

Returns SUCCESS: NV item read successfully

NV_OPER_FAILED: failure reading NV item

Copyright © 2015–2016, Texas Instruments Incorporated

Because SNV is shared with other modules in the *Bluetooth* low energy SDK such as the GapBondMgr, carefully manage the NV item IDs. By default, the IDs available to the customer are defined in bcomdef.h.

```
// Customer NV Items - Range  0x80 - 0x8F - This must match the number of Bonding entries
#define BLE_NVID_CUST_START             0x80  //!< Start of the Customer's NV IDs
#define BLE_NVID_CUST_END               0x8F  //!< End of the Customer's NV IDs
```

3.10.5 Customer Configuration Area

The Customer Configuration Area (CCA) occupies the last page of flash and lets a customer configure various chip and system parameters in the Customer Configuration (CCFG) table . The CCFG table is defined in ccfg_appBLE.c which can be found in the start-up folder of the application project. The last 86 bytes of the CCA sector are reserved by the system for the CCFG table. By default, the linker allocates the unused flash of the CCA sector to the application image for code and data use. The linker can be modified to reserve the entire sector for customer parameter data (for example, board serial number and other identity parameters).

The CCA region is defined linker file of the application by the FLASH_LAST_PAGE symbol; placement is based on the IDE:

For CCS:

```
FLASH_LAST_PAGE (RX) :  origin = FLASH_SIZE - 0x1000, length = 0x1000
…
.ccfg           :   > FLASH_LAST_PAGE (HIGH)
```

For IAR:

```
define region FLASH_LAST_PAGE = mem:[from(FLASH_SIZE) - 0x1000 to FLASH_SIZE-1];
…
place at end of FLASH_LAST_PAGE { readonly section .ccfg };
```

See the *TI CC26xx Technical Reference Manual* (SWCU117) for details on CCFG fields and related configuration options.

3.11 Memory Management (RAM)

Similar to flash, the RAM is shared between the application and stack projects. The RAM sections are configured in their respective linker files.

- Application Image: RAM space for the application and shared heaps. This image is configured in the linker configuration file of the application: cc26xx_ble_app.icf (IAR) and cc2650f128_tirtos_ccs.cmd (CCS).
- Stack Image: RAM space for the .bss and .data sections of the stack. This image is configured in the linker configuration file of the stack: cc26xx_ble_stack.icf (IAR) and cc2650f128_tirtos_ccs_stack.cmd (CCS).

Copyright © 2015–2016, Texas Instruments Incorporated

3.11.1 RAM Memory Map

Figure 3-7 shows the system memory map for the default SimpleBLEPeripheral project. This is a summary and the exact memory placement for a given compilation can be found in the SimpleBLEPeripheralApp.map and SimpleBLEPeripheralStack.map files in the output folder in IAR or the FlashROM folder in CCS. See Section 0 for more information about these files. In Figure 3-7, the application linker file contains symbols pointed with a solid arrow and the stack linker file contains symbols pointed with a dashed arrow.

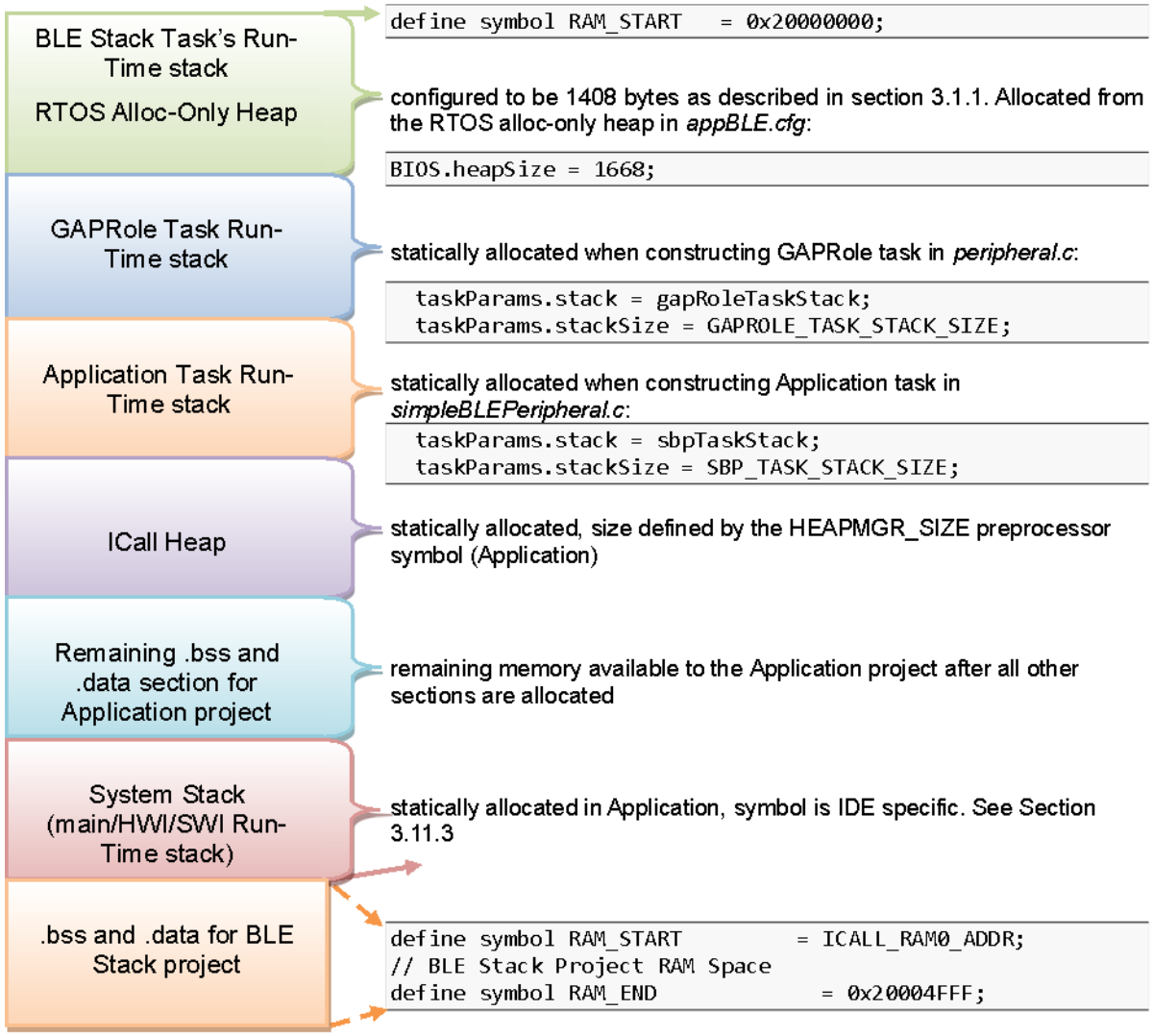


Figure 3-7. System Memory Map

3.11.2 Application and Stack RAM Boundary

The application and stack RAM memory maps are based on the common ICALL_RAM0_ADDR defined symbol. This value defines the hardcoded RAM boundary for the end of the RAM space of the application and the start of the image of the stack .BSS and .DATA sections. Unlike the flash boundary, elements of the stack project (such as task stacks and heaps) are allocated in the application project. To ensure proper linking, both the application and stack projects must use the same ICALL_RAM0_ADDR value. By default, ICALL_RAM0_ADDR is configured to allocate unused RAM to the application project but can be modified manually or automatically through the boundary tool. For information on manually modifying the RAM boundary address, see Section 3.11.4. For information on using the boundary tool to configure the RAM boundary address, see Section 3.12.

Copyright © 2015–2016, Texas Instruments Incorporated

3.11.3 System Stack

Besides the RTOS and ICall heaps, consider other sections of memory. As described in Section 3.3.1, each task has its own runtime stack for context switching. Another runtime stack is used by the RTOS for main(), HWIs, and SWIs. This system stack is allocated in the application linker file to be placed at the end of the RAM of the application.

For IAR, this RTOS system stack is defined by the CSTACK symbol:

```
////////////////////////////////////////////////////////////////////////////////
// Stack
//

define symbol STACK_SIZE           = 0x400;
define symbol STACK_START          = RAM_END + 1;
define symbol STACK_END            = STACK_START - STACK_SIZE;
define block CSTACK with alignment = 8, size = STACK_SIZE { section .stack };
//
define symbol STACK_TOP            = RAM_END + 1;
export symbol STACK_TOP;
//
place at end of RAM { block CSTACK };
```

In IAR, to change the size of the CSTACK, adjust the STACK_SIZE symbol value in the linker file of the application.

For CCS, the RTOS system stack is defined by the Program.stack parameter in the appBLE.cfg RTOS configuration file:

```
/* main() and Hwi, Swi stack size */
Program.stack = 1024;
```

and placed by the linker in the RAM space of the application:

```
/* Create global constant that points to top of stack */
/* CCS: Change stack size under Project Properties    */
__STACK_TOP = __stack + __STACK_SIZE;
```

3.11.4 Manually Modifying the RAM Boundary

The boundary tool is used to adjust the ICALL_RAM0_ADDR application-stack boundary such that the maximum amount of RAM is available to the application project. Although not required, the ICALL_RAM0_ADDR can be manually adjusted by performing the following steps.

For IAR:

1. Disable the boundary tool in the stack project (see Section 3.12 for configuring the boundary tool).
2. Adjust ICALL_STACK0_ADDR in IAR-Boundary.xcl.

```
--config_def ICALL_RAM0_ADDR=0x200043AC
```

3. Rebuild the application and stack projects.
4. Verify there are no build errors in both projects.

For CCS:

1. Disable the boundary tool in the stack project (see Section 3.12 for configuring the boundary tool).
2. Adjust ICALL_RAM0_ADDR in TOOLS/ccsLinkerDefines.cmd.

```
--define=ICALL_RAM0_ADDR=0x200043AC
```

3. Rebuild the application and stack projects.
4. Verify there are no build errors in both projects.

Copyright © 2015–2016, Texas Instruments Incorporated

Some points to remember when modifying ICALL_RAM0_ADDR:

- The ICALL_RAM0_ADDR value must be a 4-byte aligned address.
- Both the application and stack projects must be cleaned and rebuilt when ICALL_RAM0_ADDR is modified.
- Always rebuild the stack project until it builds and links correctly, then rebuild the application.
- If a linker error occurs after manually adjusting ICALL_RAM0_ADDR, verify that each project has adequate RAM allocated to it.

3.11.5 Dynamic Memory Allocation

The system uses two heaps for dynamic memory allocation. The application designer must understand the use of each heap to maximize the use of available memory.

The RTOS is configured with a small heap in the appBLE.cfg RTOS configuration file:

```
var HeapMem = xdc.useModule('xdc.runtime.HeapMem');

BIOS.heapSize = 1668;
```

This heap (HeapMem) is used to initialize RTOS objects and allocate the task runtime stack of the *Bluetooth* low energy protocol stack. TI chose this size of this heap to meet the system initialization requirements. Due to the small size of this heap, TI does not recommend allocating memory from the RTOS heap for general application use. For more information on the TI-RTOS heap configuration, see the Heap Implementations section of the TI-RTOS SYS/BIOS Kernel User's Guide.

The application must use a separate heap. The ICall module statically initializes an area of application RAM, heapmgrHeapStore, which can be used by the various tasks. The size of this ICall heap is defined by the HEAPMGR_SIZE preprocessor definition of the application and is set to 2672 by default for the SimpleBLEPeripheral project. Although the ICall heap is defined in the application project, this heap is also shared with the *Bluetooth* low energy protocol stack. APIs that allocate memory (such as GATT_bm_alloc()) allocate memory from the ICall heap. To increase the size of the ICall heap, adjust the value of the preprocessor symbol HEAPMGR_SIZE in the application project.

To profile the amount of ICall heap used, define the HEAPMGR_METRICS preprocessor symbol in the application project. See heapmgr.h in $BLE_INSTALL$\Components\applib\heap for heap metrics.

The following is an example of dynamically allocating a variable length (n) array using the ICall heap:

```
//define pointer
uint8_t *pArray;

// Create dynamic pointer to array.
if (pArray = (uint8_t*)ICall_malloc(n*sizeof(uint8_t)))
{
  //fill up array
}
else
{
  //not able to allocate
}
```

The following is an example of freeing the previous array:

```
ICall_free(pMsg->payload);
```


Copyright © 2015–2016, Texas Instruments Incorporated

3.11.6 Initializing RTOS Objects

Due to the limited size of the RTOS heap, TI recommends constructing and not creating RTOS objects. Consider the difference between the Clock_construct() and Clock_create() functions. The following shows their definitions from the SYS/BIOS API:

```
Clock_Handle Clock_create(Clock_FuncPtr clockFxn, UInt timeout, const Clock_Params *params, Error_Block *eb);
    // Allocate and initialize a new instance object and return its handle

Void Clock_construct(Clock_Struct *structP, Clock_FuncPtr clockFxn, UInt timeout, const Clock_Params *params);
    // Initialize a new instance object inside the provided structure
```

By declaring a static Clock_Struct object and passing this object to Clock_construct(), the .DATA section for the actual Clock_Struct is used; not the limited RTOS heap. Clock_create() would cause the RTOS to allocate the Clock_Struct using the limited heap of the RTOS.

This example shows how clocks and RTOS objects should be initialized throughout a project. If creating RTOS objects, the size of the RTOS heap may require adjustment in appBLE.cfg.

3.12 Configuration of RAM and Flash Boundary Using the Boundary Tool

The boundary tool is a utility to adjust the respective ICALL_STACK0_ADDR (Flash) and ICALL_RAM0_ADDR (RAM) boundaries shared between the application and stack projects. The boundary tool adjusts the boundaries such that unused flash and RAM is allocated to the application project. This tool eliminates the requirement to manually adjust the respective RAM and flash boundaries when working with the dual-project environment. No project files are modified by the boundary tool. The boundary tool does not modify any source code or perform any compiler or linker optimization; the tool adjusts the respective flash and RAM boundary addresses based on analysis of the map and linker configuration files of the project.

The boundary tool is installed to the following path:

C:\Program Files (x86)\Texas Instruments\Boundary

This path has a ReadMe.txt file that contains additional information about the tool.

3.12.1 Configuring Boundary Tool

The boundary tool uses an XML file, BoundaryConfig.xml, in the install path of the tool to configure default tool options. TI recommends keeping these default values.

Each project in the SDK has a set of configuration files that the linker and compiler of the IDE use to set or adjust the respective flash and RAM values. These configuration files are in each project at the following location:

$BLE_INSTALL$\Projects\ble\<PROJECT>\CC26xx\<IDE>\Config

Where <PROJECT> is the project (for example, SimpleBLEPeripheral) and <IDE> is either IAR or CCS.

- Boundary Linker Configuration File: IAR-Boundary.xcl [IAR] or ccsLinkerDefines.cmd [CCS]. Defines the ICALL_STACK0_ADDR and ICALL_RAM0_ADDR boundary address. This file is in the TOOLS IDE folder and is updated by the boundary tool when an adjustment is required.
- Boundary C Definition File: IAR-Boundary.cdef [IAR] or ccsCompilerDefines.bcfg [CCS]. Due to a limitation of the IAR and CCS linker, ICALL_STACK0_ADDR must also be defined in this file to the same value as the linker configuration file. This file is in the TOOLS IDE folder and is updated by the boundary tool when an adjustment is required.

Copyright © 2015–2016, Texas Instruments Incorporated

3.12.2 Operating the Boundary Tool

The boundary tool (boundary.exe) is invoked as an IDE post-build operation of the stack project. If an adjustment to the RAM and/or flash boundary is required, the boundary tool updates the Boundary Linker Configuration and C Definition Files previously listed and generates a post-build error to notify that a change occurred. To incorporate the updated configuration values, perform a Project→ Rebuild All on the stack project and then the application project. The stack project must build and link correctly before the application can be rebuilt.

In addition to the code and memory sizes, the boundary tool takes into account the number of reserved flash pages when calculating the ICALL_STACK0_ADDR value. Examples of reserved flash pages include the CCA page. Reserved flash pages are defined in the linker configuration file of the stack project.

A prerequisite to using the boundary tool is both the application and stack projects must first compile and link. If a linker error occurs, verify that the change did not exceed the maximum flash and/or RAM size of the device. A linker error may occur when the stack project is configured to use features that require additional flash memory than the default configuration. To allow the stack project to link, manually set the RAM and flash boundary addresses to their maximum values in the Boundary Linker Configuration and C Definition Files:

```
ICALL_RAM0_ADDR=0x20000000
ICALL_STACK0_ADDR=0x00000000
```

NOTE: See Section 3.10.3 and Section 3.11.4 for adjusting the flash and RAM values. These values are used only temporarily to let the stack project link successfully.

When the stack project can link successfully, the boundary tool readjusts the respective boundaries to the optimal value by generating a build error. Perform a Rebuild All as required in both projects.

Unless IAR is configured to Show All Build Messages, the boundary tool should not generate a build error. When IAR is configured to Show All Build Messages, The Project Boundary Address Has Been Moved displays in the build output window. In IAR, set Tools→ Options→ Messages→ Show Build Messages to All.

The following is an example build message output when a boundary change has been performed:

```
////////////////////////////////////////////////////////////////////////////
Boundary Operation Complete
////////////////////////////////////////////////////////////////////////////
<<< WARNING >>>
The Project Boundary Address Has Been Moved Or A Config File Change Is Required
Rebuild This Project For The Address Or Config File Change To Take Effect
////////////////////////////////////////////////////////////////////////////
```

3.12.3 Disabling the Boundary Tool

To disable the boundary tool, do the following.

1. Open the project options for the stack project.
2. Select Build Actions (IAR) or Steps in the CCS build window (CCS).
3. Remove the post-build command line (see Figure 3-8).

TI recommends keeping a copy of this command in case the boundary tool must be restored later. When the boundary tool is disabled, the respective boundary linker and compiler configuration files can be edited according to the procedures in Section 3.10.3 and Section 3.11.4.

Copyright © 2015–2016, Texas Instruments Incorporated

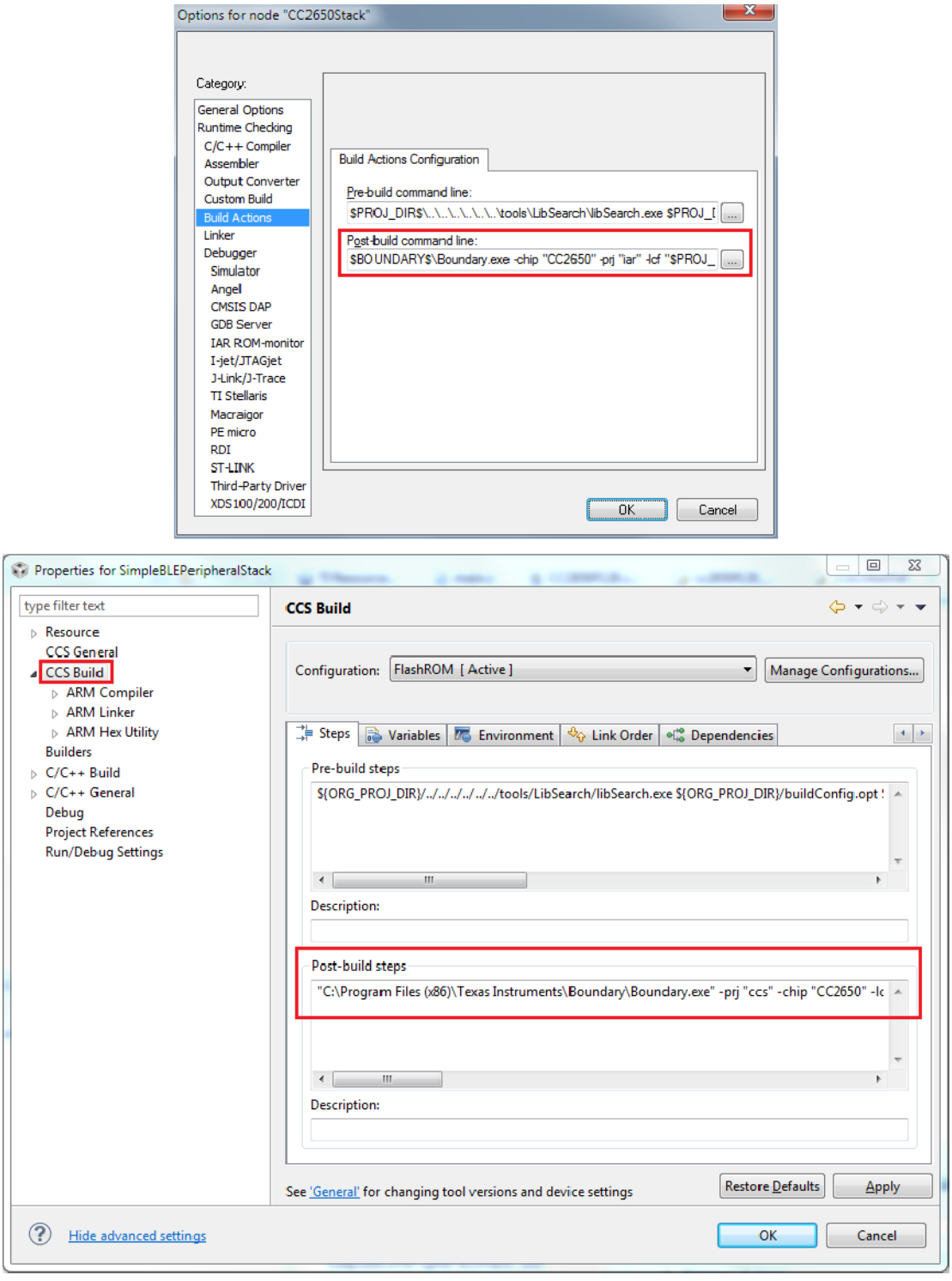


Figure 3-8. Disabling Boundary Tool from Stack Project in IAR (top) and CCS (bottom)

Copyright © 2015–2016, Texas Instruments Incorporated

The Application

This section describes the application portion of the SimpleBLEPeripheral project, which includes the following:

- Pre-RTOS initialization
- SimpleBLEPeripheral task: This is the application task, which is the lowest priority task in the system. The code for this task is in simpleBLEPeripheral.c and simpleBLEPeripheral.h in the Application IDE folder.
- ICall: This interface module abstracts communication between the stack and other tasks.

NOTE: The GAPRole task is also part of the application project workspace. The functionality of this task relates more closely to the protocol stack.

Section 5.2 describes this functionality.

4.1 Start-Up in main()

The main() function inside of main.c in the IDE Start-up folder is the starting point at run time. This point is where the board is brought up with interrupts disabled and drivers are initialized. Also in this function, power management is initialized and the tasks are created or constructed. In the final step, interrupts are enabled and the SYS/BIOS kernel scheduler is started by calling BIOS_start(), which does not return. See Chapter 8 for information on the start-up sequence before main() is reached.

```
int main()
{
  PIN_init(BoardGpioInitTable);

#ifndef POWER_SAVING
    /* Set constraints for Standby, powerdown and idle mode */
    Power_setConstraint(Power_SB_DISALLOW);
    Power_setConstraint(Power_IDLE_PD_DISALLOW);
#endif // POWER_SAVING

    /* Initialize ICall module */
    ICall_init();

    /* Start tasks of external images - Priority 5 */
    ICall_createRemoteTasks();

    /* Kick off profile - Priority 3 */
    GAPRole_createTask();

    SimpleBLEPeripheral_createTask();

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

    return 0;
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

Chapter 3 describes how the application and GAPRole tasks are constructed. The stack task is created here as well in ICall_createRemoteTasks(). The ICall module is initialized through ICall_init(). In terms of the IDE workspace, main.c exists in the application project (when the project is compiled and placed in the allocated section of flash of the application).

4.2 ICall

4.2.1 Introduction

Indirect Call Framework (ICall) is a module that provides a mechanism for the application to interface with the *Bluetooth* low energy protocol stack services (that is, *Bluetooth* low energy stack APIs) as well as certain primitive services provided by the RTOS (for example, thread synchronization). ICall allows the application and protocol stack to operate efficiently, communicate, and share resources in a unified RTOS environment.

The central component of the ICall architecture is the dispatcher, which facilitates the application program interface between the application and the *Bluetooth* low energy protocol stack task across the dual-image boundary. Although most ICall interactions are abstracted within the *Bluetooth* low energy protocol stack APIs (for example, GAP, HCI, and so forth), the application developer must understand the underlying architecture for the *Bluetooth* low energy protocol stack to operate properly in the multithreaded RTOS environment.

The ICall module source code is provided in the ICall and ICall *Bluetooth* low energy IDE folders in the application project.

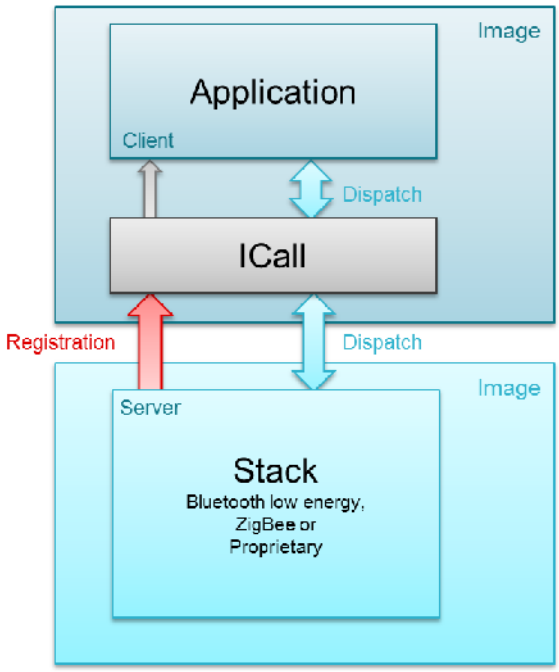


Figure 4-1. ICall Application – Protocol Stack Abstraction

Copyright © 2015–2016, Texas Instruments Incorporated

4.2.2 *ICall Bluetooth* low energy Protocol Stack Service

As Figure 4-1 shows, the ICall core use case involves messaging between a server entity (that is, the *Bluetooth* low energy stack task) and a client entity (for example, the application task).

NOTE: The ICall framework is not the GATT server and client architecture as defined by the *Bluetooth* low energy protocol.

The reasoning for this architecture is as follows:

- To enable independent updating of the application and *Bluetooth* low energy protocol stack
- To maintain API consistency as software is ported from legacy platforms (that is, OSAL for the CC254x) to the TI-RTOS of the CC2640

The ICall *Bluetooth* low energy protocol stack service serves as the application interface to *Bluetooth* low energy stack APIs. When a *Bluetooth* low energy protocol stack API is called by the application internally, the ICall module routes (that is, dispatches) the command to the *Bluetooth* low energy protocol stack and routes messages from the *Bluetooth* low energy protocol stack to the application when appropriate.

Because the ICall module is part of the application project, the application task can access ICall with direct function calls. Because the *Bluetooth* low energy protocol stack executes at the highest priority, the application task blocks until the response is received. Certain protocol stack APIs may respond immediately, but the application thread blocks as the API is dispatched to the *Bluetooth* low energy protocol stack through ICall. Other *Bluetooth* low energy protocol stack APIs may also respond asynchronously to the application through ICall (for example, event updates) with the response sent to the event handler of the application task.

4.2.3 *ICall Primitive Service*

ICall includes a primitive service that abstracts various operating system-related functions. Due to shared resources and to maintain interprocess communication, the application must use the following ICall primitive service functions:

- Messaging and Thread Synchronization
- Heap Allocation and Management

Some of these are abstracted to Util functions (see the relevant module in Chapter 3).

4.2.3.1 Messaging and Thread Synchronization

The Messaging and Thread Synchronization functions provided by ICall enable designing an application to protocol stack interface in the multithreaded RTOS environment.

In ICall, messaging between two tasks occurs by sending a block of message from one thread to the other through a message queue. The sender allocates a memory block, writes the content of the message into the memory block, and then sends (that is, enqueues) the memory block to the recipient. Notification of message delivery occurs using a signaling semaphore. The receiver wakes up on the semaphore, copies the message memory block (or blocks), processes the message, and returns (frees) the memory block to the heap.

The stack uses ICall for notifying and sending messages to the application. ICall delivers these service messages, the application stask receives them, and the messages are processed in the context of the application.

4.2.3.2 Heap Allocation and Management

ICall provides the application with global heap APIs for dynamic memory allocation. The size of the ICall heap is configured with the HEAPMGR_SIZE preprocessor-defined symbol in the application project. See Section 3.11.5 for more details on managing dynamic memory. ICall uses this heap for all protocol stack messaging and to obtain memory for other ICall services. TI recommends that the application uses these ICall APIs to allocate dynamic memory.

Copyright © 2015–2016, Texas Instruments Incorporated

4.2.4 ICall Initialization and Registration

To instantiate and initialize the ICall service, the application must call the following functions in main() before starting the SYS/BIOS kernel scheduler:

```
/* Initialize ICall module */
ICall_init();
/* Start tasks of external images - Priority 5 */
ICall_createRemoteTasks();
```

Calling ICall_init() initializes the ICall primitive service (for example, heap manager) and framework. Calling ICall_createRemoteTasks() creates but does not start the *Bluetooth* low energy protocol stack task. Before using ICall protocol services, the server and client must enroll and register with ICall. The server enrolls a service, which is defined at build time. Service function handler registration uses a globally defined unique identifier for each service. For example, *Bluetooth* low energy uses ICALL_SERVICE_CLASS_BLE for receiving *Bluetooth* low energy protocol stack messages through ICall.

The following is a call to enroll the *Bluetooth* low energy protocol stack service (server) with ICall in OSAL_ICallBle.c:

```
// ICall enrollment
/* Enroll the service that this stack represents */
ICall_enrollService(ICALL_SERVICE_CLASS_BLE, NULL, &entity, &sem);
```

The registration mechanism is used by the client to send and/or receive messages through the ICall dispatcher.

For a client (for example, application task) to use the *Bluetooth* low energy stack APIs, the client must first register its task with ICall. This registration usually occurs in the task initialization function of the application. The following is an example from SimpleBLEPeripheral_int() in simpleBLEPeripheral.c:

```
// Register the current thread as an ICall dispatcher application
// so that the application can send and receive messages.
ICall_registerApp(&selfEntity, &sem);
```

The application supplies the selfEntity and sem inputs. These imputs are initialized for the task of the client (for example, application) when the ICall_registerApp() returns are initialized. These objects are subsequently used be ICall to facilitate messaging between the application and server tasks. The sem argument represents the semaphore for signaling and the selfEntity represents the destination message queue of the task. Each task registering with ICall have unique sem and selfEntity identifiers.

NOTE: *Bluetooth* low energy protocol stack APIs defined in ICallBLEApi.c and other ICall primitive services are not available before ICall registration.

Copyright © 2015–2016, Texas Instruments Incorporated

4.2.5 ICall Thread Synchronization

The ICall module switches between application and stack threads through Preemption and Semaphore Synchronization services provided by the RTOS. The two ICall functions to retrieve and enqueue messages are not blocking functions. These functions check whether there is a received message in the queue and if there is no message, the functions return immediately with ICALL_ERRNO_NOMSG return value. To allow a client or a server thread to block until it receives a message, ICall provides the following function which blocks until the semaphore associated with the caller RTOS thread is posted:

```
//static inline ICall_Errno ICall_wait(uint_fast32_t milliseconds)
ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);
```

milliseconds is a time-out period in milliseconds. If not already returned after this time-out period, the function returns with ICALL_ERRNO_TIMEOUT. If ICALL_TIMEOUT_FOREVER is passed as milliseconds, the ICall_wait() blocks until the semaphore is posted. Allowing an application or a server thread to block yields the processor resource to other lower priority threads or conserves energy by shutting down power and/or clock domains when possible.

The semaphore associated with an RTOS thread is signaled by either of the following conditions:

- A new message is queued to the RTOS thread queue of the application.
- ICall_signal() is called to unblock the semaphore.

ICall_signal() is used so an application or a server can add its own event to unblock ICall_wait() and synchronize the thread. ICall_signal() accepts semaphore handle as its sole argument as follows:

```
//static inline ICall_Errno ICall_signal(ICall_Semaphore msgsem)
ICall_signal(sem);
```

The semaphore handle associated with the thread is obtained through either ICall_enrollService() call or ICall_registerApp() call.

NOTE: Do not call an ICall function from a stack callback. This action can cause ICall to abort (with ICall_abort()) and break the system.

Copyright © 2015–2016, Texas Instruments Incorporated

4.2.6 Example ICall Usage

Figure 4-2 shows an example command being sent from the application to the *Bluetooth* low energy protocol stack through the ICall framework with a corresponding return value passed back to the application. ICall_init() initializes the ICall module instance and ICall_createRemoteTasks() creates a task per external image with an entry function at a known address. After initializing ICall, the application task registers with ICall through ICall_registerApp. After the SYS/BIOS scheduler starts and the application task runs, the application sends a protocol command defined in ICallBLEAPI.c such as GAP_GetParamValue(). The protocol command is not executed in the thread of the application but is encapsulated in an ICall message and routed to the *Bluetooth* low energy protocol stack task through the ICall framework. This command is sent to the ICall dispatcher where it is dispatched and executed on the server side (that is, *Bluetooth* low energy stack). The application thread meanwhile blocks (that is, waits) for the corresponding command status message (that is, status and GAP parameter value). When the *Bluetooth* low energy protocol stack finishes executing the command, the command status message response is sent through ICall back to the application thread.

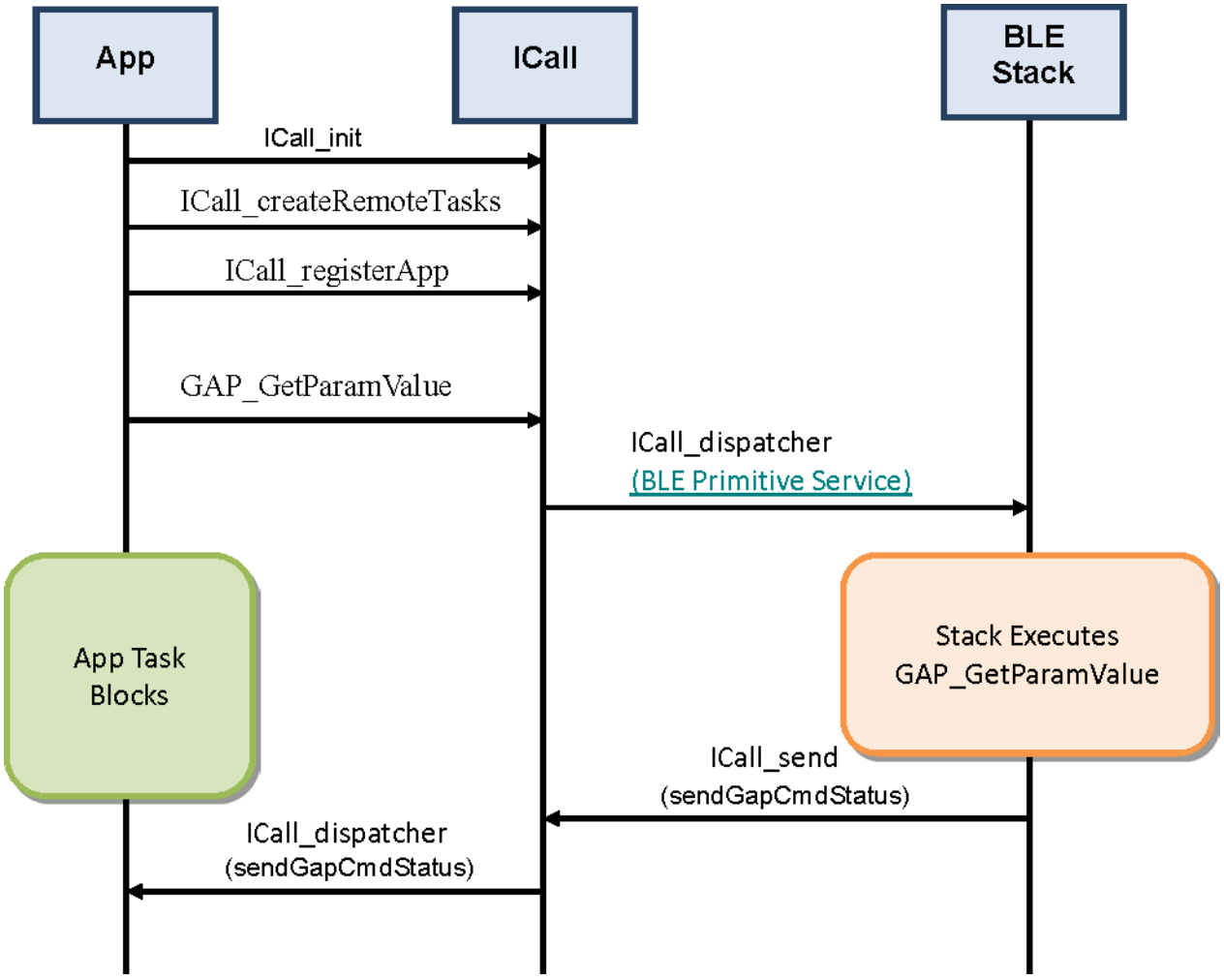


Figure 4-2. ICall Messaging Example

Copyright © 2015–2016, Texas Instruments Incorporated

4.3 General Application Architecture

This section describes in detail how an application task is constructed.

4.3.1 Application Initialization Function

Section 3.3 describes how a task is constructed. After the task is constructed and the SYS/BIOS kernel scheduler is started, the function that was passed during task construction is run when the task is ready (for example, SimpleBLEPeripheral_taskFxn). This function must first call an application initialization function. For example, in SimpleBLEPeripheral.c:

```
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
  // Initialize application
  SimpleBLEPeripheral_init();

  // Application main loop
  for (;;)
  {
...
```

This initialization function (SimpleBLEPeripheral_init()) configures several services for the task and sets several hardware and software configuration settings and parameters. The following list contains some common examples:

- Initializing the GATT client
- Registering for callbacks in various profiles
- Setting up the GAPRole
- Setting up the Bond Manager
- Setting up the GAP layer
- Configuring hardware modules such as LCD, SPI, and so forth

For more information on all of these examples, see their respective sections in this guide.

NOTE: In the application initialization function, ICall_registerApp() must be called before any stack API is called.

Copyright © 2015–2016, Texas Instruments Incorporated

4.3.2 Event Processing in the Task Function

After the initialization function shown in the previous code snippet, the task function enters an infinite loop so that it continuously processes as an independent task and does not run to completion. In this infinite loop, the taskremains blocked and waits until a semaphore signals a new reason for processing:

```
    ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

    if (errno == ICALL_ERRNO_SUCCESS)
    {
...
```

When an event or other stimulus occurs and is processed, the task waits for the semaphore and remains in a blocked state until there is another reason to process. Figure 4-3 shows this flow.

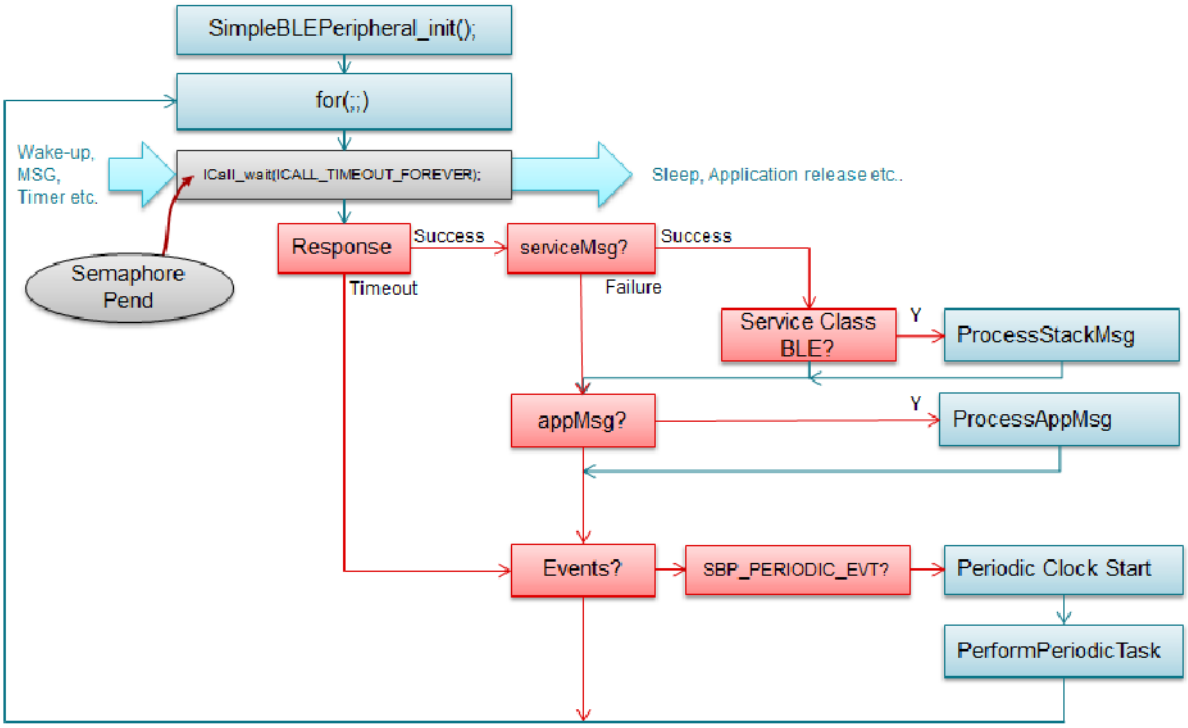


Figure 4-3. SBP Task Flow Chart

As shown in Figure 4-3, various reasons cause the semaphore to be posted to and the task to become active to process.

Copyright © 2015–2016, Texas Instruments Incorporated

4.3.2.1 Task Events

Task events are set when the *Bluetooth* low energy protocol stack sets an event in the application task through ICall. An example of a task event is when the HCI_EXT_ConnEventNoticeCmd() is called (see Section H.1) to indicate the end of a connection event. An example of a task event that signals the end of a connection event is shown in the task function of the SimpleBLEPeripheral:

```
if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)

{
  if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
  {
    ICall_Event *pEvt = (ICall_Event *)pMsg;

    // Check for BLE stack events first
    if (pEvt->signature == 0xffff)
    {
      if (pEvt->event_flag & SBP_CONN_EVT_END_EVT)
      {
        // Try to retransmit pending ATT response (if any)
        SimpleBLEPeripheral_sendATTRsp();
      }
    ...
  }

  if (pMsg)
  {
    ICall_freeMsg(pMsg);
  }
}
```

NOTE: In the code, the pEvt->signature is always equal to 0xFFFF if the event is coming from the *Bluetooth* low energy protocol stack.

When selecting an event value for an intertask event, the value must be unique for the given task and must be a power of 2 (so only 1 bit is set). Because the pEvt->event variable is initialized as uint16_t, this initialization allows for a maximum of 16 events. The only event values that cannot be used are those already used for *Bluetooth* low energy OSAL global events (stated in bcomdef.h):

```
/*********************************************************************
 * BLE OSAL GAP GLOBAL Events
 */
#define GAP_EVENT_SIGN_COUNTER_CHANGED  0x4000  //!< The device level sign counter changed
```

NOTE: These intertask events are a different set of events than the intratask events mentioned in Section 4.3.2.4.

Copyright © 2015–2016, Texas Instruments Incorporated

4.3.2.2 Intertask Messages

These messages are passed from another task (such as the *Bluetooth* low energy protocol stack) through ICall to the application task. Some possible examples are as follows:

- A confirmation sent from the protocol stack in acknowledgment of a successful over-the-air indication
- An event corresponding to an HCI command (see Section 5.6)
- A response to a GATT client operation (See Section 5.3.3.1)

The following is an example of this from the main task loop of the SimpleBLEPeripheral.

```
if (ICall_fetchServiceMsg(&src, &dest,
                          (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
{
  uint8 safeToDealloc = TRUE;

  if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
  {
    ICall_Event *pEvt = (ICall_Event *)pMsg;
    ...
    else
    {
      // Process inter-task message
      safeToDealloc = SimpleBLEPeripheral_processStackMsg((ICall_Hdr *)pMsg);
    }
  }

  if (pMsg && safeToDealloc)
  {
    ICall_freeMsg(pMsg);
  }
}
```

4.3.2.3 Messages Posted to the RTOS Queue of the Application Task

These messages have been enqueued using the SimpleBLEPeripheral_enqueueMsg() function. Because these messages are posted to a queue, they are processed in the order in which they occurred. A common example of this is an event received in a callback function (see Section 5.3.4.2.4).

```
// If RTOS queue is not empty, process app message.

if (!Queue_empty(appMsgQueue))
{
  sbpEvt_t *pMsg = (sbpEvt_t *)Util_dequeueMsg(appMsgQueue);
  if (pMsg)
  {
    // Process message.
    SimpleBLEPeripheral_processAppMsg(pMsg);

    // Free the space from the message.
    ICall_free(pMsg);
  }
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

4.3.2.4 Events Signaled Through the Internal Event Variable

These asyncronous events are signaled to the application task for processing by setting the appropriate bit in the events variable of the application task, where each bit corresponds to a defined event.

```
// Internal Events for RTOS application
#define SBP_STATE_CHANGE_EVT                  0x0001
#define SBP_CHAR_CHANGE_EVT                   0x0002
#define SBP_PERIODIC_EVT                      0x0004
```

The function that sets this bit in the events variable must also post to the semaphore to wake up the application for processing. An example of this process is the clock handler that handles clock timeouts (see Section 3.4.2). The following is an example of processing the periodic event from the main task function of the SimpleBLEPeripheral:

```
    if (events & SBP_PERIODIC_EVT)
    {
      events &= ~SBP_PERIODIC_EVT;

      Util_startClock(&periodicClock);

      // Perform periodic application task
      SimpleBLEPeripheral_performPeriodicTask();
    }
```

NOTE: When adding an event, the event must be unique for the given task and must be a power of 2 (so that only one bit is set). Because the events variable is initialized as uint16_t, this initialization allows for a maximum of 16 internal events.

Copyright © 2015–2016, Texas Instruments Incorporated

4.3.3 Callbacks

The application code also includes various callbacks to protocol stack layers, profiles, and RTOS modules. To ensure thread safety, processing must be minimized in the actual callback and the bulk of the processing should occur in the application context. Two functions are defined per callback (consider the GAPRole state change callback):

- **The actual callback**: This function is called in the context of the calling task or module (for example, the GAPRole task). To minimize processing in the calling context, this function should enqueue an event to the queue of the application for processing.

```
static void SimpleBLEPeripheral_stateChangeCB(gaprole_States_t newState)
{
  SimpleBLEPeripheral_enqueueMsg(SBP_STATE_CHANGE_EVT, newState);
}
```

- **The function to process in the application context**: When the application wakes up due to the enqueue from the callback, this function is called when the event is popped from the application queue and processed.

```
static void SimpleBLEPeripheral_processStateChangeEvt(gaprole_States_t newState)
{
   ...
}
```

See Section 5.2.1 for a flow diagram of this process.

Copyright © 2015–2016, Texas Instruments Incorporated

The Bluetooth low energy Protocol Stack

This section describes the functionality of the *Bluetooth* low energy protocol stack and provides a list of APIs to interface with the protocol stack. The stack project and its associated files serve to implement the *Bluetooth* low energy protocol stack task. This is the highest priority task in the system and it implements the *Bluetooth* low energy protocol stack as shown in Figure 1-2.

Most of the *Bluetooth* low energy protocol stack is object code in a single library file (TI does not provide the protocol stack source code as a matter of policy). A developer must understand the functionality of the various protocol stack layers and how they interact with the application and profiles. This section explains these layers.

5.1 Generic Access Profile (GAP)

The GAP layer of the *Bluetooth* low energy protocol stack is responsible for connection functionality. This layer handles the access modes and procedures of the device including device discovery, link establishment, link termination, initiation of security features, and device configuration. See Figure 5-1 for more details.

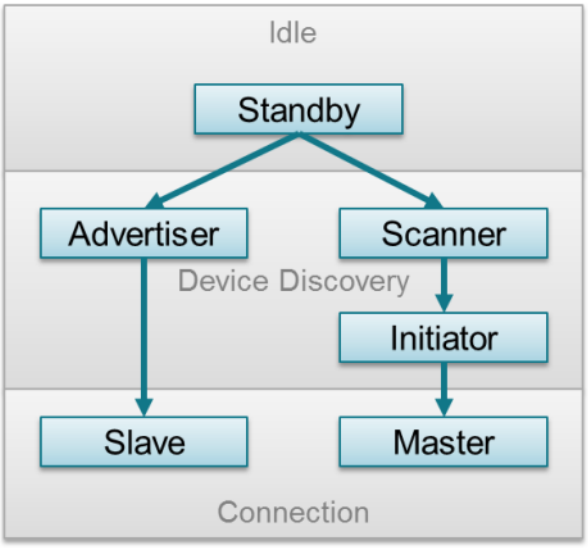


Figure 5-1. GAP State Diagram

Based on the role for which the device is configured, Figure 5-1 shows the states of the device. The following describes these states.

- **Standby**: The device is in the initial idle state upon reset.
- **Advertiser**: The device is advertising with specific data letting any initiating devices know that it is a connectable device (this advertisement contains the device address and can contain some additional data such as the device name).
- **Scanner**: When receiving the advertisement, the scanning device sends a scan request to the advertiser. The advertiser responds with a scan response. This process is called device discovery. The scanning device is aware of the advertising device and can initiate a connection with it.
- **Initiator**: When initiating, the initator must specify a peer device address to which to connect. If an advertisement is received matching that address of the peer device, the initiating device then sends

Copyright © 2015–2016, Texas Instruments Incorporated

out a request to establish a connection (link) with the advertising device with the connection parameters described in Section 5.1.1.

- **Slave/Master**: When a connection is formed, the device functions as a slave if the advertiser and a master if the initiator.

5.1.1 Connection Parameters

This section describes the connection parameters which are sent by the initiating device with the connection request and can be modified by either device when the connection is established. These parameters are as follows:

- **Connection Interval** – In *Bluetooth* low energy connections, a frequency-hopping scheme is used. The two devices each send and receive data from one another only on a specific channel at a specific time. These devices meet a specific amount of time later at a new channel (the link layer of the *Bluetooth* low energy protocol stack handles the channel switching). This meeting is where the two devices send and receive data is known as a *connection event*. If there is no application data to be sent or received, the two devices exchange link layer data to maintain the connection. The connection interval is the amount of time between two connection events in units of 1.25 ms. The connection interval can range from a minimum value of 6 (7.5 ms) to a maximum of 3200 (4.0 s). See Figure 5-2 for more details.

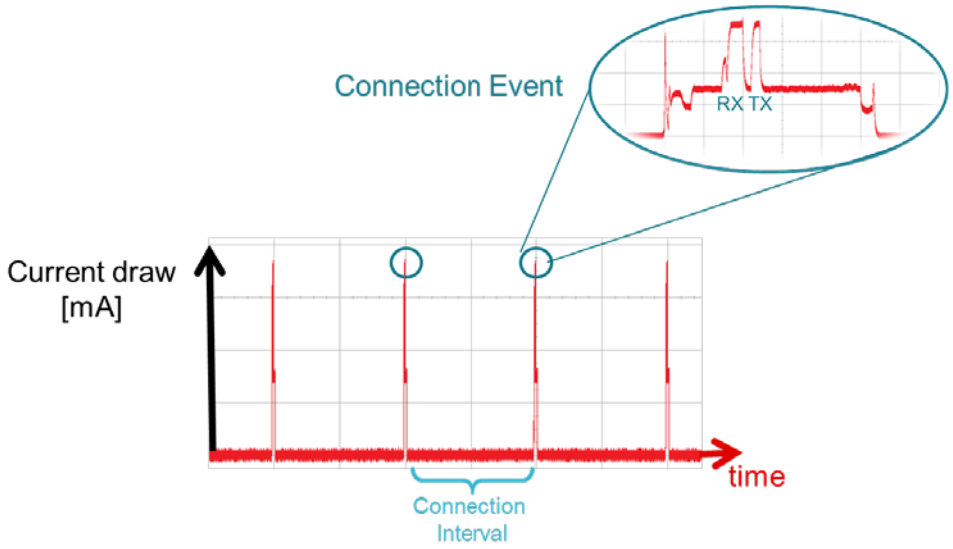


Figure 5-2. Connection Event and Interval

Different applications may require different connection intervals. As described in Section 5.1.3, these requirements affect the power consumption of the device. For more detailed information on power consumption, see *Measuring Bluetooth Smart Power Consumption Application Report* (SWRA478).

Copyright © 2015–2016, Texas Instruments Incorporated

- **Slave Latency** – This parameter gives the slave (peripheral) device the option of skipping a number of connection events. This ability gives the peripheral device some flexibility. If the peripheral does not have any data to send, it can skip connection events, stay asleep, and save power. The peripheral device selects whether to wake or not on a per connection event basis. The peripheral can skip connection events but must not skip more than allowed by the slave latency parameter or the connection fails. See Figure 5-3 for more details.

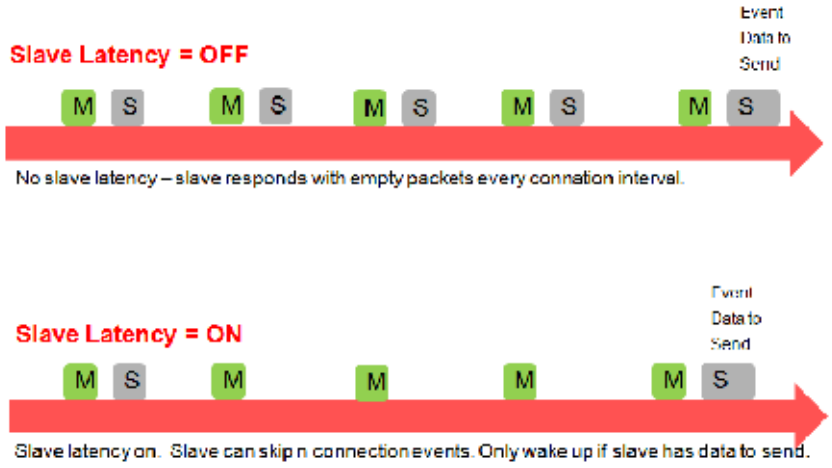


Figure 5-3. Slave Latency

- **Supervision Time-out** – This time-out is the maximum amount of time between two successful connection events. If this time passes without a successful connection event, the device terminates the connection and returns to an unconnected state. This parameter value is represented in units of 10 ms. The supervision time-out value can range from a minimum of 10 (100 ms) to 3200 (32.0 s). The time-out must be larger than the effective connection interval (see Section 5.1.2 for more details).

5.1.2 Effective Connection Interval

The effective connection interval is equal to the amount of time between two connection events, assuming that the slave skips the maximum number of possible events if slave latency is allowed (the effective connection interval is equal to the actual connection interval if slave latency is set to 0).

The slave latency value represents the maximum number of events that can be skipped. This number can range from a minimum value of 0 (meaning that no connection events can be skipped) to a maximum of 499. The maximum value must not make the effective connection interval (see the following formula) greater than 16 s. The interval can be calculated using the following formula:

Effective Connection Interval = (Connection Interval) × (1 + [Slave Latency])

Consider the following example:

- Connection Interval: 80 (100 ms)
- Slave Latency: 4
- Effective Connection Interval: (100 ms) × (1 + 4) = 500 ms

When no data is being sent from the slave to the master, the slave transmits during a connection event once every 500 ms.

Copyright © 2015–2016, Texas Instruments Incorporated

5.1.3 Connection Parameter Considerations

In many applications, the slave skips the maximum number of connection events. Consider the effective connection interval when selecting or requesting connection parameters. Selecting the correct group of connection parameters plays an important role in power optimization of the *Bluetooth* low energy device. The following list gives a general summary of the trade-offs in connection parameter settings.

Reducing the connection interval does as follows:

- Increases the power consumption for both devices
- Increases the throughput in both directions
- Reduces the time for sending data in either direction

Increasing the connection interval does as follows:

- Reduces the power consumption for both devices
- Reduces the throughput in both directions
- Increases the time for sending data in either direction

Reducing the slave latency (or setting it to zero) does as follows:

- Increases the power consumption for the peripheral device
- Reduces the time for the peripheral device to receive the data sent from a central device

Increasing the slave latency does as follows:

- Reduces power consumption for the peripheral during periods when the peripheral has no data to send to the central device
- Increases the time for the peripheral device to receive the data sent from the central device

5.1.4 Connection Parameter Limitations with Multiple Connections

Due to controller processing limitations in the stack, additional rules exist (beyond what the specification defines) that must be followed when multiple simultaneous connections exist. These rules are as follows.

- All intervals of the connection must be multiples of each other. If not, the connection does not form or the parameters do not update.
- If additional connections are desired, the controller must have enough processing time to scan for a new connection. Multiple connections at short connection intervals limit and possibly remove opportunities for scanning. Updating connection parameters to a longer connection interval might be required to add a new connection. The user can then update the parameters to make them shorter when the connection is formed.

Copyright © 2015–2016, Texas Instruments Incorporated

5.1.5 Connection Parameter Update

In some cases, the central device requests a connection with a peripheral device containing connection parameters that are unfavorable to the peripheral device. In other cases, a peripheral device might have the desire to change parameters in the middle of a connection, based on the peripheral application. The peripheral device can request the central device to change the connection settings by sending a Connection Parameter Update Request. For *Bluetooth* 4.1-capable devices, this request is handled directly by the Link Layer. For *Bluetooth* 4.0 devices, the L2CAP layer of the protocol stack handles the request. The *Bluetooth* low energy stack automatically selects the update method.

This request contains four parameters: minimum connection interval, maximum connection interval, slave latency, and time-out. These values represent the parameters that the peripheral device needs for the connection (the connection interval is given as a range). When the central device receives this request, it can accept or reject the new parameters.

Sending a Connection Parameter Update Request is optional and is not required for the central device to accept or apply the requested parameters. Some applications try to establish a connection at a faster connection interval to allow for a faster service discovery and initial setup. These applications later request a longer (slower) connection interval to allow for optimal power usage.

Depending on the GAPRole, connection parameter updates can be sent asynchronously with the GAPRole_SendUpdateParam() or GAPCentralRole_UpdateLink() command. See the API in Section B.1 and Section C.1, respectively. The peripheral GAPRole can be configured to automatically send a parameter update a certain amount of time after establishing a connection. For example, the SimpleBLEPeripheral application uses the following preprocessor-defined symbols:

```
#define DEFAULT_ENABLE_UPDATE_REQUEST         TRUE
#define DEFAULT_DESIRED_MIN_CONN_INTERVAL     80
#define DEFAULT_DESIRED_MAX_CONN_INTERVAL     800
#define DEFAULT_DESIRED_SLAVE_LATENCY         0
#define DEFAULT_DESIRED_CONN_TIMEOUT          1000
#define DEFAULT_CONN_PAUSE_PERIPHERAL         6
```

Six seconds after a connection is established, the GAP layer automatically sends a connection parameter update. See Section 5.2.1 for an explanation of how the parameters are configured, and Section B.2 for a more detailed description of these parameters. This action can be disabled by setting DEFAULT_ENABLE_UPDATE_REQUEST to FALSE.

5.1.6 Connection Termination

Either the master or the slave can terminate a connection for any reason. One side initiates termination and the other side must respond before both devices exit the connected state.

5.1.7 Connection Security

GAP also handles the initiation of security features during a *Bluetooth* low energy connection. Certain data may be readable or writeable only in an authenticated connection. When a connection is formed, two devices can go through a process called *pairing*. When pairing is performed, keys are established to encrypt and authenticate the link. The peripheral device requires that the central device provide a passkey to complete the pairing process. This could be a fixed value, such as 000000, or could be a randomly generated value that is provided to the user (such as on a display). After the central device sends the correct passkey, the two devices exchange security keys to encrypt and authenticate the link.

The same central and peripheral devices regularly connect and disconnect from each other. *Bluetooth* low energy has a security feature that lets two devices to give each other a long-term set of security keys when pairing. This feature is called *bonding*. Bonding lets the two devices to quickly reestablish encryption and authentication after reconnecting without going through the full pairing process every time that they connect if they store the long-term key information. In the SimpleBLEPeripheral application, the management of the GAP role is handled by the GAP role profile and the management of bonding information is handled by the GAP security profile.

Copyright © 2015–2016, Texas Instruments Incorporated

5.1.8 GAP Abstraction

The application and profiles can directly call GAP API functions to perform *Bluetooth* low energy-related functions such as advertising or connecting. Most of the GAP functionality is handled by the GAPRole Task. Figure 5-4 shows this abstraction hierarchy.

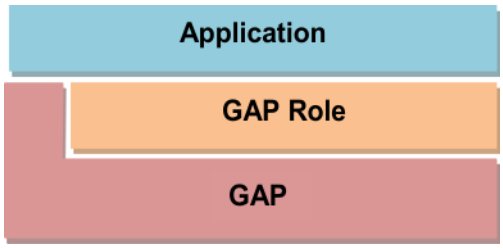


Figure 5-4. GAP Abstraction

Access the GAP layer through direct calls or through the GAPRole task as described in **Section 5.2**. Use the GAPRole task rather than direct calls when possible. Section 5.1.9 describes the functions and parameters that are not handled or configured through the GAPRole task and must be modified directly through the GAP layer.

5.1.9 Configuring the GAP Layer

The GAP layer functionality is mostly defined in library code. The function headers can be found in gap.h in the protocol stack project. Most of these functions are used by the GAPRole and do not need to be called directly. For reference, the GAP API is defined in Appendix D. Several parameters exist which may be desirable to modify before starting the GAPRole. These parameters can be set or get through the GAP_SetParamValue() and GAP_GetParamValue() functions and include advertising and scanning intervals, windows, and so forth (see the API for more information). The following is the configuration of the GAP layer in SimpleBLEPeripheral_init():

```
// Set advertising interval
{
  uint16_t advInt = DEFAULT_ADVERTISING_INTERVAL;

  GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MIN, advInt);
  GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MAX, advInt);
  GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MIN, advInt);
  GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MAX, advInt);
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

5.2 GAPRole Task

The GAPRole task is a separate task which offloads the application by handling most of the GAP layer functionality. This task is enabled and configured by the application during initialization. Based on this configuration, many *Bluetooth* low energy protocol stack events are handled directly by the GAPRole task and never passed to the application. Callbacks exist that the application can register with the GAPRole task so that the application task can be notified of certain events and proceed accordingly.

Based on the configuration of the device, the GAP layer always operates in one of four roles:

- Broadcaster – The advertiser is nonconnectable.
- Observer – The device scans for advertisements but cannot initiate connections.
- Peripheral – The advertiser is connectable and operates as a slave in a single link-layer connection.
- Central – The device scans for advertisements and initiates connections and operates as a master in a single or multiple link-layer connections. The *Bluetooth* low energy central protocol stack supports up to three simultaneous connections.

The *Bluetooth* low energy specification allows for certain combinations of multiple-roles, which are supported by the *Bluetooth* low energy protocol stack. For configuration of the *Bluetooth* low energy stack features, see Section 10.4.

Copyright © 2015–2016, Texas Instruments Incorporated

5.2.1 Peripheral Role

The peripheral GAPRole task is defined in peripheral.c and peripheral.h. Section A.1 describes the full API including commands, configurable parameters, events, and callbacks. The steps to use this module are as follows:

1. Initialize the GAPRole parameters (see Appendix B).

NOTE: This initialization should occur in the application initialization function (that is SimpleBLEPeripheral_init()).

```
{
    // For all hardware platforms, device starts advertising upon initialization
    uint8_t initialAdvertEnable = TRUE;

    uint16_t advertOffTime = 0;

    uint8_t enableUpdateRequest = DEFAULT_ENABLE_UPDATE_REQUEST;
    uint16_t desiredMinInterval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
    uint16_t desiredMaxInterval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
    uint16_t desiredSlaveLatency = DEFAULT_DESIRED_SLAVE_LATENCY;
    uint16_t desiredConnTimeout = DEFAULT_DESIRED_CONN_TIMEOUT;

    // Set the GAP Role Parameters
    GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
                         &initialAdvertEnable);
    GAPRole_SetParameter(GAPROLE_ADVERT_OFF_TIME, sizeof(uint16_t),
                         &advertOffTime);
    GAPRole_SetParameter(GAPROLE_SCAN_RSP_DATA, sizeof(scanRspData),
                         scanRspData);
    GAPRole_SetParameter(GAPROLE_ADVERT_DATA, sizeof(advertData), advertData);

    GAPRole_SetParameter(GAPROLE_PARAM_UPDATE_ENABLE, sizeof(uint8_t),
                         &enableUpdateRequest);
    GAPRole_SetParameter(GAPROLE_MIN_CONN_INTERVAL, sizeof(uint16_t),
                         &desiredMinInterval);
    GAPRole_SetParameter(GAPROLE_MAX_CONN_INTERVAL, sizeof(uint16_t),
                         &desiredMaxInterval);
    GAPRole_SetParameter(GAPROLE_SLAVE_LATENCY, sizeof(uint16_t),
                         &desiredSlaveLatency);
    GAPRole_SetParameter(GAPROLE_TIMEOUT_MULTIPLIER, sizeof(uint16_t),
                         &desiredConnTimeout);
}
```

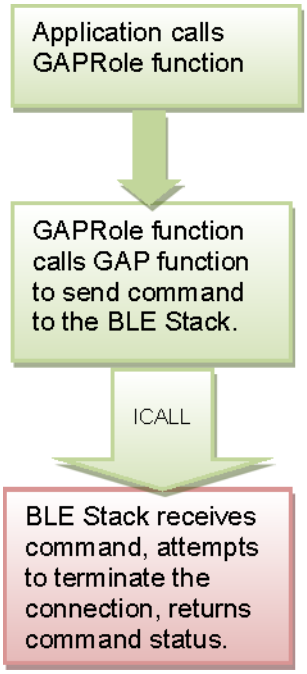
2. Initialize the GAPRole task.

NOTE: This initialization should occur in the application initialization function. This initialization involves passing function pointers to application callback functions. **Section B.3** defines these callbacks.

```
// Start the Device
VOID GAPRole_StartDevice(&SimpleBLEPeripheral_gapRoleCBs);
```


Copyright © 2015–2016, Texas Instruments Incorporated

3. Send GAPRole commands from the application (the following is an example of the application using GAPRole_TerminateConnection(). Green corresponds to the app context and red corresponds to the *Bluetooth* low energy protocol stack context).



simpleBLEPeripheral.c:

```
GAPRole_TerminateConnection();
```

peripheral.c

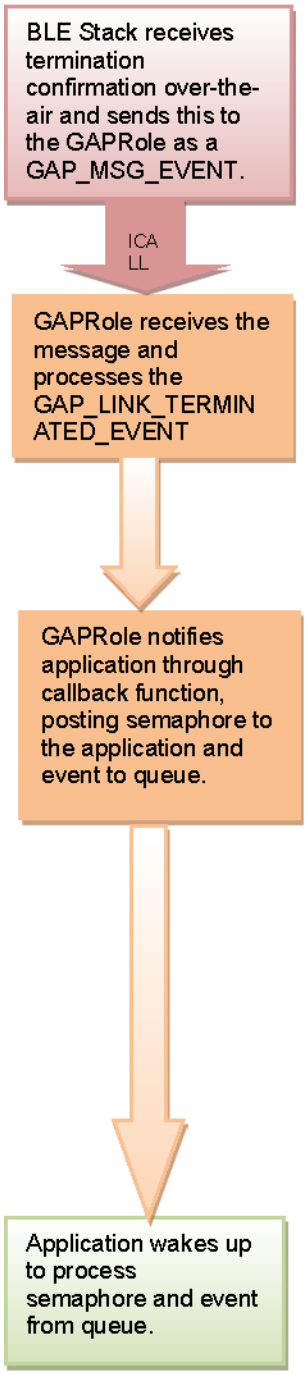
```
bStatus_t GAPRole_TerminateConnection(void)
{
  if (gapRole_state == GAPROLE_CONNECTED)
  {
    Return (GAP_TerminateLinkReq(selfEntity,
          gapRole_ConnectionHandle,
          HCI_DISCONNECT_REMOTE_USER_TERM));
...
```

Library Code

NOTE: The return value from the *Bluetooth* low energy protocol stack only indicates whether the attempt to terminate the connection initiated successfully. The actual termination of connection event is returned asynchronously. The API in Section B.3 lists the return parameters for each command and associated callback function events.

4. The GAPRole task processes most of the GAP-related events passed to it from the *Bluetooth* low energy protocol stack. The GAPRole task also forwards some events to the application. (The following is an example tracing the GAP_LINK_TERMINATED_EVENT from the *Bluetooth* low energy protocol stack to the application. Green corresponds to the app context. Orange corresponds to the GAPRole context. Red corresponds to the protocol stack context.)

Copyright © 2015–2016, Texas Instruments Incorporated



Library Code

```
case GAP_LINK_TERMINATED_EVENT:
    {
    ...
       notify = TRUE;
```

Peripheral.c:

```
// Notify the application
if (pGapRoles_AppCGs && pGapRoles_AppCGs >pfnStateChange)
{
  pGapRoles_AppCGs->pfnStateChange(gapRole_state);
}
...
```

simpleBLEPeripheral.c:

```
static void SimpleBLEPeripheral_stateChangeCB(gaprole_States_t
newState)
{
  SimpleBLEPeripheral_enqueueMsg(SBP_STATE_CHANGE_EVT, newState);
}
```

simpleBLEPeripheral.c:

```
static void SimpleBLEPeripheral_processAppMsg(sbpEvt_t *pMsg)
{
  switch (pMsg->event)
  {
    case SBP_STATE_CHANGE_EVT:

SimpleBLEPeripheral_processStateChangeEvt((gaprole_States_t)pMsg-
>status);
...
```

```
static void
SimpleBLEPeripheral_processStateChangeEvt(gaprole_States_t newState)
{
  switch ( newState )
  {
    case GAP_LINK_TERMINATED_EVENT:
...
```


Copyright © 2015–2016, Texas Instruments Incorporated

5.2.2 Central Role

The central GAPRole task is defined in central.c and central.h. **Appendix C** describes the full API including commands, configurable parameters, events, and callbacks. The steps to use this module are as follows.

1. Initialize the GAPRole parameters.

NOTE: **Appendix C** defines these parameters. This initialization should occur in the application initialization function (that is, SimpleBLECentral_init()).

```
// Setup GAP
GAP_SetParamValue(TGAP_GEN_DISC_SCAN, DEFAULT_SCAN_DURATION);
GAP_SetParamValue(TGAP_LIM_DISC_SCAN, DEFAULT_SCAN_DURATION);
GGS_SetParameter(GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN,
                 (void *)attDeviceName);
```

2. Start the GAPRole task.

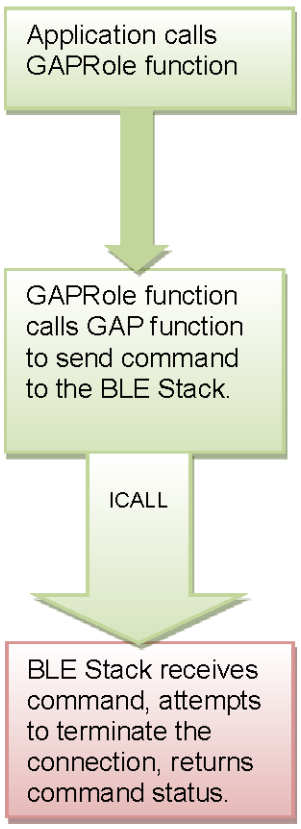
NOTE: Starting the GAPRole task should occur in the application initialization function. This action involves passing function pointers to application callback functions. Section C.3 defines these callbacks.

```
// Start the Device
VOID GAPCentralRole_StartDevice(&SimpleBLECentral_roleCB);
```


Copyright © 2015–2016, Texas Instruments Incorporated

3. Send GAPRole commands from the application.

NOTE: The following is an example of the application using GAPCentralRole_StartDiscovery(). Green corresponds to the app context and red corresponds to the *Bluetooth* low energy protocol stack context.



simpleBLEcentral.c

```
GAPCentralRole_StartDiscovery(DEFAULT_DISCOVERY_MODE,
   DEFAULT_DISCOVERY_ACTIVE_SCAN,
   DEFAULT_DISCOVERY_WHITE_LIST);
```

central.c

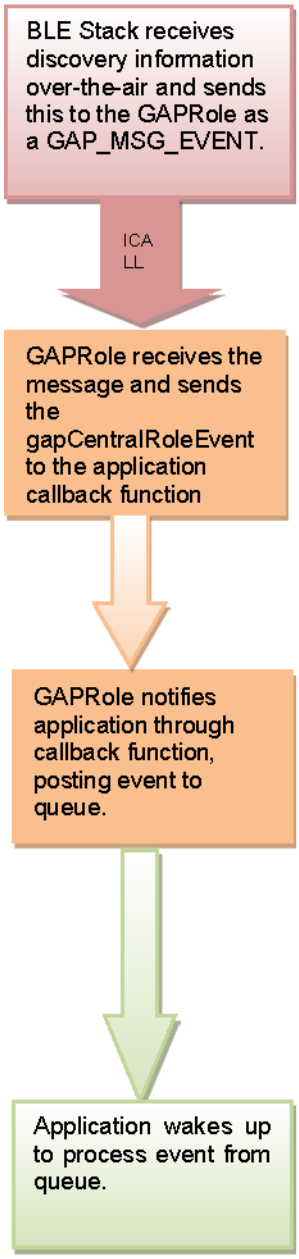
```
bStatus_t GAPCentralRole_StartDiscovery(uint8_t mode, uint8_t
activeScan, uint8_t whiteList)
{
  gapDevDiscReq_t params;

  params.taskID =
ICall_getLocalMsgEntityId(ICALL_SERVICE_CLASS_BLE_MSG,
                                                selfEntity);
  params.mode = mode;
  params.activeScan = activeScan;
  params.whiteList = whiteList;

  return GAP_DeviceDiscoveryRequest(&params);
}
```

Library Code

NOTE: The return value from the *Bluetooth* low energy protocol stack indicates only whether or not the attempt to perform device discovery was initiated. The actual termination of connection event is returned asynchronously. The API lists the return parameters for each command and associated callback events.

4. The GAPRole task processes most of the GAP-related events passed to it from the *Bluetooth* low energy protocol stack. The task also forwards some events to the application. (The following is an example tracing the GAP_DEVICE_DISCOVERY_EVENT from the *Bluetooth* low energy protocol stack to the application. Green corresponds to the app context. Orange corresponds to the GAPRole context. Red corresponds to the protocol stack context.)

Copyright © 2015–2016, Texas Instruments Incorporated



Library Code

central.c:

```
static void gapRole_processGAPMsg(gapEventHdr_t *pMsg)
{
...
  // Pass event to app
  if (pGapCentralRoleCB && pGapCentralRoleCB->eventCB)
  {
    return (pGapCentralRoleCB-
      >eventCB((gapCentralRoleEvent_t *)pMsg));
  }
...
```

SimpleBLECentral.c:

```
// Forward the role event to the application
if (SimpleBLECentral_enqueueMsg(SBC_STATE_CHANGE_EVT,
                                SUCCESS,(uint8_t*)pEvent))
...
```

SimpleBLECentral.c:

```
static void SimpleBLECentral_processAppMsg(sbcEvt_t *pMsg)
{
  switch (pMsg->event)
  {
    case SBC_STATE_CHANGE_EVT:
      SimpleBLECentral_processStackMsg((ICall_Hdr *)pMsg-
       >pData);
…
```

SimpleBLECentral.c:

```
static void SimpleBLECentral_processRoleEvent(gapCentralRoleEvent_t
*pEvent)
{
  switch (pEvent->gap.opcode)
  {
     case GAP_DEVICE_DISCOVERY_EVENT:
      {
...
```


Copyright © 2015–2016, Texas Instruments Incorporated

5.3 Generic Attribute Profile (GATT)

Just as the GAP layer handles most connection-related functionality, the GATT layer of the *Bluetooth* low energy protocol stack is used by the application for data communication between two connected devices. Data is passed and stored in the form of characteristics which are stored in memory on the *Bluetooth* low energy device. From a GATT standpoint, when two devices are connected they are each in one of two roles.

- The **GATT server** is the device containing the characteristic database that is being read or written by a GATT client.
- The **GATT client** is the device that is reading or writing data from or to the GATT server.

Figure 5-5 shows this relationship in a sample *Bluetooth* low energy connection where the peripheral device (that is, a SensorTag) is the GATT server and the central device (that is, a smart phone) is the GATT client.

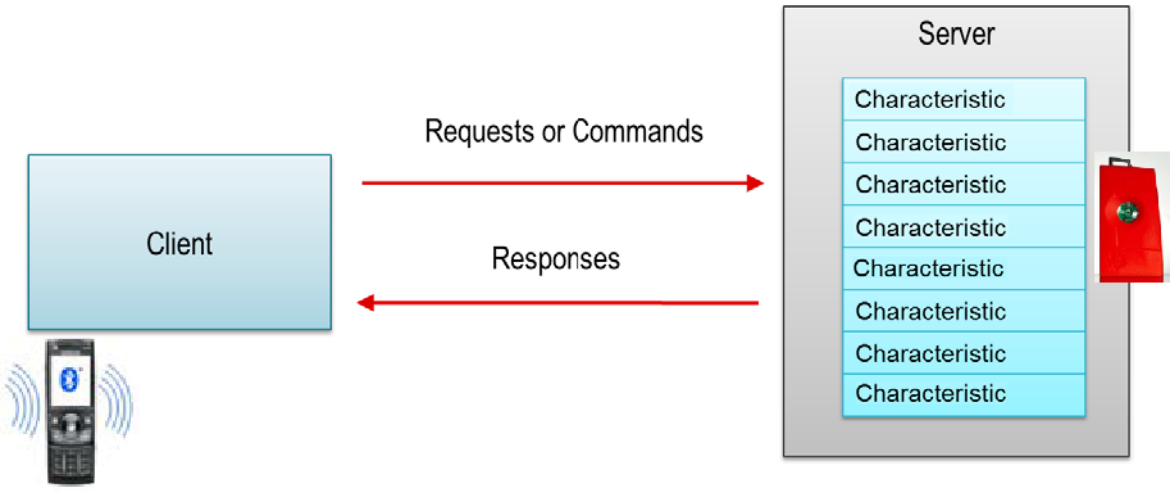


Figure 5-5. GATT Client and Server

The GATT roles of client and server are independent from the GAP roles of peripheral and central. A peripheral can be either a GATT client or a GATT server, and a central can be either a GATT client or a GATT server. A peripheral can act as both a GATT client and a GATT server.

5.3.1 GATT Characteristics and Attributes

While characteristics and attributes are sometimes used interchangeably when referring to *Bluetooth* low energy, consider characteristics as groups of information called attributes. Attributes are the information actually transferred between devices. Characteristics organize and use attributes as data values, properties, and configuration information. A typical characteristic is composed of the following attributes.

- Characteristic Value: data value of the characteristic
- Characteristic Declaration: descriptor storing the properties, location, and type of the characteristic value
- Client Characteristic Configuration: a configuration that allows the GATT server to configure the characteristic to be notified (sent to GATT server) or indicated (sent to GATT server and expecting an ACK)
- Characteristic User Description: an ASCII string describing the characteristic

These attributes are stored in the GATT server in an attribute table. In addition to the value, the following properties are associated with each attribute.

- Handle: the index of the attribute in the table (Every attribute has a unique handle.)
- Type: indicates what the attribute data represents (referred to as a UUID [universal unique identifier]. Some of these are *Bluetooth* SIG-defined and some are custom.)
- Permissions: enforces if and how a GATT client device can access the value of an attribute

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.2 GATT Services and Profile

A GATT service is a collection of characteristics. For example, the heart rate service contains a heart rate measurement characteristic and a body location characteristic, among others. Multiple services can be grouped together to form a profile. Many profiles only implement one service so the two terms are sometimes used interchangeably. There are four GATT profiles for the SimpleBLEPeripheral application.

- Mandatory GAP Service: This service contains device and access information such as the device name, vendor identification, and product identification. This service is a part of the *Bluetooth* low energy protocol stack and required for every *Bluetooth* low energy device as per the *Bluetooth* low energy specification. The source code for this service is not provided as it is built into the stack library.
- Mandatory GATT Service: This service contains information about the GATT server, is a part of the *Bluetooth* low energy protocol stack, and is required for every GATT server device as per the *Bluetooth* low energy specification. The source code for this service is not provided as it is built into the stack library.
- Device Info Service: This service exposes information about the device such as the hardware, software version, firmware version, regulatory information, compliance information, and manufacturer name. The Device Info Service is part of the *Bluetooth* low energy protocol stack and configured by the application.
- SimpleGATTProfile Service: This service is a sample profile for testing and for demonstration. The full source code is provided in the simpleGATTProfile.c and simpleGATTProfile.h files.

Figure 5-6 shows the portion of the attribute table in the SimpleBLEPeripheral project corresponding to the simpleGATTProfile service. TI intends this section as an introduction to the attribute table. For information on how this profile is implemented in the code, see **Section 5.3.4.2**.

Handle	Juid	Uuid Description	Value	Properties
0x001F	0x2800	GATT Primary Service Declaration	FU:FF	
0x0020	0x2803	GATT Characteristic Declaration	0A:21:00:F1:FF	
0x0021	0xFFF1	Simple Profile Char 1	01	Rd Wr 0x0A
0x0022	0x2901	Characteristic User Description	Characteristic 1	
0x0023	0x2803	GATT Characteristic Declaration	02:24:00:F2:FF	
0x0024	0xFFF2	Simple Profile Char 2	02	Rd 0x02
0x0025	0x2901	Characteristic User Description	Characteristic 2	
0x0026	0x2803	GATT Characteristic Declaration	08:27:00:F3:FF	
0x0027	0xFFF3	Simple Profile Char 3		Wr 0x08
0x0028	0x2901	Characteristic User Description	Characteristic 3	
0x0029	0x2803	GATT Characteristic Declaration	10:2A:00:F4:FF	
0x002A	0xFFF4	Simple Profile Char 4		Nfy 0x10
0x002B	0x2902	Client Characteristic Configuration	00:00	
0x002C	0x2901	Characteristic User Description	Characteristic 4	
0x002D	0x2803	GATT Characteristic Declaration	02:2E:00:F5:FF	
0x002E	0xFFF5	Simple Profile Char 5		Rd 0x02
0x002F	0x2901	Characteristic User Description	Characteristic 5	

Figure 5-6. simpleGATTProfile Characteristic Table from BTool

The simpleGATTProfile contains the following characteristics:

- SIMPLEPROFILE_CHAR1 – a 1-byte value that can be read or written from a GATT client device
- SIMPLEPROFILE_CHAR2 – a 1-byte value that can be read from a GATT client device but cannot be written
- SIMPLEPROFILE_CHAR3 – a 1-byte value that can be written from a GATT client device but cannot be read
- SIMPLEPROFILE_CHAR4 – a 1-byte value that cannot be directly read or written from a GATT client device (This value is notifiable: This value can be configured for notifications to be sent to a GATT client device.)
- SIMPLEPROFILE_CHAR5 – a 5-byte value that can be read (but not written) from a GATT client

Copyright © 2015–2016, Texas Instruments Incorporated

device

The following is a line-by-line description of the simple profile attribute table, referenced by the following handle.

- 0x001F is the simpleGATTprofile service declaration. This declaration has a UUID of 0x2800 (*Bluetooth*-defined GATT_PRIMARY_SERVICE_UUID). The value of this declaration is the UUID of the simpleGATTprofile (custom-defined).
- 0x0020 is the SimpleProfileChar1 characteristic declaration. This declaration can be thought of as a pointer to the SimpleProfileChar1 value. The declaration has a UUID of 0x2803 (*Bluetooth*-defined GATT_CHARACTER_UUID). The value of the declaration characteristic, as well as all other characteristic declarations, is a 5-byte value explained here (from MSB to LSB):
  - Byte 0: the properties of the SimpleProfileChar1 as defined in the *Bluetooth* specification (The following are some of the relevant properties.)
    - 0x02: permits reads of the characteristic value
    - 0x04: permits writes of the characteristic value (without a response)
    - 0x08: permits writes of the characteristic value (with a response)
    - 0x10: permits of notifications of the characteristic value (without acknowledgment)
    - 0x20: permits notifications of the characteristic value (with acknowledgment)

  The value of 0x0A means the characteristic is readable (0x02) and writeable (0x08).
  - Bytes 1–2: the byte-reversed handle where the SimpleProfileChar1 value is (handle 0x0021)
  - Bytes 3–4: the UUID of the SimpleProfileChar1 value (custom-defined 0xFFF1)
- 0x0021 is the SimpleProfileChar1 value. This value has a UUID of 0xFFF1 (custom-defined). This value is the actual payload data of the characteristic. As indicated by its characteristic declaration (handle 0x0020), this value is readable and writeable.
- 0x0022 is the SimpleProfileChar1 user description. This description has a UUID of 0x2901 (*Bluetooth*-defined). The value of this description is a user-readable string describing the characteristic.
- 0x0023 – 0x002F are attributes that follow the same structure as the simpleProfileChar1 described previously with regard to the remaining four characteristics. The only different attribute, handle 0x002B, is described as follows.
- 0x002B is the SimpleProfileChar4 client characteristic configuration. This configuration has a UUID of 0x2902 (*Bluetooth*-defined). By writing to this attribute, a GATT server can configure the SimpleProfileChar4 for notifications (writing 0x0001) or indications (writing 0x0002). Writing 0x0000 to this attribute disable notifications and indications.

5.3.3 GATT Client Abstraction

Like the GAP layer, the GATT layer is also abstracted. This abstraction depends on whether the device is acting as a GATT client or a GATT server. As defined by the *Bluetooth* Specification, the GATT layer is an abstraction of the ATT layer.

GATT clients do not have attribute tables or profiles as they are gathering, not serving, information. Most of the interfacing with the GATT layer occurs directly from the application. In this case, use the direct GATT API described in **Appendix D**. Figure 5-7 shows the abstraction.

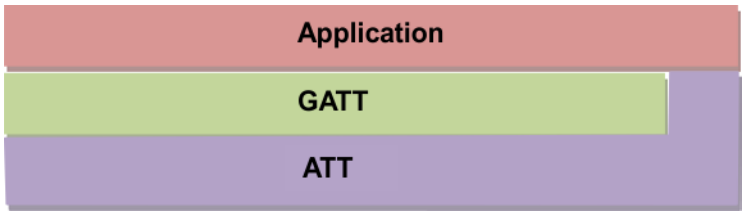


Figure 5-7. GATT Client Abstraction

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.3.1 Using the GATT Layer Directly

This section describes how to use the GATT layer in the application. The functionality of the GATT layer is implemented in the library but header functions can be found in the gatt.h file. Appendix D has the complete API for the GATT layer. Specification of the Bluetooth System, Covered Core Package, Version: 4.1 provides more information on the functionality of these commands. These functions is used primarily for GATT client applications. A few server-specific functions are described in the API. Most of the GATT functions returns ATT events to the application so consider the ATT API in **Appendix D**. The general procedure to use the GATT layer when functioning as a GATT client (that is, in the SimpleBLECentral project) is as follows:

1. Initialize the GATT client.

```
VOID GATT_InitClient();
```

2. Register to receive incoming ATT indications and notifications.

```
GATT_RegisterForInd(selfEntity);
```

3. Perform a GATT client procedure.

NOTE: The example uses GATT_WriteCharValue(), which is triggered by a left key press in the SimpleBLECentral application. Green corresponds to the app context and red corresponds to the protocol stack context.

Copyright © 2015–2016, Texas Instruments Incorporated

4. Receive and handle the response to the GATT client procedure in the application. In this example, the application receives an ATT_WRITE_RSP event. See Section D.6 for a list of GATT commands and their corresponding ATT events. Green corresponds to the app context and red corresponds to the protocol stack context.

Library Code

simpleBLECentral.c:

```
if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) ==
   ICALL_ERRNO_SUCCESS)
   {
   if ((src == ICALL_SERVICE_CLASS_BLE) && (dest ==
      selfEntity))
      {
         // Process inter-task message
         SimpleBLECentral_processStackMsg((ICall_Hdr *)pMsg);
...
```

SimpleBLECentral.c:

```
static void SimpleBLECentral_processStackMsg(ICall_Hdr *pMsg)
{
  switch (pMsg->event)
  {
    case GATT_MSG_EVENT:
      SimpleBLECentral_processGATTMsg((gattMsgEvent_t
        *)pMsg);
      break;
...
```

```
static void SimpleBLECentral_processGATTMsg(gattMsgEvent_t *pMsg)
{
   ...
  else if ((pMsg->method == ATT_WRITE_RSP)  ||
           ((pMsg->method == ATT_ERROR_RSP) &&
            (pMsg->msg.errorRsp.reqOpcode == ATT_WRITE_REQ)))
  {
    ...
    else
    {
      // After a succesful write, display the value that was
      // written and increment value
      LCD_WRITE_STRING_VALUE("Write sent:", charVal++, 10,
         LCD_PAGE2);
...
```

NOTE: Even though the event sent to the application is an ATT event, it is sent as a GATT protocol stack message (GATT_MSG_EVENT).

5. A GATT client may also receive asynchronous data from the GATT server as indications or notifications other than receiving responses to its own commands. Registering to receive these ATT notifications and indications is required as in Step 2. These notifications and indications are also be sent as ATT events in GATT messages to the application and must be handled as described in Section 5.3.2.

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.4 GATT Server Abstraction

As a GATT server, most of the GATT functionality is handled by the individual GATT profiles. These profiles use the GattServApp (a configurable module that stores and manages the attribute table). Figure 5-8 shows this abstraction hierarchy.

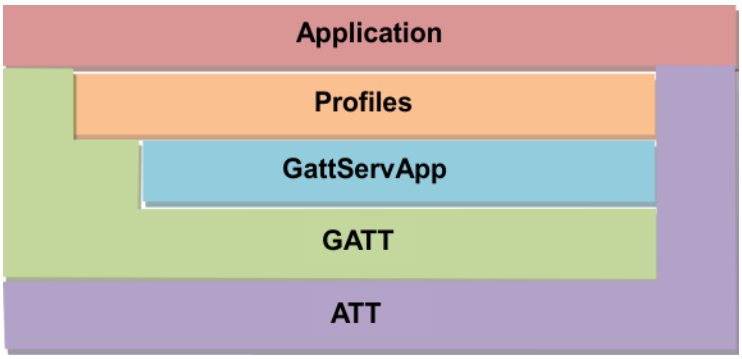


Figure 5-8. GATT Server Abstraction

The design process involves creating GATT profiles that configure the GATTServApp module and use its API to interface with the GATT layer. In this case of a GATT server, direct calls to GATT layer functions are unnecessary. The application then interfaces with the profiles.

5.3.4.1 GATTServApp Module

The GATTServApp stores and manages the application-wide attribute table. Various profiles use this module to add their characteristics to the attribute table. The *Bluetooth* low energy stack uses this module to respond to discovery requests from a GATT client. For example, a GATT client may send a Discover all Primary Characteristics message. The *Bluetooth* low energy stack on the GATT server receives this message and uses the GATTServApp to find and send over-the-air all of the primary characteristics stored in the attribute table. This type of functionality is beyond the scope of this document and is implemented in the library code. The GATTServApp functions accessible from the profiles are defined in gattservapp_util.c and described in the API in Appendix E. These functions include finding specific attributes and reading or modifying client characteristic configurations. See Figure 5-9 for more information.

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.4.1.1 Building up the Attribute Table

Upon power-on or reset, the application builds the GATT table by using the GATTServApp to add services. Each service consists of a list of attributes with UUIDs, values, permissions, and read and write call-backs. As Figure 5-9 shows, all of this information is passed through the GATTServApp to GATT and stored in the stack.

Attribute table initialization must occur in the application initialization function, that is, simpleBLEPeripheral_init().

```
 // Initialize GATT attributes
GGS_AddService(GATT_ALL_SERVICES);           // GAP
GATTServApp_AddService(GATT_ALL_SERVICES);   // GATT attributes
DevInfo_AddService();                        // Device Information Service
SimpleProfile_AddService(GATT_ALL_SERVICES); // Simple GATT Profile
```

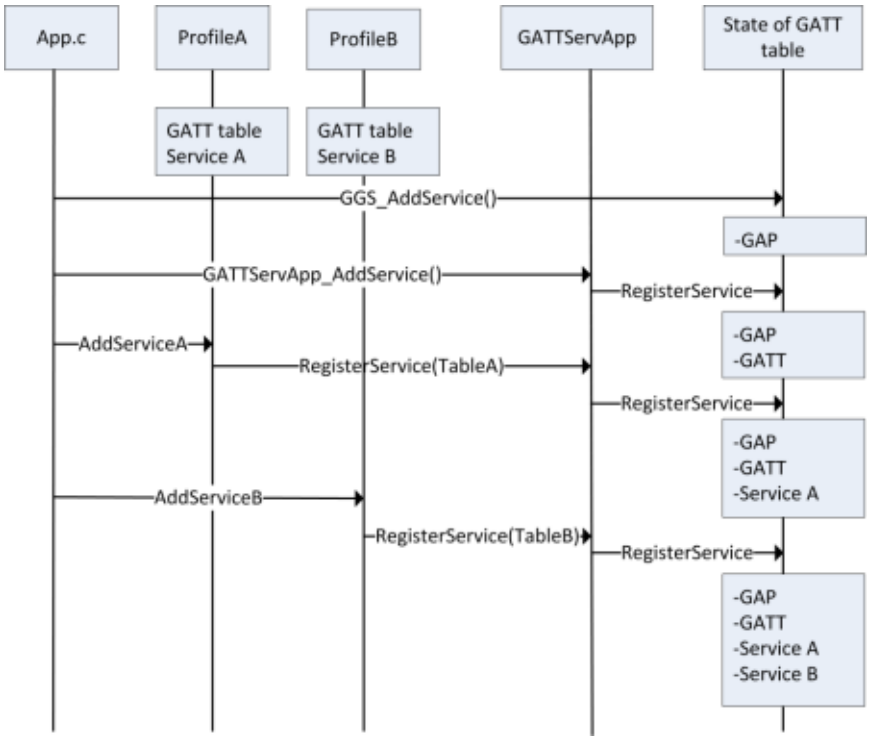


Figure 5-9. Attribute Table Initialization

5.3.4.2 Profile Architecture

This section describes the general architecture for all profiles and provides specific functional examples in relation to the simpleGATTProfile in the SimpleBLEPeripheral project. See Section 5.3.2 for an overview of the simpleGATTProfile. To interface with the application and *Bluetooth* low energy protocol stack at minimum, each profile must contain all the sub-elements of Section 5.3.4.2.1.

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.4.2.1 Attribute Table Definition

Each service or group of GATT attributes must define a fixed size attribute table that gets passed into GATT. This table in simpleGATTProfile.c is defined as the following.

```
static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED]
...
```

Each attribute in this table is of the following type.

```
typedef struct attAttribute_t
{
  gattAttrType_t type; //!< Attribute type (2 or 16 octet UUIDs)
  uint8 permissions;   //!< Attribute permissions
  uint16 handle;       //!< Attribute handle - assigned internally by attribute server
  uint8* const pValue; //!< Attribute value - encoding of the octet array is defined in
                       //!< the applicable profile. The maximum length of an attribute
                       //!< value shall be 512 octets.
} gattAttribute_t;
```

The specific elements of this attribute type are detailed as follows.

- type is the UUID associated with the attribute and is defined as follows:

```
typedef struct
{
  uint8 len;         //!< Length of UUID
  const uint8 *uuid; //!< Pointer to UUID
} gattAttrType_t;
```

  The length can be either ATT_BT_UUID_SIZE (2 bytes), or ATT_UUID_SIZE (16 bytes). The *uuid is a pointer to a number either reserved by *Bluetooth* SIG (defined in gatt_uuid.c) or a custom UUID defined in the profile.

- permissions enforces how and if a GATT client device can access the value of the attribute. Possible permissions are defined in gatt.h as follows:

```
GATT_PERMIT_READ     // Attribute is Readable
GATT_PERMIT_WRITE          // Attribute is Writable
GATT_PERMIT_AUTHEN_READ    // Read requires Authentication
GATT_PERMIT_AUTHEN_WRITE   // Write requires Authentication
GATT_PERMIT_AUTHOR_READ    // Read requires Authorization
GATT_PERMIT_AUTHOR_WRITE   // Write requires Authorization
GATT_PERMIT_ENCRYPT_READ   // Read requires Encryption
GATT_PERMIT_ENCRYPT_WRITE  // Write requires Encryption
```

  Section 5.3.5 further describes authentication, authorization, and encryption.

- handle is a placeholder in the table where GATTServApp assigns a handle. This placeholder is not customizable. Handles are assigned sequentially.
- pValue is a pointer to the attribute value. The size is unable to change after initialization. The maximum size is 512 octets.

The following sections provide examples of attribute definitions for common attribute types.

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.4.2.1.1 Service Declaration

Consider the following simpleGATTProfile service declaration attribute:

```
// Simple Profile Service
{
  { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
  GATT_PERMIT_READ,                         /* permissions */
  0,                                        /* handle */
  (uint8 *)&simpleProfileService            /* pValue */
},
```

The type is set to the *Bluetooth* SIG-defined primary service UUID (0x2800). A GATT client must read this attribute, so the permission is set to GATT_PERMIT_READ. The pValue is a pointer to the UUID of the service, custom-defined as 0xFFF0.

```
// Simple Profile Service attribute
static CONST gattAttrType_t simpleProfileService = { ATT_BT_UUID_SIZE,
   simpleProfileServUUID };
```

5.3.4.2.1.2 Characteristic Declaration

Consider the following simpleGATTProfile simpleProfileCharacteristic1 declaration.

```
    // Characteristic 1 Declaration
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &simpleProfileChar1Props
    },
```

The type is set to the *Bluetooth* SIG-defined characteristic UUID (0x2803).

A GATT client must read this so the permission is set to GATT_PERMIT_READ.

Section 5.3.1 describes the value of a characteristic declaration. For functional purposes, the only information required to be passed to the GATTServApp in pValue is a pointer to the properties of the characteristic value. The GATTServApp adds the UUID and the handle of the value. These properties are defined as follows.

```
// Simple Profile Characteristic 1 Properties
static uint8 simpleProfileChar1Props = GATT_PROP_READ | GATT_PROP_WRITE;
```

NOTE: An important distinction exists between these properties and the GATT permissions of the characteristic value. These properties are visible to the GATT client stating the properties of the characteristic value. The GATT permissions of the characteristic value affect its functionality in the protocol stack. These properties must match that of the GATT permissions of the characteristic value. Section 5.3.4.2.1.3 expands on this idea.

5.3.4.2.1.3 Characteristic Value

Consider the simpleGATTProfile simpleProfileCharacteristic1 value.

```
    // Characteristic Value 1
    {
      { ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
      GATT_PERMIT_READ | GATT_PERMIT_WRITE,
      0,
      &simpleProfileChar1
    },
```


Copyright © 2015–2016, Texas Instruments Incorporated

The type is set to the custom-defined simpleProfilechar1 UUID (0xFFF1). The properties of the characteristic value in the attribut table must match the properties from the characteristic value declaration. The pValue is a pointer to the location of the actual value, statically defined in the profile as follows.

```
// Characteristic 1 Value
static uint8 simpleProfileChar1 = 0;
```

5.3.4.2.1.4 Client Characteristic Configuration

Consider the simpleGATTProfile simpleProfileCharacteristic4 configuration.

```
      // Characteristic 4 configuration
      {
        { ATT_BT_UUID_SIZE, clientCharCfgUUID },
        GATT_PERMIT_READ | GATT_PERMIT_WRITE,
        0,
        (uint8 *)&simpleProfileChar4Config
      },
```

The type is set to the *Bluetooth* SIG-defined client characteristic configuration UUID (0x2902) GATT clients must read and write to this attribute so the GATT permissions are set to readable and writeable. The pValue is a pointer to the location of the client characteristic configuration array, defined in the profile as follows.

```
static gattCharCfg_t *simpleProfileChar4Config;
```

NOTE: Client characteristic configuration is represented as an array because this value must be cached for each connection. The catching of the client characteristic configuration is described in more detail in Section 5.3.4.2.2.

5.3.4.2.2 Add Service Function

As described in Section 5.3.4.1, when an application starts up it requires adding the GATT services it supports. Each profile needs a global AddService function that can be called from the application. Some of these services are defined in the protocol stack, such as GGS_AddService and GATTServApp_AddService. User-defined services must expose their own AddService function that the application can call for profile initialization. Using SimpleProfile_AddService() as an example, these functions should do as follows.

- Allocate space for the client characteristic configuration (CCC) arrays. As an example, a pointer to one of these arrays was initialized in the profile as described in Section 5.3.4.2.1.4.

  In the AddService function, the supported connections is declared and memory is allocated for each array. Only one CCC is defined in the simpleGATTProfile but there can be multiple CCCs.

```
simpleProfileChar4Config = (gattCharCfg_t *)ICall_malloc( sizeof(gattCharCfg_t) *
  linkDBNumConns );
if ( simpleProfileChar4Config == NULL )
{
  return ( bleMemAllocError );
}
```

- Initialize the CCC arrays. CCC values are persistent between power downs and between bonded device connections. For each CCC in the profile, the GATTServApp_InitCharCfg() function must be called. This function tries to initialize the CCCs with information from a previously bonded connection and set the initial values to default values if not found.

```
GATTServApp_InitCharCfg( INVALID_CONNHANDLE, simpleProfileChar4Config );
```

- Register the profile with the GATTServApp. This function passes the attribute table of the profile to the

Copyright © 2015–2016, Texas Instruments Incorporated

GATTServApp so that the attributes of the profile are added to the application-wide attribute table managed by the protocol stack and handles are assigned for each attribute. This also passes pointers to the callbacks of the profile to the stack to initiate communication between the GATTServApp and the profile.

```
status = GATTServApp_RegisterService( simpleProfileAttrTbl,
            GATT_NUM_ATTRS( simpleProfileAttrTbl ), GATT_MAX_ENCRYPT_KEY_SIZE,
            &simpleProfileCBs );
```

5.3.4.2.3 Register Application Callback Function

Profiles can relay messages to the application using callbacks. In the SimpleBLEPeripheral project, the simpleGATTProfile calls an application callback whenever the GATT client writes a characteristic value. For these application callbacks to be used, the profile must define a Register Application Callback function that the application uses to set up callbacks during its initialization. The register application callback function fo the simpleGATTProfile is the following.

```
bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks )
{
  if ( appCallbacks )
  {
    simpleProfile_AppCBs = appCallbacks;

    return ( SUCCESS );
  }
  else
  {
    return ( bleAlreadyInRequestedMode );
  }
}
```

Where the callback typedef is defined as the following.

```
typedef void (*simpleProfileChange_t)( uint8 paramID );

typedef struct
{
  simpleProfileChange_t        pfnSimpleProfileChange;  // Called when characteristic value
changes
} simpleProfileCBs_t;
```

The application must then define a callback of this type and pass it to the simpleGATTProfile with the SimpleProfile_RegisterAppCBs() function. This occurs in simpleBLEPeripheral.c as follows.

```
// Simple GATT Profile Callbacks
static simpleProfileCBs_t SimpleBLEPeripheral_simpleProfileCBs =
{
  SimpleBLEPeripheral_charValueChangeCB // Characteristic value change callback
};
…
// Register callback with SimpleGATTprofile
SimpleProfile_RegisterAppCBs(&SimpleBLEPeripheral_simpleProfileCBs);
```

See Section 5.3.4.2.4 for further information on how this callback is used.

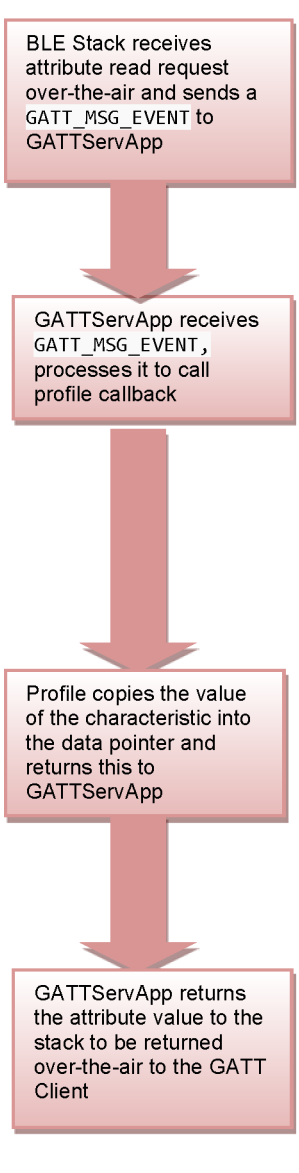
5.3.4.2.4 Read and Write Callback Functions

The profile must define Read and Write callback functions which the protocol stack calls when one of the attributes of the profile are written to or read from. The callbacks must be registered with GATTServApp as mentioned in Section 5.3.4.2.2. These callbacks perform the characteristic read or write and other processing (possibly calling an application callback) as defined by the specific profile.

Copyright © 2015–2016, Texas Instruments Incorporated

5.3.4.2.4.1 Read Request from Client

When a read request from a GATT Client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is readable, call the read call-back profile. The profile copies in the value, performs any profile-specific processing, and notifies the application if desired. This procedure is illustrated in the following flow diagram for a read of simpleprofileChar1 in the simpleGATTProfile. Red corresponds to processing in the protocol stack context.



Library Code

gattservapp.c:

```
if ( useCB == TRUE )
{
  // Use Service's read callback to process the request
  pfnGATTReadAttrCB_t pfnCB = gattServApp_FindReadAttrCB( service );
  if ( pfnCB != NULL )
  {
    // Read the attribute value
    status = (*pfnCB)( connHandle, pAttr, pValue, pLen, offset,
      maxLen, method );
...
```

simpleGATTProfile.c

```
static bStatus_t simpleProfile_ReadAttrCB(......)
{
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    // 16-bit UUID
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
      >type.uuid[1]);
    switch ( uuid )
    {
      case SIMPLEPROFILE_CHAR1_UUID:
        *pLen = 1;
        pValue[0] = *pAttr->pValue;
        break;
...
```

Library Code

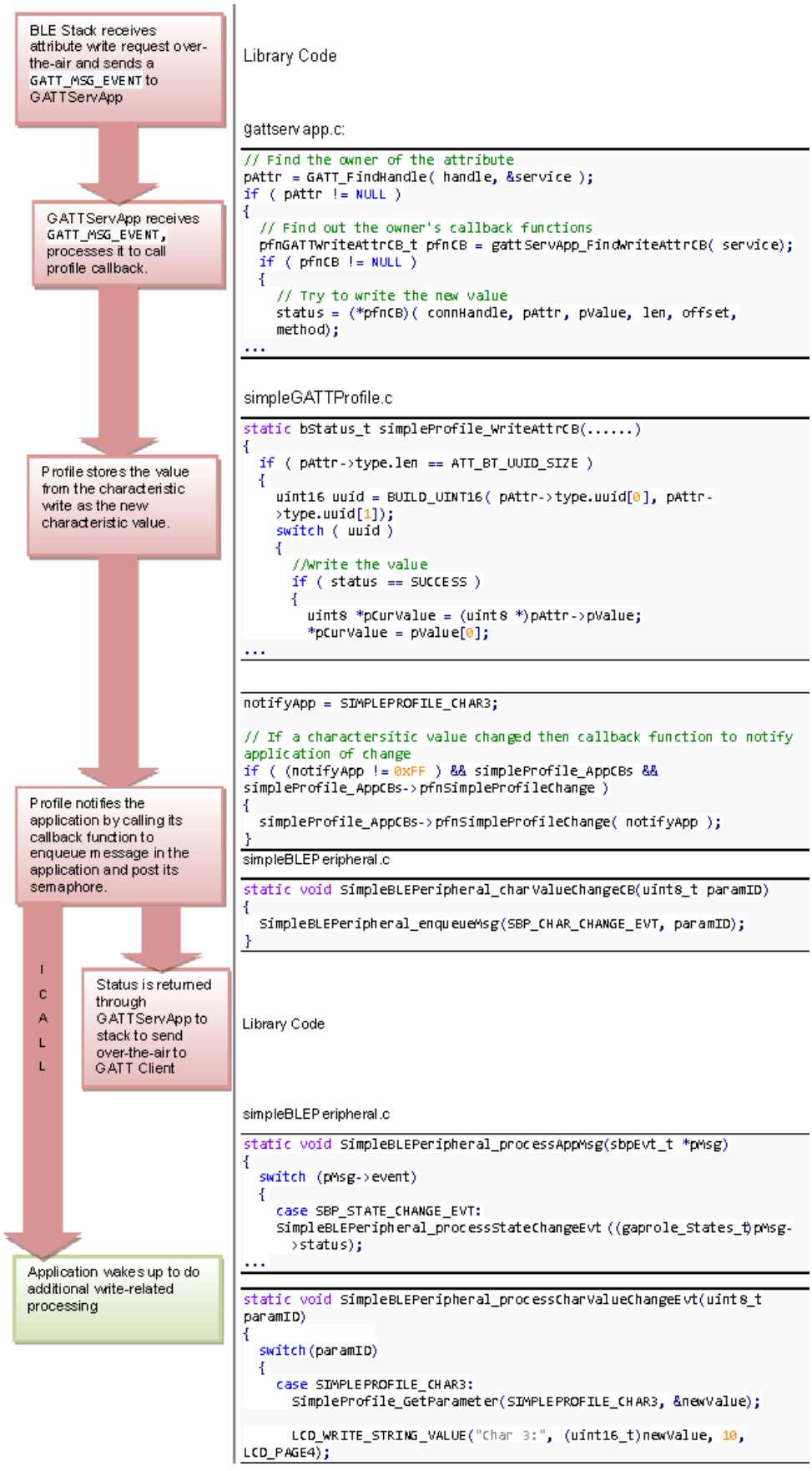
Copyright © 2015–2016, Texas Instruments Incorporated

NOTE: The processing is in the context of the protocol stack. If any intensive profile-related processing that must be done in the case of an attribute read, this should be split up and done in the context of the Application task. See the following write request for more information.

5.3.4.2.4.2 Write Request from Client

When a write request from a GATT client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is write, call the write callback of the profile. The profile stores the value to be written, performs any profile-specific processing, and notifies the application if desired. The following flow diagram shows this procedure for a write of simpleprofileChar3 in the simpleGATTProfile. Red corresponds to processing in the protocol stack context and green is processing in the application context.

Copyright © 2015–2016, Texas Instruments Incorporated



Copyright © 2015–2016, Texas Instruments Incorporated

NOTE: Minimizing the processing in protocol stack context is important. In this example, additional processing beyond storing the attribute write value in the profile (that is, writing to the LCD) occurs in the application context by enqueuing a message in the queue of the application.

5.3.4.2.5 Get and Set Functions

The profile containing the characteristics shall provide set and get abstraction functions for the application to read and write a characteristic of the profile. The set parameter function also includes logic to check for and implement notifications and indications if the relevant characteristic has notify or indicate properties. Figure 5-10 and the following code show this example for setting simpleProfileChacteristic4 in the simpleGATTProfile.

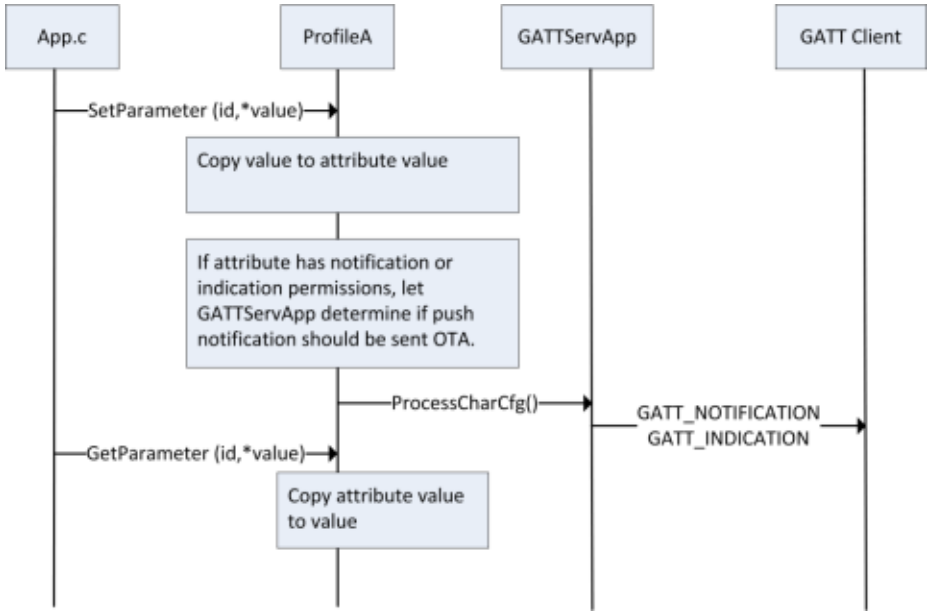


Figure 5-10. Get and Set Profile Parameter

For example, the application initializes simpleProfileCharacteristic4 to 0 in SimpleBLEPeripheral.c through the following.

```
uint8_t charValue4 = 4;
SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8_t), &charValue4);
```


Copyright © 2015–2016, Texas Instruments Incorporated

The code for this function is displayed in the following code snippet (from simpleGATTProfile.c). Besides setting the value of the static simpleProfileChar4, this function also calls GATTServApp_ProcessCharCfg because it has notify properties. This action forces GATTServApp to check if notifications have been enabled by the GATT Client. If so, the GATTServApp sends a notification of this attribute to the GATT Client.

```
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value )
{
  bStatus_t ret = SUCCESS;
  switch ( param )
  {
    case SIMPLEPROFILE_CHAR4:
      if ( len == sizeof ( uint8 ) )
      {
        simpleProfileChar4 = *((uint8*)value);

        // See if Notification has been enabled
        GATTServApp_ProcessCharCfg( simpleProfileChar4Config, &simpleProfileChar4, FALSE,
                                    simpleProfileAttrTbl, GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                                    INVALID_TASK_ID, simpleProfile_ReadAttrCB );
      }
```

5.3.5 Allocating Memory for GATT Procedures

To support fragmentation, GATT and ATT payload structures must be dynamically allocated for commands sent wirelessly. For example, a buffer must be allocated when sending a GATT_Notification. The stack does this allocation if the preferred method to send a GATT notification or indication is used: calling a SetParameter function of the profile (that is, SimpleProfile_SetParameter()) and calling GATTServApp_ProcessCharCfg() as described in Section 5.3.4.2.5.

If using GATT_Notification() or GATT_Indication() directly, memory management must be added as follows.

1. Try to allocate memory for the notification or indication payload using GATT_bm_alloc().
2. Send notification or indication using GATT_Notification() or GATT_Indication() if the allocation succeeds.

NOTE: If the return value of the notification or indication is SUCCESS (0x00), the stack freed the memory.

3. Use GATT_bm_free() to free the memory if the return value is something other than SUCCESS (for example, blePending).

Copyright © 2015–2016, Texas Instruments Incorporated

The following is an example of this in the gattServApp_SendNotiInd() function in the gattservapp_util.c file.

```
noti.pValue = (uint8 *)GATT_bm_alloc( connHandle, ATT_HANDLE_VALUE_NOTI,
                                      GATT_MAX_MTU, &len );
if ( noti.pValue != NULL )
{
  status = (*pfnReadAttrCB)( connHandle, pAttr, noti.pValue, &noti.len,
                             0, len, GATT_LOCAL_READ );
  if ( status == SUCCESS )
  {
    noti.handle = pAttr->handle;

    if ( cccValue & GATT_CLIENT_CFG_NOTIFY )
    {
      status = GATT_Notification( connHandle, &noti, authenticated );
    }
    else // GATT_CLIENT_CFG_INDICATE
    {
      status = GATT_Indication( connHandle, (attHandleValueInd_t *)&noti,
                                authenticated, taskId );
    }
  }

  if ( status != SUCCESS )
  {
    GATT_bm_free( (gattMsg_t *)&noti, ATT_HANDLE_VALUE_NOTI );
  }
}
else
{
  status = bleNoResources;
}
```

For other GATT procedures, take similar steps as noted in the API in Appendix D

5.3.6 Registering to Receive Additional GATT Events in the Application

Using GATT_RegisterForMsgs() (see Appendix D), receiving additional GATT messages to handle certain corner cases is possible. This possibility can be seen in SimpleBLEPeripheral_processGATTMsg(). The following three cases are currently handled.

- GATT server in the stack was unable to send an ATT response (due to lack of available HCI buffers): Attempt to transmit on the next connection interval.

```
// See if GATT server was unable to transmit an ATT response
if (pMsg->hdr.status == blePending)
{
  // No HCI buffer was available. Let's try to retransmit the response
  // on the next connection event.
  if (HCI_EXT_ConnEventNoticeCmd(pMsg->connHandle, selfEntity,
                                 SBP_CONN_EVT_END_EVT) == SUCCESS)
  {
    // First free any pending response
    SimpleBLEPeripheral_freeAttRsp(FAILURE);

    // Hold on to the response message for retransmission
    pAttRsp = pMsg;

    // Don't free the response message yet
    return (FALSE);
  }
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

- An ATT flow control violation: The application is notified that the connected device has violated the ATT flow control specification. No more ATT requests or indications can be sent wirelessly during the connection. The application may want to terminate the connection due to this violation. As an example in SimpleBLEPeripheral, the LCD is updated.

```
else if (pMsg->method == ATT_FLOW_CTRL_VIOLATED_EVENT)
{
  // ATT request-response or indication-confirmation flow control is
  // violated. All subsequent ATT requests or indications will be dropped.
  // The app is informed in case it wants to drop the connection.

  // Display the opcode of the message that caused the violation.
  LCD_WRITE_STRING_VALUE("FC Violated:", pMsg->msg.flowCtrlEvt.opcode,
                         10, LCD_PAGE5);
}
```

- An ATT MTU size is updated: The application is notified in case this affects its processing in any way. See Section 5.5.2for more information on the MTU. As an example in SimpleBLEPeripheral, the LCD is updated.

```
else if (pMsg->method == ATT_MTU_UPDATED_EVENT)
{
  // MTU size updated
  LCD_WRITE_STRING_VALUE("MTU Size:", pMsg->msg.mtuEvt.MTU, 10, LCD_PAGE5);
}
```

5.4 GAP Bond Manager

The GAP Bond Manager is a configurable module that offloads most of the security mechanisms from the application. Table 5-1 lists the terminology.

Table 5-1. GAP Bond Manager Terminology

Term	Description
Pairing	The process of exchanging keys
Encryption	Data is encrypted after pairing, or re-encryption (a subsequent connection where keys are looked up from nonvolatile memory).
Authentication	The pairing process completed with MITM (Man in the Middle) protection.
Bonding	Storing the keys in nonvolatile memory to use for the next encryption sequence.
Authorization	An additional application level key exchange in addition to authentication
OOB	Out of Band. Keys are not exchanged over the air, but rather over some other source such as serial port or NFC. This also provides MITM protection.
MITM	Man in the Middle protection. This prevents an attacker from listening to the keys transferred over the air to break the encryption.
Just Works	Pairing method where keys are transferred over the air without MITM

The general process that the GAPBondMgr uses is as follows.

1. The pairing process exchanges keys through the following methods:
   - Just Works
   - MITM
2. Encrypt the link with keys from Step 1.
3. The bonding process stores keys in secure flash (SNV).
4. Use the keys stored in SNV to encrypt the link when reconnecting.

Copyright © 2015–2016, Texas Instruments Incorporated

NOTE: Performing all of these steps is unnecessary. For example, two devices may choose to pair but not bond. The GAPBondMgr uses of the SNV flash area for storing bond information. For more information on SNV, see Section 3.10 .

5.4.1 Using GAPBondMgr

This section describes what the application must do to configure, start, and use the GAPBondMgr. The GAPRole handles some of the GAPBondMgr functionality. The GAPBondMgr is defined in gapbondmgr.c and gapbondmgr.h. Appendix D describes the full API including commands, configurable parameters, events, and callbacks. The SimpleBLECentral project is the example because it uses the callback functions from the GAPBondMgr.The general steps to use this module are as follows.

1. Initialize the GAPBondMgr parameters (see Section D.2).

NOTE: Do this initialization in the application initialization function (that is SimpleBLECentral_init()). Consider the following parameters. For the example, the pairMode is changed to initiate pairing.

```
// Setup the GAP Bond Manager
{
  uint32_t passkey = DEFAULT_PASSCODE;
  uint8_t pairMode = GAPBOND_PAIRING_MODE_INITIATE;
  uint8_t mitm = DEFAULT_MITM_MODE;
  uint8_t ioCap = DEFAULT_IO_CAPABILITIES;
  uint8_t bonding = DEFAULT_BONDING_MODE;

  GAPBondMgr_SetParameter(GAPBOND_DEFAULT_PASSCODE, sizeof(uint32_t),
                          &passkey);
  GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &pairMode);
  GAPBondMgr_SetParameter(GAPBOND_MITM_PROTECTION, sizeof(uint8_t), &mitm);
  GAPBondMgr_SetParameter(GAPBOND_IO_CAPABILITIES, sizeof(uint8_t), &ioCap);
  GAPBondMgr_SetParameter(GAPBOND_BONDING_ENABLED, sizeof(uint8_t), &bonding);
}
```

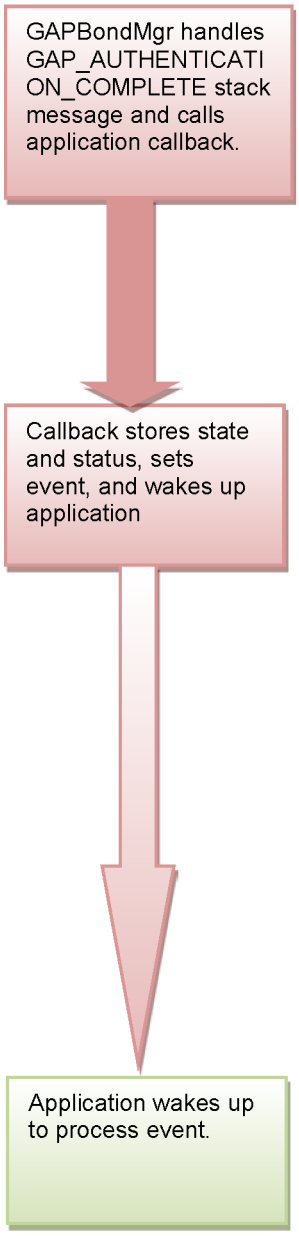
2. Register application callbacks with the GAPBondMgr.

NOTE: Register the GAPBondMgr callbacksin the application initialization function after the GAPRole profile is started. Section D.3 defines these callbacks.

```
// Start the Device
VOID GAPRole_StartDevice(&SimpleBLEPeripheral_gapRoleCBs);
```

3. The GAPBondMgr is configured and operate mostly autonomously from the perspective of the application. When a connection is established the GAPBondMgr initiates pairing and bonding depending on the configuration parameters from Step 1. A few parameters can be set asynchronously at this point such as GAPBOND_ERASE_ALLBONDS. See Appendix F for more information. Most communication between the GAPBondMgr and the application at this point occurs through the callbacks which were registered in Step 2. The following is a flow diagram example from SimpleBLECentral of the GAPBondMgr notifying the application that pairing has completed. The following sections expand on the details of these callbacks. Red corresponds to processing in the protocol stack context and green to the application context.

Copyright © 2015–2016, Texas Instruments Incorporated



gapbondmgr.c:

```
uint16 GAPBondMgr_ProcessEvent( uint8 task_id, uint16 events )
{
  if ( events & GAP_BOND_SYNC_CC_EVT )
  {
    if ( gapBondMgr_SyncCharCfg( pAuthEvt->connectionHandle ) )
    {
      if ( pGapBondCB && pGapBondCB->pairStateCB )
      {
        pGapBondCB->pairStateCB( pAuthEvt->connectionHandle,
         GAPBOND_PAIRING_STATE_COMPLETE, SUCCESS );
...
```

simpleBLECentral.c:

```
static void SimpleBLECentral_pairStateCB(uint16_t connHandle, uint8_t
state, uint8_t status)
{
  pairState = state;
  pairStatus = status;

  events |= SBC_PAIRING_STATE_EVT;

  Semaphore_post(sem);
}
```

SimpleBLECentral.c:

```
if (events & SBC_PAIRING_STATE_EVT)
{
  events &= ~SBC_PAIRING_STATE_EVT;

  SimpleBLECentral_processPairState(pairState, pairStatus);
}
```

```
static void SimpleBLECentral_processPairState(uint8_t state, uint8_t
status)
{
...
  else if (state == GAPBOND_PAIRING_STATE_COMPLETE)
  {
    if (status == SUCCESS)
    {
      LCD_WRITE_STRING("Pairing success", LCD_PAGE2);
    }
    else
    {
      LCD_WRITE_STRING_VALUE("Pairing fail:", status, 10, LCD_PAGE2);
    }
  }
...
```

5.4.2 GAPBondMgr Examples for Security Modes

This section provides message diagrams for the types of security that can be implemented. These modes assume acceptable I/O capabilities are available for the security mode. See the Specification of the Bluetooth System, Covered Core Package, Version: 4.1 on how I/O capabilities affect pairing.

5.4.2.1 Pairing Disabled

With pairing set to FALSE, the protocol stack automatically rejects any attempt at pairing.

```
uint8 pairMode = GAPBOND_PAIRING_MODE_NO_PAIRING;
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &pairMode);
```


Copyright © 2015–2016, Texas Instruments Incorporated

5.4.2.2 Just Works Pairing Without Bonding

Just Works pairing allows encryption without MITM authentication and is vulnerable to MITM attacks. Configure the GAPBondMgr for Just Works pairing as follows.

```
uint8 mitm = FALSE;
uint8 bonding = FALSE;
GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION, sizeof ( uint8 ), &mitm );
GAPBondMgr_SetParameter( GAPBOND_BONDING_ENABLED, sizeof ( uint8 ), &bonding );
```

Figure 5-11 gives an overview of this process for peripheral device.

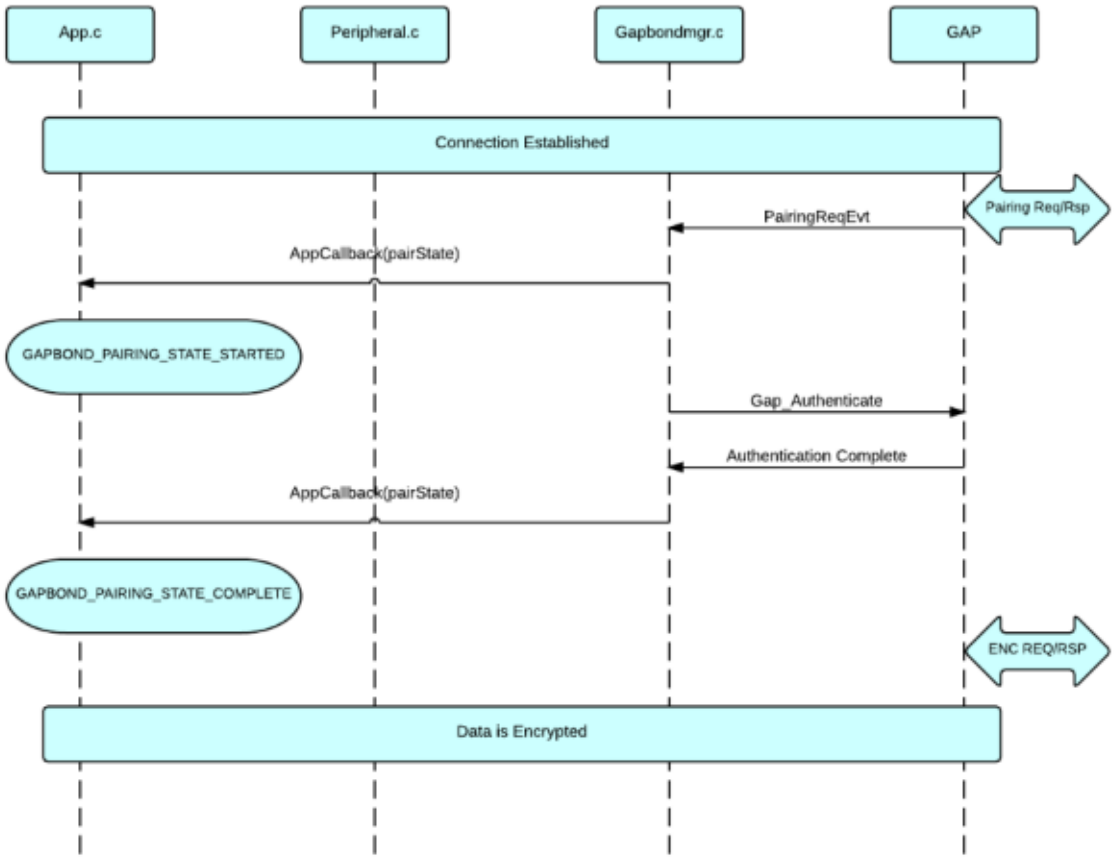


Figure 5-11. Just Works Pairing

Copyright © 2015–2016, Texas Instruments Incorporated

As shown in Figure 5-11, the GAPBondMgr pairing states are passed to the application callback at the appropriate time. GAPBOND_PAIRING_STATE_STARTED is passed when the pairing request or response is initially sent or received. GAPBOND_PAIRING_STATE_COMPLETE is sent when the pairing completes. Just Works pairing requires the pair state callback. SeeSection F.3 for more information.

5.4.2.3 Just Works Pairing With Bonding Enabled

To enable bonding with Just Works pairing, the settings should be used as follows.

```
uint8 mitm = FALSE;
uint8 bonding = TRUE;
GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION, sizeof ( uint8 ), &mitm );
GAPBondMgr_SetParameter( GAPBOND_BONDING_ENABLED, sizeof ( uint8 ), &bonding );
```

Figure 5-12 is an overview of this process for peripheral device.

Copyright © 2015–2016, Texas Instruments Incorporated

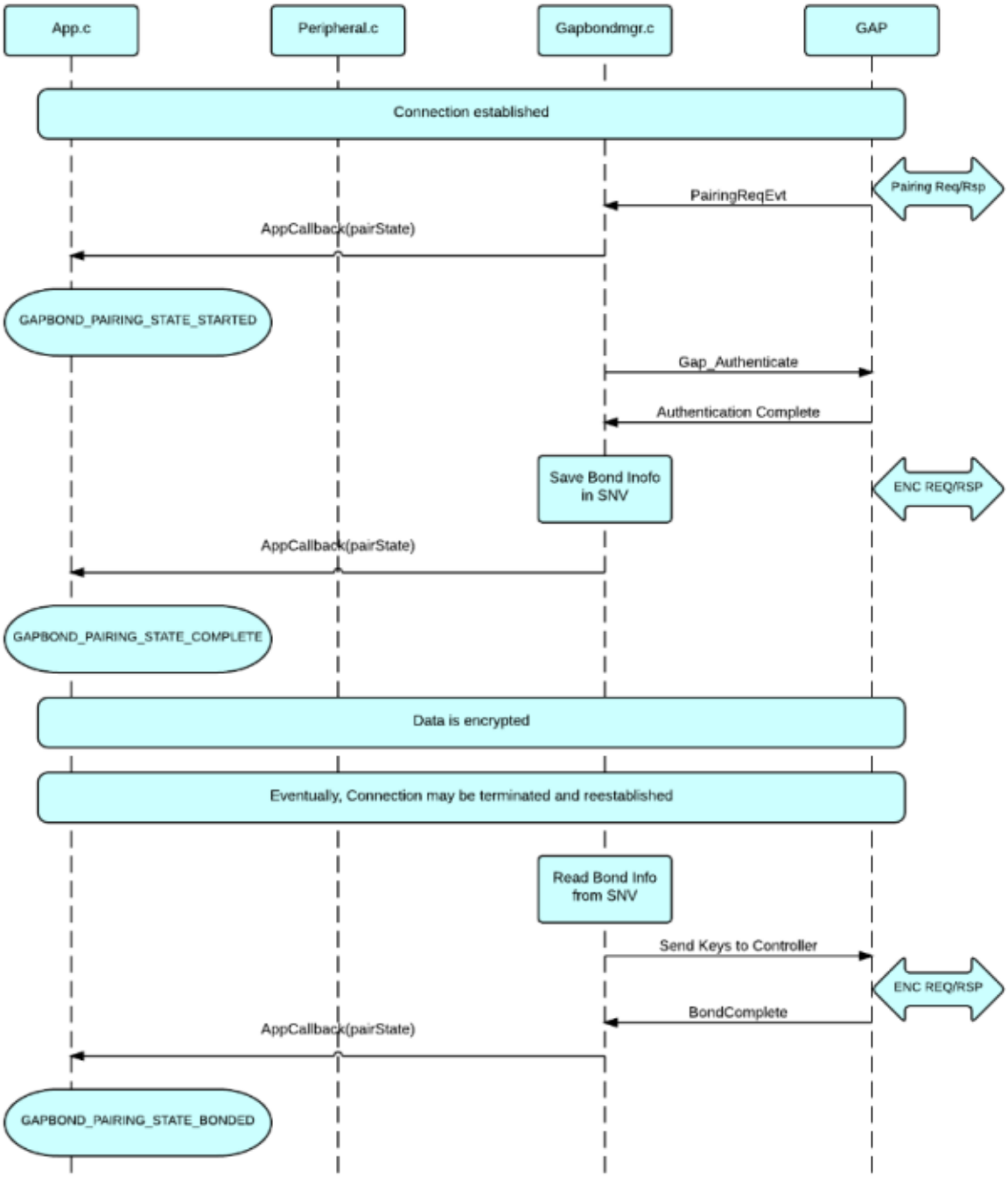


Figure 5-12. Bonding After Just Works Pairing

NOTE: GAPBOND_PAIRING_STATE_COMPLETE is only passed to the application pair state callback when initially connecting, pairing, and bonding. For future connections, the security keys are loaded from flash, skipping the pairing process. In this case, only PAIRING_STATE_BONDED is passed to the application pair state callback.

Copyright © 2015–2016, Texas Instruments Incorporated

5.4.2.4 Authenticated Pairing

Authenticated pairing requires Man In The Middle (MITM) protection, a method of transferring a passcode between the devices. To prevent the passcode from being intercepted, do not send the passcode wirelessly. Display the passcode on one device using an LCD screen or a serial number on the device and entered on the other device. Another option is OOB (out-of-band) passcode transfer using NFC, but that is beyond the scope of this document.

To pair with MITM authentication, use the following settings.

```
uint8 mitm = TRUE;
GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION, sizeof ( uint8 ), &mitm );
```


Copyright © 2015–2016, Texas Instruments Incorporated

Authenticated pairing requires an additional step in the security process as shown in Figure 5-13; entering a passcode. After pairing is started, the GAPBondMgr notifies the application that a passcode is required. Depending on the I/O capabilities of the device that determine its role in the passcode display and entering process, the device must display or enter the passcode. If displaying, the device must send this passcode back to the GAPBondMgr.

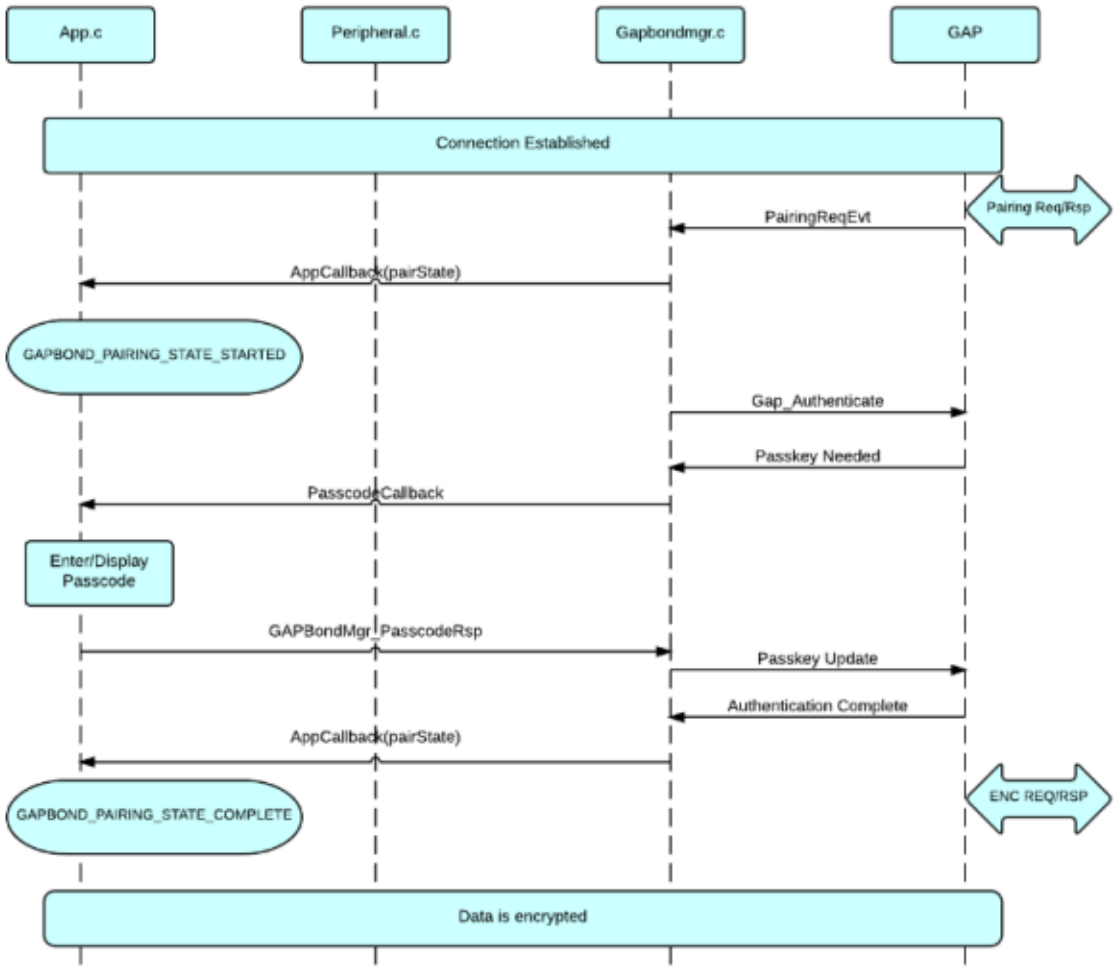


Figure 5-13. Pairing With MITM Authentication

For successful authenticated pairing, add a passcode function to the GAPBondMgr application callbacks as follows.

```
// Bond Manager Callbacks.
static const gapBondCBs_t BondCB =
{
  passcodeCB,
  pairStateCB
};
```

When the GAPBondMgr requires a passcode, it uses this callback to request a passcode from the application. Depending on the I/O capabilities of the device, it displays either a passcode or reads an entered passcode. In both cases, this passcode must be sent to the GAPBondMgr using the GAPBondMgr_PasscodeRsp() function. The following is the SimpleBLECentral example.

Copyright © 2015–2016, Texas Instruments Incorporated

```
static void SimpleBLECentral_processPasscode(uint16_t connectionHandle,
                                             uint8_t uiOutputs)
{
  uint32_t  passcode;

  // Create random passcode
  passcode = TRNGNumberGet(TRNG_LOW_WORD);
  passcode %= 1000000;

  // Display passcode to user
  if (uiOutputs != 0)
  {
    LCD_WRITE_STRING_VALUE("Passcode:", passcode, 10, LCD_PAGE4);
  }

  // Send passcode response
  GAPBondMgr_PasscodeRsp(connectionHandle, SUCCESS, passcode);
}
```

In this example, a random passcode is created and displayed on an LCD screen. The other connected device must then enter this passcode.

5.4.2.4.1 Authenticated Pairing with Bonding Enabled

After pairing and encrypting with MITM authentication, bonding occurs as shown in Section 5.4.2.3

5.5 Logical Link Control and Adaptation Layer Protocol (L2CAP)

The L2CAP layer sits on top of the HCI layer on the host side and transfers data between the upper layers of the host (GAP, GATT, application) and the lower layer protocol stack. This layer is responsible for protocol multiplexing capability, segmentation, and reassembly operation for data exchanged between the host and the protocol stack. L2CAP permits higher-level protocols and applications to transmit and receive upper layer data packets (L2CAP service data units, SDU) up to 64KB long. See Figure 5-14 for more information.

NOTE: The actual size is limited by the amount of memory available on the specific device being implemented. L2CAP also permits per-channel flow control and retransmission.

Copyright © 2015–2016, Texas Instruments Incorporated

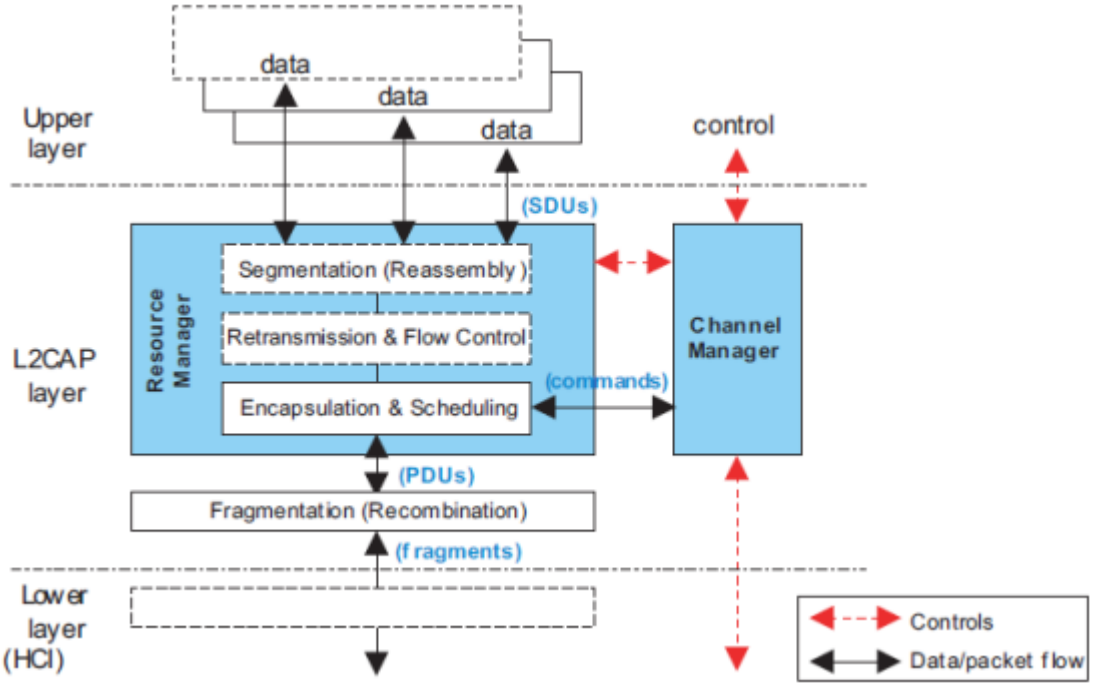


Figure 5-14. L2CAP Architectural Blocks

5.5.1 General L2CAP Terminology

Term	Description
L2CAP channel	The logical connection between two endpoints in peer devices, characterized by their Channel Identifiers (CIDs)
SDU or L2CAP SDU	Service Data Unit: a packet of data that L2CAP exchanges with the upper layer and transports transparently over an L2CAP channel using the procedures specified in this document
PDU or L2CAP PDU	Protocol Data Unit: a packet of data containing L2CAP protocol information fields, control information, and/or upper layer information data
Maximum Transmission Unit (MTU)	The maximum size of payload data, in octets, that the upper layer entity can accept (that is, the MTU corresponds to the maximum SDU size).
Maximum PDU Payload Size (MPS)	The maximum size of payload data in octets that the L2CAP layer entity can accept (that is, the MPS corresponds to the maximum PDU payload size).

Copyright © 2015–2016, Texas Instruments Incorporated

5.5.2 Maximum Transmission Unit (MTU)

The *Bluetooth* low energy stack supports fragmentation and recombination of L2CAP PDUs at the link layer. This fragmentation support allows L2CAP and higher-level protocols built on top of L2CAP, such as the attribute protocol (ATT), to use larger payload sizes, and reduce the overhead associated with larger data transactions. When fragmentation is used, larger packets are split into multiple link layer packets and reassembled by the link layer of the peer device. Figure 5-15 shows this relationship.

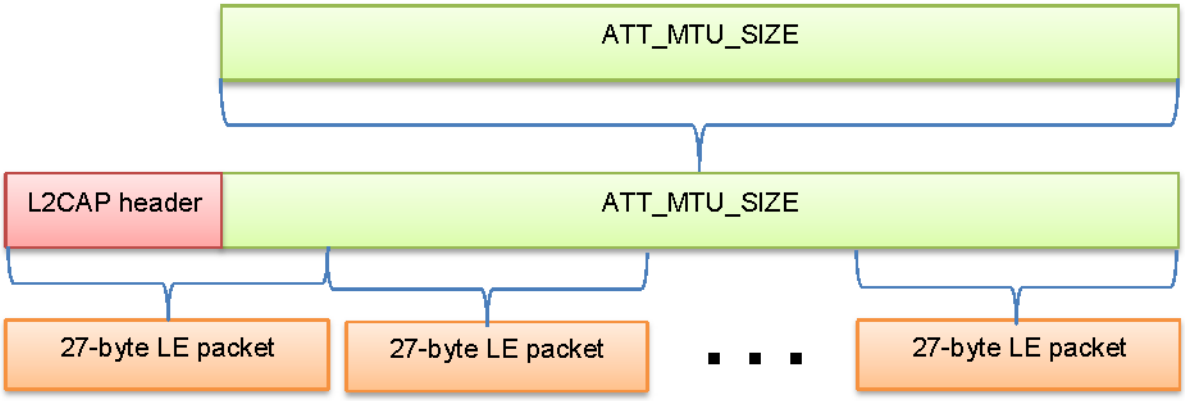


Figure 5-15. L2CAP Packet Fragmentation

The size of the L2CAP PDU also defines the size of the Attribute Protocol Maximum Transmission Unit (ATT_MTU). By default, LE devices assume the size of the L2CAP PDU is 27 bytes, which corresponds to the maximum size of the LE packet that can transmit in a single connection event packet. In this case, the L2CAP protocol header is 4 bytes, resulting in a default size for ATT_MTU of 23.

5.5.2.1 Configuring for Larger MTU Values

A client device can request a larger ATT_MTU during a connection by using the GATT_ExchangeMTU() command (see the API defined in Section D.3). During this procedure, the client (that is, Central) informs the server of its maximum supported receive MTU size and the server (that is, Peripheral) responds with its maximum supported receive MTU size. Only the client can initiate this procedure. When the messages are exchanged, the ATT_MTU for the duration of the connection is the minimum of the client MTU and server MTU values. If the client indicates it can support an MTU of 200 bytes and the server responds with a maximum size of 150 bytes, the ATT_MTU size is 150 for that connection. For more information, see the MTU Exchange section of Specification of the Bluetooth System, Covered Core Package, Version: 4.1.

Take the following steps to configure the stack to support larger MTU values.

1. Set the MAX_PDU_SIZE definition in bleUserConfig.h (see **Section 5.7**) to the maximum desired size of the L2CAP PDU size. The maximum ATT_MTU size is always 4 bytes less than the value of the MAX_PDU_SIZE.
2. Call GATT_ExchangeMTU() after a connection is formed (GATT client only). The MTU parameter passed into this function must be less than or equal to the definition from step 1.
3. Receive the ATT_MTU_UPDATED_EVENT in the calling task to verify that the MTU was successfully updated. This update requires the calling task to have registered for GATT messages. See **Section 5.3.6** for more information.

Though the stack can be configured to support a MAX_PDU_SIZE up to 255 bytes, each *Bluetooth* low energy connection initially uses the default 27 bytes (ATT_MTU = 23 bytes) value until the exchange MTU procedure results in a larger MTU size. The exchange MTU procedure must be performed on each *Bluetooth* low energy connection and must be initiated by the client.

Copyright © 2015–2016, Texas Instruments Incorporated

Increasing the size of the ATT_MTU increases the amount of data that can be sent in a single ATT packet. The longest attribute that can be sent in a single packet is (ATT_MTU-1) bytes. Procedures, such as notifications, have additional length restrictions. If an attribute value has a length of 100 bytes, a read of this entire attribute requires a read request to obtain the first (ATT_MTU-1) bytes, followed by multiple read blob request to obtain the subsequent (ATT_MTU-1) bytes. To transfer the entire 100 bytes of payload data with the default ATT_MTU = 23 bytes, five request or response procedures are required, each returning 22 bytes. If the exchange MTU procedure was performed and an ATT_MTU was configured to 101 bytes (or greater), the entire 100 bytes could be read in a single read request or response procedure.

NOTE: Due to memory and processing limitations, not all *Bluetooth* low energy systems support larger MTU sizes. Know the capabilities of expected peer devices when defining the behavior of the system. If the capability of peer devices is unknown, design the system to work with the default 27-byte L2CAP PDU/23-byte ATT_MTU size. For example, sending notifications with a length greater than 20 bytes (ATT_MTU-3) bytes results in truncation of data on devices that do not support larger MTU sizes.

5.5.3 L2CAP Channels

L2CAP is based around channels. Each endpoint of an L2CAP channel is referred to by a channel identifier (CID).See Volume 3, Part A, Section 2.1 of the Specification of the Bluetooth System, Covered Core Package, Version: 4.1 for more details on L2CAP Channel Identifiers. Channels can be divided into fixed and dynamic channels. For example, data exchanged over the GATT protocol uses channel 0x0004. A dynamically allocated CID is allocated to identify the logical link and the local endpoint. The local endpoint must be in the range from 0x0040 to 0xFFFF. This endpoint is used in the connection-orientated L2CAP channels described in the following section.

Copyright © 2015–2016, Texas Instruments Incorporated

5.5.4 L2CAP Connection-Oriented Channel (CoC) Example

The *Bluetooth* low energy stack SDK provides APIs to create L2CAP CoC channels to transfer bidirectional data between two *Bluetooth* low energy devices supporting this feature. This feature is enabled by default in the protocol stack. Figure 5-16 shows a sample connection and data exchange process between master and slave device using a L2CAP connection-oriented channel in LE Credit Based Flow Control Mode.

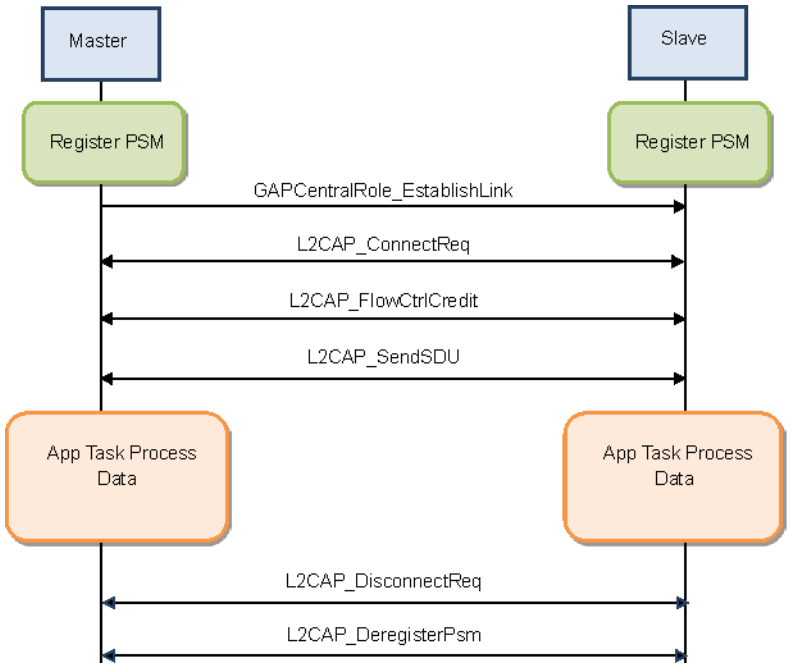


Figure 5-16. Sample Connection and Data Exchange Between a Master and Slave Device Using a L2CAP Connection-Oriented Channel in LE Credit Based Flow Control Mode

For more information on these L2CAP APIs, refer to the L2CAP API in Appendix G.

5.6 HCI

The HCI layer is a thin layer which transports commands and events between the host and controller. In a pure network processor application (that is, the HostTest project), the HCI layer is implemented through a transport protocol such as SPI or UART. In embedded SOC projects or the SimpleNP project, the HCI layer is implemented through function calls and callbacks. All of the commands and events discussed so far, such as ATT, GAP, and so forth, pass from the given layer through the HCI layer to the controller. The controller sends data to the layers through the HCI layer.

5.6.1 HCI Extension Vendor-Specific Commands

A number of HCI Extension Vendor-Specific Commands extend some of the functionality of the controller for use by the application or host. See Appendix H for a description of available HCI Extension Commands and examples for use in an SOC project.

5.6.2 Receiving HCI Extension Events in the Application

Similar to the GAP and ATT layers, HCI Extension Commands result in HCI Extension Events passing from the controller to the host. To receive these messages, a task must register with the stack using the GAP_RegisterForMsgs() function. The following is an example of this in SimpleBLEPeripheral.c.

```
// Register with GAP for HCI/Host messages
GAP_RegisterForMsgs(selfEntity);
```


Copyright © 2015–2016, Texas Instruments Incorporated

HCI events are passed as stack messages to the calling task. The following is an example of receiving these events in the SimpleBLEPeripheral_processStackMsg() function.

```
case HCI_GAP_EVENT_EVENT:
  {
    // Process HCI message
    switch(pMsg->status)
    {
      case HCI_COMMAND_COMPLETE_EVENT_CODE:
        // Process HCI Command Complete Event
        break;

      default:
        break;
    }
  }
  break;
```

5.7 Run-Time *Bluetooth* low energy Protocol Stack Configuration

The *Bluetooth* low energy protocol stack can be configured with various parameters that control its runtime behavior and RF antenna layout. The available configuration parameters are described in the bleUserConfig.h file in the ICallBLE IDE folder of the application. During initialization, these parameters are supplied to the *Bluetooth* low energy protocol stack by the user0Cfg structure, declared in main.c.

```
// BLE user defined configuration
bleUserCfg_t user0Cfg = BLE_USER_CFG;
```

Because the bleUserConfig.h file is shared with projects within the SDK, TI recommends defining the configuration parameters in the preprocessor symbols of the application when using a nondefault value. For example, to change the maximum PDU size from the default 27 to 162, set the preprocessor symbol MAX_PDU_SIZE = 162 in the application project. Increasing certain parameters may increase heap memory use by the protocol stack; adjust the HEAPMGR_SIZE as required. See Section 9.2 lists the available configuration parameters.

Table 5-2. *Bluetooth* low energy Stack Configuration Parameters

Parameter	Description
MAX_NUM_BLE_CONNS	Maximum number of simultaneous *Bluetooth* low energy connections. Default is 1 for Peripheral and Central roles. Maximum value is based on GAPRole.
MAX_NUM_PDU	Maximum number of *Bluetooth* low energy HCI PDUs. Default is 27. If the maximum number of connections is set to 0, then this number should also be set to 0.
MAX_PDU_SIZE	Maximum size in bytes of the *Bluetooth* low energy HCI PDU. Default is 27. Valid range is 27 to 255. The maximum ATT_MTU is MAX_PDU_SIZE - 4. See Section 5.5.2.1.
L2CAP_NUM_PSM	Maximum number of L2CAP Protocol/Service Multiplexers (PSM). Default is 3.
L2CAP_NUM_CO_CHANNELS	Maximum number of L2CAP Connection-Oriented (CO) Channels. Default is 3.
PM_STARTUP_MARGIN	Defines time in microseconds (µs) the system will wake up before the start of the connection event. Default is 300. This value is optimized for the example projects.
RF_FE_MODE_AND_BIAS	Defines the RF antenna front end and bias configuration. Set this value to match the actual hardware antenna layout. This value must be set directly in the bleUserConfig.h file by adding a board-type preprocessor defined symbol. Default values are based on Evaluation Module (EM) boards.

Copyright © 2015–2016, Texas Instruments Incorporated

5.8 Configuring *Bluetooth* low energy Protocol Stack Features

The *Bluetooth* low energy protocol stack can be configured to include or exclude certain *Bluetooth* low energy features by changing the library configuration in the stack project. Selecting the correct stack configuration is essential in optimizing the amount of flash memory available to the application. To conserve memory, exclude certain *Bluetooth* low energy protocol stack features that may not be required. The available *Bluetooth* low energy features are defined in the buildConfig.opt file in the TOOLS IDE folder of the stack project. Based on the features selected in the buildConfig.opt file, the LibSearch.exe tool selects the respective precompiled library during the build process of the stack project. Table 5-3 lists a summary of configurable features. See the buildConfig.opt file for additional details.

Table 5-3. *Bluetooth* low energy Protocol Stack Features

Feature	Description
HOST_CONFIG	*Bluetooth* low energy host configuration and associated GAP Role
GATT_DB_OFF_CHIP	GATT Database is off chip. Applicable to the HostTest project only.
DGAP_PRIVACY GAP_PRIVACY_RECONNECT	GAP Privacy Feature, applicable to the Peripheral Privacy feature only
GAP_BOND_MGR	Includes the Bond Manager. Required when using SNV.
L2CAP_CO_CHANNELS	Includes support for L2CAP Connection-Oriented Channels
GATT_NO_SERVICE_CHANGED	Whether or not to include the GATT service changed characteristic
HCI_TL_xxxx	Include HCI Transport Layer (FULL, PTM or NONE).
CTRL_V41_CONFIG	*Bluetooth* low energy Core Specifications V4.1 Controller Feature Partition Build Configuration

Copyright © 2015–2016, Texas Instruments Incorporated

Peripherals and Drivers

The TI-RTOS provides a suite of CC26xx peripheral drivers that can be added to an application. The drivers provide a mechanism for the application to interface to the CC26xx onboard peripherals and communicate with external devices.

6.1 Adding a Driver

The TI-RTOS drivers (and corresponding DriverLib) are provided in source and precompiled library form. By default, the FlashROM project configuration links to the prebuilt library in the following Linker→ Library IAR project options.

- **DriverLib**: $CC26XXWARE$\driverlib\bin\iar\driverlib.lib
- **TI-RTOS Drivers**: $TI_RTOS_DRIVERS_BASE$\ti\drivers\lib\drivers_cc26xxware.arm3

The $CC26XXWARE$ and $TI_RTOS_DRIVERS_BASE$ IAR argument variables refer to the installation location and can be viewed in IAR Tools→ Configure Custom Argument Variables menu. For CCS, the corresponding path variables are defined in the Project Options→ Resource→ Linked Resources, Path Variables tab.

To use a precompiled driver, include the C include file of respective driver in the application file (or files) where driver APIs are referenced.

For example, to add the PIN driver for reading or controlling an output I/O pin, do the following.

```
#include <ti/drivers/pin/PINCC26XX.h>
```

To override a specific prebuilt version of a driver, include the respective C source and include file (or files) to the project within the IDE. The IDE uses the source version (or versions) included in the project instead of the respective prebuilt library version. This override option is useful in cases where the prebuilt drivers are used for other drivers, but source-level debugging is available within the IDE for specific driver (or drivers).

NOTE: Due to flash size considerations, CCS uses the source drivers for all project configurations.

For FlashOnly project configurations required for Over-the-Air (OAD) download, the driver and DriverLib source must be directly added to the project. For a description of available features and driver APIs, refer to the TI-RTOS API Reference.

6.2 Board File

The board file sets the parameters of the fixed driver configuration for a specific board configuration, such as configuring the GPIO table for the PIN driver or defining which pins are allocated to the I2C, SPI or UART driver.

The board files for the SmartRF06 Evaluation Board are in the following path:

$TI_RTOS_DRIVERS_BASE$\ti\boards\SRF06EB\<Board_Type>

$TI_RTOS_DRIVERS_BASE$ is the path to the TI-RTOS driver installation and <Board_Type> is the actual Evaluation Module (EM). To view the actual path to the installed TI-RTOS version, see the following:

- IAR: Tools→ Configure Custom Argument Variables
- CCS: Project Options→ Resources→ Linked Resources, Path Variables tab

Copyright © 2015–2016, Texas Instruments Incorporated

The <Board_Type> is based on the preprocessor symbol and search path setting in the application project.

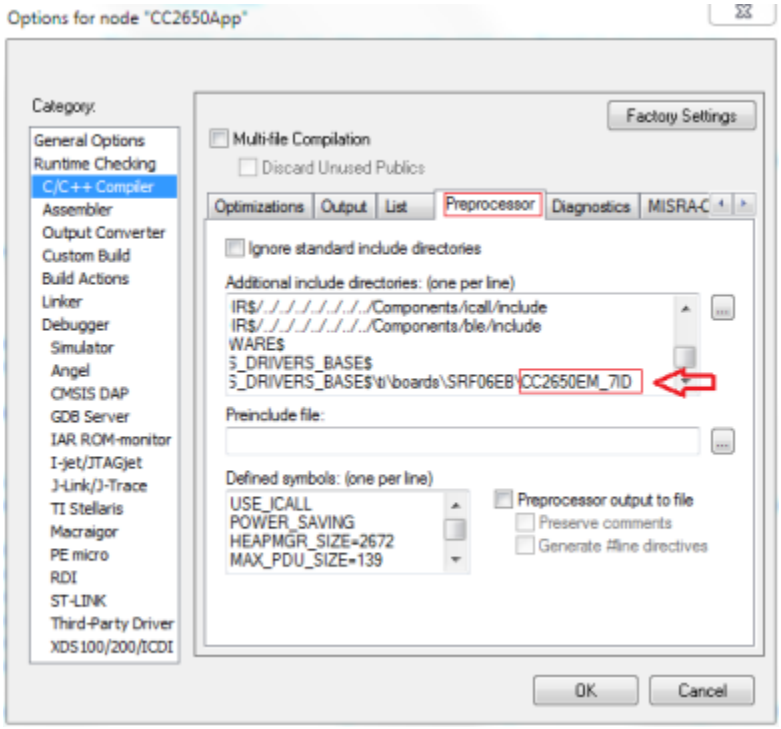
To change EM board type, do the following.

1. Update the Search Path to point to the source.
2. Include the file for the board type.

CC2640 EM options include CC2650EM_7ID, CC2650EM_5XD, or CC2650EM_4XS.

In IAR, do the following.

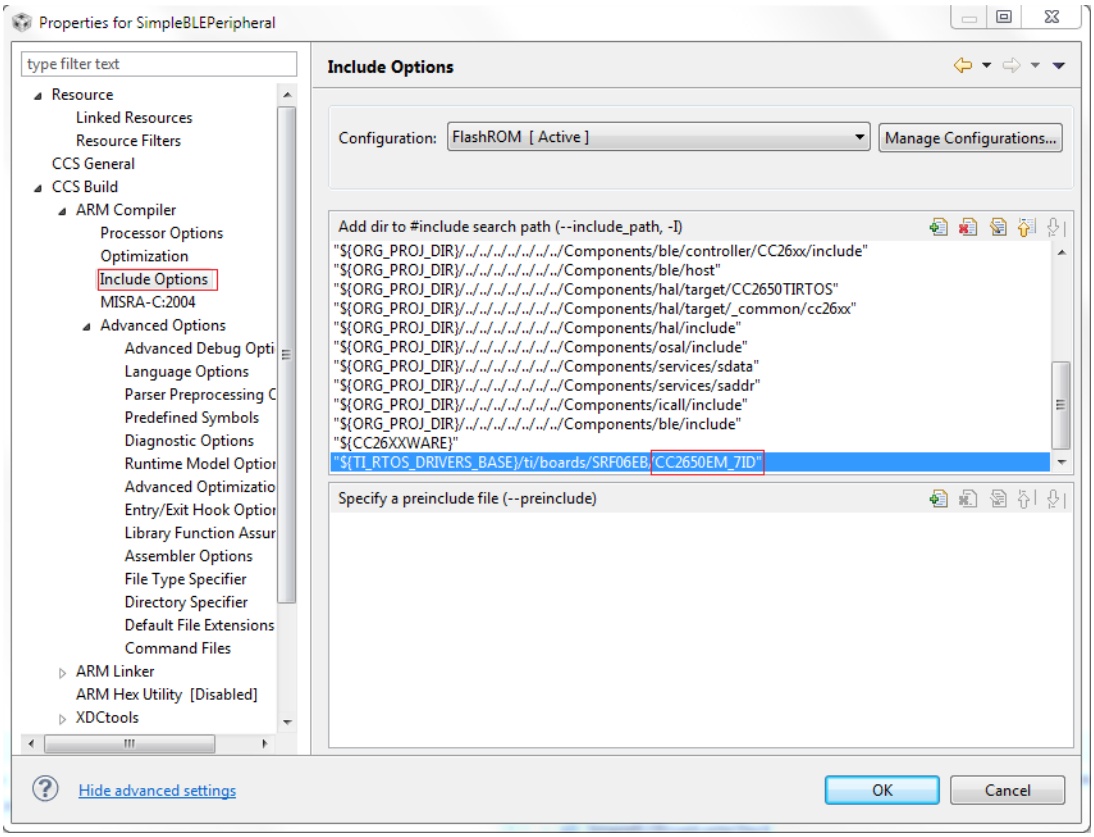
1. Open the Project Options for the application.
2. Locate the board type on the last line of the Additional include directories box in the Preprocessor tab.
3. Edit the following entry.



Copyright © 2015–2016, Texas Instruments Incorporated

In CCS, do the following.

1. Open the Project Properties for the application.
2. Locate the board type on the last line of the Include Options under CCS Build.
3. Edit this entry.



At a minimum, the board file must contain a PIN_Config structure that places all configured and unused pins in a default, safe state and defines the state when the pin is used.

```
/*
 *  ======================== IO driver initialization ========================
 *  From main, PIN_init(BoardGpioInitTable) should be called to setup safe
 *  settings for this board.
 *  When a pin is allocated and then de-allocated, it will revert to the state
 *  configured in this table.
*/
PIN_Config BoardGpioInitTable[] = {

    Board_LED1       | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW   | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED2       | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW   | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED3       | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW   | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_LED4       | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW   | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    Board_KEY_SELECT | PIN_INPUT_EN  | PIN_PULLUP | PIN_HYSTERESIS,
    Board_KEY_DOWN   | PIN_INPUT_EN  | PIN_PULLUP | PIN_HYSTERESIS,
    Board_UART_TX    | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH   | PIN_PUSHPULL,
    PIN_TERMINATE    /* Terminate list */
};
```

This structure is used to initialize the pins in main() as in Section 4.1.

```
PIN_init(BoardGpioInitTable);
```


Copyright © 2015–2016, Texas Instruments Incorporated

6.3 Available Drivers

This section describes each available driver and provide a basic example of adding the driver to the SimpleBLEPeripheral project. For more detailed information on each driver, see the TI-RTOS API Reference.

6.3.1 PIN

The PIN driver allows control of the I/O pins for software-controlled general-purpose I/O (GPIO) or connections to hardware peripherals. The SimpleBLECentral or SensorTagExample projects use the PIN driver. As stated in Section 6.2, the pins must first be initialized to a safe state in main(). After this initialization, any module can use the PIN driver to configure a set of pins for use. The following is an example of configuring the SimpleBLEPeripheral task to use one pin as an interrupt and another as an output to toggle when the interrupt occurs. IOID_x pin numbers map to DIO pin numbers as referenced in *TI CC26xx Technical Reference Manual* (SWCU117). The following table lists pins used and their mapping on the Smart RF 06 board.

Signal Name	Pin ID	SmartRF 06 Mapping:
Board_LED1	IOID_25	RF2.11 (LED1)
Board_KEY_UP	IDIO_19	RF1.10 (BTN_UP)

The following simpleBLEPeripheral.c code modifications are required.

1. Include PIN driver files.

```
#include <ti/drivers/pin/PINCC26XX.h>
```

2. Declare the pin configuration table and pin state and handle variables to be used by the SimpleBLEPeripheral task.

```
static PIN_Config SBP_configTable[] =
{
  Board_LED1  | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW   | PIN_PUSHPULL | PIN_DRVSTR_MAX,
  Board_KEY_UP  | PIN_INPUT_EN  | PIN_PULLUP | PIN_HYSTERESIS,
  PIN_TERMINATE
};

static PIN_State sbpPins;
static PIN_Handle hSbpPins;
```

3. Declare the ISR to be performed in the hwi context.

NOTE: This ISR is setting an event in the application task and waking it up to minimize processing in the hwi context.

```
static void buttonHwiFxn(PIN_Handle hPin, PIN_Id pinId)
{
  // set event in SBP task to process outside of hwi context
  events |= SBP_BTN_EVT;

  // Wake up the application.
  Semaphore_post(sem);
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

4. Define the event and related processing (in SimpleBLEPeripheral_taskFxn()) to handle event from above ISR.

```
#define SBP_BTN_EVT                           0x0010
    if (events & SBP_BTN_EVT)
    {
      events &= ~SBP_BTN_EVT; //clear event

      //toggle LED1
      if (LED_value)
      {
        PIN_setOutputValue(hSbpPins, Board_LED1 , LED_value--);
      }
      else
      {
        PIN_setOutputValue(hSbpPins, Board_LED1, LED_value++);
      }
    }
```

5. Open the pins for use and configure the interrupt in SimpleBLEPeripheral_init().

```
// Open pin structure for use
hSbpPins = PIN_open(&sbpPins, SBP_configTable);
// Register ISR
PIN_registerIntCb(hSbpPins, buttonHwiFxn);
// Configure interrupt
PIN_setConfig(hSbpPins, PIN_BM_IRQ, Board_KEY_UP | PIN_IRQ_NEGEDGE);
// Enable wakeup
PIN_setConfig(hSbpPins, PINCC26XX_BM_WAKEUP, Board_KEY_UP|PINCC26XX_WAKEUP_NEGEDGE);
```

6. Compile.
7. Download.
8. Run.

NOTE: Pushing the Up button on the SmartRF06 toggles the LED1. No debouncing is implemented.

6.3.2 UART

There are many ways to configure the UART driver. See TI-RTOS API Reference for more information. An example project that uses the UART driver is HostTest. In addition to a UART driver, the HostTest project includes additional GPIOs for power management, a packet parser, and other items that are beyond the scope of this documentation. This section shows an example for using the UART driver with the default settings from UART_Params_init(): blocking mode, baud rate 115200, and so forth.

The following example uses the UART peripheral defined in the board file.

Signal Name	Pin ID	SmartRF 06 Mapping:
Board_UART_RX	IOID_2	RF1.7 (UART_RX)
Board_UART_TX	IDIO_3	RF1.9 (UART_TX)

Perform the following steps to add the UART driver:

1. Include the UART driver.

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

2. Declare the UART handle and parameter structures as local variables.

Copyright © 2015–2016, Texas Instruments Incorporated

```
static UART_Handle SbpUartHandle;
static UART_Params SbpUartParams;
```

3. Initialize the UART driver in SimpleBLEPeripheral_init()).

```
  UART_Params_init(&SbpUartParams);
  SbpUartHandle = UART_open(CC2650_UART0, &SbpUartParams);
```

4. Perform a sample 5-byte UART write.

```
  uint8 txbuf[] = {0,1,2,3,4};
  UART_write(SbpUartHandle, txbuf, 5);
```

6.3.3 SPI

There are many ways to configure the SPI driver. See the TI-RTOS API Reference for more information. An example project that uses the SPI driver is the LCD in the simpleBLEPeripheral project. This section provides an example for using the SPI driver with the default settings from SPI_Params_init(): as a master, blocking mode, and so forth. The board file for simpleBLEPeripheral declares two SPI peripherals where the LCD peripheral already uses Board_SPI0. The following example uses Board_SPI1.

Signal Name	Pin ID	SmartRF 06 Mapping:
Board_SPI1_MISO	IOID_24	RF2.10
Board_SPI1_MOSI	IOID_23	RF2.5
Board_SPI1_CLK	IOID_30	RF2.12
Board_SPI1_CSN	IOID_26	RF2.6

NOTE: Remove the jumpers on these pins to disconnect them from any SmartRF06 peripherals.

Modify the board file (Board.c) as follows.

NOTE: While the Board_SPI1 peripheral is declared in the board.c file, its pins are not set to a safe initialization state in BoardGpioInitTable.

1. Add the following line to this table.

```
Board_SPI1_CSN    | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH    | PIN_PUSHPULL
```

2. The board.c file assumes that the CSN pin is controlled by the application. For simplicity in this example, this pin is controlled by the SPI driver.
3. Assign the CSN pin to be used by the SPI driver.

```
    {   /* SRF06EB_CC2650_SPI1 */
        .baseAddr = SSI1_BASE,
        .intNum = INT_SSI1,
        .defaultTxBufValue = 0,
        .powerMngrId = PERIPH_SSI1,
        .rxChannelBitMask  = 1<<UDMA_CHAN_SSI1_RX,
        .txChannelBitMask  = 1<<UDMA_CHAN_SSI1_TX,
        .mosiPin = Board_SPI1_MOSI,
        .misoPin = Board_SPI1_MISO,
        .clkPin = Board_SPI1_CLK,
        .csnPin = Board_SPI1_CSN,
    }
```


Copyright © 2015–2016, Texas Instruments Incorporated

Perform the following steps to add the SPI driver.

1. Include the SPI driver.

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

2. Declare the SPI handle and parameter structures as local variables.

```
static SPI_Handle SbpSpiHandle;
static SPI_Params SbpSpiParams;
```

3. Initialize the SPI driver in SimpleBLEPeripheral_init()).

```
  SPI_Params_init(&SbpSpiParams);
  SbpSpiHandle = SPI_open(CC2650_SPI1, &SbpSpiParams);
```

4. Perform a sample 5-byte SPI transfer.

```
  uint8 txbuf[] = {0,1,2,3,4};
  uint8 rxbuf[5];
  SPI_Transaction spiTransaction;
  spiTransaction.arg = NULL;
  spiTransaction.count = 5;
  spiTransaction.txBuf = txbuf;
  spiTransaction.rxBuf = rxbuf;
  SPI_transfer(SbpSpiHandle, &spiTransaction);
```

6.3.4 I^2C

There are many ways exist to configure the I^2C driver. See the TI-RTOS API Reference for more information. An example project that uses the I^2C driver is the SensorTag. This section provides an example using the I^2C driver with the default settings from I2C_Params_init(): as a master, in blocking mode. Because there is no I^2C peripheral on the SmartRF06 board, the LED1 and LED2 pins are used for I2C as follows.

Signal Name	Pin ID	SmartRF 06 Mapping:
Board_SDA	IOID_25	RF2.11 (LED1)
Board_SCL	IDIO_27	RF2.13 (BTN_UP)

NOTE: Remove the jumpers on these pins to disconnect them from the LEDs.

Modify the board as follows.

1. Comment out the LED pins.
2. Define the I^2C pins in the Board.h file.

```
//#define Board_LED1                      IOID_25
//#define Board_LED2                      IOID_27
#define Board_SDA                       IOID_25
#define Board_SCL                       IOID_27
```

3. Replace the LED configuration in the initial GPIO configuration table with the I^2C pin configuration in the Board.c file.

```
PIN_Config BoardGpioInitTable[] = {
  Board_SDA   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW    | PIN_PUSHPULL | PIN_DRVSTR_MAX,
  Board_SCL   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW    | PIN_PUSHPULL | PIN_DRVSTR_MAX,
```


Copyright © 2015–2016, Texas Instruments Incorporated

4. Declare the I²C structures in the Board.c file.

```
/* Place into subsections to allow the TI linker to remove items properly */
#if defined(__TI_COMPILER_VERSION__)
#pragma DATA_SECTION(i2cCC26XX_config, ".const:i2cCC26XX_config")
#pragma DATA_SECTION(i2cCC26XXHWAttrs, ".const:i2cCC26XXHWAttrs")
#endif

/* Include drivers */
#include <ti/drivers/I2C/I2CCC26XX.h>

/* I2C objects */
I2CCC26XX_Object i2cCC26XXObjects[CC2650_I2CCOUNT];

/* I2C hardware parameter structure, also used to assign UART pins */
const I2CCC26XX_HWAttrs i2cCC26XXHWAttrs[CC2650_I2CCOUNT] = {
    {    /* CC2650_I2C0 */
        .baseAddr = I2C0_BASE,
        .intNum = INT_I2C,
        .powerMngrId = PERIPH_I2C0,
        .sdaPin = Board_SDA,
        .sclPin = Board_SCL
    },
};

/* I2C configuration structure */
const I2C_Config I2C_config[] = {
    { &I2CCC26XX_fxnTable, &i2cCC26XXObjects[0], &i2cCC26XXHWAttrs[0] },
    { NULL, NULL, NULL }
};
```


Copyright © 2015–2016, Texas Instruments Incorporated

When the Board.c file has been modified to add an I²C peripheral, perform the steps to initialize I²C and a sample transaction from the SimpleBLEPeripheral task as follows. Code changes are in the simpleBLEPeripheral.c file.

1. Include the I²C driver.

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

2. Declare the I²C handle and parameter structures as local variables.

```
static I2C_Handle SbpI2cHandle;
static I2C_Params SbpI2cParams;
```

3. Initialize the I²C driver in SimpleBLEPeripheral_init()).

```
  I2C_Params_init(&SbpI2cParams);
  SbpI2cHandle = I2C_open(CC2650_I2C0, &SbpI2cParams);
```

4. Perform a sample 5-byte I²C transfer.

```
  I2C_Transaction i2cTransaction;
  uint8 txBuf[] = {0,1,2,3,4};
  uint8 rxBuf[5];
  i2cTransaction.writeBuf = txBuf;
  i2cTransaction.writeCount = 5;
  i2cTransaction.readBuf = rxBuf;
  i2cTransaction.readCount = 5;
  i2cTransaction.slaveAddress = 0x0A; //arbitrary for demo
  I2C_transfer(SbpI2cHandle, &i2cTransaction);
```

5. Compile.
6. Download.
7. Run.

Copyright © 2015–2016, Texas Instruments Incorporated

Sensor Controller

The sensor controller engine (SCE) is an autonomous processor within the CC2640. The SCE can control the peripherals in the sensor controller independently of the main CPU. The main CPU does not have to wake up to (for example) execute an ADC sample or poll a digital sensor over SPI, and it saves both current and wake-up time that would otherwise be wasted. A PC tool lets you configure the sensor controller and choose which peripherals are controlled and which conditions wake up the main CPU. The sensor controller studio (SCS) is a stand-alone IDE to develop and compile microcode for execution on the SCE. Refer to Sensor Controller Studio for more details on the SCS, including documentation embedded within the SCS IDE.

Copyright © 2015–2016, Texas Instruments Incorporated

For a complete description of the CC2640 reset sequence, see *TI CC26xx Technical Reference Manual* (SWCU117).

8.1 Programming Internal Flash With the ROM Bootloader

The CC2640 internal flash memory can be programmed using the bootloader in the ROM of the device. Both UART and SPI protocols are supported. For more details on the programming protocol and requirements, see Chapter 9 of *TI CC26xx Technical Reference Manual* (SWCU117).

NOTE: Because the ROM bootloader uses predefined DIO pins for internal flash programming, allocate these pins in the layout of your board. For details on the pins allocated to the bootloader based on the chip package type, see *TI CC26xx Technical Reference Manual* (SWCU117) .

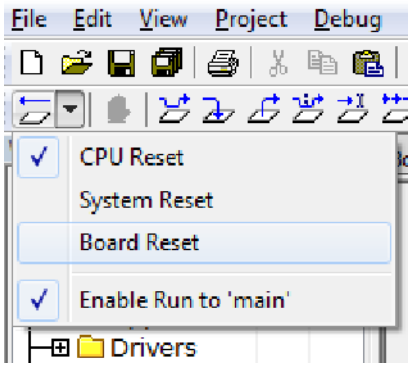
8.2 Resets

Use only hard resets to reset the device. From software, a reset can occur through one of the following.

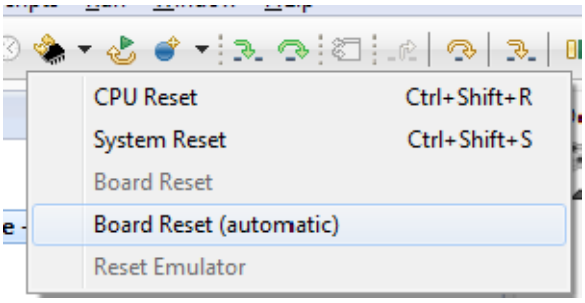
```
HCI_EXT_ResetSystemCmd(HCI_EXT_RESET_SYSTEM_HARD);
```

```
HAL_SYSTEM_RESET();
```

In IAR, select the Board Reset option from the following Reset (back arrow) Debug Menu drop-down box.



In CCS, select Board Reset from the reset menu.



Copyright © 2015–2016, Texas Instruments Incorporated

Development and Debugging

9.1 Debug Interfaces

The CC2640 platform supports the cJTAG (2-wire) and JTAG (4-wire) interfaces. Any debuggers that support cJTAG, like the TI XDS100v3 and XDS200, work natively. Others interfaces, like the IAR I-Jet and Segger J-Link, can only be used in JTAG mode but their drivers inject a cJTAG sequence which enables JTAG mode when connecting. The hardware resources included on the devices for debugging are listed as follows. Not all debugging functionality is available in all combinations of debugger and IDE.

- Breakpoint unit (FBP) – 6 instruction comparators, 2 literal comparators
- Data watchpoint unit (DWT) – 4 watchpoints on memory access
- Instrumentation Trace Module (ITM) – 32 × 32 bit stimulus registers
- Trace Port Interface Unit (TPIU) – serialization and time-stamping of DWT and ITM events

The SmartRF06 Board contains a XDS100v3 debug probe. This debugger is used by default in the sample projects.

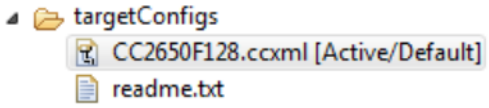
9.1.1 Connecting to the XDS Debugger

If only one debugger is attached, the IDE uses it automatically. If multiple debuggers are connected, you must choose the individual debugger. The following steps detail how to select a debugger in CCS and IAR.

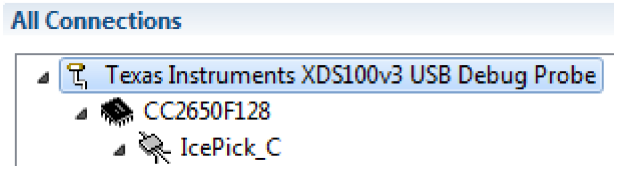
9.1.1.1 Debugging Using CCS

To debug using CCS, do as follows:

1. Open the target configuration file.
2. Open the Advanced pane.



3. Choose the top-level debugger entry.



4. Choose to select by serial number.
5. Enter the serial number.

To find the serial number for XDS100v3 debuggers, do as follows.

1. Open a command prompt.

Copyright © 2015–2016, Texas Instruments Incorporated

2. Run C:\ti\ccsv6\ccs_base\common\uscif\xds100serial.exe to get a list of serial numbers of the connected debuggers.

9.1.1.2 Debugging Using IAR

To debug using IAR, do as follows.

1. Open the project options (Project→ Options).
2. Go to the Debugger entry.
3. Go to Extra options.
4. Add the following command line option: --drv_communication=USB:#select

Adding this command line option makes the IAR prompt which debugger to use for every connection.

9.2 Breakpoints

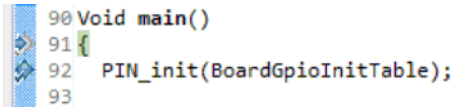
Both IAR and CCS reserve one of the instruction comparators. Five hardware breakpoints are available for debugging. This section describes setting breakpoints in IAR and CCS.

9.2.1 Breakpoints in CCS

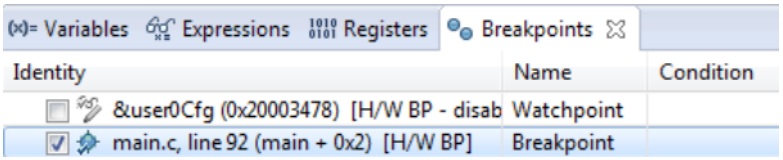
To toggle a breakpoint, do any of the following.

- Double-click the area to the left of the line number.
- Press Ctrl+Shift+B.
- Right-click on the line.
  - Select Breakpoint→ Hardware Breakpoint.

A breakpoint set on line 92 looks like the following.



For an overview of the active and inactive breakpoints, click on View→ Breakpoints.



To set a conditional break, do as follows.

1. Right-click the breakpoint in the overview.
2. Choose Properties.

When debugging, Skip Count and Condition can help skip a number of breaks or only break if a variable is a certain value.

NOTE: Conditional breaks require a debugger response and may halt the processor long enough to break. For example, a conditional break can break an active *Bluetooth* low energy connection if the condition is false or the skip count has yet to be reached.

9.2.2 Breakpoints in IAR

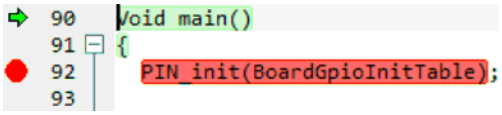
To toggle a breakpoint, do any of the following.

- Single-click the area to the left of the line number.
- Go to the line.
  - Press F9.

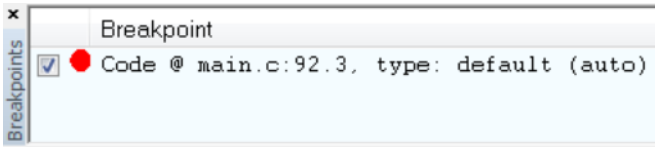
Copyright © 2015–2016, Texas Instruments Incorporated

- Right-click on the line.
  - Select Toggle Breakpoint (Code).

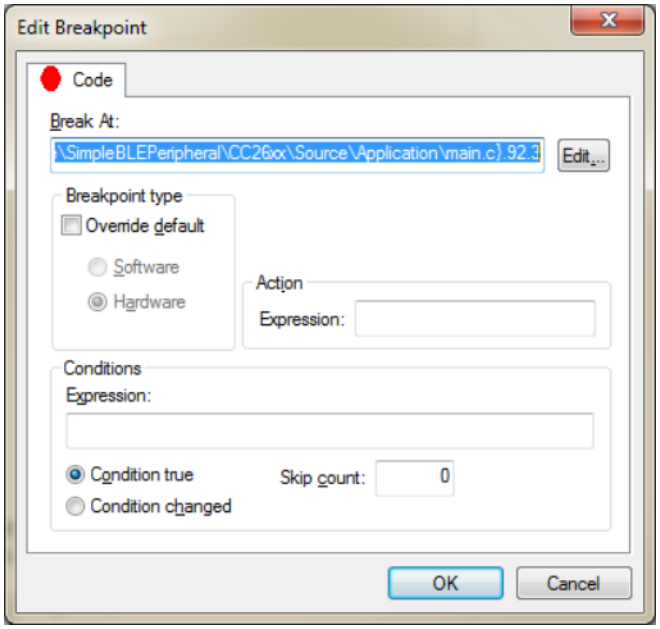
A breakpoint set on line 92 looks like the following.



For an overview of the active and inactive breakpoints, click View→ Breakpoints.



To set a conditional break, do as follows.



1. Right-click the breakpoint in the overview.
2. Choose Edit....

When debugging, Skip Count and Condition can help skip a number of breaks or only break if a variable is a certain value.

NOTE: Conditional breaks require a debugger response and may halt the processor long enough to break. For example, a conditional break can break an active *Bluetooth* low energy connection if the condition is false or the skip count has not been reached.

9.2.3 *Considerations When Using Breakpoints With an Active Bluetooth low energy Connection*

Because the *Bluetooth* low energy protocol is timing sensitive, any breakpoints break the execution long enough to lose network timing and break the link. Place breakpoints as close as possible to where the relevant debug information can be read or step through the relevant code segment to debug. This closeness also lets you experiment on breakpoint placements by restarting debugging and repeating the conditions that cause the code to hit the breakpoint.

Copyright © 2015–2016, Texas Instruments Incorporated

9.2.4 Considerations no Breakpoints and Compiler Optimization

When the compiler is optimizing code, toggling a breakpoint on a line of C code may not result in the expected behavior. Some examples include the following.

- Code is removed or not compiled in: Toggling a breakpoint in the IDE results in a breakpoint some other unintended place and not on the selected line. Some IDEs disable breakpoints on nonexisting code.
- Code block is part of a common subexpression: For example, a breakpoint might toggle inside a function called from one other function, but can also break due to a call from another unintended function.
- An if clause is represented by a conditional branch in assembly: A breakpoint inside an if clause always breaks on the conditional statement, even if not executed.

TI recommends selecting an optimization level as low as possible when debugging. See Section 9.4 for information on modifying optimization levels.

9.3 Watching Variables and Registers

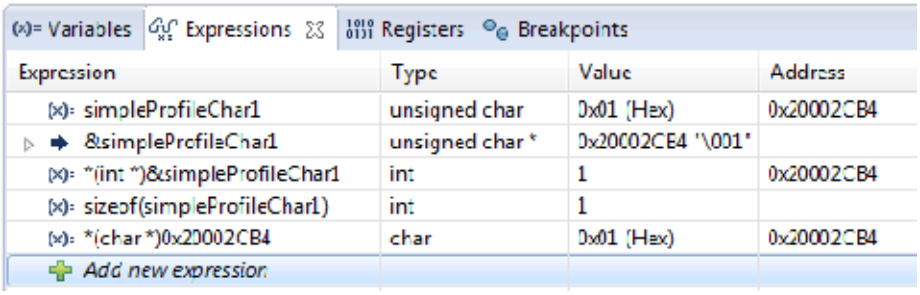
IAR and CCS provide several ways of viewing the state of a halted program. Global variables are statically placed during link-time and can end up anywhere in the RAM available to the project or potentially in flash if they are declared as a constant value. These variables can be accessed at any time through the Watch and Expression windows.

Unless removed due to optimizations, global variables are always available in these views. Local variables or variables that are only valid inside a limited scope are placed on the stack of the active task. Such variables can also be viewed with the Watch or Expression views, but can also be automatically displayed when breaking or stepping through code. To view the variables through IAR and CCS, do as follows.

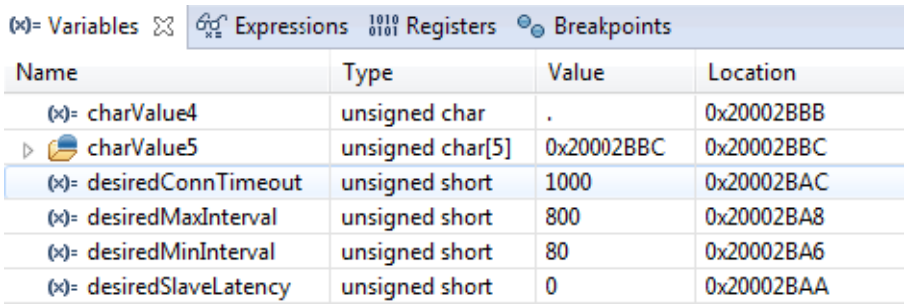
9.3.1 Variables in CCS

You can view Global Variables by doing either of the following.

- Select View→ Expressions.
- Select a variable name in code.
  - Right-click and select Add Watch Expression.



Select View→ Variables to automatically viewed Local Variables.

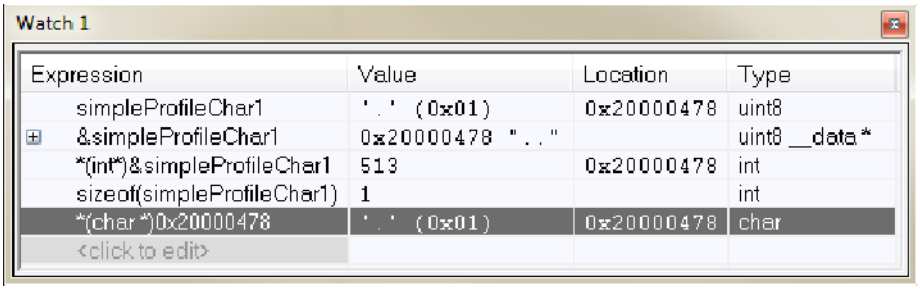


Copyright © 2015–2016, Texas Instruments Incorporated

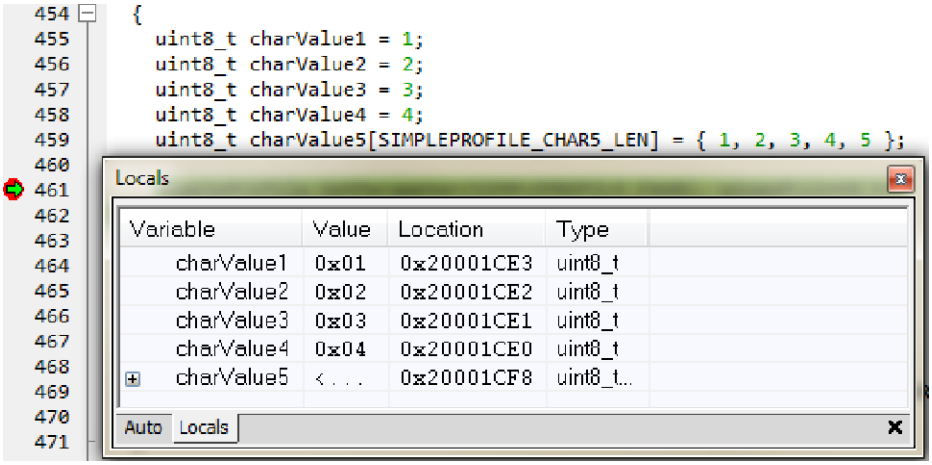
9.3.2 Variables in IAR

To view Global Variables, do either of the following.

- Right-click on the variable.
  - Select Add to Watch: varName.
- Select View→ Watch n.
  - Enter the name of the variable.



View→ Locals show the local variables in IAR.



9.3.3 Considerations When Viewing Variables

Local variables are often placed in CPU registers and not on the stack. These variables also have a limited lifetime even within the scope in which they are valid, depending on the optimization performed. Both CCS and IAR may struggle to show a particular variable due to its limited lifetime. The solution when debugging is as follows.

- Move the variable to global scope, so it remains accessible in RAM.
- Make the variable volatile, so the compiler fails to use a limited scope.
- Make a shadow copy of the variable that is global and volatile.

NOTE: IAR may remove the variable during optimization. If so, add the __root directive to volatile.

9.4 Memory Watchpoints

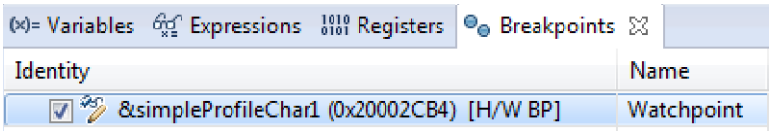
As mentioned in Chapter 9. the DWT module contains four memory watchpoints that allow breakpoints on memory access. The hardware match functionality looks only at the address. If intended for use on a variable, the variable must be global. Using watchpoints is described for IAR and CCS as follows.

NOTE: If a data watchpoint with value match is used, two of the four watchpoints are used.

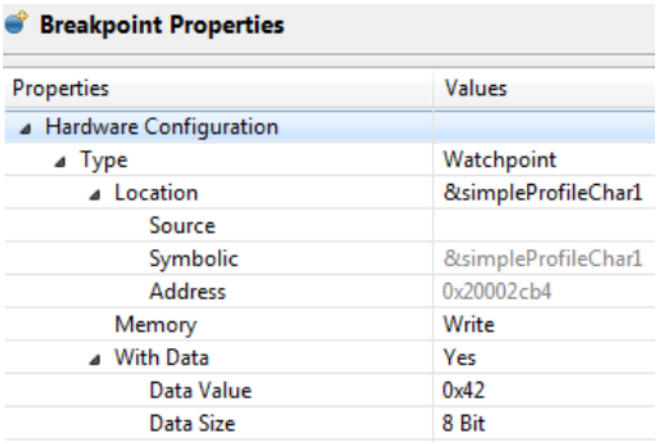
Copyright © 2015–2016, Texas Instruments Incorporated

9.4.1 Watchpoints in CCS

1. Right-click on a global variable.
2. Select Breakpoint→ Hardware Watchpoint to add it to the breakpoint overview.



3. Right-click and edit the Breakpoint Properties to configure the watchpoint.

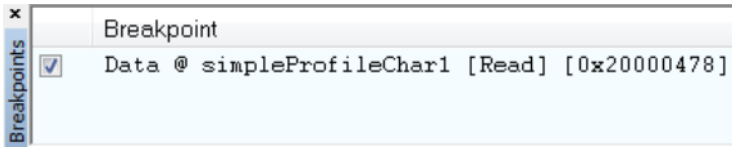


This example configuration ensures that if 0x42 is written to the memory location for Characteristic 1 in the *Bluetooth* low energy SimpleBLEPeripheral example project. The device halts execution.

9.4.2 Watchpoints in IAR

NOTE: IAR currently does not support the watchpoint functionality with the XDS debuggers, but an IAR I-Jet can be used to accomplish this.

1. Right-click a variable.
2. Select Set Data Breakpoint for myVar to add it to the active breakpoints.



3. Right-click from the breakpoints view.
4. Choose Edit... to set up whether the watchpoint should match on read, write, or any access.

Copyright © 2015–2016, Texas Instruments Incorporated

Figure 9-1 shows a break on read access when the value matches 0x42.

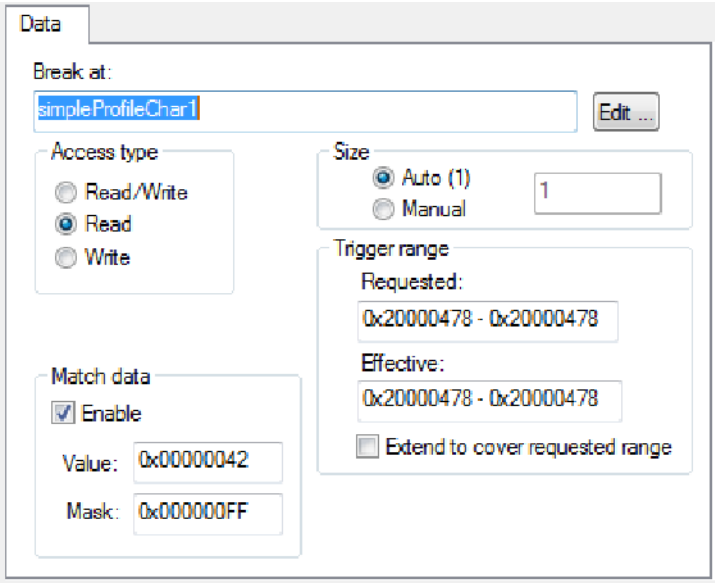


Figure 9-1. Break on Read Access

9.5 TI-RTOS Object Viewer

Both IAR and CCS include the RTOS Object Viewer (ROV) plug-in that provides insight into the current state of TI-RTOS, including task states, stacks, and so forth. Because both CCS and IAR have a similar interface, these examples discuss only CCS.

To access the ROV in IAR, do as follows.

1. Use the TI-RTOS menu on the menu bar.
2. Select a subview.

To access the ROV in CCS, do as follows.

1. Click the Tools menu.
2. Click RTOS Object View.

This section discusses some ROV views useful for debugging and profiling.

9.5.1 Scanning the BIOS for Errors

The BIOS→ Scan for errors view sweep through the available ROV modules and report any errors. This functionality can be a point to start if anything has gone unpredictably wrong. This scan only shows errors related to TI-RTOS modules and only the errors it can catch. See Figure 9-2.

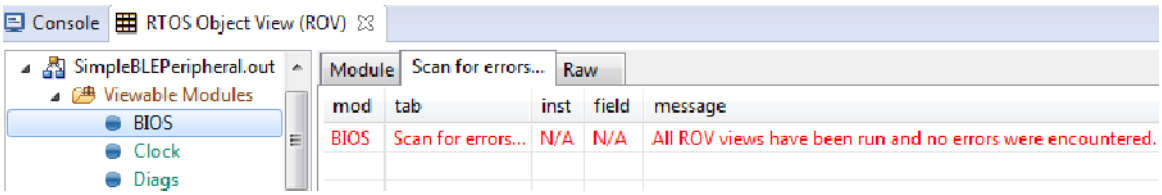


Figure 9-2. Error Scan

Copyright © 2015–2016, Texas Instruments Incorporated

9.5.2 Viewing the State of Each Task

The Task→ Detailed view is useful for seeing the state of each task and its related runtime stack usage. This example shows the state the first time the user-thread is called. Figure 9-3 shows the Idle task, the GAPRole task, the SimpleBLEPeripheral task, and the *Bluetooth* low energy stack task, represented by its ICall proxy.

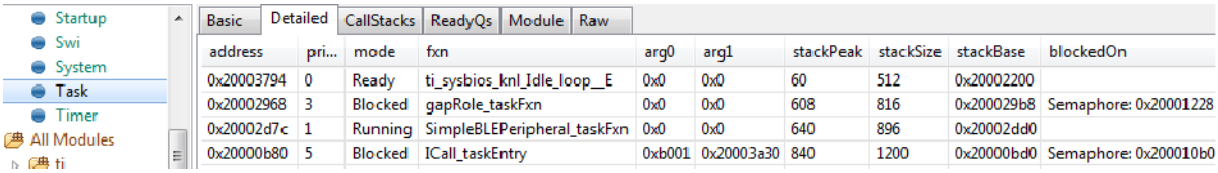


Figure 9-3. Viewing State of RTOS Tasks

The following list explains the column in Figure 9-3 (see Section 3.3 for more information on runtime stacks).

- address: This column shows the memory location of the Task_Struct instance for each task.
- priority: This column shows the TI-RTOS priority for the task.
- mode: This column shows the current state of the task.
- fxn: This column shows the name of the entry function of the task.
- arg0, arg1: These columns show abitrary values that can be given to entry function of the task. In the image, the ICall_taskEntry is given 0xb001, which is the flash location of the entry function of the RF stack image and 0x20003a30 (the location of bleUserCfg_t user0Cfg, defined in main()).
- stackPeak: This column shows the maximum run-time stack memory used based on watermark in RAM, where the stacks are prefilled with 0xBE and there is a sentinel word at the end of the run-time stack.

NOTE: Function calls may push the stack pointer out of the run-time stack, but not actually write to the entire area. A stack peak near stackSize but not exceeding it may indicate stack overflow.

- stackSize: This column shows the size of the runtime stack, configured when instantiating a task.
- stackBase: This column shows the logical top of the runtime stack of the task (usage starts at stackBase + stackSize and grows down to this address).
- blockedOn: This column shows the type and address of the synchronization object; the thread is blocked on if available. For semaphores, the addresses are listed under Semaphore→ Basic.

9.5.3 Viewing the System Stack

The Hwi→ Module view allows profiling of the system stack used during boot or for main(), Hwi execution, and SWI execution. See Section 3.11.3 for more information on the system stack. For more information, see Figure 9-4 for the more details.

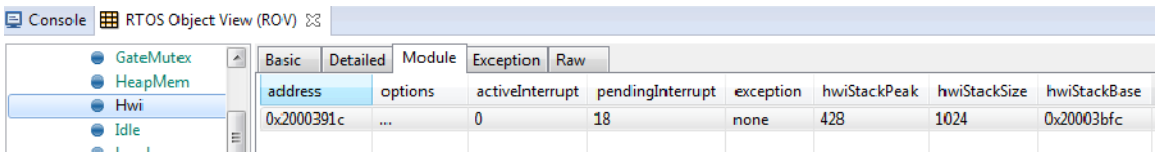


Figure 9-4. Viewing the System Stack in Hwi

The hwiStackPeak, hwiStackSize, and hwiStackBase can be used to check for system stack overflow.

Copyright © 2015–2016, Texas Instruments Incorporated

9.5.4 Viewing Power Manager Information

The Power→ Module view shows a value that is *logical or* of all the constraints currently enforced through the Power API. The value in the example (0x06) indicates Standby Disallow (0x4) and Shutdown Disallow (0x02); these numeric preprocessor symbols are subject to change. See TI-RTOS Power Management for CC26xx for more information.

9.6 Profiling the ICall Heap Manager (heapmgr.h)

As described in Section 3.11.5, the ICall Heap Manager and its heap are used to allocate messages between the *Bluetooth* low energy stack task and the application task and as dynamic memory allocations in the tasks.

Profiling functionality is provided for the ICall heap but is not enabled by default. Therefore, it must be compiled in by adding HEAPMGR_METRICS to the defined preprocessor symbols. This functionality is useful for finding potential sources for unexplained behavior and to optimize the size of the heap. When HEAPMGR_METRICS is defined, the variables and functions listed as follows become available.

Global variables:

- heapmgrBlkMax: the maximum amount of simultaneous allocated blocks
- heapmgrBlkCnt: the current amount of allocated blocks
- heapmgrBlkFree: the current amount of free blocks
- heapmgrMemAlo: the current total memory allocated in bytes
- heapmgrMemMax: the maximum amount of simultaneous allocated memory in blocks (this value must not exceed the size of the heap)
- heapmgrMemUb: the furthest memory location of an allocated block, measured as an offset from the start of the heap
- heapmgrMemFail: the amount of memory allocation failure (instances where ICall_malloc() has returned NULL)

Functions:

- void ICall_heapGetMetrics(u16 *pBlkMax, u16 *pBlkCnt, u16 *pBlkFree, u16 *pMemAlo, u16 *pMemMax, u16 *pMemUb)
  - returns the previously described variables in the pointers passed in as parameters
- int heapmgrSanityCheck(void)
  - returns 0 if the heap is ok; otherwise, returns a nonzero (that is, an array access has overwritten a header in the heap)

9.7 Optimizations

While debugging, turn off or lower optimizations to ease single-stepping through code. This optimization is possible at the following levels.

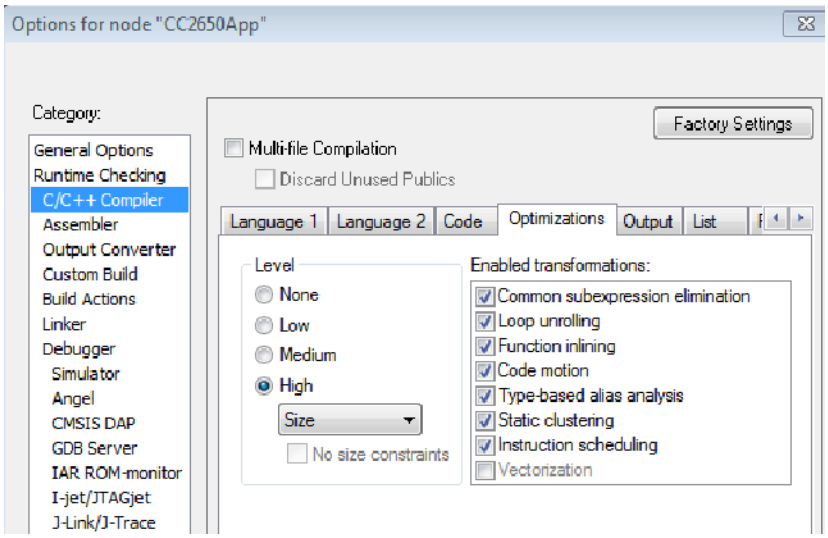
Copyright © 2015–2016, Texas Instruments Incorporated

9.7.1 Project-Wide Optimizations

There may not be enough available flash to do project-wide optimizations. The following screen shots show how to set up optimization options for IAR and CCS for the entire project.

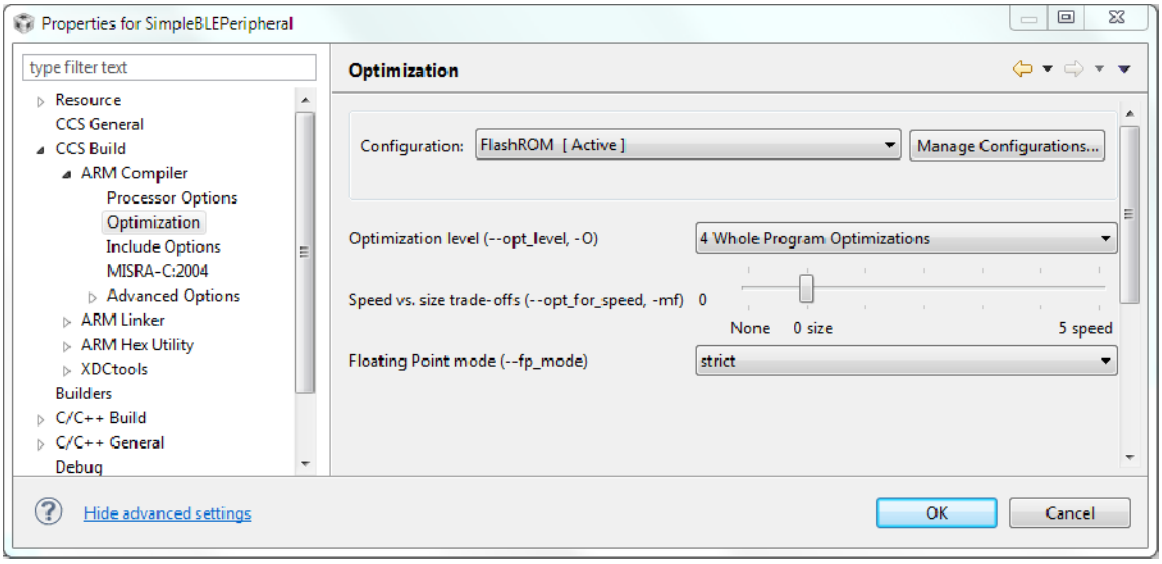
In IAR:

Project Options→ C/C++ Compiler→ Optimizations



In CCS:

Project Properties→ CCS Build→ ARM Compiler→ Optimization



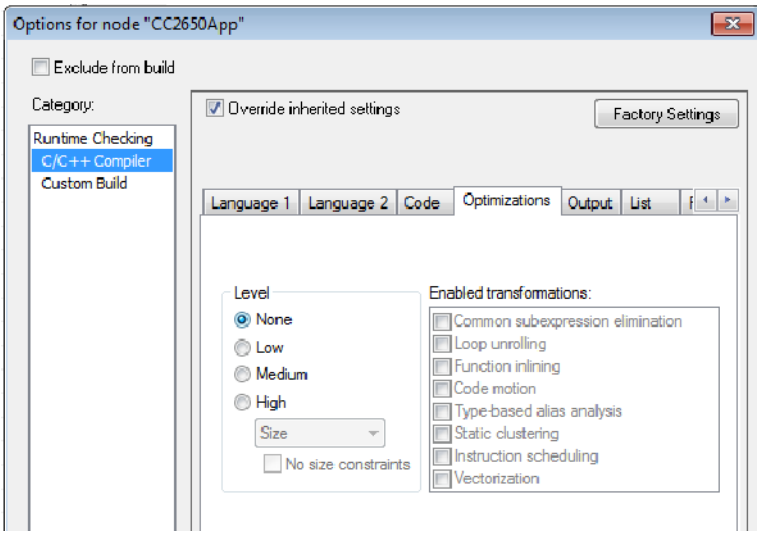
Copyright © 2015–2016, Texas Instruments Incorporated

9.7.2 Single-File Optimizations

In IAR:

1. Right-click on the file in the Workspace pane.
2. Choose Options.
3. Check Override inherited Settings.
4. Choose the optimization level.

NOTE: Do single-file optimizations with care because this also overrides the project-wide preprocessor symbols.



In CCS:

1. Right-click on the file in the Workspace pane.
2. Choose Properties.
3. Change the optimization level of the file using the same menu in the CCS project-wide optimization menu.

9.7.3 Single-Function Optimizations

Using compiler directives, you can control the optimization level of a single function.

In IAR:

Use #pragma optimize=none before the function definition to deoptimize the entire function, that is, as follows.

```
#pragma optimize=none
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
  // Initialize application
  SimpleBLEPeripheral_init();

  // Application main loop
  for (;;)
...
```


Copyright © 2015–2016, Texas Instruments Incorporated

9.8 Deciphering CPU Exceptions

Several possible exception causes exist. If an exception is caught, an exception handler function can be called. Depending on the project settings, this handler may be a default handler in ROM, which is just an infinite loop or a custom function called from this default handler instead of a loop.

When an exception occurs, the exception may be caught and halted in debug mode immediately, depending on the debugger. If the execution halted manually later through the Break debugger, it is then stopped within the exception handler loop.

9.8.1 Exception Cause

With the default setup using TI-RTOS, the exception cause can be found in the System Control Space register group (CPU_SCS) in the register CFSR (Configurable Fault Status Register). The ARM Cortex-M3 User Guide describes this register. Most exception causes fall into the following three categories.

- Stack overflow or corruption leads to arbitrary code execution.
  - Almost any exception is possible.
- A NULL pointer has been dereferenced and written to.
  - Typically (IM)PRECISERR exceptions
- A peripheral module (like UART, Timer, and so forth) is accessed without being powered.
  - Typically (IM)PRECISERR exceptions

The CFSR is available in View→ Registers in IAR and CCS.

When an access violation occurs, the exception type is IMPRECISERR because writes to flash and peripheral memory regions are mostly buffered writes.

If the CFSR:BFARVALID flag is set when the exception occurs (typical for PRECISERR), the BFAR register in CPU_SCS can be read out to find which memory address caused the exception.

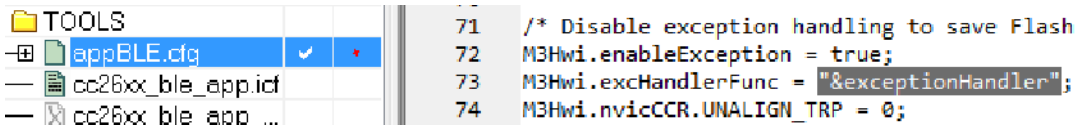
If the exception is IMPRECISERR, PRECISERR can be forced by manually disabling buffered writes. Set [CPU_SCS:ACTRL:DISDEFWBUF] to 1, by either manually setting the bit in the register view in IAR/CCS or by including <inc/hw_cpu_scs.h> from Driverlib and calling the following.

```
HWREG(CPU_SCS_BASE + CPU_SCS_O_ACTLR) = CPU_SCS_ACTLR_DISDEFWBUF;
```

NOTE: This negatively affects performance.

9.8.2 Using a Custom Exception Handler

You can use a custom exception handler instead of the default exception handler from ROM. In the sample projects, this handler is configured in appBle.cfg, through the M3Hwi.excHandlerFunc property.



When this function is called, the Core-M3 processor has pushed the core registers R0-3, R12, PC, LR, and xPSR on the run-time stack of the active task when the exception was registered. The TI-RTOS exception handler has also pushed R4-11 onto the runtime stack.

Copyright © 2015–2016, Texas Instruments Incorporated

9.8.3 Parsing the Exception Frame

The custom exception handler must be of the following type.

```
void exceptionHandler(struct exceptionFrame *e, unsigned int execLr){..}
```

execLr is the LR value set by the Core-M3 and e points to the following structure that describes the CPU state (core registers) when the exception occurred.

```
struct execptionFrame
{
  unsigned int _r4;
  unsigned int _r5;
  unsigned int _r6;
  unsigned int _r7;
  unsigned int _r8;
  unsigned int _r9;
  unsigned int _r10;
  unsigned int _r11;
  unsigned int _r0;
  unsigned int _r1;
  unsigned int _r2;
  unsigned int _r3;
  unsigned int _r12;
  unsigned int _lr;
  unsigned int _pc;
  unsigned int _xpsr;
};
```

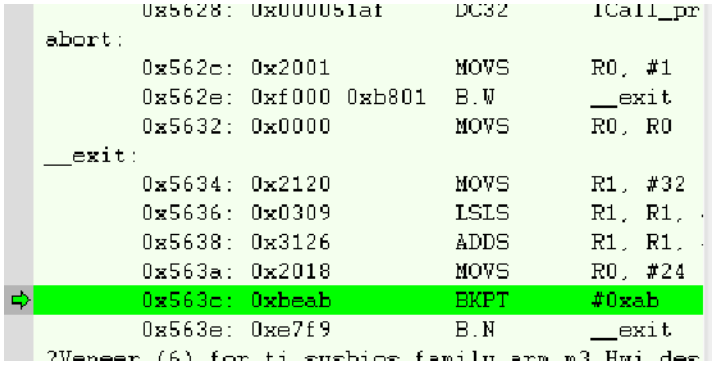
NOTE: Due to optimization, these variables are often not shown properly in the watch windows of the IDE. The following shows the sample IAR implementation.

```
#pragma optimize=none
void exceptionHandler(struct execptionFrame *e, unsigned int eLR)
{
  static __root unsigned int failPC = 0;
  static __root unsigned int lr = 0;

  failPC = e->_pc; // This is the Program Counter when the exception happened

  lr = eLR;        // The exception LR

  while(1);
}
```


Copyright © 2015–2016, Texas Instruments Incorporated

9.9 Debugging a Program Exit

The program must never exit the main() function. If this occurs, the disassembly looks like the following.



This code sequence can be seen in the disassembly when the ICall_abort() function is called, which can be caused by the following:

- Calling an ICall function from a stack callback
- Misconfiguring of additional ICall tasks or entities
- Incorrect ICall task registering

A breakpoint can be set in the ICall_abort function to trace from where this error is coming.

9.10 Debugging Memory Problems

This section describes how to debug a situation where the program runs out of memory, either on the heap or on the runtime stack for the individual thread contexts. Exceeding array bounds or dynamically allocating too little memory for a structure corrupts the memory and can cause an exception like INVPC, INVSTATE, IBUSERR to appear in the CFSR register.

9.10.1 Task and System Stack Overflow

If an overflow on the runtime stack of the task or the system stack occurs (as found using the ROV plug-in as in Section 9.5.2 and Section 9.5.3), perform the following steps.

1. Note the current size of the runtime stack of each task.
2. Increase by a few 100 bytes as described in Section 3.3.1 and Section 3.11.3.
3. Reduce the runtime stack sizes so that they are larger than their respective stackPeaks to save some memory.

Copyright © 2015–2016, Texas Instruments Incorporated

9.10.2 Dynamic Allocation Errors

Section 9.6 describes how to use the ICall Heap profiling functionality. To check if dynamic allocation errors occurred, do as follows:

1. Determine if memAlo or memMax approaches the preprocessor-defined HEAPMGR_SIZE.
2. Check memFail to see if allocation failures have occurred.
3. Call the sanity check function.

If the heap is sane but there are allocation errors, increase HEAPMGR_SIZE and see if the problem continues.

You can set a breakpoint in heapmgr.h in HEAPMGR_MALLOC() on the line hdr = NULL; to find the allocation that is failing.

9.11 Preprocessor Options

Preprocessor symbols configure system behavior, features, and resource usage at compile time. Some symbols are required as part of the *Bluetooth* low energy system, while others are configurable. See Section 2.6 for details on accessing preprocessor symbols within the IDE. Symbols defined in a particular project are defined in all files within the project.

9.11.1 Modifying

To disable a symbol, put an *x* in front of the name. To disable power management, change POWER_SAVING to xPOWER_SAVING.

Copyright © 2015–2016, Texas Instruments Incorporated

9.11.2 Options

Table 9-1 lists the preprocessor symbols used by the application in the SimpleBLEPeripheral project. Symbols that must remain unmodified are marked with an *N* in the Modify column while modifiable; configurable symbols are marked with a *Y*.

Table 9-1. Application Preprocessor Symbols

Preprocessor Symbol	Description	Modify
USE_ICALL	Required to use ICall *Bluetooth* low energy and primitive services.	N
POWER_SAVING	Power management is enabled when defined, and disabled when not defined. Requires same option in stack project.	Y
HEAPMGR_SIZE=2672	Defines the size in bytes of the ICall heap. Memory is allocated in .bss section.	Y
ICALL_MAX_NUM_TASKS=3	Defines the number of ICall aware tasks. Modify only if adding a new RTOS task that uses ICall services.	Y
ICALL_MAX_NUM_ENTITIES=6	Defines maximum number of entities that use ICall, including service entities and application entities. Modify only if adding a new RTOS task that uses ICall services.	Y
TI_DRIVERS_SPI_DMA_INCLUDED	Includes SPI DMA driver. Pins associated with the SPI are defined in the board file. This preprocessor symbol is required when using the LCD driver on the CC2650EM 7x7 Evaluation Module.	Y
TI_DRIVERS_LCD_INCLUDED	Includes SmartRF06 LCD driver. This preprocessor symbol is required to use the LCD on the CC2650EM 7x7 Evaluation Module. The SPI DMA driver is a required to use the LCD driver.	Y
MAX_NUM_BLE_CONNS=1	This is the maximum number of simultaneous *Bluetooth* low energy collections allowed. Adding more connections uses more RAM and may require increasing HEAPMGR_SIZE. Profile heap usage accordingly.	Y
CC26XXWARE	This includes DriverLib.	N
CC26XX	This selects the chipset.	N
xdc_runtime_Assert_DISABLE_ALL	Disables XDC run-time assert	N
xdc_runtime_Log_DISABLE_ALL	Disables XDC run-time logging	N
HEAPMGR_METRICS	Enables collection of ICall heap metrics. See Section 9.6 for details on how to profile heap usage.	Y

Copyright © 2015–2016, Texas Instruments Incorporated

Table 9-2 lists the only stack preprocessor options that may be modified:

Table 9-2. Stack Preprocessor Symbols

Preprocessor Symbol	Description	Modify
POWER_SAVING	Power management is enabled when defined, and disabled when not defined. Requires the same option in application project.	Y
GATT_NO_CLIENT	When defined, the GATT client is not included to save flash. GATT client is excluded from most peripheral projects, included in central and certain peripheral projects (for example, TimeApp).	Y
OSAL_SNV	Select the number of NV pages to use for SNV. Each page is 4kB of flash. A minimum of one page is required when GAP_BOND_MANAGER is defined. See Section 3.10.4	Y

9.12 Check System Flash and RAM Usage With Map File

Both application and stack projects produce a map file which can be used to compute the combined flash and RAM system memory usage. Both projects have their own memory space and both map files must be analyzed to determine the total system memory usage. The map file is in the output folder of the respective project in IAR. To compute the total memory usage, do as follows.

1. Open the application map file (that is, SimpleBLEPeripheralApp.map).

   NOTE: At the end of the file, three lines contain a breakdown of memory usage for read-only code, read-only data, and read/write data.

2. Add the two values for read-only code and read-only data memory.

   NOTE: This sum is the total flash memory usage for the application project. The read/write data memory is the total RAM usage by the application project.

3. Note these values.
4. Open the stack map file.
5. Compute the same flash and RAM values for the stack project.
6. Add the total flash memory value from the application with the total flash usage of the stack to determine the total system flash usage.
7. Add the total RAM usage from the application with the stack to get the total system RAM usage.

For CCS, the map file of the respective project gives a summary of flash and RAM usage. To determine the remaining available memory for each project, see Section 3.10 (flash) and Section 3.11 (RAM). Due to section placement and alignment requirements, some remaining memory may be unavailable. The map file memory usage is valid only if the project builds and links successfully.

Copyright © 2015–2016, Texas Instruments Incorporated

Creating a Custom *Bluetooth* low energy Application

As a *Bluetooth* low energy system designer, you must have a firm grasp on the general system architecture, application, and *Bluetooth* low energy stack framework to implement a custom *Bluetooth* Smart application. This section provides indications on where and how to start writing a custom application. Decide what role and purpose the custom application should have. If an application is tied to a specific service or profile, start with that. An example is the Heart Rate sensor project. Start with one of the following SimpleBLE sample projects.

- SimpleBLECentral
- SimpleBLEPeripheral
- SimpleBLEBroadcaster
- SimpleBLEObserver

10.1 Adding a Board File

After you select the reference application, preprocessor symbol and add a board file that matches the custom board layout. The following steps provide guidance on adding a custom board file to the project.

1. Create a custom board file (TI recommends using the Evaluation Module [EM] board file as a starting reference).
2. Modify the PIN structure.
3. Add peripheral driver initialization objects according to the board design.
4. Remove the existing EM board C.
5. Include files from the folder of the start-up application.
6. Add the custom board file to the application project.
7. Update the C compiler search path of the IDE to point to the header file of the new board file.
8. Define a new board file identifier.
9. Add the RF front-end and bias configuration to the bleUserConfig.h file.
10. Refer to the direction in this file for guidance on adding a new custom board RF configuration.
11. Rebuild the application project.

10.2 Configuring Parameters for Custom Hardware

1. Set parameters, such as the sleep clock accuracy of the 32.768-kHz crystal.
2. Define the CCFG parameters.

For a description of CCFG configuration parameters, see *TI CC26xx Technical Reference Manual*, (SWCU117).

10.3 Creating Additional Tasks

Many implementations can use the RTOS environment to operate in the application task framework. If the system design requires the addition of an additional RTOS task, see Section 3.3.1 for guidance on adding a task.

10.4 Configuring the *Bluetooth* low energy Stack

Configure the *Bluetooth* low energy protocol stack with parameter and features. Section 5.7 and Section 5.8 describe available options.

Copyright © 2015–2016, Texas Instruments Incorporated

10.4.1 Minimizing Flash Usage

To increase the available flash memory allocated to the application project, you must minimize the flash usage of the stack by including only *Bluetooth* low energy features required to implement the defined role of the device. This increase of available application flash memory comes at the tradeoff of some features. To minimize flash usage, do the following.

1. Verify that your application uses the optimize for flash size compiler optimization settings (default for TI projects).
2. Use only one page of SNV or do not use any NV pages.
3. Set the NO_OSAL_SNV stack preprocessor option. (Do not include GAP_BOND_MGR in buildConfig.opt because it removes pairing and bonding capability. See Section 3.10.4 for a description of SNV.)
4. Exclude the GATT client functionality by defining the GATT_NO_CLIENT predefined symbol in the stack project for peripheral devices. (Peripheral devices do not implement the GATT client role.)
5. Do not include the GATT service changed characteristic by defining GATT_NO_SERVICE_CHANGED in buildConfig.opt.
6. Exclude *Bluetooth* 4.1 specific features from the *Bluetooth* low energy stack for devices that use only *Bluetooth* 4.0.

For example, the *Bluetooth* 4.1 controller and L2CAP connection-oriented channels features can be excluded by commenting out the following lines from buildConfig.opt.

```
/* L2CAP Connection Oriented Channels */
/* -DL2CAP_CO_CHANNELS */
..
/* BLE Core Spec V4.1 Controller Feature Partition Build Configuration. Comment out to
use default Controller Configuration  */
/* -DCTRL_V41_CONFIG=PING_CFG+SLV_FEAT_EXCHG_CFG+CONN_PARAM_REQ_CFG+MST_SLV_CFG */
```

10.5 Defining *Bluetooth* low energy Behavior

This step involves using *Bluetooth* low energy protocol stack APIs to define the system behavior and adding profiles, defining the GATT database, configuring the security model, and so forth. Use the concepts explained in Chapter 5 as well as the *Bluetooth* low energy API reference in Appendix A.

Copyright © 2015–2016, Texas Instruments Incorporated

Porting from CC254x to CC2640

11.1 Introduction

TI-RTOS is the new operating environment for *Bluetooth* low energy projects on CC26xx devices. This software is a multithreaded environment where the protocol stack, application, and its profiles exist on different threads. TI-RTOS has similar features to OSAL but different mechanisms for accomplishing them. This section covers the main differences between TI-RTOS and OSAL when developing applications on top of the *Bluetooth* low energy protocol stack. Although the incorporation of the RTOS is a major architecture change, *Bluetooth* low energy APIs and related procedures are similar to CC254x.

This section covers the following topics:

- OSAL
- Application and stack separation with ICall
- Threads, semaphores, and queues
- Peripheral Drivers
- Event Processing

Most of these differences are unique to TI-RTOS. This section covers these differences and how they relate to OSAL.

11.2 OSAL

A major change in moving to TI-RTOS is the complete removal of the application from the OSAL environment. While the stack code uses OSAL within its own thread, the application thread can only use the APIs of OSAL that are defined in ICallBleAPI.c. Many functions such as osal_memcpy(), osal_memcmp(), and osal_mem_alloc() are unavailable. These functions have been replaced by TI-RTOS, C run time, and ICall APIs.

11.3 Application and Stack Separation With ICall

In the CC2640 *Bluetooth* low energy protocol stack, the application is a separate image from the stack image unlike the OSAL method, which consists of only a single image. The benefit for this separation is detailed in the ICall (see Section 4.2). This structure allows independent upgrading of the application and stack images.

The address of the startup entry for the stack image is known by the application image at build time so the application image knows where the stack image starts. Messages between the application and stack pass through a framework developed called ICall short for indirect function calls. This functionality lets the application call the same APIs used in OSAL but is parsed by the ICall and sent to the stack for processing. Many of these stack functions are defined in ICallBleAPI.c for the application to use transparently while ICall handles the sending and receiving from the stack transparently.

Copyright © 2015–2016, Texas Instruments Incorporated

11.4 Threads, Semaphores, and Queues

Unlike single-threaded operating systems such as OSAL, TI-RTOS is multithreaded with custom priorities for each thread. The TI-RTOS handles thread synchronization and APIs are provided for the application threads to use to maintain synchronization between different threads. Semaphores are the prime source of synchronization for applications. The semaphores are used to pass event messages to the event processor of the application.

Profile callbacks that run in the context of the *Bluetooth* low energy protocol stack thread are made re-entrant by storing event data and posting a semaphore of the application to process in the context of the application. Similarly, key press events and clock events that run in ISR context also post semaphores to pass events to the application. Unique to TI-RTOS, queues are how applications process events in the order the events were called and make callback functions from profiles and the stack re-entrant. The queues also provide a FIFO ordering for event processing. An example project may use a queue to manage internal events from an application profile or a GAP profile role (for example, Peripheral or Central). ICall uses a queue and it is accessed through the ICall API. For a description of the TI-RTOS objects used by the *Bluetooth* low energy stack SDK, see Chapter 3.

11.5 Peripheral Drivers

Aside from switching to an RTOS-based environment, peripheral drivers represent a significant change from the CC254x architecture. Any drivers used by the CC254x software must be ported to the respective TI-RTOS driver interfaces. For details on adding and using a CC26xx peripheral driver, see Chapter 6.

11.6 Event Processing

Similar to OSAL, each RTOS task has two functions that implement the fundamental tasks for an application: SimpleBLEPeripheral_init() and SimpleBLEPeripheral_taskFxn().

SimpleBLEPeripheral_init() contains ICall registration routines and initialization functions for the application profiles and the GAP and GATT roles. Function calls that are normally in the START_DEVICE_EVT event of the CC254x application are also made in the SimpleBLEPeripheral_init() function. The initialization includes setting up callbacks that the application should receive from the profile and stack layers. For more details on callbacks, see Section 4.3.3.

SimpleBLEPeripheral_taskFxn() contains an infinite loop in which events are processed. After entry of the loop and having just finished initialization, the application task calls ICall_wait() to block on its semaphore until an event occurs. For more information on how the application processes different events, see Section 4.3.2.2.

Similar to osal_set_event() in a CC254x application, the application task can post the semaphore of the applciation with a call to Semaphore_post(sem) after setting an event such as in SimpleBLEPeripheral_clockHandler(). An alternative way is to enqueue a message using SimpleBLEPeripheral_enqueueMsg(,) which preserves the order in which the events are processed. Similar to osal_start_timerEx() in a CC254x application, you can use a clock to set an event after a predetermined amount of time using Util_constructClock().This function can also set a periodic event as shown in the SimpleBLEPeripheral project.

Events come from within the same task, the profiles, and the stack. Events from the stack are handled first with a call to ICall_fetchServiceMsg() similar to osal_msg_receive() in a CC254x application. Internal events and messages from the profiles and the GAP role profiles received in callback functions must be treated as re-entrant are handled in the SimpleBLEPeripheral_taksFxn() function too. In many cases such as in GAP role profile callbacks, you must place events in a queue to preserve the order in which messages arrive. For more information, see Section 4.3 for general overview of application architecture.

Copyright © 2015–2016, Texas Instruments Incorporated

Sample Applications

The purpose of this section is to give an overview of the sample applications included in the TI *Bluetooth* low energy stack software development kit. Some of these implementations are based on specifications that have been adopted by the *Bluetooth* Special Interest Group (*Bluetooth* SIG), while others are based on specifications that have not been finalized. Some applications are not based on any standardized profile being developed by the *Bluetooth* SIG but are custom implementations developed by TI.

All projects contain an IAR and a CCS implementation. Also, each project contains a release and a debug configuration to assist in the development process. Except for the SimpleLink *Bluetooth* Smart CC2650 SensorTag, TI intends most sample applications described in this section to run on the SmartRF06 Evaluation Board using a CC26xx Evaluation Module.

12.1 Blood Pressure Sensor

This sample project implements the Blood Pressure profiles in a *Bluetooth* low energy peripheral device to provide an example blood pressure monitor (BPM) using simulated measurement data. The application implements the Sensor role of the blood pressure profile. The project is based on the adopted profile and service specifications for blood pressure. The project also includes the Device Information Service. The project is configured to run on the SmartRF06 board.

12.1.1 Interface

This application has two button inputs.

SmarfRF Button Right—When disconnected, this button toggles advertising on and off. When connected, this button increases the value of various measurements.

SmartRF Button Up—This button cycles through measurement formats.

12.1.2 Operation

The following steps detail how to use the Blood Pressure Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the blood pressure sensor from a blood pressure collector peer device.

NOTE: The peer device discovers the blood pressure service and configures it to enable indication or notifications of the blood pressure measurement. The peer device may also discover the device information service for more information such as the manufacturing and serial number. When blood pressure measurements have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats in the following order:
   - MMHG | TIMESTAMP | PULSE | USER | STATUS
   - MMHG | TIMESTAMP
   - MMHG
   - KPA
   - KPA | TIMESTAMP
   - KPA |TIMESTAMP | PULSE

Copyright © 2015–2016, Texas Instruments Incorporated

If the peer device initiates pairing, the blood pressure sensor requires a passcode. The default passcode is 000000. When the connection terminates, the BPM does not begin advertising until the button is pressed. The peer device may also query the blood pressure for read-only device information. The GATT_DB excel sheet for this project lists further details on the supported items (for example, model number, serial number, and so forth).

12.2 Heart Rate Sensor

This sample project implements the Heart Rate and Battery profiles in a *Bluetooth* low energy peripheral device to provide an example heart rate sensor using simulated measurement data. The application implements the Sensor" role of the Heart Rate profile and the Battery Reporter role of the Battery profile. The project is based on adopted profile and service specifications for Health Rate. The project also includes the Device Information Service. The project is configured to run on the SmartRF06 board.

12.2.1 Interface

When the left button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF is connected, it cycles through different heart rate sensor data formats. When in a connection and the battery characteristic is enabled for notification, the battery level is periodically notified.

12.2.2 Operation

The following steps detail how to use the Heart Rate Sensor sample project:

1. Power up the device.
2. Press the left button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the heart rate sensor from a heart rate collector peer device.

NOTE: The peer device discovers the heart rate service and configure it to enable notifications of the heart rate measurement. The peer device may also discover and configure the battery service for battery level-state notifications. When heart rate measurement notifications have been enabled the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
   - Sensor contact not supported
   - Sensor contact not detected
   - Sensor contact and energy expended set
   - Sensor contact and R-R Interval set
   - Sensor contact, energy expended, and R-R Interval set
   - Sensor contact, energy expended, R-R Interval, and UINT16 heart rate set
   - Nothing set

If the peer device initiates pairing, the devices pair. Only just works pairing is supported by the application (pairing without a passcode). The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason, the application starts advertising at the slow interval. The advertising intervals and durations are configurable in file heartrate.c.

12.3 Cycling Speed and Cadence (CSC) Sensor

This sample project implements the CSC profile in a *Bluetooth* low energy peripheral device to provide a sample application of sensor that would be placed on a bicycle, using simulated measurement data. The application implements the Sensor role of the CSC. This profile also uses of the optional Device Info Service, which has default values that may be altered at compile or run time to help identify a specific *Bluetooth* low energy device. This project is configured to run on the SmartRF06 board.

Copyright © 2015–2016, Texas Instruments Incorporated

12.3.1 Interface

When the right button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF, it cycles through different cycling speed and cadence sensor data formats.

Pressing the select key initiates a soft reset. A soft reset includes the following.

- Terminating all current connections
- Clearing all bond data
- Clearing white list of all peer addresses

12.3.2 Operation

The following steps detail how to use the Cycling Speed and Cadence Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the cycling sensor from a CSC collector peer device.

NOTE: The peer device receives a slave security request and initiates a bond. When bonded, the collector discovers the CSC service and configures it to enable CSC measurements. When CSC measurement notifications have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
   - Sensor at rest (no speed or cadence detected)
   - Sensor detecting speed but no cadence
   - Sensor detecting cadence but no speed
   - Sensor detecting speed and cadence

The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds. If the sensor successfully bonds to a peer device and stores the address of the device in its white list, then for the first 10 seconds of advertising the sensor only tries to connect to any device addresses stored in its white list. After 10 seconds, the sensor tries to connect to any peer device that attempts to connect. Independent of the white list, a 30-second period of slow interval advertising passes after 30 seconds of fast interval connection. The device sleeps until you press the right button before advertising again. If the device terminates connection for any other reason, the sensor advertises for 60 seconds at a slow interval and then sleeps if no connection is made. The sensor advertises only if you press the right button.

12.3.3 Neglect Timer

This device has a compile time option that lets the sensor terminate a connection if there is no input for 15 seconds. After the device has connected and notifications are disabled, the application starts a timer. This timer is restarted whenever a read or write request comes from the peer device and is disabled while notifications are enabled. If the value USING_NEGLECT_TIMEOUT is set to FALSE at compile, this timer is permanently disabled at run time.

12.4 Running Speed and Cadence (RSC) Sensor

This sample project implements the RSC profile in a *Bluetooth* low energy peripheral device to provide a sample application of sensor on a bicycle using simulated measurement data. The application implements the Sensor role of the RSC Profile. This profile also makes use of the optional Device Info Service in the same manner as the Cycling Sensor. This project is configured to run on the SmartRF06 Board.

Copyright © 2015–2016, Texas Instruments Incorporated

12.4.1 Interface

When the right button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF is connected, it cycles through different running speed and cadence sensor data formats.

Pressing the select key initiates a soft reset. This reset includes the following.

- Terminating all current connections
- Clearing all bond data
- Clearing the white list of all peer addresses

12.4.2 Operation

The following steps detail how to use the Running Speed and Cadence Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the cycling sensor from an RSC collector peer device.

NOTE: The peer device receives a slave security request and initiates a bond. When bonded, the collector discovers the RSC service and configures it to enable running speed and cadence measurements. When RSC measurement notifications have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows.
   - At rest: neither instantaneous stride length nor total distance is included in measurement
   - Stride: instantaneous stride length is included in measurement
   - Distance: total distance is included in measurement
   - All: both stride length and total distance are included in measurement

The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application advertises at the fast interval for 30 seconds. If the sensor successfully bonds to a peer device and stores the address of the device in its white list, the sensor tries only to connect to any device addresses stored in its white list for the first 10 seconds of advertising. After 10 seconds, the sensor tries to connect to any peer device trying to connect. After 30 seconds of fast interval connection, a 30-second period of slow interval advertising passes independent of white list use. The device sleeps and waits for you to press the right button before resuming advertising. If the device terminates connection for any other reason, the sensor advertises for 60 seconds at a slow interval and then sleeps if no connection is made. The device advertises again only if the right button is pressed.

12.4.3 Neglect Timer

This device has a compile time option that lets the sensor terminate a connection if there is no input for 15 seconds. After the device has connected and notifications are disabled, the application starts a timer. This timer is restarted whenever a read or write request comes from the peer device and is disabled while notifications are enabled. If the value USING_NEGLECT_TIMEOUT is set to FALSE at compile, this timer is permanently disabled at run time.

12.5 Glucose Collector

This sample project implements a glucose collector. The application is designed to connect to the glucose sensor sample application to demonstrate the operation of the Glucose Profile. The project is configured to run on the SmartRF06.

Copyright © 2015–2016, Texas Instruments Incorporated

12.5.1 Interface

The SmartRF buttons and display provide an interface for the application. The buttons are as follows.

- Up: If not connected, start or stop device discovery. If connected to a glucose sensor, request the number of records that meet configured filter criteria.
- Left: Scroll through device discovery results. If connected to a glucose sensor, send a record access abort message.
- Select: Connect or disconnect to or from the selected device.
- Right: If connected, request records that meet configured filter criteria.
- Down: If connected, clear records that meet configured filter criteria. If not connected, erase all bonds.

The LCD display displays the following information.

- BD address of the device
- Device discovery results
- Connection state
- Pairing and bonding status
- Number of records requested
- Sequence number, glucose concentration, and Hba1c value of received glucose measurement and context notifications

12.5.2 Record Access Control Point

The Glucose Profile uses a characteristic called the record access control point to perform operations on glucose measurement records stored by the glucose sensor. The following different operations can be performed.

- Retrieve stored records.
- Delete stored records.
- Abort an operation in progress.
- Report number of stored records.

The glucose collector sends control point messages to a sensor by using write requests, while the sensor sends control point messages to the glucose collector by using indications. When records are retrieved, the glucose measurement and glucose context are sent though notifications on their respective characteristics. If an expected response is not received, the operation times out after 30 seconds and the glucose collector closes the connection.

12.6 Glucose Sensor

This sample project implements the Glucose Profile in a *Bluetooth* low energy peripheral device to provide an example glucose sensor using simulated measurement data. The application implements the Sensor role of the Glucose Profile. The application is compiled to run on a SmartRF06 board.

12.6.1 Interface

When the right button is disconnected, it toggles advertising on and off. When the up button is connected, it sends a glucose measurement and glucose context.

Copyright © 2015–2016, Texas Instruments Incorporated

12.6.2 Operation

The following steps detail how to use the Glucose Sensor sample projects:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the glucose sensor from a glucose collector peer device.

NOTE: The peer device discovers the glucose service and configures it to enable notifications of the glucose measurement. the device may also enable notifications of the glucose measurement context. When glucose measurement notifications have been enabled a simulated measurement can be sent by pressing the up button. If the peer device has also enabled notifications of the glucose measurement context then this is sent following the glucose measurement. The peer device may also write commands to the record access control point to retrieve or erase stored glucose measurement records. The sensor has four hardcoded simulated records. If the peer device initiates pairing then the devices pair. Only just works pairing is supported by the application (pairing without a passcode).

12.7 HID–Emulated Keyboard

This sample project implements the HID-Over-GATT profile in a *Bluetooth* low energy peripheral device to provide an example of how a HID keyboard can be emulated with a simple four button remote control device. The project is based on adopted profile and service specifications for HID-Over-GATT and Scan Parameters. The project also includes the Device Information Service and Battery Service. This project is configured to run on the SmartRF06 board

12.7.1 Interface

When the following are connected, they send the following key presses:

- The left button sends a left arrow key.
- The right button sends a right arrow key.
- The up button sends an up arrow key.
- The down button sends a down arrow key.

A secure connection must be established before key presses is sent to the peer device.

12.7.2 Operation

The following steps detail how to use the HID-Emulated Keyboard sample project:

1. Power up the device.

NOTE: The device advertises by default.

2. Initiate a device discovery and connection procedure to discover and connect to the HID device from a HID Host peer device.

NOTE: The peer device discovers the HID service and recognizes the device as a keyboard. The peer device may also discover and configure the battery service for battery level-state notifications. By default, the HID device requires security and uses just works pairing. After a secure connection is established and the HID host configures the HID service to enable notifications, the HID device can send HID key presses to the HID host. A notification is sent when a button is pressed and when a button is released. If there is no HID activity for a period of time (20 seconds by default) the HID device disconnects. When the connection is closed, the HID device advertises again.

Copyright © 2015–2016, Texas Instruments Incorporated

12.8 HostTest–*Bluetooth* low energy Network Processor

The HostTest project implements a pure *Bluetooth* low energy network processor to use with an external microcontroller or a PC software application such as BTool. Communication occurs through the HCI interface.

12.9 KeyFob

The KeyFob application demonstrates the following:

- Report battery level
- Report 3-axis accelerometer readings
- Report proximity changes
- Report key press changes

The following GATT services are used:

- Device Information
- Link Loss
- Immediate Alert (for the Reporter role of the Proximity Profile and for the Target role of the Find Me Profile)
- Tx Power (for the Report role of the Proximity Profile)
- Battery
- Accelerometer
- SimpleKeys

The accelerometer and simple keys profiles are unaligned with official SIG profiles but are an example of the implementation of the profile service. The device information service and proximity-related services are based on adopted specifications.

12.9.1 Interface

This application uses two buttons for input and an LED and buzzer for output.

Right Button

When disconnected, this button toggles advertising on and off. When connected, this button registers a key press that can be enabled to notify a peer device or may be read by a peer device.

Left Button

When connected, this button registers a key press that can be enabled to notify a peer device or may be read by a peer device.

Buzzer

The buzzer activates if a Link Loss Alert is triggered.

12.9.2 Battery Operation

The Battery Profile allows for the USB dongle to read the percentage of battery remaining on the SmartRF by reading the value of <BATTERY_LEVEL_UUID>.

12.9.3 Accelerometer Operation

The SmartRF does not communicate with an accelerometer, so the accelerometer data is always set to 0. The accelerometer must be enabled <ACCEL_ENABLER_UUID> by writing a value of 01. When the accelerometer is enabled, each axis can be configured to send notifications by writing 01 00 to the characteristic configuration for each axis <GATT_CLIENT_CHAR_CFG_UUID>. The values can be read by reading <ACCEL_X_UUID>, <ACCEL_Y_UUID>, and <ACCEL_Z_UUID>.

Copyright © 2015–2016, Texas Instruments Incorporated

12.9.4 Keys

The simple keys service on the SmartRF lets the device send notifications of key presses and releases to a central device. The application registers with HAL to receive a callback in case HAL detects a key change. The peer device can read the value of the keys by reading <SK_KEYPRESSED_UUID>. The peer device can enable key press notifications by writing 01 to <GATT_CLIENT_CHAR_CFG_UUID>. A value of 00 indicates that neither key is pressed. A value of 01 indicates that the left key is pressed. A value of 02 indicates that the right key is pressed. A value of 03 indicates that both keys are pressed.

12.9.5 Proximity

One of the services of the Proximity Profile is the link loss service, which lets the proximity reporter begin an alert if the connection drops. The link loss alert can be set by writing a value to <PROXIMITY_ALERT_LEVEL_UUID>.

The default alert value setting is 00, which indicates no alert. To turn on the alert, write a 1-byte value of 01 (low alert) or 02 (high alert). By default, the link does not time out until 20 seconds have passed without receiving a packet. This Supervision Timeout value can be changed in the Connection Services tab. The time-out value must be set before the connection is established. After completing the write, move the SmartRF device far enough away from the USB Dongle until disconnected. Alternatively, disconnect the USB Dongle from the PC. When the time-out expires, the alarm is triggered. If a low alert is set, the SmartRF makes a low-pitched beep. If a high alert is set, the SmartRF makes a high-pitched beep. In either case, the SmartRF beeps 10 times and then stops. To stop the beeping, either connect with the SmartRF or press the left button.

12.10 SensorTag

The SimpleLink *Bluetooth* Smart CC2650 SensorTag 2.0 is a *Bluetooth* low energy peripheral slave device that runs on the CC2650 SensorTag reference hardware platform. The SimpleLink CC2650 is a multistandard wireless MCU that supports *Bluetooth* low energy and other wireless protocols. Software developed with the *Bluetooth* low energy stack is binary compatible with the CC2650. The SensorTag 2.0 includes multiple peripheral sensors with a complete software solution for sensor drivers interfaced to a GATT server running on TI *Bluetooth* low energy protocol stack. The GATT server contains a primary service for each sensor for configuration and data collection. For a description of the available sensors, see http://www.ti.com/sensortag.

12.10.1 Operation

On start-up, the SensorTag advertises with a 100-ms interval. The connection is established by a central device and the sensors can then be configured to provide measurement data. The central device could be any *Bluetooth* low energy-compliant device and the main focus is on *Bluetooth* low energy-compliant mobile phones, running either Android™ or iOS®. The central device operates as follows:

- Scans and discovers the SensorTag (the scan response contains name SensorTag)
- Establishes connection based on user-defined connection parameters
- Performs service discovery (discovers characteristics by UUID)
- Operates as a GATT client (write to and read from Characteristic Value)

The central device initiates the connection and becomes the master. To obtain the data, first activate the corresponding sensor through a Characteristic Value write to appropriate service.

The most power-efficient way to obtain measurements for a sensor is as follows:

1. Enable notifications.
2. Enable the sensor.
3. Disable the sensor (with notification on) when notifications with data are obtained on the master side.

Alternatively, to not use notifications at all do as follows:

1. Enable the sensor.
2. Read the data and verify.
3. Disable the sensor.

Copyright © 2015–2016, Texas Instruments Incorporated

For the alternative, the sensor takes a varying amount of time to measure data. Depending on the connection interval (approximately 10 to 4000 ms) set by the central device, the time to measure data varies. The individual sensors require varying delays to complete measurements. TI recommends a setting of 100 ms. For fast accelerometer and magnetometer data updates, a lower setting is required. You can stop notifications and turn the sensors on or off.

12.10.2 Sensors

The following sensors support SensorTag:

- IR Temperature, both object and ambient temperature
- Accelerometer, 3-axis
- Humidity, both relative humidity and temperature
- Magnetometer, 3-axis
- Barometer, both pressure and temperature
- Gyroscope, 3-axis

12.11 SimpleBLECentral

The SimpleBLECentral project implements a simple *Bluetooth* low energy central device with GATT client functionality. This project uses the SmartRF05 + CC2650EM hardware platform. This project can be used as a framework for developing many central-role applications. This project is configured to run on the SmartRF06 board. By default, the SimipleBLECentral application is configured to filter and connect to peripheral devices with the TI Simple Profile UUID. To modify this behavior, set DEFAULT_DEV_DISC_BY_SVC_UUID to FALSE in SimpleBLECentral.c.

12.11.1 Interface

The SmartRF buttons and display provide an interface for the application. The buttons are as follows:

- Up: If disconnected, start or stop device discovery. If connected to a SimpleBLEPeripheral, alternate sample read and write requests.
- Left: Scroll through device discovery results.
- Select: Connect or disconnect to or from the selected device.
- Right: If connected, send a parameter update request.
- Down: If connected, start or cancel RSSI polling.

The LCD display displays the following information:

- BD address of the device
- Device discovery results
- Connection state
- Pairing and bonding status
- Attribute read or write value after parameter update

12.12 SimpleBLEPeripheral

The SimpleBLEPeripheral project implements a simple *Bluetooth* low energy peripheral device with GATT services and demonstrates the TI Simple Profile. This project can be a framework for developing many different peripheral-role applications. The *Software Developer's Guide* explains this project.

12.13 SimpleAP

The SimpleAP project demonstrates the TI Simple Profile running on a CC2640 configured as an application processor (AP) interfacing to a CC2640, through SPI or UART, running the SimpleNP network processor application. The project provides project build configurations for the SimpleAP running on a SensorTag and SmartRF06 + CC2650EM evaluation module. With SimpleAP project configuration, processing of GATT profile and service data is handled on the SimpleAP, while the GATT database and *Bluetooth* low energy stack reside on the SimpleNP network processor.

Copyright © 2015–2016, Texas Instruments Incorporated

12.14 SimpleNP

The SimpleNP project implements the TI Simple Network Processor *Bluetooth* low energy device configuration. In this configuration, the CC2640 operates as a *Bluetooth* low energy network processor with the application and profiles executing off-chip on a host MCU. The SimpleNP network processor is for designs that require adding a *Bluetooth* low energy capability to an existing embedded system with a host MCU or application processor. The Simple Network Processor API Guide has further details on how to interface to the SimpleNP, including the API interface.

12.15 TimeApp

This sample project implements time and alert-related profiles in a *Bluetooth* low energy peripheral device to provide an example of how *Bluetooth* low energy profiles are used in a product like a watch. The project is based on adopted profile specifications for Time, Alert Notification, and Phone Alert Status. All profiles are implemented in the client role. The following Network Availability Profile, Network Monitor role has been implemented, based on Network Availability Draft Specification d05r04 (UCRDD). This project has been configured for the SmartRF06 board.

12.15.1 Interface

The interface for the application consists of the SmartRF06 buttons and display. The buttons are used as follows:

- Up: Starts or stops advertising.
- Left: If connected, sends a command to the Alert Notification control point.
- Center: If connected, disconnects. If held down on power-up, erases all bonds.
- Right: If connected, initiates a Reference Time update.
- Down: If connected, initiates a Ringer Control Point update.

The LCD display shows the following information:

- BD address of the device
- Connection state
- Pairing and bonding status
- Passcode display
- Time and date
- Network availability
- Battery state of peer device
- Alert notification messages
- Unread message alerts
- Ringer status

12.15.2 Operation

The following steps detail how to use the Time App sample project.

1. Power up the application.

   NOTE: When the application powers up it displays Time App, the BD address of the device, and a default time and date of 00:00 Jan01 2000.

2. Press Up to start advertising.
3. Connect from a peer device.

Copyright © 2015–2016, Texas Instruments Incorporated

NOTE: The connection status displays. When the application tries to discover the following services on the peer device:

- Current Time Service
- DST Change Service
- Reference Time Service
- Alert Notification Service
- Phone Alert Status Service
- Network Availability Service
- Battery Service

The discovery procedure caches handles of interest. When bonded to a peer device, the handles are saved to avoid performing the procedure again. If a service is discovered certain service characteristics are read and displayed. The network availability status and battery level is displayed and the current time updates. The application also enables notification or indication for characteristics that support these operations. The enabling of notification or indication allows the peer device to send notifications or indications updating the time, network availability, or battery status. The peer device can also send alert notification messages and unread message alerts. These updates and messages is displayed on the LCD. The peer device may initiate pairing. If a passcode is required, the application generates and displays a random passcode.

Enter this passcode on the peer device to proceed with pairing. The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason the application starts advertising at the slow interval. The advertising intervals and durations are configurable in the timeapp.c file.

12.16 Thermometer

This sample project implements a Health Thermometer and Device Information Profile in a *]* low energy peripheral device to provide an example health thermometer application using simulated measurement data. The application implements the Sensor role of the Health Thermometer Profile. The project is based on the adopted profile and service specifications for Health Thermometer. The project also includes the Device Information Service. This project has been configured to run on the SmartRF06 board.

12.16.1 Interface

This application has two buttons inputs.

Right Button

When the button is disconnected and unconfigured to take measurements, it toggles advertising on and off. When connected or configured to take measurements, pressing this button increases the temperature by 1°C. After a 3°C increase in temperature, the interval is set to 30 seconds. This thermometer application also sends an indication to the peer with this interval change if configured.

Up Button

This button cycles through different measurement formats.

Copyright © 2015–2016, Texas Instruments Incorporated

12.16.2 Operation

The following steps detail how to use the Thermometer sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the thermometer sensor from a thermometer collector peer device.

NOTE: The peer device discovers the thermometer service and configures it to enable indication or notifications of the thermometer measurement. The peer device may also discover the device information service for more information such as manufacturing and serial number. When thermometer measurements have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
   - CELSIUS | TIMESTAMP | TYPE
   - CELSIUS | TIMESTAMP
   - CELSIUS
   - FAHRENHEIT
   - FAHRENHEIT | TIMESTAMP
   - FAHRENHEIT | TIMESTAMP | TYPE

If the peer device initiates pairing, the HT requests a passcode. The passcode is 000000.

The thermometer operates in the following states:

- Idle – In this state, the thermometer does nothing until you press the button on the right to start advertising.
- Idle Configured – The thermometer waits the interval before taking a measurement and proceeding to Idle Measurement Ready state.
- Idle Measurement Ready – The thermometer has a measurement ready and advertises to allow connection. The thermometer periodically advertises in this state.
- Connected Not Configured – The thermometer may be configured to enable measurement reports. The thermometer does not send stored measurements until the CCC is enabled. When the thermometer is connected, it sets a timer to disconnect in 20 seconds.
- Connected Configured – The thermometer does send any stored measurements if CCC is set to send measurement indications.
- Connected Bonded – The thermometer sends any stored measurements if CCC was previously set to send measurement indications.

The peer device may also query the thermometers read only device information. Examples are model number, serial number, and so forth.

Copyright © 2015–2016, Texas Instruments Incorporated

GAP API

A.1 Commands

This section details the GAP commands from gap.h that the application uses. All other GAP commands are abstracted through the GAPRole or the GAPBondMgr. The return values described in this section are the return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

uint16 GAP_GetParamValue (gapParamIDs_t paramID)

Get a GAP parameter.

Parameters parameter ID (Section A.2)

Returns GAP Parameter Value if successful

0xFFFF if paramID invalid

bStatus_t GAP_SetParamValue (gapParamIDs_t paramID, uint16 paramValue)

Set a GAP parameter.

Parameters paramID – parameter ID (Section A.2)

paramValue – new param value

Returns SUCCESS (0x00)

INVALIDPARAMETER (0x02): paramID is invalid

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t bStatus_t GAP_ConfigDeviceAddr(uint8 addrType, uint8 *pStaticAddr);

Setup the address type of the device. If ADDRTYPE_PRIVATE_RESOLVE is specified, the stack changes the address periodically.

Parameters

addrType – address Type

- ADDRTYPE_PUBLIC
- ADDRTYPE_STATIC
- ADDRTYPE_PRIVATE_NONRESOLVE
- ADDRTYPE_RESOLVE

pStaticAddr – address to use for static or private nonresolvable address types

Returns

SUCCESS (0x00): address type updated

bleNotReady (0x10): this command must be called after GAP_DeviceInit() is called and the initialization process is complete

bleIncorrectMode (0x12): this cannot be done during a connection

INVALIDPARAMETER (0x02): invalid address type passed into function

void GAP_RegisterForMsgs(uint8 taskID);

Register a given task ID to receive extra (unprocessed) HCI status and complete events, and other Host events.

Parameters

taskID – task ID to send events to

A.2 Configurable Parameters

ParamID	Description
TGAP_GEN_DISC_ADV_MIN	Time (ms) to remain advertising in general discovery mode. Setting this to 0 turns off this timeout, advertising infinitely. Default is 0 (continue indefinitely)
TGAP_LIM_ADV_TIMEOUT	Time (sec) to remain advertising in limited discovery mode. Default is 180 s.
TGAP_GEN_DISC_SCAN	Time (ms) to perform scanning for general discovery.
TGAP_LIM_DISC_SCAN	Time (ms) to perform scanning for limited discovery.
TGAP_CONN_EST_ADV_TIMEOUT	Advertising timeout (ms) when performing connection establishment.
TGAP_CONN_PARAM_TIMEOUT	Timeout (ms) for link layer to wait to receive connection parameter update response.
TGAP_LIM_DISC_ADV_INT_MIN	Minimum advertising interval in limited discovery mode (n × 0.625 ms)
TGAP_LIM_DISC_ADV_INT_MAX	Maximum advertising interval in limited discovery mode (n × 0.625 ms)
TGAP_GEN_DISC_ADV_INT_MIN	Minimum advertising interval in general discovery mode (n × 0.625 ms)
TGAP_GEN_DISC_ADV_INT_MAX	Maximum advertising interval in general discovery mode (n × 0.625 ms)
TGAP_CONN_ADV_INT_MIN	Minimum advertising interval when in connectable mode (n × 0.625 ms)
TGAP_CONN_ADV_INT_MAX	Maximum advertising interval when in connectable mode (n × 0.625 ms)
TGAP_CONN_SCAN_INT	Scan interval used during Link Layer Initiating state, when in connectable mode (n × 0.625 ms)
TGAP_CONN_SCAN_WIND	Scan window used during Link Layer Initiating state, when in connectable mode (n × 0.625 ms)
TGAP_CONN_HIGH_SCAN_INT	Scan interval used during Link Layer Initiating state, when in connectable mode, high duty scan cycle scan parameters (n × 0.625 ms)
TGAP_CONN_HIGH_SCAN_WIND	Scan window used during Link Layer Initiating state, when in connectable mode, high duty scan cycle scan parameters (n × 0.625 ms)
TGAP_GEN_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in general discovery proc (n × 0.625 ms).
TGAP_GEN_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in general discovery proc (n × 0.625 ms)

Copyright © 2015–2016, Texas Instruments Incorporated

ParamID	Description
TGAP_LIM_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in limited discovery proc (n × 0.625 ms)
TGAP_LIM_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in limited Discovery proc (n × 0.625 ms)
TGAP_CONN_EST_INT_MIN	Minimum Link Layer connection interval, when using connection establishment proc (n × 1.25 ms)
TGAP_CONN_EST_INT_MAX	Maximum Link Layer connection interval, when using connection establishment proc (n × 1.25 ms)
TGAP_CONN_EST_SCAN_INT	Scan interval used during Link Layer Initiating state, when using connection establishment proc (n × 0.625 ms)
TGAP_CONN_EST_SCAN_WIND	Scan window used during Link Layer Initiating state, when using connection establishment proc (n × 0.625 ms)
TGAP_CONN_EST_SUPERV_TIMEOUT	Link Layer connection supervision timeout, when using connection establishment proc (n × 10 ms)
TGAP_CONN_EST_LATENCY	Link Layer connection slave latency, when using connection establishment proc (in number of connection events)
TGAP_CONN_EST_MIN_CE_LEN	Local informational parameter about minimum length of connection required, when using connection establishment proc (n × 0.625 ms)
TGAP_CONN_EST_MAX_CE_LEN	Local informational parameter about maximum length of connection required, when using connection establishment proc (n × 0.625 ms).
TGAP_PRIVATE_ADDR_INT	Minimum Time Interval between private (resolvable) address changes. In minutes (default 15 min)
TGAP_CONN_PAUSE_CENTRAL	Central idle timer. In seconds (default 1 s)
TGAP_CONN_PAUSE_PERIPHERAL	Minimum time upon connection establishment before the peripheral starts a connection update procedure. In seconds (default 5 seconds)
TGAP_SM_TIMEOUT	Time (ms) to wait for security manager response before returning bleTimeout. Default is 30 s.
TGAP_SM_MIN_KEY_LEN	SM Minimum Key Length supported. Default 7.
TGAP_SM_MAX_KEY_LEN	SM Maximum Key Length supported. Default 16.
TGAP_FILTER_ADV_REPORTS	TRUE to filter duplicate advertising reports. Default TRUE.
TGAP_SCAN_RSP_RSSI_MIN	Minimum RSSI required for scan responses to be reported to the app. Default –127.
TGAP_REJECT_CONN_PARAMS	Whether or not to reject Connection Parameter Update Request received on Central device. Default FALSE.

Copyright © 2015–2016, Texas Instruments Incorporated

A.3 Events

This section details the events relating to the GAP layer that can be returned to the application from the *Bluetooth* low energy stack. Some of these events are passed directly to the application and some are handled by the GAPRole or GAPBondMgr layers. The events are passed as a GAP_MSG_EVENT with header:

```
typedef struct
{
  osal_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
  uint8 opcode;                    //!< GAP type of command. Ref: @ref
GAP_MSG_EVENT_DEFINES
} gapEventHdr_t;
```

The following is a list of the possible hdr and the associated events. See gap.h for all other definitions used in these events.

- GAP_DEVICE_INIT_DONE_EVENT: Sent when the Device Initialization is complete

```
typedef struct
{
  osal_event_hdr_t  hdr;               //!< GAP_MSG_EVENT and status
  uint8 opcode;                        //!< GAP_DEVICE_INIT_DONE_EVENT
  uint8 devAddr[B_ADDR_LEN];           //!< Device's BD_ADDR
  uint16 dataPktLen;                   //!< HC_LE_Data_Packet_Length
  uint8 numDataPkts;                   //!< HC_Total_Num_LE_Data_Packets
} gapDeviceInitDoneEvent_t;
```

- GAP_DEVICE_DISCOVERY_EVENT: Sent when the Device Discovery Process is complete

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_DEVICE_DISCOVERY_EVENT
  uint8 numDevs;         //!< Number of devices found during scan
  gapDevRec_t *pDevList; //!< array of device records
} gapDevDiscEvent_t;
```

- GAP_ADV_DATA_UPDATE_DONE_EVENT: Sent when the Advertising Data or SCAN_RSP Data has been updated

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_ADV_DATA_UPDATE_DONE_EVENT
  uint8 adType;          //!< TRUE if advertising data, FALSE if SCAN_RSP
} gapAdvDataUpdateEvent_t;
```

- GAP_MAKE_DISCOVERABLE_DONE_EVENT: Sent when the Make Discoverable Request is complete

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_MAKE_DISCOVERABLE_DONE_EVENT
  uint16 interval;       //!< actual advertising interval selected by controller
} gapMakeDiscoverableRspEvent_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

- GAP_END_DISCOVERABLE_DONE_EVENT: Sent when the Advertising has ended

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_END_DISCOVERABLE_DONE_EVENT
} gapEndDiscoverableRspEvent_t;
```

- GAP_LINK_ESTABLISHED_EVENT: Sent when the Establish Link Request is complete

```
typedef struct
{
  osal_event_hdr_t  hdr;     //!< GAP_MSG_EVENT and status
  uint8 opcode;              //!< GAP_LINK_ESTABLISHED_EVENT
  uint8 devAddrType;         //!< Device address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 devAddr[B_ADDR_LEN]; //!< Device address of link
  uint16 connectionHandle;   //!< Connection Handle from controller used to ref the
device
  uint16 connInterval;       //!< Connection Interval
  uint16 connLatency;        //!< Conenction Latency
  uint16 connTimeout;        //!< Connection Timeout
  uint8 clockAccuracy;       //!< Clock Accuracy
} gapEstLinkReqEvent_t;
```

- GAP_LINK_TERMINATED_EVENT: Sent when a connection was terminated

```
typedef struct
{
  osal_event_hdr_t  hdr;   //!< GAP_MSG_EVENT and status
  uint8 opcode;            //!< GAP_LINK_TERMINATED_EVENT
  uint16 connectionHandle; //!< connection Handle
  uint8 reason;            //!< termination reason from LL
} gapTerminateLinkEvent_t;
```

- GAP_LINK_PARAM_UPDATE_EVENT: Sent when an Update Parameters Event is received

```
typedef struct
{
  osal_event_hdr_t hdr;     //!< GAP_MSG_EVENT and status
  uint8 opcode;             //!< GAP_LINK_PARAM_UPDATE_EVENT
  uint8 status;             //!< bStatus_t
  uint16 connectionHandle;  //!< Connection handle of the update
  uint16 connInterval;      //!< Requested connection interval
  uint16 connLatency;       //!< Requested connection latency
  uint16 connTimeout;       //!< Requested connection timeout
} gapLinkUpdateEvent_t;
```

- GAP_RANDOM_ADDR_CHANGED_EVENT: Sent when a random address is changed

```
typedef struct
{
  osal_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
  uint8 opcode;                       //!< GAP_RANDOM_ADDR_CHANGED_EVENT
  uint8 addrType;                     //!< Address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 newRandomAddr[B_ADDR_LEN];    //!< the new calculated private addr
} gapRandomAddrEvent_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

- GAP_SIGNATURE_UPDATED_EVENT: Sent when the device's signature counter is updated

```
typedef struct
{
  osal_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
  uint8 opcode;                       //!< GAP_SIGNATURE_UPDATED_EVENT
  uint8 addrType;                     //!< Device's address type for devAddr
  uint8 devAddr[B_ADDR_LEN];          //!< Device's BD_ADDR, could be own address
  uint32 signCounter;                 //!< new Signed Counter
} gapSignUpdateEvent_t;
```

- GAP_AUTHENTICATION_COMPLETE_EVENT: Sent when the Authentication (pairing) process is complete

```
typedef struct
{
  osal_event_hdr_t  hdr;            //!< GAP_MSG_EVENT and status
  uint8 opcode;                     //!< GAP_AUTHENTICATION_COMPLETE_EVENT
  uint16 connectionHandle;          //!< Connection Handle from controller used to ref
the device
  uint8 authState;                  //!< TRUE if the pairing was authenticated (MITM)
  smSecurityInfo_t *pSecurityInfo; //!< BOUND - security information from this device
  smSigningInfo_t *pSigningInfo;   //!< Signing information
  smSecurityInfo_t *pDevSecInfo;   //!< BOUND - security information from connected
device
  smIdentityInfo_t *pIdentityInfo; //!< BOUND - identity information
} gapAuthCompleteEvent_t;
```

- GAP_PASSKEY_NEEDED_EVENT: Sent when a Passkey is required (part of the pairing process)

```
typedef struct
{
  osal_event_hdr_t  hdr;         //!< GAP_MSG_EVENT and status
  uint8 opcode;                  //!< GAP_PASSKEY_NEEDED_EVENT
  uint8 deviceAddr[B_ADDR_LEN]; //!< address of device to pair with, and could be
either public or random.
  uint16 connectionHandle;       //!< Connection handle
  uint8 uiInputs;                //!< Pairing User Interface Inputs - Ask user to input
passcode
  uint8 uiOutputs;               //!< Pairing User Interface Outputs - Display passcode
} gapPasskeyNeededEvent_t;
```

- GAP_SLAVE_REQUESTED_SECURITY_EVENT: Sent when a Slave Security Request is received

```
typedef struct
{
  osal_event_hdr_t  hdr;         //!< GAP_MSG_EVENT and status
  uint8 opcode;                  //!< GAP_SLAVE_REQUESTED_SECURITY_EVENT
  uint16 connectionHandle;       //!< Connection Handle
  uint8 deviceAddr[B_ADDR_LEN]; //!< address of device requesting security
  uint8 authReq;                 //!< Authentication Requirements: Bit 2: MITM, Bits 0-
1: bonding (0 - no bonding, 1 - bonding)

} gapSlaveSecurityReqEvent_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

- GAP_DEVICE_INFO_EVENT: Sent during the Device Discovery Process when a device is discovered

```
typedef struct
{
  osal_event_hdr_t  hdr;    //!< GAP_MSG_EVENT and status
  uint8 opcode;             //!< GAP_DEVICE_INFO_EVENT
  uint8 eventType;          //!< Advertisement Type: @ref
GAP_ADVERTISEMENT_REPORT_TYPE_DEFINES
  uint8 addrType;           //!< address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 addr[B_ADDR_LEN];   //!< Address of the advertisement or SCAN_RSP
  int8 rssi;                //!< Advertisement or SCAN_RSP RSSI
  uint8 dataLen;            //!< Length (in bytes) of the data field (evtData)
  uint8 *pEvtData;          //!< Data field of advertisement or SCAN_RSP
} gapDeviceInfoEvent_t;
```

- GAP_BOND_COMPLETE_EVENT: Sent when the bonding (bound) process is complete

```
typedef struct
{
  osal_event_hdr_t  hdr;   //!< GAP_MSG_EVENT and status
  uint8 opcode;            //!< GAP_BOND_COMPLETE_EVENT
  uint16 connectionHandle; //!< connection Handle
} gapBondCompleteEvent_t;
```

- GAP_PAIRING_REQ_EVENT: Sent when an unexpected Pairing Request is received

```
typedef struct
{
  osal_event_hdr_t hdr;    //!< GAP_MSG_EVENT and status
  uint8 opcode;            //!< GAP_PAIRING_REQ_EVENT
  uint16 connectionHandle; //!< connection Handle
  gapPairingReq_t pairReq; //!< The Pairing Request fields received.
} gapPairingReqEvent_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

GAPRole Peripheral Role API

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

B.1 Commands

bStatus_t GAPRole_SetParameter(uint16_t param, uint8_t len, void *pValue)

Set a GAP Role parameter.

Parameters: param – Profile parameter ID (see Section B.2)

len – length of data to write

pValue – pointer to value to set parameter. This pointer is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)

INVALIDPARAMETER (0x02): param was invalid

bleInvalidRange (0x18): len is invalid for the given param

blePending (0x16): previous param update has not been completed

bleIncorrectMode (0x12): can not start connectable advertising because nonconnectable advertising is enabled

bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID (Section B.2)

pValue – pointer to location to get parameter. This pointer is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)

INVALIDPARAMETER (0x02): param was in valid

bStatus_t GAPRole_StartDevice(gapRolesCBs_t *pAppCallbacks)

Initializes the device as a peripheral and configures the application callback function.

Parameters pAppCallbacks – pointer to application callbacks (Section B.3)

Returns SUCCESS (0x00)

bleAlreadyInRequestedMode (0x11): device was already initialized

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GAPRole_TerminateConnection(void)

Terminates an existing connection.

Returns

SUCCESS (0x00): connection termination process has started

bleIncorrectMode (0x12): there is no active connection

bleInvalidTaskID (0x03): application did not register correctly with ICall

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A): disconnect is already in process

bStatus_t GAPRole_SendUpdateParam(uint16_t minConnInterval, uint16_t maxConnInterval, uint16_t latency, uint16_t connTimeout, uint8_t handleFailure)

Update the parameters of an existing connection. See Section 5.1 for more details.

Parameters

connInterval – the new connection interval

latency – the new slave latency

connTimeout – the new timeout value

handle failure– what to do if the update does not occur. Available actions:

- GAPROLE_NO_ACTION 0 // Take no action upon unsuccessful parameter updates
- GAPROLE_RESEND_PARAM_UPDATE 1 // Continue to resend request until successful update
- GAPROLE_TERMINATE_LINK 2 // Terminate link upon unsuccessful parameter updates

Returns

SUCCESS (0x00): parameter update process has started

bleNotConnected (0x14): there is no connection so can not update parameters

B.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPROLE_PROFILEROLE	R	uint8	GAP profile role (peripheral)
GAPROLE_IRK	R/W	uint8[16]	Identity resolving key. Default is all 0, which means the IRK will be randomly generated.
GAPROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0, which means the SRK will be randomly generated.
GAPROLE_SIGNCOUNTER	R/W	uint32	Sign counter.
GAPROLE_BD_ADDR	R	uint8[6]	Device address read from controller. This can be set with the HCI_EXT_SetBDADDRCmd().
GAPROLE_ADVERT_ENABLED	R/W	uint8	Enable or disable advertising. Default is TRUE = enabled.
GAPROLE_ADVERT_OFF_TIME	R/W	uint16	How long to remain off after advertising stops before starting again. Default is 30 s. If set to 0, advertising will not start again.
GAPROLE_ADVERT_DATA	R/W	<uint8[32]	Advertisement data. Default is 02:01:01. This third byte sets limited / general advertising.
GAPROLE_SCAN_RSP_DATA	R/W	<uint8[32]	Scan Response data. Default is all 0s.
GAPROLE_ADV_EVENT_TYPE	R/W	uint8	Advertisement type. Default is GAP_ADTYPE_IND (from gap.h)
GAPROLE_ADV_DIRECT_TYPE	R/W	uint8	Direct advertisement type. Default is ADDRTYPE_PUBLIC (from gap.h)
GAPROLE_ADV_DIRECT_ADDR	R/W	uint8[6]	Direct advertisement address. Default is 0.

Copyright © 2015–2016, Texas Instruments Incorporated

ParamID	R/W	Size	Description
GAPROLE_ADV_CHANNEL_MAP	R/W	uint8	Which channels to advertise on. Default is GAP_ADVCHAN_ALL (from gap.h)
GAPROLE_ADV_FILTER_POLICY	R/W	uint8	Policy for filtering advertisements. Ignored in direct advertising
GAPROLE_CONNHANDLE	R	uint16	Handle of current connection.
GAPROLE_PARAM_UPDATE_ENABLE	R/W	uint8	TRUE to request a connection parameter update upon connection. Default = FALSE.
GAPROLE_MIN_CONN_INTERVAL	R/W	uint16	Minimum connection interval to allow (n × 125 ms). Range: 7.5 ms to 4 s. Default is 7.5 ms. Also used for param update.
GAPROLE_MAX_CONN_INTERVAL	R/W	uint16	Maximum connection interval to allow (n × 125 ms). Range: 7.5 ms to 4 s. Default is 7.5 ms. Also used for param update.
GAPROLE_SLAVE_LATENCY	R/W	uint16	Slave latency to use for a param update. Range: 0 – 499. Default is 0.
GAPROLE_TIMEOUT_MULTIPLIER	R/W	uint16	Supervision time-out to use for a param update (n × 10 ms). Range: 100 ms to 32 s. Default is 1000 ms.
GAPROLE_CONN_BD_ADDR	R	uint8[6]	Address of connected device.
GAPROLE_CONN_INTERVAL	R	uint16	Current connection interval.
GAPROLE_CONN_LATENCY	R	uint16	Current slave latency.
GAPROLE_CONN_TIMEOUT	R	uint16	Current supervision time-out.
GAPROLE_PARAM_UPDATE_REQ	W	uint8	Set this to true to send a param update request.
GAPROLE_STATE	R	uint8	Gap peripheral role state (enumerated in gaprole_States_t in peripheral.h).

B.3 Callbacks

Callback are functions whose pointers are passed from the application to the GAPRole so that the GAPRole can return events to the application. They are passed as the following:

```
typedef struct
{
  gapRolesStateNotify_t     pfnStateChange;  //!< Whenever the device changes state
} gapRolesCBs_t;
```

See the SimpleBLEPeripheral application for an example.

B.3.1 State Change Callback (pfnStateChange)

This callback passes the current GAPRole state to the application whenever the state changes. This function is of the following type:

```
typedef void (*gapRolesStateNotify_t)(gaprole_States_t newState);
```

The various GAPRole states (newState) are as follows:

- GAPROLE_INIT //!< Waiting to be started
- GAPROLE_STARTED //!< Started but not advertising
- GAPROLE_ADVERTISING //!< Currently advertising
- GAPROLE_ADVERTISING_NONCONN //!< Currently using non-connectable advertising
- GAPROLE_WAITING //!< Device is started and in a waiting period before advertising again
- GAPROLE_WAITING_AFTER_TIMEOUT //!< Device timed out from a connection but is not yet advertising, the device is in waiting period before advertising again.
- GAPROLE_CONNECTED //!< In a connection
- GAPROLE_CONNECTED_ADV //!< In a connection + advertising
- GAPROLE_ERROR //!< Error occurred – invalid state

Copyright © 2015–2016, Texas Instruments Incorporated

GAPRole Central Role API

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command never gets processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

C.1 Commands

bStatus_t GAPCentralRole_StartDevice(gapCentralRoleCB_t *pAppCallbacks)

Start the device in Central role. This function is typically called once during system startup.

Parameters pAppCallbacks – pointer to application callbacks

Returns SUCCESS (0x00)

bleAlreadyInRequestedMode (0x11): Device already started

bStatus_t GAPCentralRole_SetParameter(uint16_t param, uint8_t len, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID (Section C.2)

len – length of data to write

pValue – pointer to value to set parameter. This is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)

INVALIDPARAMETER (0x02): param was not valid

bleInvalidRange (0x18): len is invalid for the given param

bStatus_t GAPCentralRole_GetParameter (uint16_t param, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID (Section C.2)

pValue – pointer to buffer to contain the read data

Returns SUCCESS (0x00)

INVALIDPARAMETER (0x02): param was not valid

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GAPCentralRole_TerminateLink (uint16_t connHandle);

Terminates an existing connection.

Parameters

connHandle – connection handle of link to terminate or...

0xFFFE – cancel the current link establishment request or...

0xFFFF – terminate all links

Returns

SUCCESS (0x00) – termination has started

bleIncorrectMode (0x12) – there is no active connection

bleInvalidTaskID (0x03) – application did not register correctly with ICall

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A) – terminate procedure already started

bStatus_t GAPCentralRole_EstablishLink(uint8_t highDutyCycle, uint8_t whiteList, uint8_t addrTypePeer, uint8_t *peerAddr)

Establish a link to a peer device.

Parameters

highDutyCycle – TRUE to high duty cycle scan, FALSE if not

whiteList – determines use of the white list

addrTypePeer – address type of the peer device

peerAddr – peer device address

Returns

SUCCESS (0x00): link establishment has started

bleIncorrectMode (0x12): invalid profile role

bleNotReady (0x10): a scan is in progress

bleAlreadyInRequestedMode (0x11): unable to process at this time

bleNoResources (0x15): too many links

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GAPCentralRole_UpdateLink(uint16_t connHandle, uint16_t connIntervalMin, uint16_t connIntervalMax, uint16_t connLatency, uint16_t connTimeout)

Update the link connection parameters.

Parameters

connHandle – connection handle

connIntervalMin – minimum connection interval in 1.25 ms

connIntervalMax – maximum connection interval in 1.25 ms

connLatency – number of LL latency connection events

connTimeout – connection timeout in 10 ms

Returns

SUCCESS (0x00): parameter update has started

bleNotConnected (0x14): no connection to update

INVALIDPARAMETER: connection parameters are invalid

LL_STATUS_ERROR_ILLEGAL_PARAM_COMBINATION (0x12): connection parameters do not meet *Bluetooth* low energy specification requirements: STO > (1 + Slave Latency) × (Connection Interval × 2)

LL_STATUS_ERROR_INACTIVE_CONNECTION (0x02): connHandle is inactive

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A): there is already a param update in process

LL_STATUS_ERROR_UNACCEPTABLE_CONN_INTERVAL (0x3B): connection interval does not work because it is not a multiple or divisor of intervals of the other simultaneous connection or the interval of the connection is not less than the allowed maximum connection interval as determined by the maximum number of connections times the number of slots per connection

bStatus_t GAPCentralRole_StartDiscovery(uint8_t mode, uint8_t activeScan, uint8_t whiteList)

Start a device discovery scan.

Parameters

mode – discovery mode

activeScan – TRUE to perform active scan

whiteList – TRUE to only scan for devices in the white list

Returns

SUCCESS (0x00): device discovery has started

bleAlreadyInRequestedMode (0x11): Device discovery already started

bleMemAllocError (0x13): not enough memory to allocate device discovery structure

LL_STATUS_ERROR_BAD_PARAMETER (0x12): bad parameter

bStatus_t GAPCentralRole_CancelDiscovery(void)

Cancel a device discovery scan.

Parameters

None

Returns

SUCCESS (0x00): cancelling of device discovery has started

bleInvalidTaskID (0x03): Application has not registered correctly with ICall or this is not the same task that started the discovery.

bleIncorrectMode (0x12): Not in discovery mode

Copyright © 2015–2016, Texas Instruments Incorporated

C.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPCENTRALROLE_IRK	R/W	uint8[16]	Identity resolving key. Default is all 0, which means the IRK is randomly generated.
GAPCENTRALROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0, which means the SRK is randomly generated.
GAPCENTRALROLE_SIGNCOUNTER	R/W	uint32	Sign counter.
GAPCENTRALROLE_BD_ADDR	R	uint8[6]	Device address read from controller. This can be set with the HCI_EXT_SetBDADDRCmd().
GAPCENTRALROLE_MAX_SCAN_RES	R/W	uint8	Maximum number of discover scan results to receive. Default is 8, 0 is unlimited.

C.3 Callbacks

Callbacks are functions whose pointers are passed from the application to the GAPRole so that the GAPRole can return events to the application. They are passed as follows.

```
typedef struct
{
  pfnGapCentralRoleEventCB_t  eventCB;  //!< Event callback.
} gapCentralRoleCB_t;
```

See the SimpleBLECentral application for an example.

Copyright © 2015–2016, Texas Instruments Incorporated

C.3.1 Central Event Callback (eventCB)

This callback passes GAP state change events to the application. This callback is of the following type.

```
typedef uint8_t (*pfnGapCentralRoleEventCB_t)
(
  gapCentralRoleEvent_t *pEvent         //!< Pointer to event structure.
);
```

NOTE: TRUE must be returned from this function if the GAPRole is to deallocate the event message. FALSE must be returned if the deallocation is done by the application.

Consider the following SimpleBLECentral example.

```
static uint8_t SimpleBLECentral_eventCB(gapCentralRoleEvent_t *pEvent)
{
  // Forward the role event to the application
  if (SimpleBLECentral_enqueueMsg(SBC_STATE_CHANGE_EVT,
                                  SUCCESS, (uint8_t *)pEvent))
  {
    // App will process and free the event
    return FALSE;
  }

  // Caller should free the event
  return TRUE;
}
```

If the message is successfully queued to the application, FALSE is returned because the application deallocates it later.

```
    case SBC_STATE_CHANGE_EVT:
      SimpleBLECentral_processStackMsg((ICall_Hdr *)pMsg->pData);

      // Free the stack message
      ICall_freeMsg(pMsg->pData);
      break;
```

If the message is not successfully queued to the application, TRUE is returned so that the GAPRole can deallocate the message. If the heap has enough room, the message must always be successfully enqueued. The possible GAPRole central states are listed in this section. See Section A.3 for more information on these events.

- GAP_DEVICE_INIT_DONE_EVENT
- GAP_DEVICE_DISCOVERY_EVENT
- GAP_LINK_ESTABLISHED_EVENT
- GAP_LINK_TERMINATED_EVENT
- GAP_LINK_PARAM_UPDATE_EVENT
- GAP_DEVICE_INFO_EVENT

Copyright © 2015–2016, Texas Instruments Incorporated

GATT and ATT API

This section describes the API of the GATT and ATT layers. The two sections are combined because the general procedure is to send GATT commands and receive ATT events as described in Section 5.3.3.1. The return values for the commands referenced in this section are described in Section D.4.

The possible return values are similar for all of these commands so they are described in Section D.4. The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command never gets processed by the Bluetooth low energy stack. In this case, one of the ICall return values from Appendix I is returned.

D.1 General Commands

void GATT_RegisterForMsgs(uint8 taskID);

Register a task ID to receive GATT local events and ATT response messages pending for transmission.

Parameters taskID – task ID to send events to

D.2 Server Commands

bStatus_t GATT_Indication(uint16 connHandle, attHandleValueInd_t *pInd, uint8 authenticated, uint8 taskId);

Indicates a characteristic value to a client and expect an acknowledgment.

Parameters connHandle: connection to use

pInd: pointer to indication to be sent

authenticated: whether an authenticated link is required

taskId: task to be notified of acknowledgment

NOTE: The payload must be dynamically allocated as described in Section 5.3.5.

Corresponding Events If the return status is SUCCESS, the calling application task receives a GATT_MSG_EVENT message with type ATT_HANDLE_VALUE_CFM upon an acknowledgment. Only at this point, this subprocedure is complete.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_Notification(uint16 connHandle, attHandleValueNoti_t *pNoti, uint8 authenticated)

Indicates a characteristic value to a client and expect an acknowledgment.

Parameters connHandle: connection to use

pNoti: pointer to notification to be sent

authenticated: whether an authenticated link is required

NOTE: The payload must be dynamically allocated as described in Section 5.3.5.

D.3 Client Commands

bStatus_t GATT_InitClient(void)

Initialize the GATT client in the** Bluetooth **low energy stack.

NOTE: GATT clients must call this from the application init function.

bStatus_t GATT_RegisterForInd (uint8 taskId)

Register to receive incoming ATT Indications or Notifications of attribute values.

Parameters taskId: task to forward indications or notifications to

NOTE: GATT clients must call this from the application initialization function.

bStatus_t GATT_ExchangeMTU(uint16 connHandle, attExchangeMTUReq_t *pReq, uint8 taskId);

Used by a client to set the ATT_MTU to the maximum possible that can be supported by both devices when the client supports a value greater than the default ATT_MTU.

Parameters taskId: task to forward indications or notifications to

NOTE: This function can only be called once during a connection. For more information on the MTU, see Section 5.5.2.

Corresponding Events If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message. The type of the message is either ATT_EXCHANGE_MTU_RSP (with SUCCESS or bleTimeout status) indicating a SUCCESS or ATT_ERROR_RSP (with status SUCCESS) if an error occurred on the server.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId)

Used by a client to discover all primary services on a server.

Parameters connHandle: connection to use

taskId: task to be notified of response

Corresponding Events: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_GRP_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_GRP_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscPrimaryServiceByUUID(uint16 connHandle, uint8 *pValue, uint8 len, uint8 taskId)

Used by a client to discover a specific primary service on a server when only the service UUID is known.

Parameters connHandle: connection to use

pValue: pointer to value (UUID) for which to look

len: length of value

taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_FIND_BY_TYPE_VALUE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_FIND_BY_TYPE_VALUE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_FindIncludedServices(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find included services with a primary service definition on a server.

Parameters connHandle: connection to use

startHandle: start handle of primary service in which to search

endHandle: end handle of primary service in which to search

taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue)

Get a GAP Role parameter.

Parameters param – profile parameter ID (See Section D.4)

pValue – pointer to a location to get the value. This pointer is dependent on the param ID and will be cast to the appropriate data type.

Returns SUCCESS

INVALIDPARAMETER: param was not valid

bStatus_t GATT_DiscAllChars(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find all the characteristic declarations within a service when the handle range of the service is known.

Parameters connHandle: connection to use

startHandle: start handle of service in which to search

endHandle: end handle of service in which to search

taskId: task to be notified of response

Corresponding Events: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscCharsByUUID(uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

Used by a client to discover service characteristics on a server when the service handle range and characteristic UUID is known.

Parameters connHandle: connection to use

pReq: pointer to request to be sent, including start and end handles of service and UUID of characteristic value for which to search

taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_DiscAllCharDescs (uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find all the attribute handles and attribute types of the characteristic descriptor within a characteristic definition when only the characteristic handle range is known.

Parameters

connHandle: connection to use

startHandle: start handle

endHandle: end handle

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_FIND_INFO_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_FIND_INFO_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadCharValue (uint16 connHandle, attReadReq_t *pReq, uint8 taskId)

Used to read a characteristic calue from a server when the client knows the characteristic value Handle.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadUsingCharUUID (uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

Used to read a characteristic value from a server when the client only knows the characteristic UUID and does not know the handle of the characteristic.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_ReadLongCharValue (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

Used to read a characteristic value from a server when the client knows the characteristic value handle and the length of the characteristic value is longer than can be sent in a single read response attribute protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BLOB_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadMultiCharValues (uint16 connHandle, attReadMultiReq_t *pReq, uint8 taskId)

Used to read multiple characteristic values from a server when the client knows the characteristic value handles.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_MULTI_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_MULTI_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_WriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client.

Parameters

connHandle: connection to use

pReq: pointer to command to be sent

NOTE: No response are sent to the calling application task for this subprocedure. If the Characteristic Value write request is the wrong size, or has an invalid value as defined by the profile, then the write does not succeed and no error is generated by the server. The payload must be dynamically allocated as described in Section 5.3.5.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_SignedWriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Used to write a characteristic value to a server when the client knows the characteristic value handle and the ATT bearer is not encrypted. This subprocedure must only be used if the characteristic properties authenticated bit is enabled and the client and server device share a bond as defined in the GAP.

Parameters

connHandle: connection to use

pReq: pointer to command to be sent

NOTE: No response is sent to the calling application task for this subprocedure. If the authenticated Characteristic Value that is written is the wrong size, or has an invalid value as defined by the profile, or the signed value does not authenticate the client, then the write does not succeed and no error is generated by the server. The payload must be dynamically allocated as described in Section 5.3.5.

bStatus_t GATT_WriteCharValue (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

Used to write a characteristic value to a server when the client knows the characteristic value handle.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in Section 5.3.5.

bStatus_t GATT_WriteLongCharValue(uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

Used to write a characteristic value to a server when the client knows the characteristic value handle but the length of the characteristic value is longer than can be sent in a single write request attribute protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure completes when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in Section 5.3.5.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_ReliableWrites (uint16 connHandle, attPrepareWriteReq_t *pReq, uint8 numReqs, uint8 flags, uint8 taskId)

Used to write a characteristic value to a server when the client knows the characteristic value handle, and assurance is required that the correct characteristic value is going to be written by transferring the characteristic value to be written in both directions before the write is performed.

Parameters

connHandle: connection to use

pReq: pointer to requests to be sent (must be allocated)

numReqs – number of requests in pReq

flags – execute write request flags

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in Section 5.3.5.

bStatus_t GATT_ReadCharDesc (uint16 connHandle, attReadReq_t *pReq, uint8 taskId)

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure completes when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GATT_ReadLongCharDesc (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration' and the length of the characteristic descriptor declaration is longer than can be sent in a single read response attribute protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BLOB_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_WriteCharDesc (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration and the length of the characteristic descriptor declaration is longer than can be sent in a single read response attribute protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

bStatus_t GATT_WriteLongCharDesc (uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

Used to write a characteristic value to a server when the client knows the characteristic value handle but the length of the characteristic value is longer than can be sent in a single write request attribute protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

NOTE: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status), or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in Section 5.3.5.

Copyright © 2015–2016, Texas Instruments Incorporated

D.4 Return Values

- SUCCESS (0x00): Command was executed as expected (See the individual command API for corresponding events to expect.)
- INVALIDPARAMETER (0x02): Invalid connection handle or request field
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): attribute requires authentication
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): key size used for encrypting is insufficient
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): attribute requires encryption
- MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available (Retry later.)
- bleNotConnected (0x14): the device is not connected
- blePending (0x17):
  - When returned to a client function: a response is pending with the server or the GATT subprocedure is in progress
  - When returned to server function: confirmation from a client is pending
- bleTimeout (0x16): the previous transaction timed out (No more ATT and GATT messages can be sent until the connection is reestablished.)
- bleMemAllocError (0x13): memory allocation error occurred
- bleLinkEncrypted (0x19): link is already encrypted (An attribute PDU that includes an authentication signature that must not be sent on an encrypted link.)

D.5 Events

Events are received from the *Bluetooth* low energy stack in the application as a GATT_MSG_EVENT stack message sent through ICall. Events are received as the following structure where the method signifies the ATT event and the message is a combination of all the various ATT events.

```
typedef struct
{
  osal_event_hdr_t hdr; //!< GATT_MSG_EVENT and status
  uint16 connHandle;    //!< Connection message was received on
  uint8 method;         //!< Type of message
  gattMsg_t msg;        //!< Attribute protocol/profile message
} gattMsgEvent_t;
```

This section lists the various ATT events by their method and displays their structure that is used in the message payload. These events are listed in the att.h file.

- ATT_ERROR_RSP (0x01)

```
typedef struct
{
  uint8 reqOpcode; //!< Request that generated this error response
  uint16 handle;   //!< Attribute handle that generated error response
  uint8 errCode;   //!< Reason why the request has generated error response
} attErrorRsp_t;
attErrorRsp_t
```

- ATT_FIND_INFO_RSP (0x03)

```
typedef struct
{
  uint16 numInfo;      //!< Number of attribute handle-UUID pairs found
  uint8 format;       //!< Format of information data
  attFindInfo_t info; //!< Information data whose format is determined by format field
} attFindInfoRsp_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

- ATT_FIND_BY_TYPE_VALUE_RSP (0x07)

```
typedef struct
{
  uint16 numInfo;                 //!< Number of handles information found
  attHandlesInfo_t handlesInfo[ATT_MAX_NUM_HANDLES_INFO]; //!< List of 1 or more
handles information
} attFindByTypeValueRsp_t;
```

- ATT_READ_BY_TYPE_RSP (0x09)

```
typedef struct
{
  uint16 numPairs;                 //!< Number of attribute handle-UUID pairs found
  uint16 len;                      //!< Size of each attribute handle-value pair
  uint8 dataList[ATT_MTU_SIZE-2]; //!< List of 1 or more attribute handle-value pairs
} attReadByTypeRsp_t;
```

- ATT_READ_RSP (0x0B)

```
typedef struct
{
  uint16 len;                    //!< Length of value
  uint8 value[ATT_MTU_SIZE-1]; //!< Value of the attribute with the handle given
} attReadRsp_t;
```

- ATT_READ_BLOB_RSP (0x0D)

```
typedef struct
{
  uint16 len;                    //!< Length of value
  uint8 value[ATT_MTU_SIZE-1]; //!< Part of the value of the attribute with the handle
given
} attReadBlobRsp_t;
```

- ATT_READ_MULTI_RSP (0x0F)

```
typedef struct
{
  uint16 len;                     //!< Length of values
  uint8 values[ATT_MTU_SIZE-1]; //!< Set of two or more values
} attReadMultiRsp_t;
```

- ATT_READ_BY_GRP_TYPE_RSP (0x11)

```
typedef struct
{
  uint16 numGrps;                  //!< Number of attribute handle, end group handle
and value sets found
  uint16 len;                      //!< Length of each attribute handle, end group
handle and value set
  uint8 dataList[ATT_MTU_SIZE-2]; //!< List of 1 or more attribute handle, end group
handle and value
} attReadByGrpTypeRsp_t;
```

- ATT_WRITE_RSP (0x13)

  No data members

Copyright © 2015–2016, Texas Instruments Incorporated

- ATT_PREPARE_WRITE_RSP (0x17)

```
typedef struct
{
  uint16 handle;               //!< Handle of the attribute that has been read
  uint16 offset;               //!< Offset of the first octet to be written
  uint16 len;                   //!< Length of value
  uint8 value[ATT_MTU_SIZE-5]; //!< Part of the value of the attribute to be written
} attPrepareWriteRsp_t;
```

- ATT_EXECUTE_WRITE_RSP (0x19)
- ATT_HANDLE_VALUE_NOTI (0x1B)

```
typedef struct
{
  uint16 handle;               //!< Handle of the attribute that has been changed (must
be first field)
  uint16 len;                   //!< Length of value
  uint8 value[ATT_MTU_SIZE-3]; //!< New value of the attribute
} attHandleValueNoti_t;
```

- ATT_HANDLE_VALUE_IND (0x1D)

```
typedef struct
{
  uint16 handle;               //!< Handle of the attribute that has been changed (must
be first field)
  uint16 len;                   //!< Length of value
  uint8 value[ATT_MTU_SIZE-3]; //!< New value of the attribute
} attHandleValueInd_t;
```

- ATT_HANDLE_VALUE_CFM (0x1E)
  - o Empty msg field
- ATT_FLOW_CTRL_VIOLATED_EVENT (0x7E)

```
typedef struct
{
  uint8 opcode;        //!< opcode of message that caused flow control violation
  uint8 pendingOpcode; //!< opcode of pending message
} attFlowCtrlViolatedEvt_t;
```

- ATT_MTU_UPDATED_EVENT (0x7F)

```
typedef struct
{
  uint16 MTU; //!< new MTU size
} attMtuUpdatedEvt_t;
```


Copyright © 2015–2016, Texas Instruments Incorporated

D.6 GATT Commands and Corresponding ATT Events

The following table lists the possible commands that may have caused an event.

ATT Response Events	GATT API Calls
ATT_EXCHANGE_MTU_RSP	GATT_ExchangeMTU
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs, GATT_DiscAllCharDescs
ATT_FIND_BY_TYPE_VALUE_RSP	GATT_DiscPrimaryServiceByUUID
ATT_READ_BY_TYPE_RSP	GATT_PrepareWriteReq, GATT_ExecuteWriteReq, GATT_FindIncludedServices, GATT_DiscAllChars, GATT_DiscCharsByUUID, GATT_ReadUsingCharUUID,
ATT_READ_RSP	GATT_ReadCharValue, GATT_ReadCharDesc
ATT_READ_BLOB_RSP	GATT_ReadLongCharValue, GATT_ReadLongCharDesc
ATT_READ_MULTI_RSP	GATT_ReadMultiCharValues
ATT_READ_BY_GRP_TYPE_RSP	GATT_DiscAllPrimaryServices
ATT_WRITE_RSP	GATT_WriteCharValue, GATT_WriteCharDesc
ATT_PREPARE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc
ATT_EXECUTE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc

D.7 ATT_ERROR_RSP errCodes

This section lists the possible error codes in the ATT_ERROR_RSP event and their possible causes.

- ATT_ERR_INVALID_HANDLE (0x01): The attribute handle value given is not valid on this attribute server.
- ATT_ERR_READ_NOT_PERMITTED (0x02): The attribute is unable to be read.
- ATT_ERR_WRITE_NOT_PERMITTED (0x03): The attribute is unable to be written.
- ATT_ERR_INVALID_PDU (0x04): The PDU attribute is invalid.
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): The attribute requires authentication before it can be read or written.
- ATT_ERR_UNSUPPORTED_REQ (0x06): The attribute server does not support the request received from the attribute client.
- ATT_ERR_INVALID_OFFSET (0x07): The offset specified is past the end of the attribute.
- ATT_ERR_INSUFFICIENT_AUTHOR (0x08): The attribute requires an authorization before it can be read or written.
- ATT_ERR_PREPARE_QUEUE_FULL (0x09): Too many prepare writes have been queued.
- ATT_ERR_ATTR_NOT_FOUND (0x0A): No attribute exists within the attribute handle range.
- ATT_ERR_ATTR_NOT_LONG (0x0B): The attribute is unable to be read or written using the read blob request or prepare write request.
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The encryption key size for encrypting this link is insufficient.
- ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is invalid for the operation.
- ATT_ERR_UNLIKELY (0x0E): The attribute request requested has encountered an error that is unlikely and could not be completed as requested.
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): The attribute requires encryption before it can be read or written.
- ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10): The attribute type is an unsupported grouping attribute as defined by a higher layer specification.
- ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient resources exist to complete the request.

Copyright © 2015–2016, Texas Instruments Incorporated

GATTServApp API

This section details the API of the GATTServApp defined in gattservapp_util.c. These API are only the public commands that must be called by the profile and/or application.

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never get processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I are returned.

E.1 Commands

void GATTServApp_InitCharCfg(uint16 connHandle, gattCharCfg_t *charCfgTbl)

Initialize the client characteristic configuration table for a given connection. This API must be used when a service is added to the application (Section 5.3.4.2.2).

Parameters

connHandle – connection handle (0xFFFF for all connections)

charCfgTbl – client characteristic configuration table where this characteristic resides

bStatus_t GATTServApp_ProcessCharCfg(gattCharCfg_t *charCfgTbl, uint8 *pValue,uint8 authenticated, gattAttribute_t *attrTbl, uint16 numAttrs, uint8 taskId, pfnGATTReadAttrCB_t pfnReadAttrCB)

Process client characteristic configuration change

Parameters

charCfgTbl – profile characteristic configuration table

pValue – pointer to attribute value

authenticated – whether an authenticated link is required

attrTbl – whether attribute table

numAttrs – number of attributes in attribute table

tasked – task to be notified of confirmation

pfnReadAttrCB – read callback function pointer

Returns

SUCCESS (0x00): parameter was set

INVALIDPARAMETER (0x02): one of the parameters was a null pointer

ATT_ERR_INSUFFICIENT_AUTHOR (0x08): permissions require authorization

bleTimeout (0x17): ATT timeout occurred

blePending (0x16): another ATT request is pending

LINKDB_ERR_INSUFFICIENT_AUTHEN (0x05): authentication is required but link is not authenticated

bleMemAllocError (0x13): memory allocation failure occurred when allocating buffer

Copyright © 2015–2016, Texas Instruments Incorporated

gattAttribute_t *GATTServApp_FindAttr(gattAttribute_t *pAttrTbl, uint16 numAttrs, uint8 *pValue)

Find the attribute record within a service attribute table for a given attribute value pointer

Parameters

pAttrTbl – pointer to attribute table

numAttrs – number of attributes in attribute table

pValue – pointer to attribute value

Returns

Pointer to attribute record if found

NULL, if not found

bStatus_t GATTServApp_ProcessCCCWriteReq(uint16 connHandle, gattAttribute_t *pAttr, uint8 *pValue, uint16 len, uint16 offset, uint16 validCfg)

Process the client characteristic configuration write request for a given client

Parameters

connHandle – connection message was received on

pAttr – pointer to attribute value

pValue – pointer to data to be written

len – length of data

offset – offset of the first octet to be written

validCfg – valid configuration

Returns

SUCCESS (0x00): CCC was written correctly

ATT_ERR_INVALID_VALUE (0x80): an invalid value for a CCC

ATT_ERR_INVALID_VALUE_SIZE (0x0D): an invalid size for a CCC

ATT_ERR_ATTR_NOT_LONG (0x0B): offset needs to be 0

ATT_ERR_INSUFFICIENT_RESOURCES (0x11): CCC not found

Copyright © 2015–2016, Texas Instruments Incorporated

GAPBondMgr API

This section details the API of the GAPBondMgr defined in gapbondmgr.c. Many of these commands are called by the GAPRole or the *Bluetooth* low energy stack and do not need to be called from the application. The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

F.1 Commands

bStatus_t GAPBondMgr_SetParameter(uint16_t param, void *pValue)

Set a GAP bond manager parameter

Parameters

param – profile parameter ID (see Section D.2)

len – length of data to write

pValue – pointer to value to set parameter (This pointer depends on the parameter ID and will be cast to the appropriate data type.)

Returns

SUCCESS (0x00): parameter was set

INVALIDPARAMETER (0x02): param was invalid

bleInvalidRange (0x18): len is invalid for the given param

bStatus_t GAPBondMgr_GetParameter(uint16_t param, void *pValue)

Get a GAP bond manager parameter

Parameters

param – profile parameter ID (see Section D.2)

pValue – pointer to a location to get the value (This pointer is dependent on the param ID and will be cast to the appropriate data type.)

Returns

SUCCESS (0x00): param was successfully placed in pValue

INVALIDPARAMETER (0x02): param was not valid

bStatus_t GAPBondMgr_LinkEst(uint8 addrType, uint8 *pDevAddr, uint16 connHandle, uint8 role)

Notify the bond manager that a connection has been made

Parameters

addrType – address type of the peer device

peerAddr – peer device address

connHandle – connection handle

role – master or slave role

Returns

SUCCESS (0x00): GAPBondMgr was notified of link establishment

Copyright © 2015–2016, Texas Instruments Incorporated

void GAPBondMgr_LinkTerm(uint16_t connHandle)

Notify the bond manager that a connection has been terminated

Parameters connHandle – connection handle

void GAPBondMgr_SlaveReqSecurity(uint16_t connHandle)

Notify the bond manager that a slave security request is received

Parameters connHandle – connection handle

uint8 GAPBondMgr_ResolveAddr(uint8 addrType, uint8 *pDevAddr, uint8 *pResolvedAddr)

Resolve an address from bonding information

Parameters addrType – address type of the peer device

peerAddr – peer device address

pResolvedAddr – pointer to buffer to put the resolved address

Returns Bonding index (0 – (GAP_BONDINGS_MAX-1): if address was found...

GAP_BONDINGS_MAX: if address was not found

bStatus_t GAPBondMgr_ServiceChangeInd(uint16_t connectionHandle, uint8 setParam)

Set and clear the service change indication in a bond record

Parameters connHandle – connection handle of the connected device or 0xFFFF for all devices in database

setParam – TRUE to set the service change indication, FALSE to clear it

Returns SUCCESS (0x00) – bond record found and changed

bleNoResources (0x15) – no bond records found (for 0xFFFF connHandle)

bleNotConnected (0x14) – connection with connHandle is invalid

bStatus_t GAPBondMgr_UpdateCharCfg(uint16 connectionHandle, uint16 attrHandle, uint16 value)

Update the characteristic configuration in a bond record

Parameters connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database

attrHandle – attribute handle

value – characteristic configuration value

Returns SUCCESS (0x00) – bond record found and changed

bleNoResources (0x15) – no bond records found (for 0xFFFF connectionHandle)

bleNotConnected (0x14) – connection with connectionHandle is invalid

void GAPBondMgr_Register(gapBondCBs_t *pCB)

Register callback functions with the bond manager

Parameters pCB – pointer to callback function structure (see Section D.3)

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t GAPBondMgr_PasscodeRsp(uint16 connectionHandle, uint8 status, uint32 passcode)

Respond to a passcode request and update the passcode if possible

Parameters connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database

status – SUCCESS if passcode is available, otherwise see SMP_PAIRING_FAILED_DEFINES in gapbondmgr.h

passcode – integer value containing the passcode

Returns SUCCESS (0x00): connection found and passcode was changed

bleIncorrectMode (0x12): connectionHandle connection not found or pairing has not started

INVALIDPARAMETER (0x02): passcode is out of range

bleMemAllocError (0x13): heap is out of memory

uint8 GAPBondMgr_ProcessGAPMsg(gapEventHdr_t *pMsg)

A bypass mechanism to allow the bond manager to process GAP messages.

Parameters pMsg – GAP event message

Returns TRUE: safe to deallocate incoming GAP message,

FALSE: otherwise

NOTE: This is an advanced feature and must not be called unless the normal GAP Bond Manager task ID registration is overridden.

uint8 GAPBondMgr_CheckNVLen(uint8 id, uint8 len)

This function checks the length of a bond manager NV Item.

Parameters id – NV ID

len – lengths in bytes of item

Returns SUCCESS (0x00): NV item is the correct length

FAILURE (0x01): NV item is an incorrect length

Copyright © 2015–2016, Texas Instruments Incorporated

F.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPBOND_PAIRING_MODE	R/W	uint8	Default is GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_INITIATE_WAIT	R/W	uint16	Pairing Mode Initiate wait timeout. This is the time it will wait for a Pairing Request before sending the Slave Initiate Request. Default is 1000 ms
GAPBOND_MITM_PROTECTION	R/W	uint8	Man-In-The-Middle (MITM) turns on passkey protection in the pairing algorithm. Default is 0 (disabled).
GAPBOND_IO_CAPABILITIES	R/W	uint8	Default is GAPBOND_IO_CAP_DISPLAY_ONLY
GAPBOND_OOB_ENABLED	R/W	uint8	OOB data available for pairing algorithm. Default is 0(disabled).
GAPBOND_OOB_DATA	R/W	uint8[16]	OOB Data. Default is all 0s.
GAPBOND_BONDING_ENABLED	R/W	uint8	Request Bonding during the pairing process if enabled. Default is 0 (disabled).
GAPBOND_KEY_DIST_LIST		uint8	The key distribution list for bonding. Default is sEncKey, sIdKey, mIdKey, mSign enabled.
GAPBOND_DEFAULT_PASSCODE		uint32	The default passcode for MITM protection. Range is 0 to 999,999. Default is 0.
GAPBOND_ERASE_ALLBONDS	W	None	Erase all of the bonded devices.
GAPBOND_KEYSIZE	R/W	uint8	Key Size used in pairing. Default is 16.
GAPBOND_AUTO_SYNC_WL	R/W	uint8	Clears the White List adds to it each unique address stored by bonds in NV. Default is FALSE.
GAPBOND_BOND_COUNT	R	uint8	Gets the total number of bonds stored in NV. Default is 0 (no bonds).
GAPBOND_BOND_FAIL_ACTION	W	uint8	Possible actions central may take upon an unsuccessful bonding. Default is 0x02 (Terminate link upon unsuccessful bonding).
GAPBOND_ERASE_SINGLEBOND	W	uint8[9]	Erase a single bonded device. Must provide address type followed by device address.

F.3 Callbacks

These callbacks are functions whose pointers are passed from the application to the GAPBondMgr so that it can return events to the application as required. They are passed as the following structure.

```
typedef struct
{
  pfnPasscodeCB_t     passcodeCB;       //!< Passcode callback
  pfnPairStateCB_t    pairStateCB;      //!< Pairing state callback
} gapBondCBs_t;
```

F.3.1 Passcode Callback (passcodeCB)

This callback returns to the application the peer device information when a passcode is requested during the paring process. This function is defined as follows.

```
typedef void (*pfnPasscodeCB_t)
(
uint8  *deviceAddr,                       //!< address of device to pair with, and
could be either public or random.
  uint16 connectionHandle,                //!< Connection handle
  uint8  uiInputs,                        //!< Pairing User Interface Inputs - Ask
user to input passcode
  uint8  uiOutputs                        //!< Pairing User Interface Outputs -
Display passcode
 );
```

Based on the parameters passed to this callback such as the pairing user interface inputs or outputs, the application must display the passcode or initiate the entrance of a passcode.

Copyright © 2015–2016, Texas Instruments Incorporated

F.3.2 Pairing State Callback (pairStateCB)

This callback returns the current pairing state to the application whenever the state changes and as the current status of the pairing or bonding process associated with the current state. This function is defined as follows.

```
typedef void (*pfnPairStateCB_t)
(
  uint16 connectionHandle,   //!< Connection handle
  uint8  state,              //!< Pairing state @ref GAPBOND_PAIRING_STATE_DEFINES
  uint8  status              //!< Pairing status
);
```

The pairing states or state are as follows:

- GAPBOND_PAIRING_STATE_STARTED
  - The following status are possible for this state:
    - SUCCESS (0x00): pairing has been initiated
- GAPBOND_PAIRING_STATE_COMPLETE
  - The following statuses are possible for this state:
    - SUCCESS (0x00): pairing is complete (Session keys have been exchanged.)
    - SMP_PAIRING_FAILED_PASSKEY_ENTRY_FAILED (0x01): user input failed
    - SMP_PAIRING_FAILED_OOB_NOT_AVAIL (0x02): Out-of-band data not available
    - SMP_PAIRING_FAILED_AUTH_REQ (0x03): Input and output capabilities of devices do not allow for authentication
    - SMP_PAIRING_FAILED_CONFIRM_VALUE (0x04): the confirm value does not match the calculated compare value
    - SMP_PAIRING_FAILED_NOT_SUPPORTED (0x05): pairing is unsupported
    - SMP_PAIRING_FAILED_ENC_KEY_SIZE (0x06): encryption key size is insufficient
    - SMP_PAIRING_FAILED_CMD_NOT_SUPPORTED (0x07): The SMP command received is unsupported on this device
    - SMP_PAIRING_FAILED_UNSPECIFIED (0x08): encryption failed to start
    - bleTimeout (0x17): pairing failed to complete before timeout
    - bleGAPBondRejected (0x32): keys did not match
- GAPBOND_PAIRING_STATE_BONDED
  - The following statuses are possible for this state:
    - LL_ENC_KEY_REQ_REJECTED (0x06): encryption key is missing
    - LL_ENC_KEY_REQ_UNSUPPORTED_FEATURE (0x1A): feature is unsupported by the remote device
    - LL_CTRL_PKT_TIMEOUT_TERM (0x22): Timeout waiting for response
    - bleGAPBondRejected (0x32): this is received due to one of the previous three errors

Copyright © 2015–2016, Texas Instruments Incorporated

L2CAP API

G.1 Commands

This section describes the API related to setting up bidirectional communication between two *Bluetooth* low energy devices using L2CAP connection orientated channels. The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

bStatus_t L2CAP_RegisterPsm(l2capPsm_t *pPsm)

Register a protocol or service multiplexer with L2CAP

Parameters pPsm: PSM to deregister

Returns SUCCESS (0x00): Registration was successful.

INVALIDPARAMETER (0x02): maximum number of channels is greater than total supported

bleInvalidRange (0x18): PSM value is out of range

bleInvalidMtuSize (0x1B): MTU size is out of range

bleNoResources (0x15): out of resources

bleAlreadyInRequestedMode (0x11): PSM already registered

bStatus_t L2CAP_DeregisterPsm(uint8 taskId, uint16 psm)

Deregister a protocol or service multiplexer with L2CAP

Parameters taskId – the task to which PSM belongs

psm – PSM to deregister

Returns SUCCESS (0x00): Registration was successful.

INVALIDPARAMETER (0x02): PSM or task ID is invalid.

bleIncorrectMode (0x12): PSM is in use.

bStatus_t L2CAP_PsmInfo(uint16 psm, l2capPsmInfo_t *pInfo)

Get information about a given registered PSM

Parameters pPsm: PSM ID

pInfo – structure into which to copy PSM information

Returns SUCCESS (0x00): Operation was successful.

INVALIDPARAMETER (0x02): PSM is not registered.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t L2CAP_PsmChannels(uint16 psm, uint8 numCIDs, uint16 *pCIDs)

Get all active channels for a given registered PSM

Parameters

pPsm: PSM ID

numCIDs – number of CIDs can be copied

pCIDs – structure into which to copy CIDs

Returns

SUCCESS (0x00): Operation was successful.

INVALIDPARAMETER (0x02): PSM is not registered.

bStatus_t L2CAP_ChannelInfo(uint16 CID, l2capChannelInfo_t *pInfo)

Get information about an active connection-oriented channnel

Parameters

CID – local channel ID

pInfo – structure into which to copy channel information

Returns

SUCCESS (0x00): Registration was successful.

INVALIDPARAMETER (0x02): No such channel

bStatus_t L2CAP_ConnectReq(uint16 connHandle, uint16 psm, uint16 peerPsm)

Send connection request

Parameters

connHandle – connection handle

id – identifier received in connection request

result – outcome of connection request

Returns

SUCCESS (0x00): Request was sent successfully.

INVALIDPARAMETER (0x02): PSM is not registered.

MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available

bleIncorrectMode (0x12): PSM not registered

bleNotConnected (0x14): Connection is down.

bleNoResources (0x15): No available resource.

bleMemAllocError (0x13): Memory allocation error occurred.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t L2CAP_ConnectRsp(uint16 connHandle, uint8 id, uint16 result)

Send connection response.

Parameters

connHandle – connection onto which to create channel

psm – local PSM

peerPsm – peer PSM

Returns

SUCCESS: (0x00) Request was sent successfully.

INVALIDPARAMETER (0x02): PSM is not registered or Channel is closed.

MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.

bleNotConnected (0x14): Connection is down.

bleMemAllocError (0x13): Memory allocation error occurred.

L2CAP_DisconnectReq(uint16 CID)

Send disconnection request.

Parameters

CID – local CID to disconnect

Returns

SUCCESS (0x00): Request was sent successfully.

INVALIDPARAMETER (0x02): Channel ID is invalid.

MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.

bleNotConnected (0x14): Connection is down.

bleNoResources (0x15): No available resource

bleMemAllocError (0x13): Memory allocation error occurred.

Copyright © 2015–2016, Texas Instruments Incorporated

bStatus_t L2CAP_FlowCtrlCredit(uint16 CID, uint16 peerCredits)

Send flow control credit.

Parameters

CID – local CID

peerCredits – number of credits to give to peer device

Returns

SUCCESS (0x00): Request was sent successfully.

INVALIDPARAMETER (0x02): Channel is not open.

MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.

bleNotConnected (0x14): Connection is down.

bleInvalidRange (0x18): Credits is out of range.

bleMemAllocError (0x13): Memory allocation error occurred.

bStatus_t L2CAP_SendSDU(l2capPacket_t *pPkt)

Send data packet over an L2CAP connection-oriented channel established over a physical connection.

Parameters

pPkt – pointer to packet to be sent

Returns

SUCCESS (0x00): Data was sent successfully.

INVALIDPARAMETER (0x02): SDU payload is null.

bleInvalidRange (0x18): PSM value is out of range.

bleNotConnected (0x14): Connection or Channel is down.

bleMemAllocError (0x13): Memory allocation error occurred.

blePending (0x16): Another transmit in progress.

bleInvalidMtuSize (0x1B): SDU size is larger than peer MTU.

Copyright © 2015–2016, Texas Instruments Incorporated

HCI API

This section describes the vendor specific HCI Extension API, the HCI LE API, and the HCI Support API. An example is provided when more detail is required. The return values for these commands is always SUCCESS unless otherwise specified. This return value does not indicate successful completion of the command. These commands result in corresponding events that must be checked by the calling application. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the *Bluetooth* low energy stack. In this case, one of the ICall return values from Appendix I is returned.

H.1 Commands

hciStatus_t HCI_EXT_AdvEventNoticeCmd (uint8 taskID, uint16 taskEvent)

This command configures the device to set an event in the user task after each advertisement event completes. A nonzero taskEvent value is enable, while a zero valued taskEvent is disable.

Parameters taskID – task ID of the user

taskEvent – task event of the user (This event must be a single bit value.)

Returns SUCCESS: event configured correctly

LL_STATUS_ERROR_BAD_PARAMETER: more than one bit set exists

NOTE: This command does not return any events but has a meaningful return status and requires additional checks in the task function as described in Section 4.3.2.1.

Example (code additions to SimpleBLEPeripheral.c):

1. Define the event in the application.

```
// BLE Stack Events
#define SBP_ADV_CB_EVT                        0x0001
```

2. Configure the *Bluetooth* low energy protocol stack to return the event (in simpleBLEPeripheral_init()).

```
HCI_EXT_AdvEventNoticeCmd( selfEntity, SBP_ADV_CB_EVT);
```


Copyright © 2015–2016, Texas Instruments Incorporated

3. Check for and receive these events in the application (SimpleBLEPeripheral_taskFxn()).

```
  if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
  {
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
      ICall_Event *pEvt = (ICall_Event *)pMsg;

      // Check for BLE stack events first
      if (pEvt->signature == 0xffff)
      {
         if (pEvt->event_flag & SBP_ADV_CB_EVT)
         {
            // Advertisement ended. Process as desired.
         }
…
```

hciStatus_t HCI_EXT_BuildRevision(uint8 mode, uint16 userRevNum)

This command allows the embedded user code to set their own 16-bit revision number or read the build revision number of the *Bluetooth* ***low energy stack library software. The default value of the revision number is zero. When you update a*** *Bluetooth* ***low energy project by adding their own code, use this API to set your own revision number. When called with mode set to HCI_EXT_SET_APP_REVISION, the stack saves this value. No event is returned from this API when used this way. TI intended the event to be called from within the target. That the event is intended to be called from within the target does not preclude this command from being received through the HCI. No event is returned.***

Parameters

Mode – HCI_EXT_SET_APP_REVISION, HCI_EXT_READ_BUILD_REVISION

userRevNum – Any 16-bit value

Returns (only when mode == HCI_EXT_SET_USER_REVISION)

SUCCESS: build revision set successfully

LL_STATUS_ERROR_BAD_PARAMETER: an invalid mode

Corresponding Events (only when mode == HCI_EXT_SET_USER_REVISION)

HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ConnEventNoticeCmd (uint16 connHandle, uint8 taskID, uint16 taskEvent)

This command configures the device to set an event in the user task after each connection event completes. A non-zero taskEvent value is enable, while a zero valued taskEvent is disable.

Parameters

taskID – task ID of the user

taskEvent – task event of the user

Returns

SUCCESS or FAILURE

LL_STATUS_ERROR_BAD_PARAMETER: more than one bit set exists

NOTE: This command does not return any events but it has a meaningful return status and requires additional checks in the task function as described in Section 4.3.2.1.

Copyright © 2015–2016, Texas Instruments Incorporated

Example (code additions to SimpleBLEPeripheral.c):

1. Define the event in the application.

```
// BLE Stack Events
#define SBP_CON_CB_EVT                        0x0001
```

2. Configure the *Bluetooth* low energy protocol stack to return the event (in SimpleBLEPeripheral_processStateChangeEvt()) after the connection is established.

```
    case GAPROLE_CONNECTED:
      {
        HCI_EXT_ConnEventNoticeCmd ( connHandle, selfEntity, SBP_CON_CB_EVT );
```

3. Check for and receive these events in the application (SimpleBLEPeripheral_taskFxn()).

```
  if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
  {
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
      ICall_Event *pEvt = (ICall_Event *)pMsg;

      // Check for BLE stack events first
      if (pEvt->signature == 0xffff)
      {
         if (pEvt->event_flag & SBP_CON_CB_EVT)
         {
            // Connection Event ended. Process as desired.
         }
…
```

hciStatus_t HCI_EXT_DecryptCmd (uint8 *key, uint8 * encText)

This command decrypts encrypted data using the AES128.

Parameters key – Pointer to 16-byte encryption key

encText – Pointer to 16-byte encrypted data

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DisconnectImmedCmd (uint16 connHandle)

This command disconnects a connection immediately. This command is useful when a connection must be ended without the latency associated with the normal *Bluetooth* ***low energy controller terminate control procedure. The host issuing the command receives the HCI disconnection complete event with a reason status of 0x16 (that is, Connection Terminated by Local Host), followed by an HCI vendor-specific event.***

Parameters connHandle – The handle of the connection

Corresponding Events HCI_Disconnection_Complete

HCI_VendorSpecifcCommandCompleteEvent

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_EnablePTMCmd (void)

This command enables production test mode (PTM). This mode is used by the customer during assembly of their product to allow limited access to the *Bluetooth* ***low energy controller for testing and configuration. This mode remains enabled until the device is reset. See the related application note for additional details.***

Return Values HCI_SUCCESS: Successfully entered PTM

NOTE: This command causes a reset of the controller. To reenter the application, reset the device. This command does not return any events.

hciStatus_t HCI_EXT_EndModemTestCmd (void)

This command shuts down a modem test. A complete link layer reset occurs.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_GetConnInfoCmd(uint8 *numAllocConns, uint8 *numActiveConns, hciConnInfo_t *activeConnInfo)

This command acquires connection related information: number of allocated connections, the number of active connections, connection ID, connection role, peer address, and address type. The number of allocated connections can be modified with the MAX_NUM_BLE_CONNS define in bleUserConfig.h (see Section 5.7)

Parameters numAllocConns – pointer to number of build time connections allowed

numActiveConns – pointer to number of active *Bluetooth* low energy connections

activeConnInfo – pointer for active connection information

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: If all the parameters are NULL, the command is assumed to have originated from the transport layer. Otherwise, the command is assumed to have originated from a direct call by the application and any non-NULL pointer is used.

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

This API is used to start a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37-byte packet of pseudo-random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0 to 39) every 625 µs. Use the HCI_EXT_EndModemTest command to end the test.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs. The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_ModemTestRxCmd(uint8 rxFreq)

This API starts a continuous receiver modem test using a modulated carrier wave tone, at the frequency that corresponds to the specific RF channel. Any received data is discarded. Receiver gain can be adjusted using the HCI_EXT_SetRxGain command. RSSI may be read during this test by using the HCI_ReadRssi command. Use HCI_EXT_EndModemTest command to end the test.

Parameters rxFreq – selects which channel [0 to 39] on which to receive

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: The RF channel is specified, not the *Bluetooth* low energy frequency. The RF channel can be obtained from the *Bluetooth* low energy frequency as follows: RF channel = (*Bluetooth* low energy frequency – 2402) ÷ 2.

When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

hciStatus_t HCI_EXT_ModemTestTxCmd(uint8 cwMode, uint8 txFreq)

This API starts a continuous transmitter modem test, using either a modulated or unmodulated carrier wave tone, at the frequency that corresponds to the specified RF channel. Use the HCI_EXT_EndModemTest command to end the test.

Parameters cwMode – HCI_EXT_TX_MODULATED_CARRIER, HCI_EXT_TX_UNMODULATED_CARRIER

txFreq – Transmit RF channel k = 0 to 39, where *Bluetooth* low energy frequency = 2402 + (k × 2 MHz)

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: The RF channel, not the *Bluetooth* low energy frequency, is specified by txFreq. The RF channel can be obtained from the *Bluetooth* low energy frequency as follows: RF channel = (*Bluetooth* low energy frequency – 2402) ÷ 2.

When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_NumComplPktsLimitCmd (uint8 limit, uint8 flushOnEvt)

This command sets the limit on the minimum number of complete packets before a number of completed packets event is returned by the controller. If the limit is not reached by the end of a connection event, then the number of completed packets event is returned (if non-zero) based on the flushOnEvt flag. The limit can be set from one to the maximum number of HCI buffers (see the LE Read Buffer Size command in the** *Bluetooth* **Core specification). The default limit is one; the default flushOnEvt flag is FALSE.

Parameters

limit – from 1 to HCI_MAX_NUM_DATA_BUFFERS (returned by HCI_LE_ReadBufSizeCmd)

flushOnEvt

- HCI_EXT_DISABLE_NUM_COMPL_PKTS_ON_EVENT: return only a number of completed packets event when the number of completed packets is greater than or equal to the limit
- HCI_EXT_ENABLE_NUM_COMPL_PKTS_ON_EVENT: return the number of completed packets event at the end of every connection event

Corresponding Events

HCI_VendorSpecifcCommandCompleteEvent

NOTE: This command minimizes the overhead of sending multiple number of completed packet events, maximizing the processing available to increase over-the-air throughput. This command is often used with HCI_EXT_OverlappedProcessingCmd.

hciStatus_t HCI_EXT_OnePacketPerEventCmd (uint8 control)

This command configures the link layer to allow only one packet per connection event. The default system value for this feature is disabled. This command can be used to tradeoff throughput and power consumption during a connection. When enabled, power can be conserved during a connection by limiting the number of packets per connection event to one, at the expense of more limited throughput. When disabled, the number of packets transferred during a connection event is not limited, at the expense of higher power consumption per connection event.

Parameters

control – HCI_EXT_DISABLE_ONE_PKT_PER_EVT, HCI_EXT_ENABLE_ONE_PKT_PER_EVT

Corresponding Events

HCI_VendorSpecifcCommandCompleteEvent: this event is returned only if the setting is changing from enable to disable or from disable to enable

NOTE: A thorough power analysis of the system requirements to be performed before it is certain that this command saves power. Transferring multiple packets per connection event may be more power efficient.

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_PacketErrorRateCmd (uint16 connHandle, uint8 command)

This command is used to reset or read the packet error rate counters for a connection. When reset, the counters are cleared; when read, the total number of packets received, the number of packets received with a CRC error, the number of events, and the number of missed events are returned.

Parameters connId – the connection ID on which to perform the command

command – HCI_EXT_PER_RESET, HCI_EXT_PER_READ

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: The counters are 16 bits. At the shortest connection interval, 16-bit counters provide a little over 8 minutes of data.

hciStatus_t HCI_EXT_PERbyChanCmd (uint16 connHandle, perByChan_t *perByChan)

This command starts or ends the packet error rate by channel counter accumulation for a connection, and can be used by an application to make coexistence assessments. Based on the results, an application can perform an update channel classification command to limit channel interference from other wireless standards. If *perByChan is NULL, counter accumulation is discontinued. If *perByChan is not NULL, this location for the PER data has sufficient memory, based on the following type definition perByChan_t located in ll.h:

```
#define LL_MAX_NUM_DATA_CHAN 37
// Packet Error Rate Information By Channel
typedef struct
{
  uint16 numPkts[ LL_MAX_NUM_DATA_CHAN ];
  uint16 numCrcErr[ LL_MAX_NUM_DATA_CHAN ];
} perByChan_t;
```

NOTE: Ensure there is sufficient memory allocated in the perByChan structure and maintain the counters, clearing them if required before starting accumulation.

The counters are 16 bits. At the shortest connection interval, 16-bit counters provide a bit more than 8 minutes of data.

This command can be used in combination with HCI_EXT_PacketErrorRateCmd.

Parameters connHandle – The connection ID on which to accumulate the data.

perByChan – Pointer to PER by channel data, or NULL

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

This API starts a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37-byte packet of pseudo-random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0 to 39) every 625 µs. Use the HCI_EXT_EndModemTest command to end the test.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

hciStatus_t HCI_ReadBDADDRCmd(void)

This command is used to read the Bluetooth low energy address (BDADDR) of the device.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent where the BDADDR is a parameter

hciStatus_t HCI_ReadTransmitPowerLevelCmd(uint16 connHandle, uint8 txPwrType)

This command reads the transmit power level.

Parameters connHandle – connection handle

txPwrType – HCI_READ_CURRENT_TX_POWER_LEVEL or HCI_READ_MAXIMUM_TX_POWER_LEVEL

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ResetSystemCmd (uint8 mode)

This command issues a hard or soft system reset. A hard reset is caused by a watchdog timer time-out, while a soft reset is caused by jumping to the reset ISR.

Parameters mode – HCI_EXT_RESET_SYSTEM_HARD

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: The reset occurs after a 100-ms delay to allow the correspond event to be returned to the application.

Only a hard reset is allowed. A soft reset causes the command to fail. See Section 8.2.

hciStatus_t HCI_EXT_SetBDADDRCmd(uint8 *bdAddr)

This command sets the** Bluetooth **low energy address (BDADDR) of the device. This address overrides the address of the device determined when the device is reset). To restore the initialized address of the device stored in flash, issue this command with an invalid address (0xFFFFFFFFFFFF).

Parameters bdAddr – A pointer to a buffer to hold the address of this device. An invalid address (that is, all FFs) restores the address of this device to the address set at initialization.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd (uint8 control)

This command configures the link layer fast transmit response time feature. The default system value for this feature is enabled.

Parameters control – HCI_EXT_ENABLE_FAST_TX_RESP_TIME, HCI_EXT_DISABLE_FAST_TX_RESP_TIME

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: This command is only valid for a slave controller. When the host transmits data, the controller (by default) ensures the packet is sent over the link layer connection with minimal delay, even when the connection is configured to use slave latency. That is, the transmit response time is at or less than the connection interval (instead of waiting for the next effective connection interval due to slave latency). This transmit time results in lower power savings because the link layer may wake to transmit during connection events that would have been skipped due to slave latency. If saving power is more critical than fast transmit response time, you can disable this feature using this command. When disabled, the transmit response time is at or less than the effective connection interval (slave latency + 1× the connection interval).

hciStatus_t HCI_EXT_SetLocalSupportedFeaturesCmd (uint8 * localFeatures)

This command sets the local supported features of the controller.

Parameters localFeatures – A pointer to the feature set where each bit where each bit corresponds to a feature

0: Feature is not used

1: Feature can be used

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: This command can be issued either before or after one or more connections are formed. The local features set in this manner are only effective if performed before a feature exchange procedure has been initiated by the master. When this control procedure has been completed for a particular connection, only the exchanged feature set for that connection is used. Because the link layer may initiate the feature exchange procedure autonomously, use this command before the connection is formed. The features are initialized by the controller upon start-up. You are unlikely to require this command. The defined symbols for the feature values are in ll.h.

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd (uint8 txPower)

This command overrides the RF transmitter output power used by the direct test mode (DTM). The maximum transmitter output power setting used by DTM is typically the maximum transmitter output power setting for the device (that is, 5 dBm). This command changes the value used by DTM.

Parameters txPower – one of the following:

- HCI_EXT_TX_POWER_MINUS_21_DBM
- HCI_EXT_TX_POWER_MINUS_18_DBM
- HCI_EXT_TX_POWER_MINUS_15_DBM
- HCI_EXT_TX_POWER_MINUS_12_DBM
- HCI_EXT_TX_POWER_MINUS_9_DBM
- HCI_EXT_TX_POWER_MINUS_6_DBM
- HCI_EXT_TX_POWER_MINUS_3_DBM
- HCI_EXT_TX_POWER_0_DBM
- HCI_EXT_TX_POWER_1_DBM
- HCI_EXT_TX_POWER_2_DBM
- HCI_EXT_TX_POWER_3_DBM
- HCI_EXT_TX_POWER_4_DBM
- HCI_EXT_TX_POWER_5_DBM

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

NOTE: When DTM is ended by a call to HCI_LE_TestEndCmd or a HCI_Reset is used, the transmitter output power setting is restored to the default value of 0 dBm.

hciStatus_t HCI_EXT_SetSCACmd (uint16 scaInPPM)

This command sets the sleep clock accuracy (SCA) value of this device, in parts per million (PPM), from 0 to 500. For a master device, the value is converted to one of eight ordinal values representing a SCA range per Specification of the Bluetooth System, Covered Core Package, Version: 4.1, which is used when a connection is created. For a slave device, the value is directly used. The system default value for a master and slave device is 50 ppm and 40 ppm, respectively.

Parameters scaInPPM – This SCA of the device in PPM from 0 to 500.

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: This command is only allowed when the device is disconnected.

The SCA value of the device remains unaffected by an HCI Reset.

Copyright © 2015–2016, Texas Instruments Incorporated

hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd (uint8 mode)

This command enables or disables the Slave Latency Override, letting you temporarily suspend Slave Latency even though it is active for the connection. When enabled, the device wakes up for every connection until Slave Latency Override is disabled again. The default value is Disable.

Parameters control – HCI_EXT_ENABLE_SL_OVERRIDE, HCI_EXT_DISABLE_SL_OVERRIDE

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: The function applies only to devices acting in the slave role. The function can be helpful when the slave application receives something that must be handled without delay. The function does not change the slave latency connection parameter; the device wakes up for each connection event.

hciStatus_t HCI_EXT_SetTxPowerCmd(uint8 txPower)

This command sets the RF transmitter output power. The default system value for this feature is 0 dBm.

Parameters txPower – transmit power of the device, one of the following corresponding events

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent:

- HCI_EXT_TX_POWER_MINUS_21_DBM
- HCI_EXT_TX_POWER_MINUS_18_DBM
- HCI_EXT_TX_POWER_MINUS_15_DBM
- HCI_EXT_TX_POWER_MINUS_12_DBM
- HCI_EXT_TX_POWER_MINUS_9_DBM
- HCI_EXT_TX_POWER_MINUS_6_DBM
- HCI_EXT_TX_POWER_MINUS_3_DBM
- HCI_EXT_TX_POWER_0_DBM
- HCI_EXT_TX_POWER_1_DBM
- HCI_EXT_TX_POWER_2_DBM
- HCI_EXT_TX_POWER_3_DBM
- HCI_EXT_TX_POWER_4_DBM
- HCI_EXT_TX_POWER_5_DBM

hciStatus_t HCI_ReadRssiCmd(uint16 connHandle)

This command reads the RSSI of the last packet received on a connection given by the connection handle. An example of using this command can be found in the application task of the SimpleBLECentral project'.

Parameters connHandle – connection handle of connection from which to read the RSSI

Corresponding Event HCI_VendorSpecifcCommandCompleteEvent

NOTE: If the receiver modem test is running, the RF RSSI for the last received data is returned. If there is no RSSI value, then HCI_RSSI_NOT_AVAILABLE is returned

Copyright © 2015–2016, Texas Instruments Incorporated

H.2 Host Error Codes

This section lists the various possible error codes generated by the host. If an HCI extension command that sent a command status with the SUCCESS error code before processing may find an error during execution then the error is reported in the normal completion command for the original command. The error code 0x00 means SUCCESS. The possible range of failure error codes is 0x01-0xFF. The following table lists an error code description for each failure error code.

Value	Parameter Description
0x00	SUCCESS
0x01	FAILURE
0x02	INVALIDPARAMETER
0x03	INVALID_TASK
0x04	MSG_BUFFER_NOT_AVAIL
0x05	INVALID_MSG_POINTER
0x06	INVALID_EVENT_ID
0x07	INVALID_INTERRUPT_ID
0x08	NO_TIMER_AVAIL
0x09	NV_ITEM_UNINIT
0x0A	NV_OPER_FAILED
0x0B	INVALID_MEM_SIZE
0x0C	NV_BAD_ITEM_LEN
0x10	bleNotReady
0x11	bleAlreadyInRequestedMode
0x12	bleIncorrectMode
0x13	bleMemAllocError
0x14	bleNotConnected
0x15	bleNoResources
0x16	blePending
0x17	bleTimeout
0x18	bleInvalidRange
0x19	bleLinkEncrypted
0x1A	bleProcedureComplete
0x30	bleGAPUserCanceled
0x31	bleGAPConnNotAcceptable
0x32	bleGAPBondRejected
0x40	bleInvalidPDU
0x41	bleInsufficientAuthen
0x42	bleInsufficientEncrypt
0x43	bleInsufficientKeySize
0xFF	INVALID_TASK_ID

Copyright © 2015–2016, Texas Instruments Incorporated

ICall API

I.1 Commands

The ICall commands that are useful from the application task are defined in Section 4.2

I.2 Error Codes

This section lists the error codes associated with ICall failures. The following values can be returned from any function defined in ICallBleAPI.c.

Value	Error Name	Description
0x04	MSG_BUFFER_NOT_AVAIL	Allocation of ICall Message Failed
0xFF	ICALL_ERRNO_INVALID_SERVICE	The service corresponding to a passed service id is not registered
0xFE	ICALL_ERRNO_INVALID_FUNCTION	The function id is unknown to the registered handler of the service
0xFD	ICALL_ERRNO_INVALID_PARAMETER	Invalid Parameter Value
0xFC	ICALL_ERRNO_NO_RESOURCE	Not available entities, tasks, or other ICall resources
0xFB	ICALL_ERRNO_UNKNOWN_THREAD	The task is not a registered task of the entity id is not a registered entity
0xFA	ICALL_ERRNO_CORRUPT_MSG	Corrupt message error
0xF9	ICALL_ERRNO_OVERFLOW	Counter Overflow
0xF8	ICALL_ERRNO_UNDERFLOW	Counter Underflow

Copyright © 2015–2016, Texas Instruments Incorporated

Revision History

Changes from A Revision (November 2015) to B Revision	Page
• Edited and updated document to TI standards.	2

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Copyright © 2015–2016, Texas Instruments Incorporated

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 and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications 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