SimpleLink™ *Bluetooth*® low energy CC2640 wireless MCU

Software Developer's Guide

For BLE-Stack™ Version: 2.1.0

Literature Number: SWRU393A

July 2015

TABLE OF CONTENTS

TABLE OF FIGURES / DIAGRAMS

References

Note: The latest version of this guide can be found at http://www.ti.com/lit/pdf/swru393

[1] TI BLE Vendor Specific HCI Reference Guide v2.0
C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx\TI_BLE_Vendor_Specific_HCI_Guide.pdf

[2] Texas Instruments CC26xx Technical Reference Manual
http://www.ti.com/lit/pdf/swcu117

[3] Measuring Bluetooth Smart Power Consumption
http://www.ti.com/lit/pdf/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 Guide
C:\ti\tirtos_simplelink_2_13_00_06\docs\tirtos_User_Guide.pdf

[7] TI-RTOS SYS/BIOS Kernel User 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 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) web site:

[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

Useful Links

[17] TI Bluetooth LE Wiki-page: www.ti.com/ble-wiki

[18] Latest BLE Stack download & examples: www.ti.com/ble-stack

[19] Texas Instruments E2E support forum: www.ti.com/ble-forum

[20] TI Designs Reference Library: http://www.ti.com/general/docs/refdesignsearch.tsp

1 Overview

The purpose of this document is to give an overview of the Texas Instruments SimpleLink™ Bluetooth® low energy CC2640 wireless MCU software development kit in order to begin creating a *Bluetooth*® Smart custom application. This document also serves as an introduction to the Bluetooth low energy specification, referred to as BLE in this document. However, it should not be used as a substitute for the complete specification. For more details, see the Bluetooth core specification [15] or some introductory material at the Texas Instruments BLE wiki page [17].

1.1 Introduction

Version 4.1 of the Bluetooth specification allows for two systems of wireless technology: Basic Rate (BR: often referred to as "BR/EDR" for "Basic Rate / Enhanced Data Rate") and Bluetooth low energy. The BLE system was created for the purpose of transmitting very small packets of data at a time, while consuming significantly less power than BR/EDR devices.

Devices that can support BR and BLE are referred to as dual-mode devices and go under the branding *Bluetooth*® Smart Ready. Typically in a Bluetooth wireless technology system, a mobile smartphone or laptop computer will be a dual-mode device. Devices that only support BLE are referred to as single-mode devices and go under the branding *Bluetooth*® Smart. These single-mode devices are generally used for application in which low power consumption is a primary concern, such as those that run on coin cell batteries.

Figure 1: Bluetooth Smart and Bluetooth Smart Ready Branding Marks

1.2 BLE Protocol Stack Basics

The BLE protocol stack architecture is illustrated here:

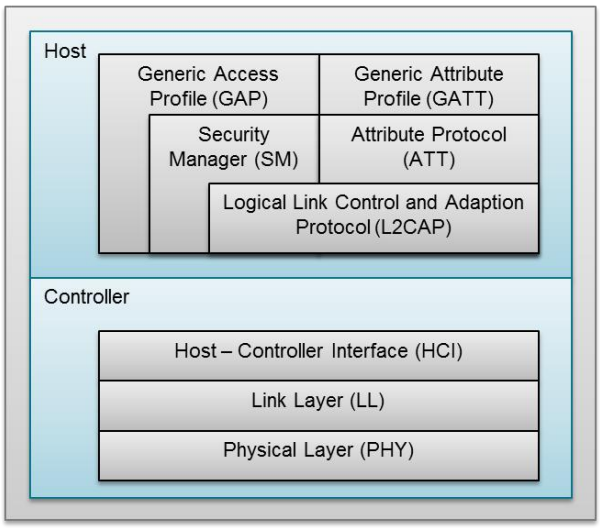


Figure 2: BLE Protocol Stack

The BLE protocol stack (referred to as "protocol stack") consists of two sections: the controller and the host. This separation of controller and host derives from standard Bluetooth BR/EDR devices, in which the two sections were often implemented separately. Any profiles and applications that are used sit on top of the GAP and GATT layers of the protocol stack.

The *physical layer* (PHY) is a 1Mbps 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 being in one of five possible states: standby, advertising, scanning, initiating, or connected. Advertisers transmit data without being in a connection, while scanners listen for advertisers. An Initiator is a device that is responding to an Advertiser with a connection request. If the Advertiser accepts, both the advertiser and initiator will enter a connected state. When a device is in a connection, it will be connected in one of two roles:

master or slave. The device that initiated the connection becomes the master, and the device that accepted the request becomes the slave.

The *host control interface* (HCI) layer provides a means of communication between the host and controller via 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 Bluetooth Core Spec [15]. Texas Instruments' proprietary commands and events are specified in the Vendor Specific Guide [1].

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 Texas Instruments' 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. In addition, GAP handles the initiation of security features. See section 5.1 for more information on Texas Instruments' implementation of the GAP layer.

The *attribute protocol* (ATT) layer protocol allows a device to expose certain pieces of data, known as "attributes", to another device.

The *generic attribute protocol* (GATT) layer is a service framework that defines the sub-procedures for using ATT. All data communications that occur between two devices in a BLE connection are handled through GATT sub-procedures. Therefore, the application and/or profiles will directly use GATT. See section 5.3 for more information on Texas Instruments' implementation of the ATT and GATT layers.

2 Texas Instruments BLE Software Development Platform

The Texas Instruments royalty-free BLE-Stack™ Software Development Kit (SDK) is a complete software platform for developing single-mode BLE applications. It is based on the SimpleLink CC2640, complete System-on-Chip (SoC) *Bluetooth*® Smart solution. The CC2640 combines a 2.4GHz RF transceiver, 128kB of in-system programmable memory, 20kB of SRAM, and a full range of peripherals. The device is centred on an ARM® Cortex-M3™ series processor that handles the application layer and BLE protocol stack, as well as an autonomous radio core centred 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, further extending the low-power capabilities of the CC2640. A block diagram is shown below and more information can be found in the CC26xx Technical Reference Manual (TRM) [2].

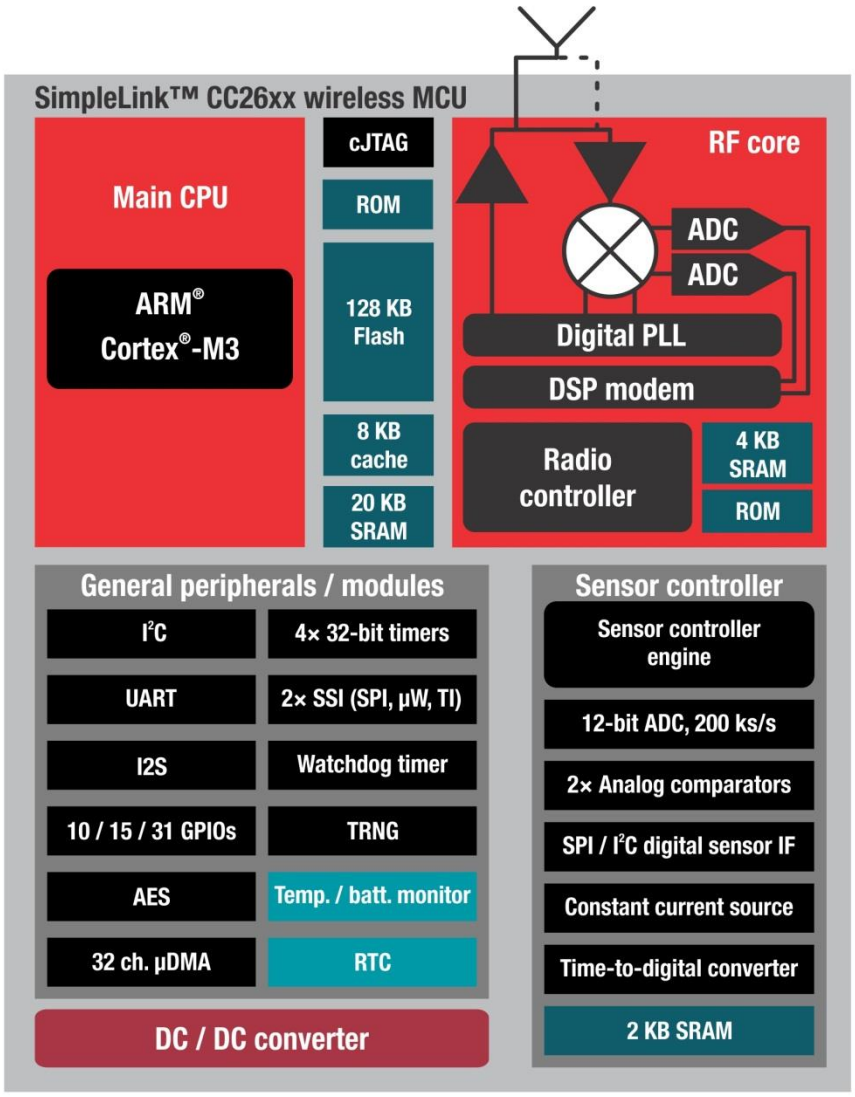


Figure 3: SimpleLink CC2640 Block Diagram

2.1 Protocol Stack / Application Configurations

The platform supports two different protocol stack / application configurations (illustrated in Figure 4)

- **Single-Device:** The controller, host, profiles, and application are all implemented on the CC2640 as a true single chip solution. This is the simplest and most common configuration when using the CC2640. This is also the configuration that most of our sample projects use. It 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 via the Simple Network Processor (SNP) API that simplifies the management of the BLE network processor. The SNP API communicates with the BLE 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 need to implement complexities associated with an HCI based protocol. In these cases, the application & profiles can be developed externally while still running the BLE protocol stack on the CC2640. For a description of the Simple Network Processor, refer to the SNP API guide [13].

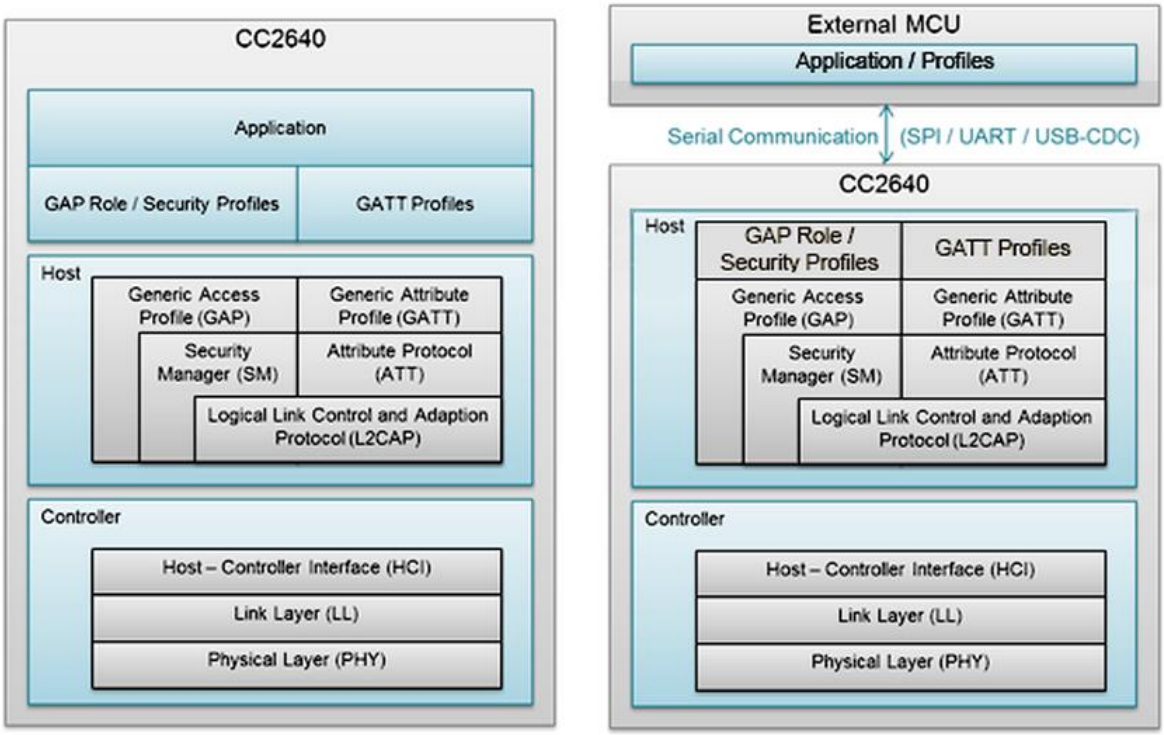


Figure 4: Single-Device and Simple Network Processor Configuration

2.2 Solution Platform

This section will describe the various components that are installed with the BLE-Stack SDK as well as the directory structure of the protocol stack and any tools which are needed for development.

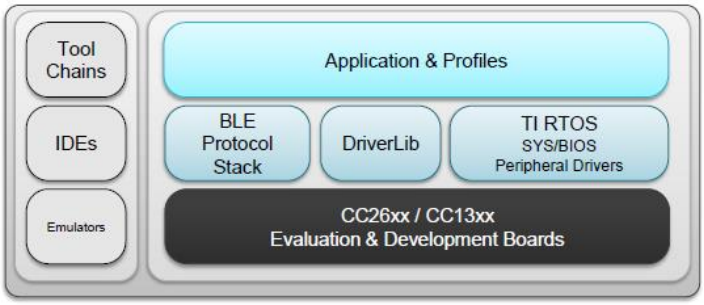


Figure 5: BLE-Stack Development System

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 etc.).
- **CC26xxware DriverLib** provides a register abstraction layer and is used by the software & drivers to control the CC2640 SoC.
- The **BLE Protocol Stack** is provided in library form with parts of the protocol stack residing in the CC2640 ROM.
- Sample **Applications and Profiles** make it easier to start development using both proprietary & generic solutions. Certain Applications & Profiles provided in the BLE SDK are fully qualified by Bluetooth SIG.

The IDE's currently supported are:

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

2.3 Directory Structure

The BLE SDK installer includes all the necessary files to get started with evaluation and creating a custom Application. There are four parts of the installed SDK:

- **Accessories:** BTool windows application, Boundary Tool, and pre-compiled "super" hex files for common applications, such as SimpleBLEPeripheral & SensorTag. Except for the SensorTag, the pre-built hex files are designed for the SmartRF06 evaluation board and the CC2650EM-7ID evaluation module.
- **Components:** BLE protocol stack services
- **Documents:** Software Developer's Guide, Vendor Specific HCI API Guide and Application Notes
- **Projects:** Application examples, proprietary and certified Bluetooth SIG Profiles

2.4 Projects

The BLE-Stack SDK installer includes a large number of projects ranging from very basic BLE functionality to use-case specific applications such as Heart Rate Sensor, Glucose Collector, etc. The basic projects which should be used as a starting point for application development are briefly described below. For more detail on these and all other included projects, please refer to Section 12.

The **SimpleBLEPeripheral** project consists of sample code that demonstrates a very simple BLE slave application in the single-device configuration. It can be used as a reference for developing a slave / peripheral application.

The **SimpleBLECentral** project demonstrates the other side of the connection. It demonstrates a simple master / central application in the single-device configuration, and can be used as a reference for developing master / central applications.

The **SimpleBLEBroadcaster** project demonstrates broadcast only role which is useful for implementing non-connectable Beacon applications, such as Apple iBeacon and Google Eddystone. Refer to the "Bluetooth low energy Beacons" Application Note (SWRA475) for more information about Bluetooth Smart Beacons.

The **SensorTag** project is a peripheral application which is configured to run on the CC2650 Sensor Tag reference hardware platform and communicate with the sensor tag's various peripheral devices (e.g. temperature sensor, gyro, magnetometer, etc.).

The **SimpleNP** project is used to build the simple network processor software for the CC2640. It currently supports a configuration for the slave / peripheral roles. Refer to the Simple Network Processor API guide [13] for APIs available in the simple network processor implementation.

The **SimpleAP** project is used to build the simple application processor software for the CC2640. It can be used to demonstrate an application (host) MCU communicating with the CC2640 running the SimpleNP network processor application. Refer to the SNP API guide located in the "Documents" folder for APIs available for the simple network processor implementation

The **HostTest** project is used to build the BLE HCI based network processor software for the CC2640. It can be configured for both master and slave roles and can be controlled by the BTool PC application. Refer to the Vendor Specific HCI API guide located in the "Documents" folder for APIs available for configuring & controlling the HostTest applicaiton.

2.5 Setting up the Development Environment

Before progressing further, it is necessary to set up the Integrated Development Environment (IDE) in order to browse through the relevant projects and view code as it is referenced in this document. All embedded software for the CC2640 is developed using *IAR Embedded Workbench for ARM* (from IAR Software) or Code Composer Studio (CCS) from Texas Instruments on a Windows 7® or later PC.. This section provides information on where to find this software and how to properly configure the workspace for each IDE.

All path and file references in this document assume that the BLE SDK has been installed to the default path henceforward referred to as **$BLE_INSTALL$**. It is strongly recommended to make a working copy of the BLE SDK prior to making any changes. The BLE SDK uses relative paths and is designed to be portable thus allowing the top-level directory to be copied to any valid path.

Note: If installing to a non-default path, ensure that the max file system name-path length is not exceeded.

2.5.1 Installing the SDK

To install the BLE-Stack SDK, run the installer "ble_cc26xx_setupwin32_2_01_00_xxxxx.exe"

- Where "xxxxx" is the SDK build revision number
- Default SDK install path is C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx

This Installer will also install the TI-RTOS bundle and XDC tools, if not already installed, as well as the BTool PC application. *Figure 6* lists the software & tools are supported and tested with this BLE-Stack SDK. Check the TI BLE Wiki page [17] for the latest supported tool versions.

Tool / Software	Version	Install Path
BLE-Stack SDK Installer	2.1.0	C:\ti\simplelink\ble_cc26xx_2_01_00_xxxxx
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

Figure 6: Supported Tools & Software

2.5.2 IAR

IAR contains many features that go beyond the scope of this document. More information and documentation can be found on IAR's website: www.iar.com.

2.5.2.1 Configuring IAR Embedded Workbench for ARM

1. Download & install IAR EW ARM version **7.40.2**. This is the official version of IAR supported & tested for this release. There are two options available for obtaining IAR:
   - **Download IAR Embedded Workbench 30-day Evaluation Edition** – This version of IAR is completely free of charge and has full functionality; however it is only a 30-day trial. It includes all of the standard features.

     IAR 30-day Evaluation Edition can be downloaded from the following URL:
     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, it is recommended to purchase the complete version of IAR without any restrictions.

     Information on purchasing the complete version of IAR can be found at the following URL:
     http://www.iar.com/en/Products/IAR-Embedded-Workbench/ARM/

2. Install the TI XDS Emulation Package (emupack):
   2.1. Run the ti_emupack_setup.exe installer found in the IAR installation:
   <iar_install>\arm\drivers\ti-xds. Typically IAR is installed to C:\Program Files (x86)\IAR Systems
   Important: Select "Run as Administrator" when performing the emupack installation
   2.2. Install it to the default location: C:\ti\ccs_base

3. For full verbosity during building, it is recommended to show all build output messages. Set Tools → Options → Messages → Show Build Messages to "All".

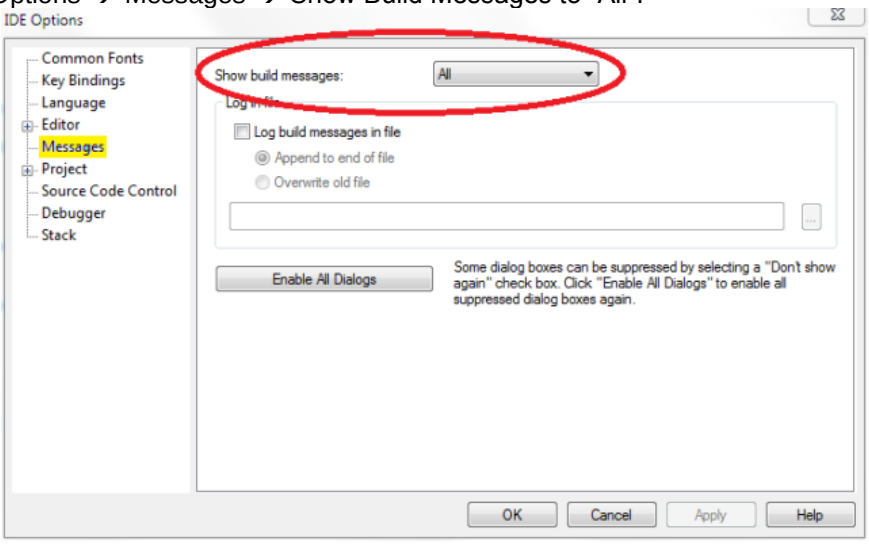


Figure 7: Full Verbosity

4. Verify Tools → Custom Argument Variables points to the installed TI-RTOS & XDC tool paths set in the "CC26xx TI-RTOS" group. The respective TI-RTOS & XDC default tool paths are shown in Figure 8. If any additional argument groups on the "Workspace" or "Global" tabs are present that conflict with the "CC26xx TI-RTOS" group, disable the group(s).

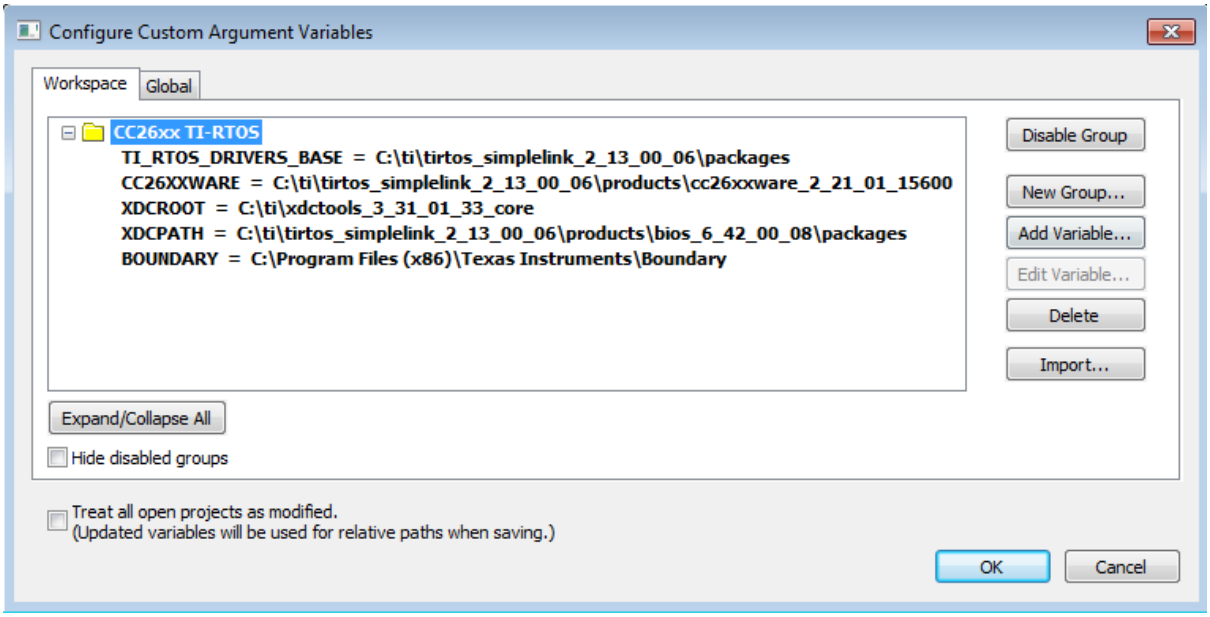


Figure 8: Custom Argument Variables

2.5.2.2 Using IAR Embedded Workbench

This section will describe how to open and build an existing project. Note that the **SimpleBLEPeripheral** project will be used as a reference throughout this document. However, all of the BLE projects included in the development kit will have a similar structure.

2.5.2.2.1 Open an Existing Project

First, open the IAR Embedded Workbench IDE from the Start Menu. Once IAR has opened up, click File → Open → Workspace. Select the following file:

$BLE_INSTALL$\Projects\ble\SimpleBLEPeripheral\CC26xx\IAR\SimpleBLEPeripheral.eww

This is the workspace file for the SimpleBLEPeripheral project. Once it is selected all of the files associated with the workspace will be visible in the Workspace pane on the left side of the screen:

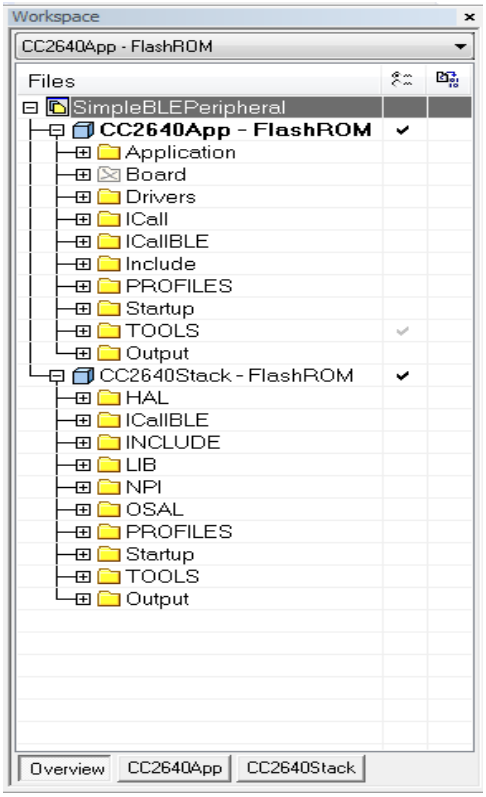


Figure 9: IAR Workspace Pane

This workspace, as well as all CC2640 project workspaces, contains two separate projects: the Application project (CC2640App) and the BLE protocol stack project (CC2640Stack). One of these can be selected as the active project by clicking the relevant tab at the bottom of the workspace pane. In the figure above, the Overview tab is selected. This will display the file structure for both projects simultaneously. In this case, the active project can be selected by using the drop-down menu at the top of the workspace pane. Each of these projects will produce a separate downloadable object. This dual-image architecture was chosen so that the Application can be updated independent of the Stack.

For practical reasons, the SimpleBLEPeripheral sample project is made primary reference target for the description of a generic application in this guide. The SimpleBLEPeripheral project implements a very basic BLE peripheral device including a GATT server with GATT services. This project can be used as a framework for developing many different peripheral-role applications.

2.5.2.2.2 **Compiling 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 (i.e., from CC2640 to CC2650) may result in build errors. Note that all CC2640/CC2650 code is binary compatible & interchangeable for BLE-Stack software builds. Also, the CPU type is the same for all silicon package types.

Because the workspace is split into two projects (Application and Stack), there is a specific sequence for compilation and download.

1. Build the Application project using Project → Make (or F7)
2. Build the Stack project using Project → Make (or F7)
3. Download the Application project using Project → Download → Download Active Application.
4. Download the Stack project using Project → Download → Download Active Application.

At this point, the Application can be debugged by selecting the Application project and choosing Project → Debug without Downloading. This option will load the debug symbols into the debugger but not program flash memory since it was previously updated using "Download Active Application".

Note that the above procedure is only needed for the initial download and whenever the Stack project is modified. As long as the Stack project is not modified, the only steps required are:

1. Build the Application using Project → Make (or F7).
2. Download the Application and debug using Project → Download and Debug (or Ctrl + D)
3. Once the Application has been downloaded (i.e., flash memory programmed), Project → Debug without Downloading can be used.

2.5.3 Code Composer Studio

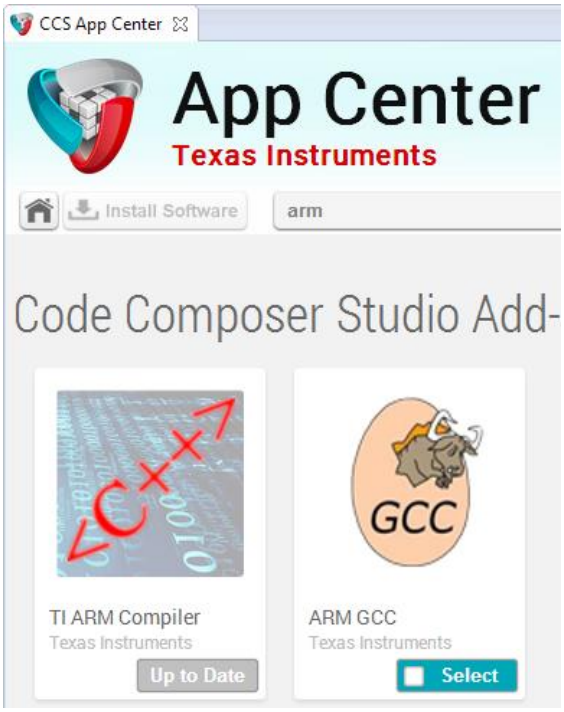
Code Composer Studio (CCS) contains many features that go beyond the scope of this document. More information and documentation can be found on CCS's website: http://www.ti.com/tool/CCSTUDIO. Make sure to check the BLE SDK release notes to see which CCS version to use as well as any required workarounds. Additionally, object code produced by CCS may differ in size and performance as compared to IAR produced object code.

2.5.3.1 Configuring CCS

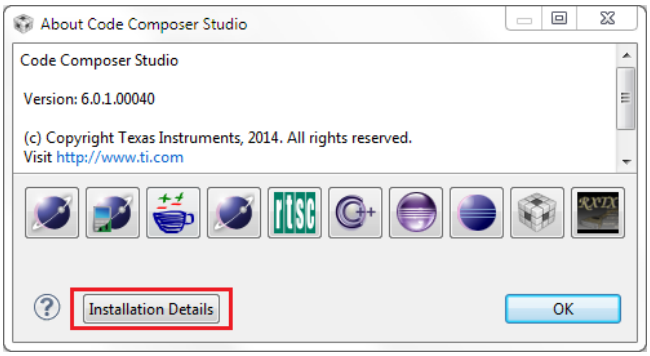
This section will describe installing & configuring the correct version of CCS and the necessary tools.

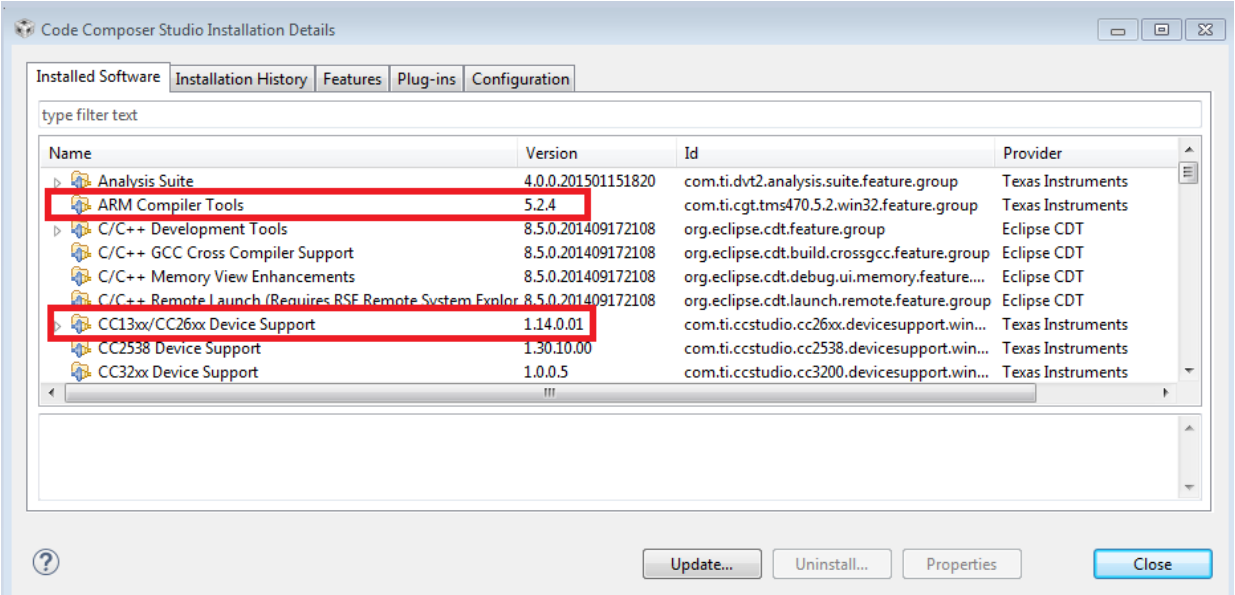
1. Download version 6.1.0 (or later) of CCS from here:
   http://processors.wiki.ti.com/index.php/Download_CCS
2. Launch the "setup_ccs_win32.exe" installer
3. On the "Processor Support" menu, expand "SimpleLink Wireless MCUs" and select:
   CC26xx Device Support
   TI ARM Complier
4. Once CCS is installed, apply all available updates by selecting Help → Check for Updates
   Note: This may require several restarts of CCS as each update is applied
5. Install latest TI ARM Compiler from the App Center inside CCS

- Select View → CCS App Center
- Select TI ARM Compiler
- Select Install Software:



6. After all updates are installed, verify the following Installation Details under Help → About Code Composer Studio:





ARM Compiler Tools: 5.2.4
CC26xx Device Support: 1.14.0.01 (or later)

2.5.3.2 Using CCS

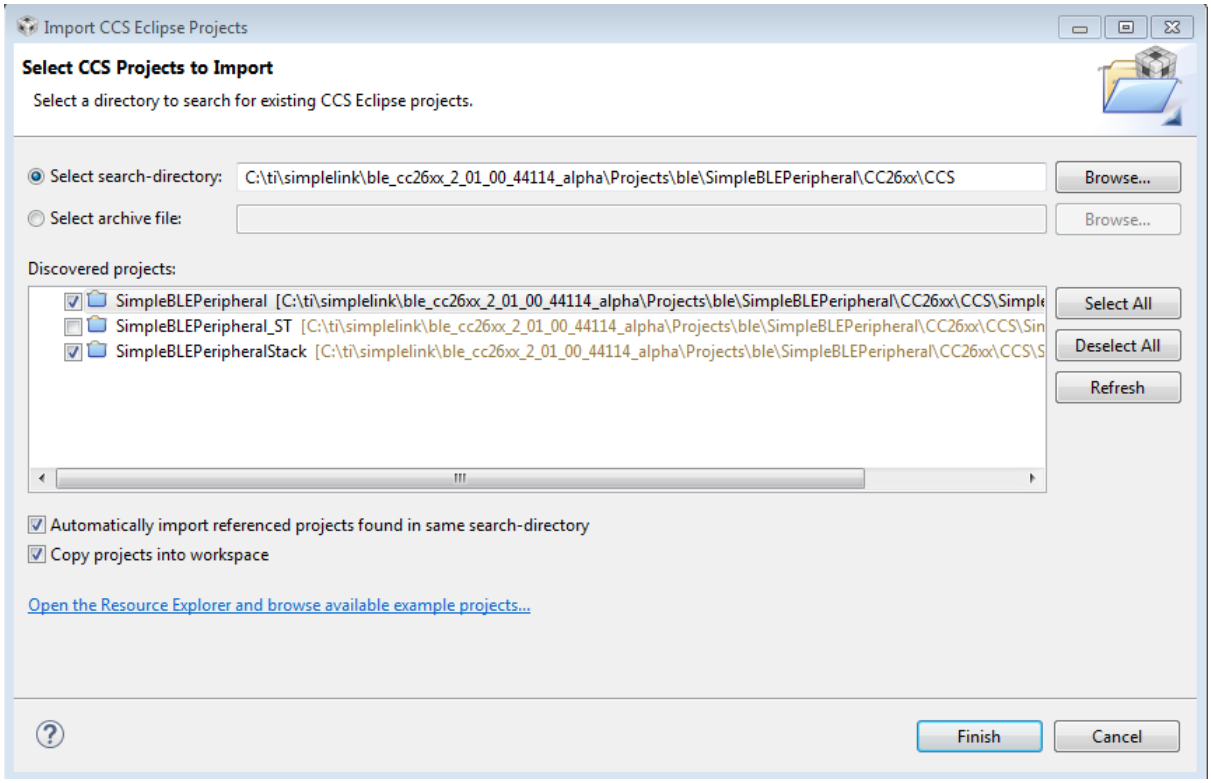
This section will describe how to open and build an existing project. Note that the SimpleBLEPeripheral project will be used as a reference. However, all of the CCS BLE projects included in the development kit will have a similar structure.

2.5.3.2.1 Import an Existing Project

First, open the CCS IDE from the Start Menu. Once CCS has opened up, click Project → Import CCS Project. Select the following directory:

$BLE_INSTALL$\Projects\ble Projects\ble\SimpleBLEPeripheral\CC26xx\CCS

This is the CCS directory for the SimpleBLEPeripheral project. CCS should discover three projects (the Application Sensor Tag, Application SmartRF, and Stack project). Check an application project (depending on your development platform) and the stack project. Then select "Copy projects into workspace". Finally select "Finish" to import:



2.5.3.2.2 Workspace Overview

This workspace, as well as all CC2640 project workspaces, will now contain two separate projects: the Application project (SimpleBLEPeripheral) and the Stack project (SimpleBLEPeripheralStack). One of these can be selected as the active project by clicking on the project name in the file explorer. In the figure below, the Application is selected as the active project. Each of these projects will produce a separate, downloadable image. This dual-image architecture was chosen so that the Application can be updated independent of the Stack.

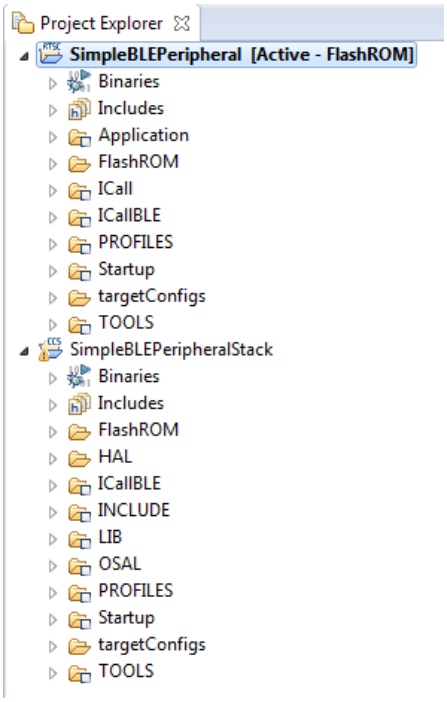


Figure 10: CCS Project Explorer Pane

For practical reasons, the SimpleBLEPeripheral sample project is made the primary reference target for the description of a generic application in this guide. The SimpleBLEPeripheral project implements a very basic BLE peripheral device including a GATT server with GATT services. This project can be used as a framework for developing many different peripheral-role applications.

2.5.3.2.3 Compiling 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 (i.e., from CC2640 to CC2650) may result in build errors. Note that all CC2640/CC2650 code is binary compatible & interchangeable for BLE-Stack software builds. Also, the CPU type is the same for all silicon package types.

Because the workspace is split into two projects (Application and Stack), there is a specific sequence for compilation and download:

1. Set the Application project as the active project and build the project using Project → Build All.
2. Set the Stack project as the active project and build the project using Project → Build All.
3. Download the Application by selecting the Application project as the active project and choosing Run → Debug. The Application can be debugged from this point.
4. Download the Stack by selecting the Stack project as the active project and choosing Run → Debug.

Note that the above procedure is only needed for the initial download and whenever the Stack project is modified. As long as the Stack project is not modified, the only steps required are:

1. Build the Application.
2. Download the Application.

2.6 Working with Hex Files

The Application & Stack projects are configured to each produce an Intel-extended hex file in the respective output folders. These hex files do not have overlapping memory regions, thus they can be individually programmed with a flash programming tool, such as SmartRF Flash Programmer 2. If desired, the Application & Stack hex files can be combined into a "super" hex file either manually or using freely available 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, make sure Python 2.7.x is installed and added to your system path environment variables.

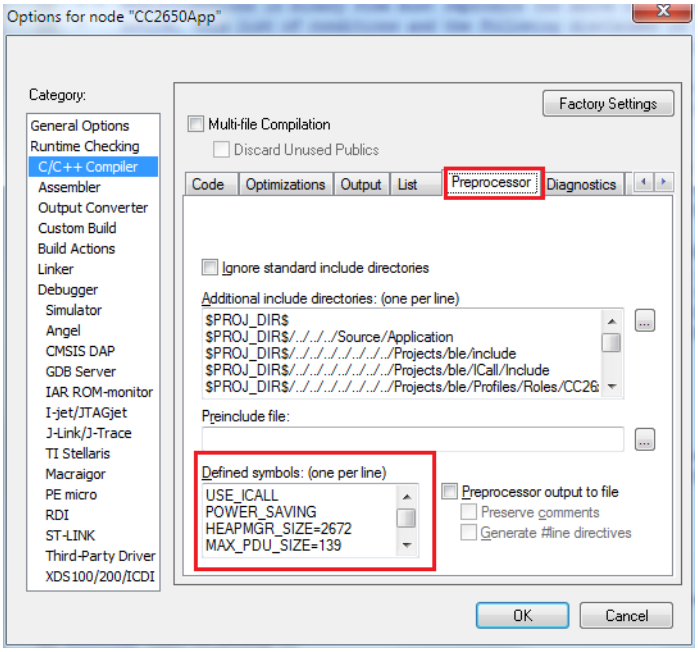
Example usage to create a merged *SimpleBLEPeripheral_merged.hex* file:

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

2.7 Accessing Preprocessor Symbols

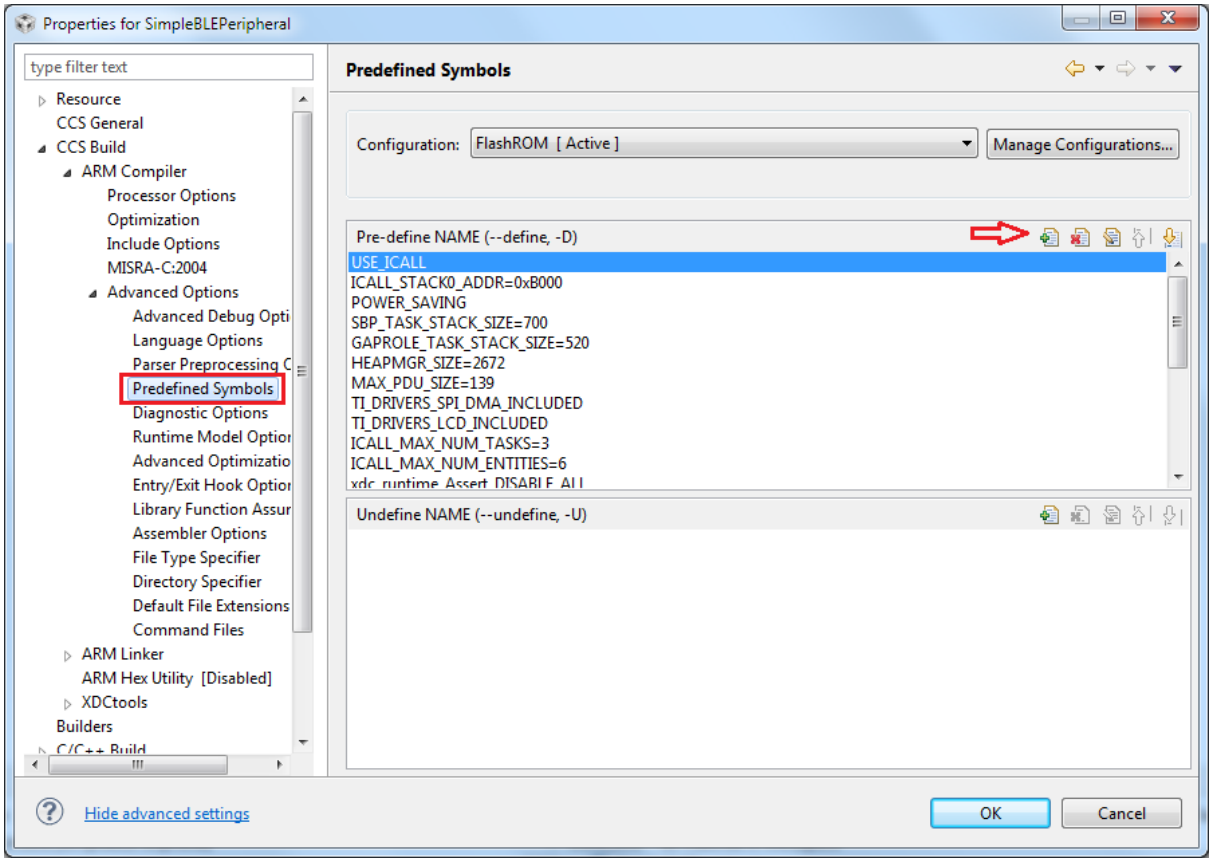
Throughout this document, various C preprocessor symbols may need to be set or adjusted at the project level. The procedure to access the pre-processor symbols (a.k.a "pre-defined symbols) is based on the IDE being used. The following procedure illustrates how to access and modify preprocessor symbols using IAR and CCS:

In IAR, preprocessor symbols are accessed by selecting & opening the respective Project Options (i.e., for Application or Stack project) in the C/C++ Compiler Category, on the "Preprocessor" tab, "Defined symbols" box:



Within the Defined symbols box, preprocessor symbols can be directly added or edited.

In CCS, preprocessor symbols are accessed by selecting & opening the respective Project Properties (i.e., for Application or Stack project), then navigating to CCS Build → ARM Compiler → Advanced Options → Predefined Symbols. To add, delete or edit a preprocessor symbol, use one of the buttons as shown by the arrow below:



2.8 Top-Level Software Architecture

At a top level, the CC2640 BLE software environment consists of three separate parts: a real-time operating system (RTOS), an Application image, and a Stack image. The TI-RTOS is a real-time, pre-emptive, multi-threaded operating system that runs the software solution with task synchronization. Both the Application and BLE protocol stack exist as separate tasks within the RTOS, with the BLE protocol stack having the highest priority. A messaging framework, Indirect Call (ICall), is used for thread-safe synchronisation between the Application and Stack. The architecture is illustrated in Figure 11.

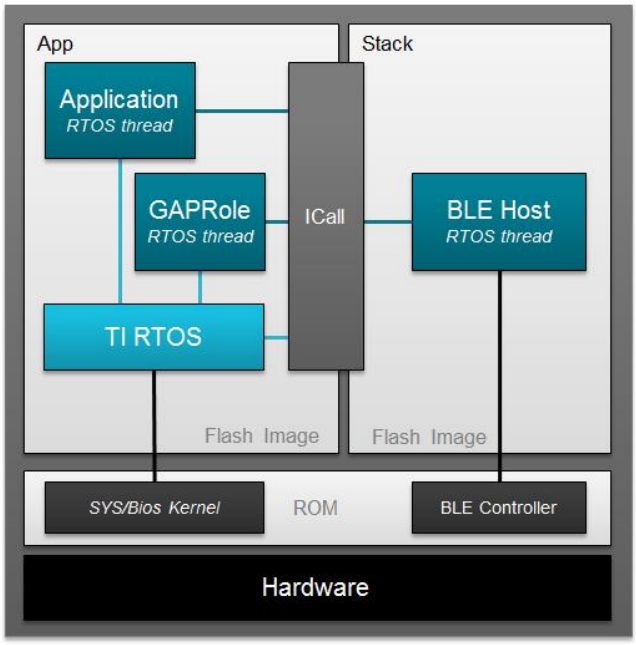


Figure 11: Top Level Software Architecture

- **The Stack image**: this includes the lower layers of the BLE protocol stack from the LL up to and including the GAP and GATT layers, as seen in *Figure 2*. Most of the BLE protocol stack code is provided as a library.
- **The Application image**: this includes the relevant profiles, application code, drivers, and the ICall module.

2.8.1 Standard Project Task Hierarchy

All projects will contain at minimum three RTOS tasks. Listed by priority such that a higher task number corresponds to a higher priority task, for the SimpleBLEPeripheral project, these tasks are:

- 5: BLE Protocol Stack task
- 3: GapRole task (peripheral role)
- 1: Application task (SimpleBLEPeripheral)

RTOS tasks in general are introduced in Section 3.3. Interfacing with the BLE protocol stack is described in Section 5, the GapRole task in described Section 5.2, and the Application task in Section 4.2.1.

3 RTOS Overview

TI-RTOS is the operating environment for BLE 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 multi-threaded operating system with tools for synchronization and scheduling (XDCTools). The SYS/BIOS kernel manages four distinct levels of execution threads: hardware interrupt service routines, software interrupt routines, tasks, and background idle functions as seen in *Figure 12*.

Note that the "TI-RTOS kernel" and "SYS/BIOS kernel" are used interchangeably and are the same.

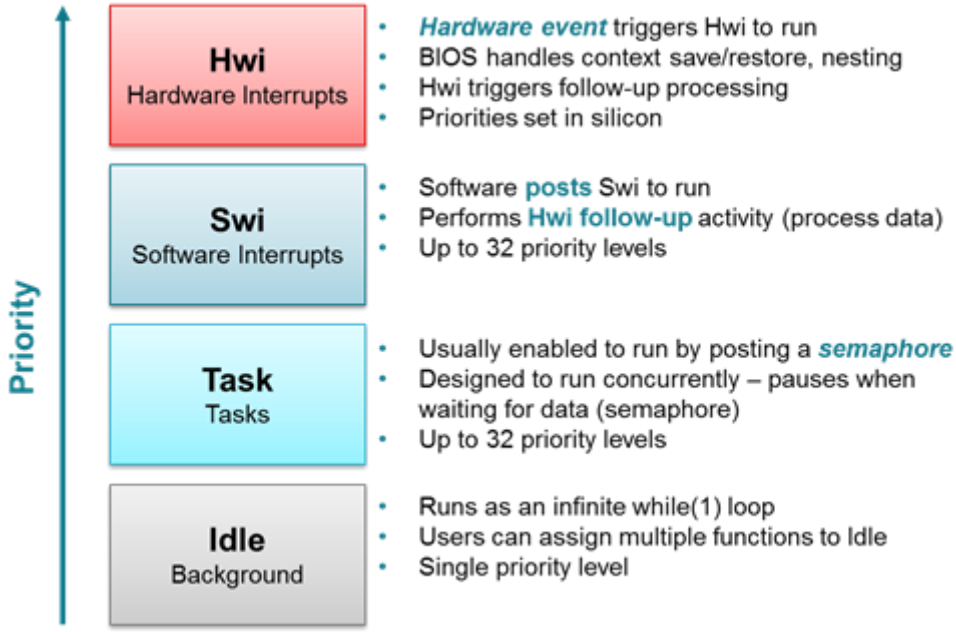


Figure 12: RTOS Execution Threads

This section will describe these four execution threads and various structures that are used throughout the RTOS for messaging and synchronization.

Note that in most cases the underlying RTOS functions have been abstracted to higher level functions in the *util.c* file. The lower-level RTOS functions are described in the SYS/BIOS module section, refer to [9]. This document also defines all of the packages & 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 (i.e. *appBLE.cfg* for the SimpleBLEPeripheral project). In the IAR project, this file is found in the Application project workspace TOOLS folder. This file defines the various SYS/BIOS and XDCTools modules that are included 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 place by recompiling the project. Note that the default project configuration is to use elements of the RTOS from the CC26xx ROM, thus some RTOS features are not available. If any features are added to the RTOS configuration file, but not supported in ROM, an "RTOS in Flash" configuration must be used. Using RTOS in Flash will consume additional flash memory. The default RTOS configuration supports all features needed by the respective example projects in the SDK.

Refer to the included TI-RTOS documentation for a full description of available configuration options.

Important: If any changes are made to the RTOS configuration file, delete the configPkg folder to force a rebuild of the RTOS. For example

$PROJ_DIR$\CC26xx\IAR\Application\CC2650\ConfigPkg

Next perform a Project → Rebuild All to rebuild the Application project 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. They are used to coordinate access to a shared resource among a set of competing tasks, i.e. the Application and BLE Stack. Semaphores are used for task synchronization and mutual exclusion.

Semaphore functionality is depicted in Figure 13. Semaphores can be counting semaphores or binary semaphores. Counting semaphores keep track of the number of times the semaphore has been posted with `Semaphore_post()`. This is useful, for example, when there are a group of resources that are shared between tasks. 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). They can be used to share a single resource between tasks. They can also be used for a basic signalling mechanism where the semaphore can be posted multiple times. Binary semaphores do not keep track of the count; they simply track whether the semaphore has been posted or not.

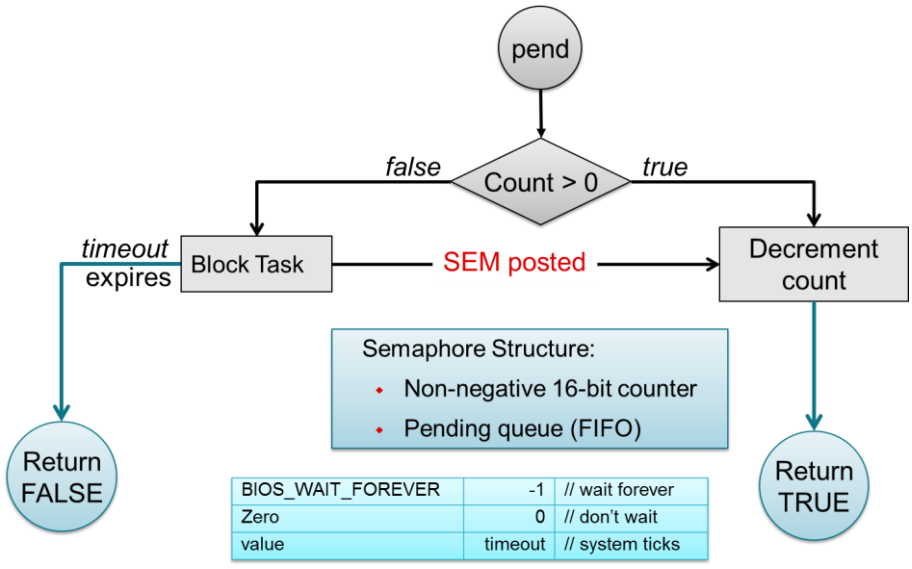


Figure 13: Semaphore Functionality

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 are used to coordinate task processing. Section 4.2 describes this in more detail.

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

3.2.2 Pending on a Semaphore

`Semaphore_pend()` is used to wait for a semaphore. It is a blocking call which will allow another task to run. The timeout parameter allows the task to wait until a timeout, wait indefinitely, or not wait at all. The return value is used to indicate if the semaphore was signaled successfully.

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

3.2.3 Posting a Semaphore

`Semaphore_post()` is used to signal 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()` simply increments the semaphore count and returns. For a binary semaphore, the count is always set to one.

```
Semaphore_post(sem);
```

3.3 Tasks

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

Each task is always in one of five modes of execution at any point in time:

- *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

There is always one (and only one) task currently *running*, even if it is only the idle task. The current 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* its own execution. In either case, the processor is switched to the highest priority task that is *ready* to run.

See the Task module in the SYS/BIOS API [9] "package ti.sysbios.knl" section for more information on these functions.

Numeric priorities are assigned to tasks and it is possible for multiple tasks to have the same priority. Tasks are readied for execution in strict priority order; tasks of the same priority are scheduled on a first-come, first-served basis. The priority of the currently running task is never lower than the priority of any ready task. Conversely, the running task is pre-empted and re-scheduled for execution whenever there exists some ready task of higher priority. In the SimpleBLEPeripheral application, the BLE 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 generally one or more callback functions.

3.3.1 Creating a Task

When a task is created, it is provided with its own run-time stack which is used for storing local variables as well as for 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 defined for C functions. This set of memory is referred to as the task's context.

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 begins execution, at its assigned priority level, after the scheduler is started.

Although it is recommended to use the existing application task for application-specific processing, if another task is required certain guidelines should be followed. When adding an additional task to the Application project, the task's priority 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;
```

Additionally, do not add a task with a priority that is equal to or higher than the BLE protocol stack task and related supporting tasks (e.g., GapRole task). See Section 2.8.1 for details on the system task hierarchy.

Finally, the task should be *constructed* with 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 gets saved when one task preempts another as a result of an interrupt thread readying a higher-priority task. Using the IDE's TI-RTOS profiling tools, the task can be analyzed to determine peak task stack usage.

Note that the term "created" is used to describe the instantiation of a task; however, the actual TI-RTOS method is to construct the task. Refer to Section 3.11.6 for details on constructing RTOS objects.

3.3.2 Creating the Task Function

As seen above, when a task is constructed, a function pointer to a task function (`SimpleBLEPeripheral_taskFxn`) is passed to the `Task_Construct` function. This is the function which the RTOS will run when the task first gets a chance to process. The general topology of this task function is depicted in Figure 14:

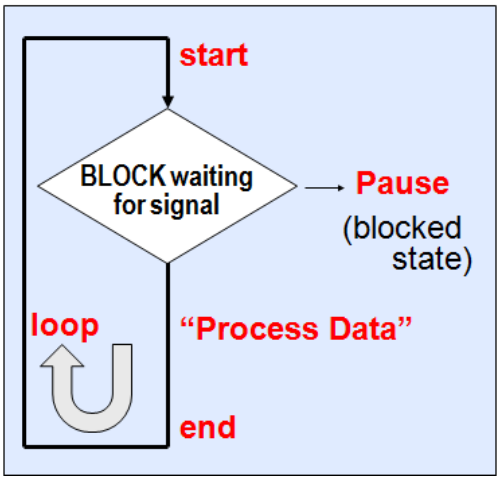


Figure 14: General Task Topology

In the SimpleBLEPeripheral task, for example, the task will likely spend most of its time in the blocked state where it is pending on a semaphore. Once its semaphore has been posted to from an ISR, callback function, queue, etc, it will become ready, then process the data and eventually return to this paused state. See Section 4.2.1 for more detail on the SimpleBLEPeripheral task functionality.

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. They are started immediately upon creation or configured to start after a delay. They can also be stopped at any time. All Clock Instances are executed when they expire in the context of a software interrupt.

The minimum resolution is the RTOS clock tick period which is set in the RTOS configuration:

```
/* 10 us tick period */
```

```
Clock.tickPeriod = 10;
```

Each tick, which is derived from the RTC, launches a Clock SWI which compares the running tick count with each Clock's period to determine if the associated function should run.

For higher resolution timers, it is recommended to use a 16-bit hardware timer channel or the Sensor Controller.

3.4.1 API

It is possible to use the RTOS clock module functions directly (see the Clock module in the SYS/BIOS API 0). For usability these 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)

Description: 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 funciton

Returns:
handle to the Clock instance

void Util_startClock(Clock_Struct *pClock)

Description: Start an (already constructed) clock.

Parameters:
pClock – pointer to clock structure

bool Util_isActive(Clock_Struct *pClock)

Description: 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)

Description: stop a clock.

Parameters:
pClock – pointer to clock structure

3.4.2 Functional Example

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

1. Define 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.
  Semaphore_post(sem);
```

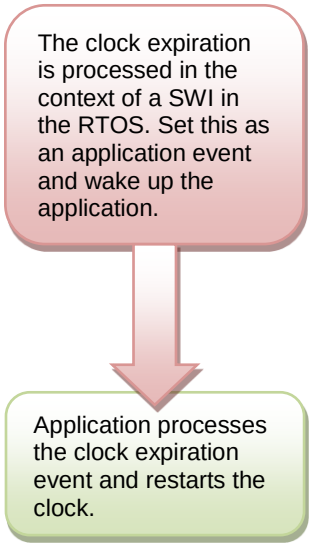
```
}
```

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 clock instance to expire and process in the application context. This is shown in the flow diagram below: green corresponds to the processor running in the application context and red corresponds to a 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 allow applications to process events in an ordered manner 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. Whereas Clocks should 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.

The Queue module provides a unidirectional method of message passing between tasks using a FIFO. In Figure 11, 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 15: Queue Messaging

3.5.1 API

The RTOS queue functions have been abstracted into functions in the *util.c* file. See the Queue module in the SYS/BIOS API [9] for the underlying functions. These utility functions combine the Queue module with the ability to notify the recipient task of an available message via semaphores. In the CC2640 software, the semaphore used for this is the same semaphore that the given task uses for task synchronization via ICall. See the `SimpleBLECentral_enqueueMsg()` function for an example of this.

Queues are commonly used to limit the processing time of application callbacks in higher level priority task's contexts. 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. This is further described in Section 3.5

Queue_Handle Util_constructQueue(Queue_Struct *pQueue)
Description: 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)
Description: Creates a queue node and puts the node in an RTOS queue. Parameters: *msgQueue* – queue handle *sem* – task's event processing semaphore that the queue is associated with *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)
Description: Dequeues the message from an RTOS queue. Parameters: *msgQueue* – queue handle Returns: NULL: no message to deuque otherwise: pointer to dequeued message.

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 task's enqueue function so that it uses the task's semaphore to wake itself up.

```
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.

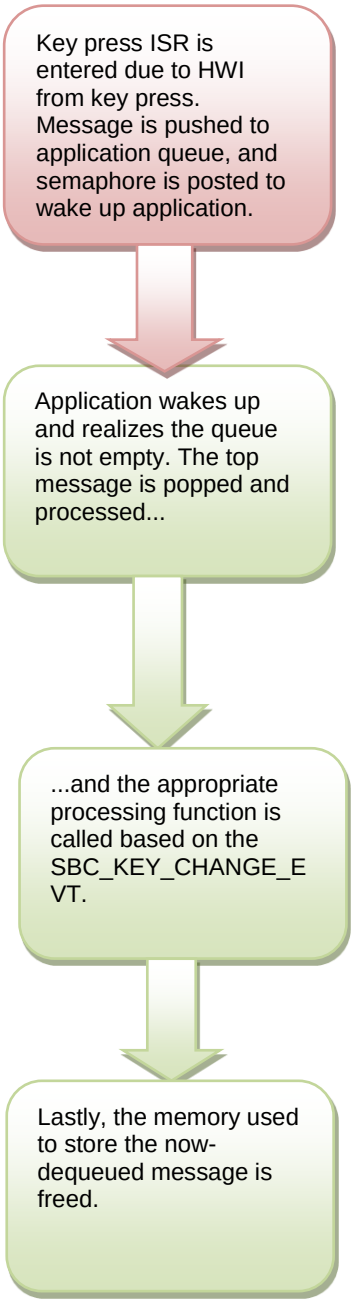
```
// Queue object used for app messages
```

```
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. This is shown in the flow diagram below: green corresponds to the processor running in the application context and red corresponds to a HWI.




```
void SimpleBLECentral_keyChangeHandler(uint8 keys)
{
  SimpleBLECentral_enqueueMsg(SBC_KEY_CHANGE_EVT, keys,
    NULL);
}
```

```
// If RTOS queue is not empty, process app message
if (!Queue_empty(appMsgQueue))
{
   sbcEvt_t *pMsg =
   (sbcEvt_t*)Util_dequeueMsg(appMsgQueue);
   if (pMsg)
   {
     // Process message
     SimpleBLECentral_processAppMsg(pMsg);
     ...
```

```
static void SimpleBLECentral_processAppMsg(sbcEvt_t *pMsg)
{
  switch (pMsg->event)
  {
    ...
    case SBC_KEY_CHANGE_EVT:
      SimpleBLECentral_handleKeys(0, pMsg->status);
      break;
    ...
  }
}
```

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

3.6 Idle Task

The Idle module is used to specify a list of functions to be called when no other tasks are running in the system. In the CC2640 software, the Idle task is responsible for running the Power Policy Manager.

3.7 Power Management

In general, all power management functionality is handled by the peripheral drivers and the BLE protocol stack. This feature can be enabled / disabled by including / excluding the POWER_SAVING preprocessor define symbol. When POWER_SAVING is enabled, the device will come in and out of sleep as is needed for BLE events, peripheral events, application timers, etc. When POWER_SAVING is not defined, the device will stay awake. See Section 9.2 for steps to modify preprocessor defines.

More information on power management functionality including the API and a sample use case for a custom UART driver can be found in the Power Management User's Guide [8] included in the RTOS install. Note that these API's will only likely be necessary in the case of a custom driver.

Also see the Measuring Power Consumption App Note [3] for steps to analyze the system power consumption and battery life.

3.8 Hardware Interrupts (HWI's)

Hardware interrupts (HWIs) handle critical processing that the application must perform in response to external asynchronous events. The SYS/BIOS target/device specific HWI modules are used to manage hardware interrupts. Specific information on the nesting, vectoring, and functionality of interrupts can be found in the Technical Reference Manual [2]. Furthermore, the SYS/BIOS User Guide details the HWI API and provides several software examples.

In general, HWI's are abstracted through the peripheral driver they pertain to (see the relevant driver in Section 6). An example of using GPIO's as HWI's can be found in Section [9]. This is the preferred method of using interrupts. Using the `Hwi_plug()` function, it is possible to write ISR's which do not interact with SYS/BIOS. However, these ISR's must do their own context preservation in order to prevent breaking the time-critical BLE Stack.

In order for the BLE protocol stack to meet RF time-critical requirements, all application-defined HWIs execute at the lowest priority. For this reason, it is not recommended to modify the default HWI priority when adding new HWIs to the system.

In general, there should be no application-defined critical sections in order to prevent breaking the RTOS or time-critical sections of the BLE protocol stack. Code executing in a critical section will prevent processing of real-time interrupt related events.

3.9 Software Interrupts (SWI's)

See the SYS/BIOS User Guide for an API and detailed information about the SWI module. Software interrupts have priorities that are higher than tasks but lower than hardware interrupts (See Figure 16). Therefore, the amount of processing done in a SWI should be extremely limited as this processing will take priority over the BLE protocol stack task. As described in Section 3.4, the clock module uses SWI's 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 SWI's. If necessary, a SWI can be implemented with the SWI module as described in the SYS/BIOS User Guide.

Note that, in order to preserve the RTOS heap, the amount of dynamically created SWI's should be limited as described in Section 3.11.6.

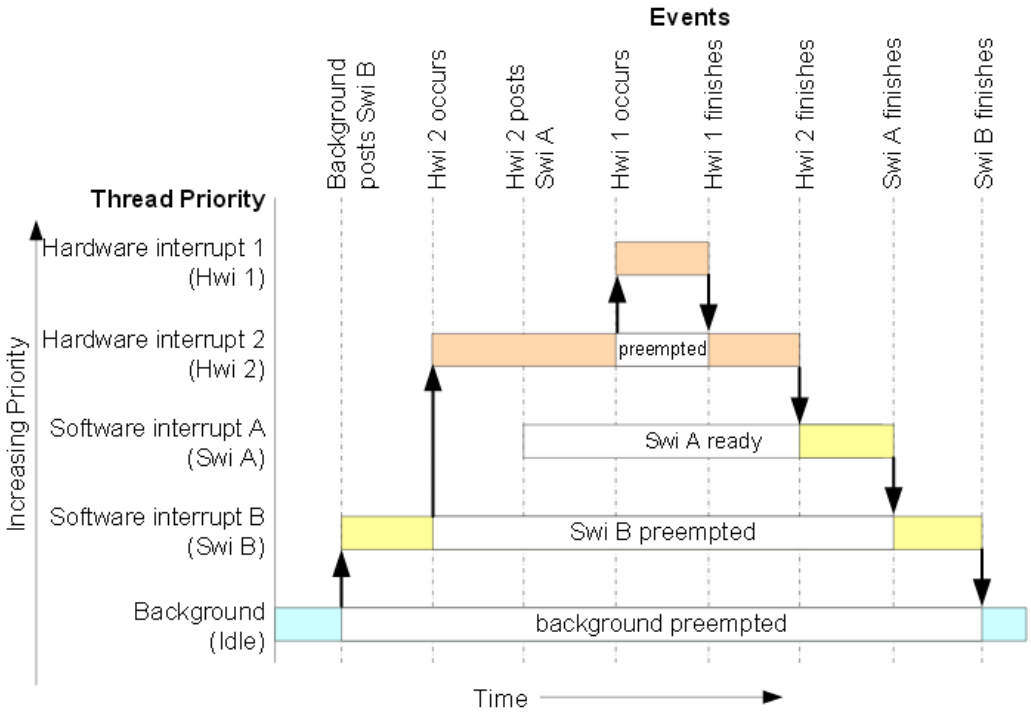


Figure 16: Preemption Scenario

3.10 Flash

The flash is split into erasable pages of 4 kB. Additionally, the Application & Stack projects must each start on a 4kB aligned flash address. The various sections of flash and their associate linker files are:

- Application Image: code space for the Application project. This is configured in the Application's linker config file: *cc26xx_ble_app.icf* (IAR) and cc2650f128_tirtos_ccs.cmd (CCS).
- Stack Image: code space for the Stack project. This is configured in the Stack's linker config file: *cc26xx_ble_stack.icf* (IAR) and *cc2650f128_tirtos_ccs_stack.cmd* (CCS).
- Simple NV (SNV): area used for non-volatile memory storage by the GAP Bond Manager and also available for Application use. 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.

3.10.1 Flash Memory Map

This section will illustrate the flash memory map at the system level. Symbols pointed with a solid arrow can be found in the Application linker file, and symbols pointed with a dashed arrow can be found in the Stack linker file.

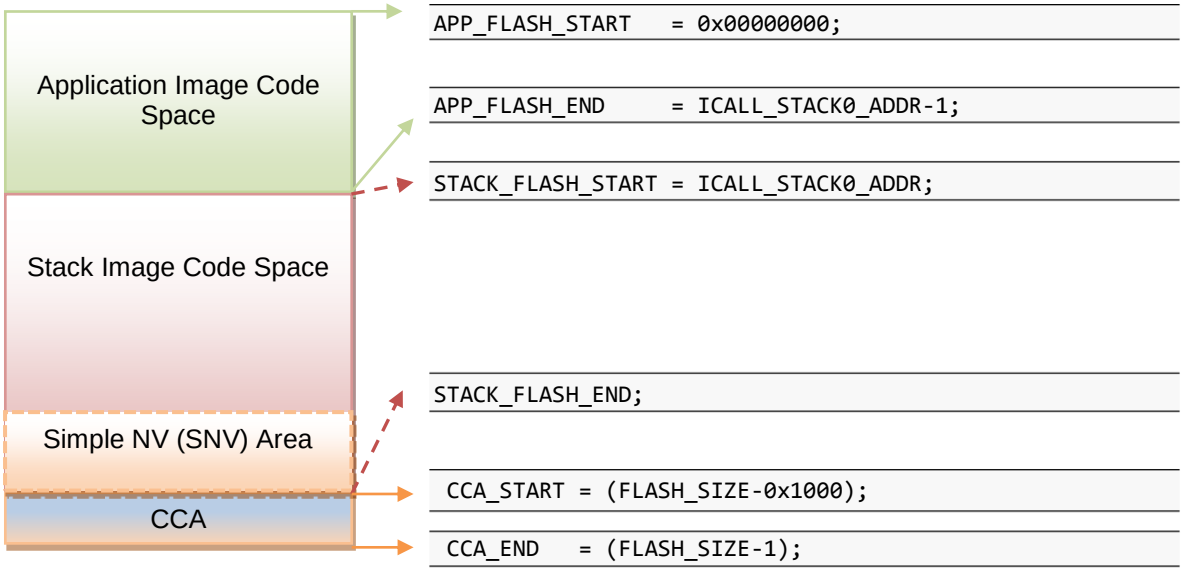


Figure 17: System Flash Map

The following table summarizes the Flash System Map definitions from Figure 17 and provides the associated linker definitions / symbols that can be found in the respective IDE linker files.

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. (ICALL_STACK0_ADDR-1)	App	ICALL_STACK0_ADDR - APP_BASE - 1	FLASH_END
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

3.10.2 Application / Stack Flash Boundary

The Application & Stack code images are based on the common ICALL_STACK0_ADDR define. 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 & Stack projects must use the same ICALL_STACK0_ADDR defined value. By default, ICALL_STACK0_ADDR is configured to allocate unused flash to the Application project but can be modified manually or automatically via the Boundary Tool. Refer to Section 3.10.3 for manually modifying the flash boundary address and Section 3.12 for using the Boundary Tool to configure the flash boundary address.

3.10.3 Manually Modifying Flash Boundary

As mentioned above, the Boundary Tool is used to adjust the ICALL_STACK0_ADDR Application-Stack Flash boundary such that the maximum amount of flash memory is allocated to the Application project. Although not typically required, the ICALL_STACK0_ADDR can be manually adjusted by performing the following steps:

Procedure 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 both Application & Stack projects

3. Adjust ICALL_STACK0_ADDR in TOOLS/*IAR-Boundary.bdef*:

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

Note: This file is shared with both Application & Stack projects

4. Rebuild the Application & Stack projects. Verify there are no build errors in both projects.

Procedure for CCS:

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/*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 & Stack projects. Verify there are no build errors in both projects.

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* & *IAR-Boundary.bdef* files for IAR, or in the *ccsLinkderDefines.cmd* & preprocessor symbol for CCS.
- Both Application & Stack projects must be rebuilt when ICALL_STACK0_ADDR is modified. Always rebuild the Stack project until it builds & links correctly, then rebuild the Application.
- If a linker error occurs after manually adjusting ICALL_STACK0_ADDR, verify that each project has adequate flash memory allocated.

3.10.4 Using Simple NV (SNV)

The SNV area of flash is used for storing persistent data in a secure manner, 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(s) 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 below, 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. The following table lists the valid values that can be configured as well as the corresponding trade-offs.

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. Refer to Section 10.4 for configuring BLE 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 will result in SNV data loss. Also, due to temporarily disabling the cache, a system performance degradation may occur during the compaction. Set pre-processor symbol OSAL_SNV=1 in the Stack project.
2	Default value. Two flash sectors are allocated to SNV. Bonding info is stored in NV. SNV data is protected against power-loss during compaction.

All other values are not valid. 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 / written to using the following API's:

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

Description: 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

Since SNV is shared with other modules in the BLE SDK such as the GapBondMgr, it is necessary to carefully manage the NV item ID's. By default, the ID's 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 (CCA)

The Customer Configuration Area (CCA) occupies the last page of flash and allows the customer to configure various chip and system parameters in the Customer Configuration Table (CCFG). The CCFG table is defined in *ccfg_appBLE.c* which can be found in the Application project's Startup folder. 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/data use, however 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 Application's linker file by the `FLASH_LAST_PAGE` symbol; placement is based on the IDE:

CCS:

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

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 };
```

Refer to the TRM [2] for details on CCFG fields & related configuration options.

3.11 Memory Management (RAM)

Similar to Flash, the RAM is shared between the Application & Stack projects. The RAM sections are configured in their respective linker files:

- Application Image: RAM space for the Application & shared heaps. This is configured in the Application's linker config file: *cc26xx_ble_app.icf* (IAR) and cc2650f128_tirtos_ccs.cmd (CCS).
- Stack Image: RAM space for the stack's .bss & .data sections. This is configured in the Stack's linker config file: *cc26xx_ble_stack.icf* (IAR) and *cc2650f128_tirtos_ccs_stack.cmd* (CCS).

3.11.1 RAM Memory Map

Figure 18 shows the system memory map for the default SimpleBLEPeripheral project. Note that this is a summary and the exact memory placement for a given compilation can be found in the *SimpleBLEPeripheralApp.map* & *SimpleBLEPeripheralStack.map* files located under the Output folder in IAR, or the FlashROM folder in CCS. See Section 0 for more information about these files. In the figure below, Symbols pointed with a solid arrow can be found in the Application linker file, and symbols pointed with a dashed arrow can be found in the Stack linker file.

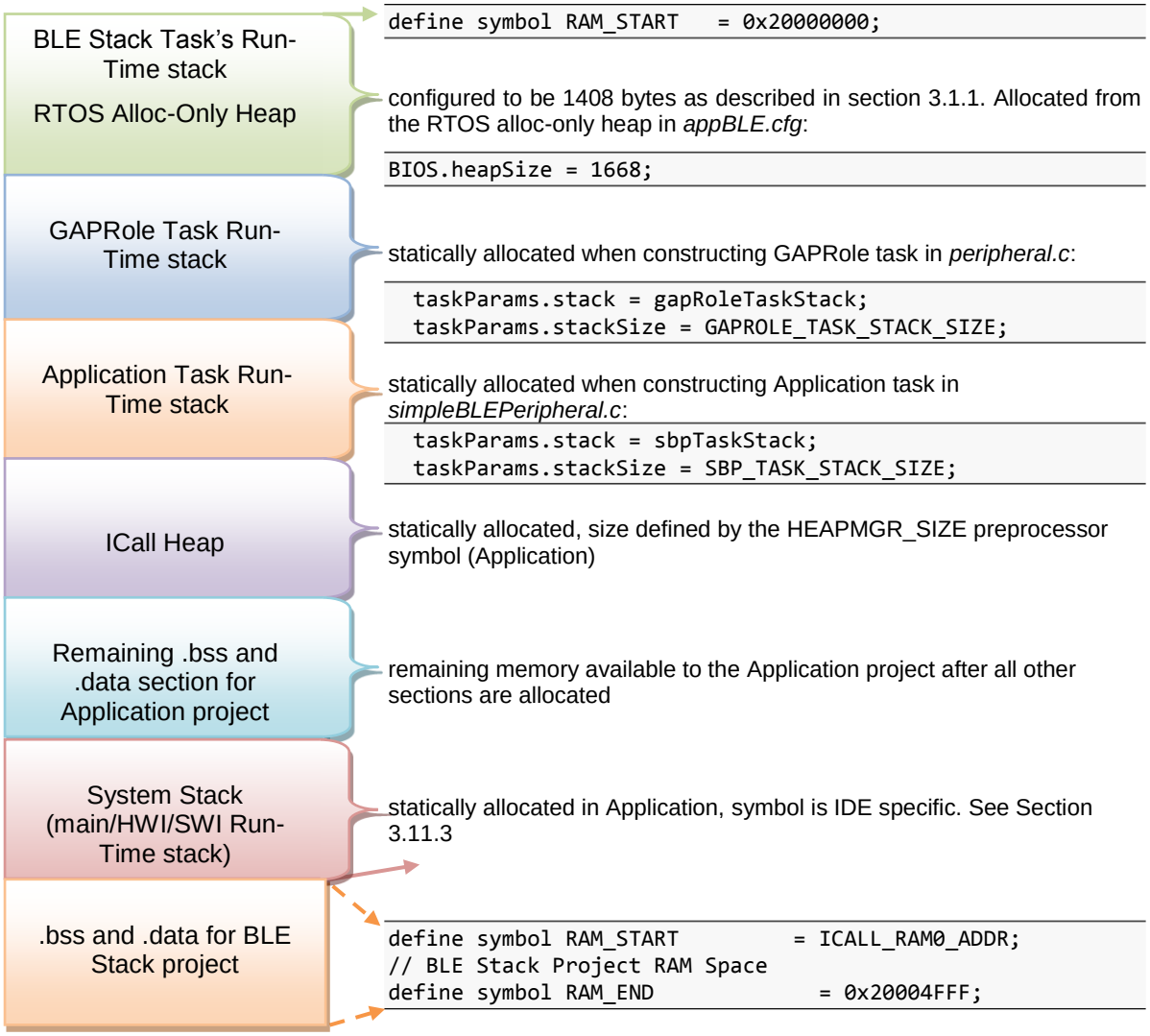


Figure 18: System Memory Map

3.11.2 Application / Stack RAM Boundary

The Application & Stack RAM memory maps are based on the common ICALL_RAM0_ADDR define. This value defines the hardcoded RAM boundary for the end of the Application's RAM space and the start of the Stack image's .BSS & .DATA sections. Note that unlike the flash boundary, elements of the Stack project, such as task stacks & heaps, are allocated in the Application project. To ensure proper linking, both the Application & 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 via the Boundary Tool. Refer to Section 0 for manually modifying the RAM boundary address and Section 3.12 for using the Boundary Tool to configure the RAM boundary address.

3.11.3 System Stack

Besides the RTOS & ICall heaps mentioned above, there are other sections of memory to consider. As described in Section 3.3.1, each task has its own run-time stack for context switching. Furthermore, there is another run-time stack used by the RTOS for main(), HWI's and SWI's. This system stack is allocated in the Application linker file to be placed at the end of the Application's RAM.

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 Application's linker file.

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 Application's RAM space:

```
/* 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

As mentioned above, 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:

Procedure 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 & Stack projects. Verify there are no build errors in both projects.

Procedure 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 & Stack projects. Verify there are no build errors in both projects.

Some points to remember when modifying ICALL_RAM0_ADDR:

- The ICALL_RAM0_ADDR value must be a 4-byte aligned address.
- Both Application & Stack projects must be cleaned & rebuilt when ICALL_RAM0_ADDR is modified. Always rebuild the Stack project until it builds & 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.

3.11.5 Dynamic Memory Allocation

The system uses two heaps for dynamic memory allocation. It is important to understand the use of each heap so that the application designer maximizes 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 for initializing RTOS objects as well as allocating the BLE protocol stack's task run-time stack. This size of this heap has been chosen to meet the system initialization requirements. Due to the small size of this heap, it is not recommended to allocate memory from the RTOS heap for general application use. For more information on the TI-RTOS heap configuration, refer to the "Heap Implementations" section of the SYS/BIOS User Guide [7].

Instead, there is a separate heap which should be used by the Application. 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 Application's preprocessor definition HEAPMGR_SIZE and is set to 2672 by default for the SimpleBLEPeripheral project. Although the ICall heap is defined in the Application project, it is also shared with the BLE protocol stack. APIs which 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 utilized, define the HEAPMGR_METRICS preprocessor symbol in the Application project. Refer to heapmgr.h in $BLE_INSTALL$\Components\applib\heap for available heap metrics.

Here 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
}
```

Here is an example of freeing the above array:

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

3.11.6 A Note on Initializing RTOS Objects

Due to the limited size of the RTOS heap, it is strongly recommended to "construct" and not "create" RTOS objects. To illustrate this, consider the difference between the Clock_construct() and Clock_create() functions. Here are 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. Conversely, Clock_create() would cause the RTOS to allocate the Clock_Struct using the RTOS's limited heap.

As much as possible, this is how clocks and RTOS objects in general, should be initialized throughout one's project. If creating RTOS objects must be used, the size of the RTOS heap may need to be adjusted in *appBLE.cfg*.

3.12 Configuration of RAM & Flash boundary using the Boundary Tool

The Boundary Tool is utility used to adjust the respective ICALL_STACK0_ADDR (Flash) and ICALL_RAM0_ADDR (RAM) boundaries shared between the Application & Stack projects. The Boundary Tool adjusts the boundaries such that unused Flash & RAM is allocated to the Application

project. This tool removes the need 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. Additionally, the Boundary Tool does NOT modify any source code nor perform any compiler / linker optimization; the tool simply adjusts the respective Flash & RAM boundary address(es) based on analysis of the project's map & linker configuration files.

The Boundary Tool is installed to the following path:

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

A ReadMe.txt file is located in this path and contains additional information about the tool.

3.12.1 Configuring Boundary Tool

The Boundary tool uses a XML file, *BoundaryConfig.xml,* located in the tool's install path to configure default tool options. It is recommended to keep these default values.

Each project in the SDK has a set of config files which are used by the IDE's linker & compiler to set/adjust the respective Flash & RAM values. These config files are located in each project at the following location:

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

Where <PROJECT> is the project (e.g. SimpleBLEPeripheral) and <IDE> is either IAR or CCS.

- Boundary Linker Config File: *IAR-Boundary.xcl* [IAR] or *ccsLinkerDefines.cmd* [CCS]. Used to define the ICALL_STACK0_ADDR & ICALL_RAM0_ADDR boundary address. This file is located in the TOOLS IDE folder and 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 & CCS linker, ICALL_STACK0_ADDR must also be defined in this file to the same value as the linker config file. This file is located in the TOOLS IDE folder and updated by the Boundary tool when an adjustment is required.

3.12.2 Boundary Tool Operation

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 needed, the Boundary Tool updates the Boundary Linker Config and C Definition Files listed above and generates a post-build error to notify that a change was performed. To incorporate the updated configuration values, perform a "Project → Rebuild All" on Stack, followed by the Application. It is imperative that the Stack project builds & links correctly before the Application can be rebuilt.

In addition to the code & 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 Stack project's linker config file.

A prerequisite to using the Boundary Tool is both the Application & Stack projects must first compile & link successfully. If a linker error occurs, first verify that the change did not exceed the maximum Flash and/or RAM size of the device. Typically 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 & Flash boundary addresses to their maximum values in the Boundary Linker Config and C Definition Files:

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

Note: Refer to Sections 3.10.3 & 3.11.4 for adjusting the Flash & RAM values. These values are only used temporarily to allow the *Stack* project to link successfully.

Once the *Stack* project is able to link successfully, the Boundary Tool will re-adjust the respective boundaries to the optimal value by generating a build error. Perform a "Rebuild All" as required in both projects.

Note: Unless IAR is configured to Show All Build Messages, there will be no reason given as to why the Boundary Tool generated a build error. When IAR is configured to show all build messages, "The Project Boundary Address Has Been Moved" will be displayed in the Build output window. In IAR set Tools → Options → Messages → Show Build Messages to "All".

Here 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 Boundary Tool

To Disable the Boundary Tool, open the project options for the *Stack* project select "Build Actions" (IAR) or "Steps" on the CCS Build window (CCS) and remove the Post-build command line as shown in Figure 19. It is advised to keep a copy of this command in case the Boundary Tool needs to be restored at a later time. Once the Boundary Tool is disabled, the respective Boundary linker & compiler config files can be manually edited as per the procedures in Sections 3.10.3 & 3.11.4.

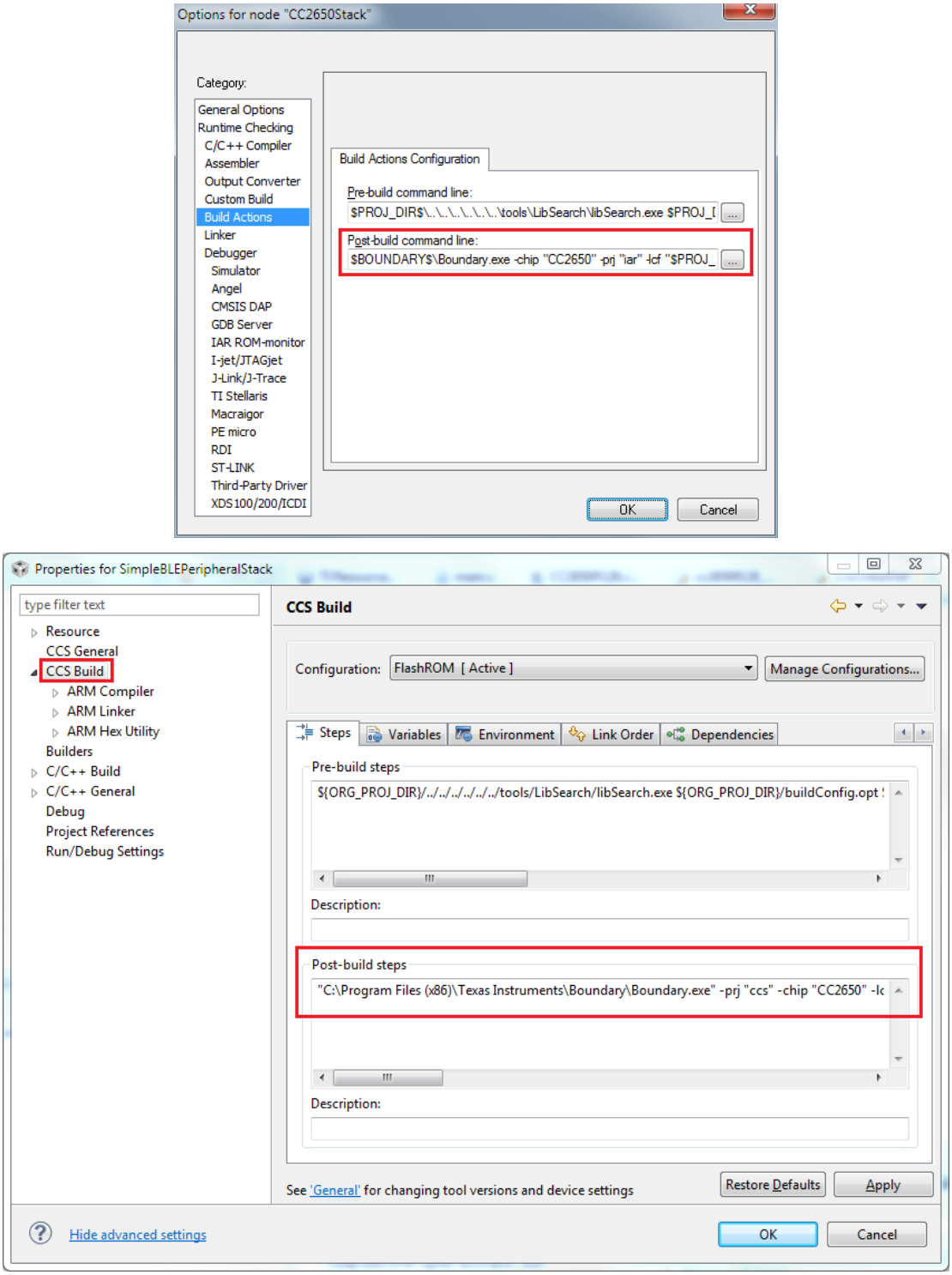


Figure 19: Disabling Boundary Tool from Stack Project in IAR (top) & CCS (bottom)

4 The Application

This section will describe the application portion of the SimpleBLEPeripheral project which includes:

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

Note that the GAPRole task is also part of the Application project workspace. However, this task's functionality relates more closely to the protocol stack so it is described in Section 5.2.

4.1 Start-up in main()

The `main()` function inside of *main.c* in the IDE "Startup" folder is the starting point at runtime. This 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 / 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 Section 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;
}
```

The Application and GAPRole tasks are constructed as described in Section 3.3. The Stack task is created here as well in `ICall_createRemoteTasks()`. The ICall module is initialized via `ICall_init()`. In terms of the IDE workspace, *main.c* exists in the Application project – meaning that when it is compiled it is placed in the application's allocated section of Flash.

4.2 ICall

4.2.1 Introduction

Indirect Call Framework (ICall) is an module that provides a mechanism for the Application to interface with the BLE protocol stack services (i.e, BLE-Stack APIs) as well as certain primitive services provided by the RTOS (e.g., thread synchronization). Combined, ICall allows both the Application and protocol stack to efficiently operate, 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 BLE protocol stack task across the dual-image boundary. Although most of the ICall interactions are abstracted within the BLE protocol stack APIs (e.g., GAP, HCI, etc.), it is important for the application developer to understand the underlying architecture such that proper BLE protocol stack operation is achieved in the multi-threaded RTOS environment.

The ICall module source code is provided in the "ICall" & "ICallBLE" IDE folders within the Application project.

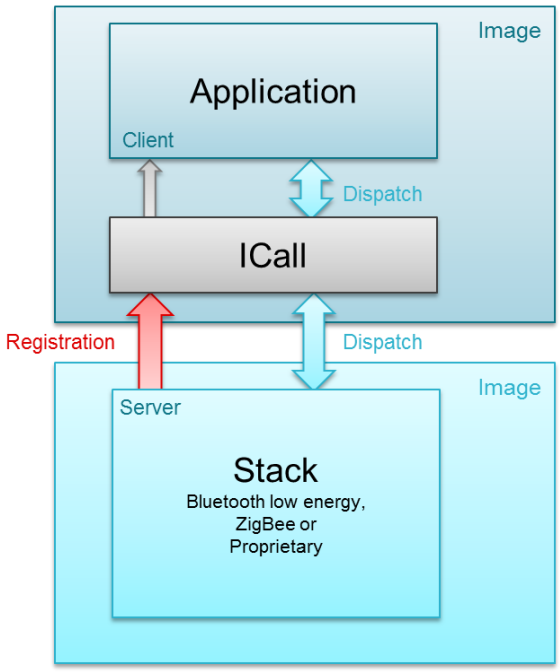


Figure 20: ICall Application - Protocol Stack Abstraction

4.2.2 ICall BLE Protocol Stack Service

As depicted in Figure *20*, the ICall core use case involves messaging between a server entity (i.e. the BLE Stack task) and a client entity (e.g. the Application task). Note that the ICall framework is not to be confused with the GATT server/client architecture as defined by the BLE protocol. The reasoning for this architecture is twofold: to enable independent updating of the application / BLE protocol stack and to maintain API consistency as the software is ported from legacy platforms (e.g. OSAL for the CC254x) to the CC2640's TI-RTOS.

The ICall BLE Protocol Stack Service serves as the Application interface to all BLE-Stack APIs. Internally, when a BLE protocol stack API is called by the Application, the ICall module routes (i.e. dispatches) the command to the BLE protocol stack, and where appropriate, routes messages from the BLE protocol stack to the Application.

Since the ICall module is part of the Application project, the Application task can access ICall with direct function calls. Note that since the BLE protocol stack executes at the highest priority, the Application task will block until the response is received. Certain protocol stack APIs may respond immediately, however the Application thread will block as the API is being dispatched to the BLE protocol stack via ICall. Other BLE protocol stack APIs may also respond asynchronously to the application via ICall (e.g. event updates) with the response sent to the application task's event handler.

4.2.3 ICall Primitive Service

In addition, ICall includes a primitive service that abstracts various operating system related functions. Due to shared resources and to maintain interprocess communication, the Application should use the following ICall primitive service functions:

- Messaging and Thread Synchronization
- Heap Allocation / Management

Some of these are abstracted to Util functions: see the relevant module in Section 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 multi-threaded RTOS environment.

Within ICall, messaging between two tasks is achieved by sending a block of message from one thread to the other via a message queue. The sender allocates a memory, writes the content of the message into the memory block and then sends (i.e, enqueues) the memory block to the recipient. Notification of message delivery is accomplished using a signalling semaphore. The receiver wakes

up on the semaphore, copies the message memory block(s), processes the message and returns (frees) the memory block to the heap.

The Stack uses ICall for notifying and sending messages to the Application. These "service messages" (e.g. state change notifications, etc.) are received by the Application task are delivered by ICall and processed in the Application's task context.

4.2.3.2 Heap Allocation and Management

ICall provides the Application with a global heap APIs for dynamic memory allocation. The size of the ICall heap is configured with the HEAPMGR_SIZE preprocessor define in the Application project. Refer to section 3.11.5 for more details on dynamic memory management. ICall uses this heap for all protocol stack messaging as well as to obtain memory for other ICall services. It is also recommended that the Application utilize these ICall APIs for dynamic memory allocation within the Application.

4.2.4 ICall Initialization and Registration

To instantiate and initialize the ICall service, the following functions must be called by the application in `main()` before to 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 (e.g., heap manager) and framework. Calling `ICall_createRemoteTasks()` creates, but does not start, the BLE protocol stack task.

Prior to using ICall protocol services, both the server and client must enroll and register with ICall. The server enrolls a service which is enumerated at build time. Service function handler registration uses a globally defined unique identifier for each service. For example, BLE uses ICALL_SERVICE_CLASS_BLE for receiving BLE protocol stack messages via ICall.

The following is a call to enroll the BLE 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 via the ICall dispatcher.

In order for a client (e.g., Application task) to utilize the BLE-Stack APIs, it must first register its task with ICall. This registration is usually done in the application's task initialization function. Here 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 which, upon return of `ICall_registerApp()`, are initialized for the client's (e.g., Application) task. These objects are subsequently used be ICall to facilitate messaging between the Application and server tasks.

The `sem` argument represents the semaphore used for signalling, whereas the `selfEntity` represents the task's destination message queue. Each task registering with ICall will have unique `sem` and `selfEntity` identifiers.

Note: BLE protocol stack APIs defined in *ICallBLEApi.c*, and other ICall primitive services, are not available prior to ICall registration.

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, as mentioned previously, are not blocking functions. That is, they will check whether there is a received message in the queue and if there is no message, the functions will return immediately with ICALL_ERRNO_NOMSG return value. To allow a client or a server thread to block till it receives a message, ICall provides the following function which will block till 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 timeout period in milliseconds, after which if the function is not already returned, the function will return with ICALL_ERRNO_TIMEOUT. If ICALL_TIMEOUT_FOREVER is passed as "milliseconds", the ICall_wait() shall block forever till the semaphore is posted. Allowing an application or a server thread to block is important in order to yield the processor resource to other lower priority threads or to conserve energy by shutting down power and/or clock domains whenever possible.

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

- A new message is queued to the Application's RTOS thread queue.
- `ICall_signal()` is called for the semaphore

`ICall_signal()` is provided so that an application or a server can add its own event to unblock the `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 that it is not possible to call an ICall function from a stack callback. This will cause ICall to abort (`with ICall_abort()`) and break the system.

4.2.6 Example ICall Usage

Figure 21 shows an example command being sent from the application to the BLE protocol stack via the ICall framework with a corresponding return value passed back to the application. `ICall_init()` initializes ICall module instance itself 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 via `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()`. Note that the protocol command is not executed in the application's thread, but is instead encapsulated in an ICall message and routed to the BLE protocol stack task via the ICall framework. In other words, this command is sent to the ICall dispatcher where it is dispatched and executed on the server side (i.e, BLE Stack). The Application thread meanwhile blocks (i.e, waits) for the corresponding command status message (i.e, status and GAP parameter value). When the BLE protocol stack finishes executing the command, the command status message response is sent via ICall back to the application thread.

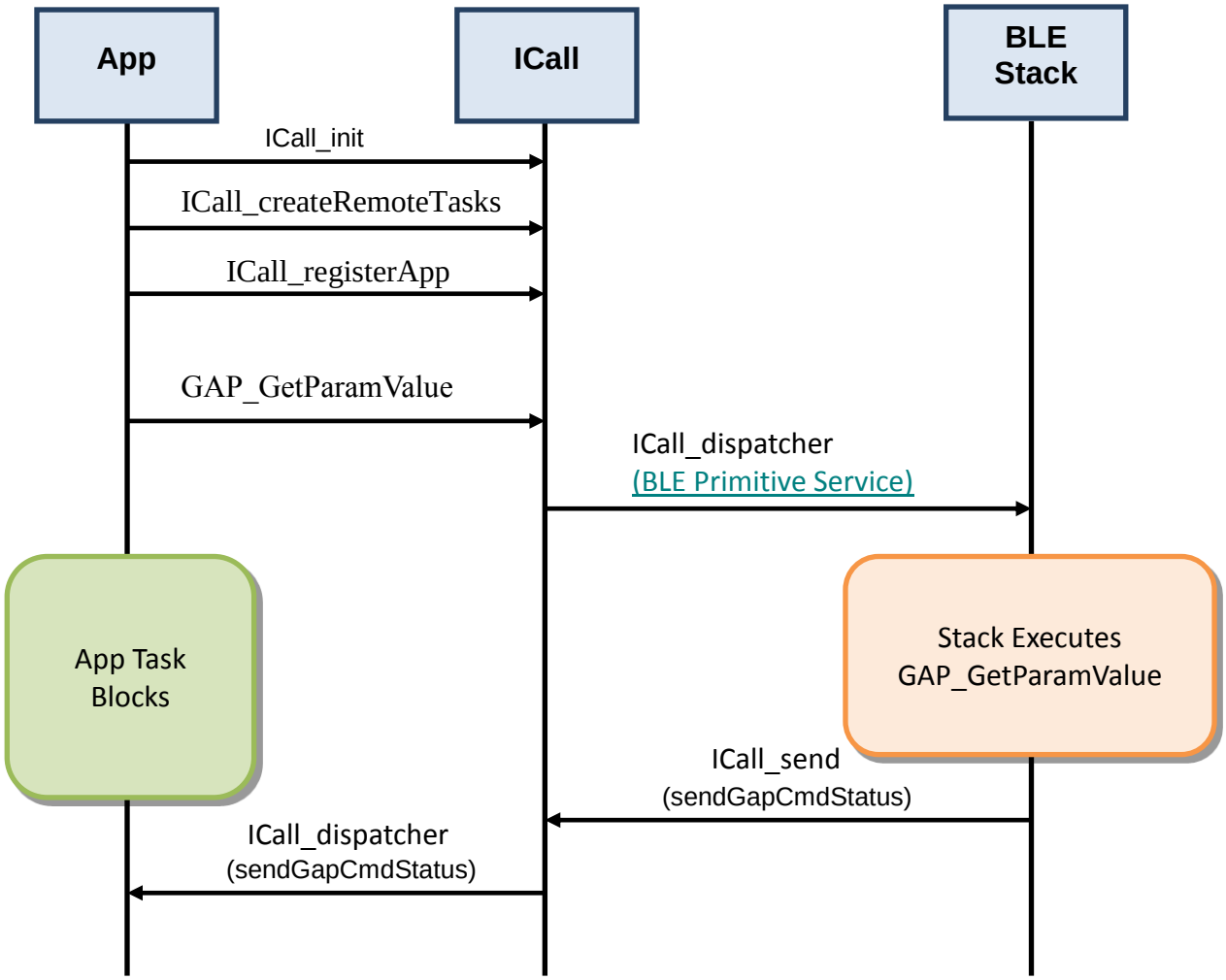


Figure 21 ICall Messaging Example

4.3 General Application architecture

This section will describe how an Application task is architected in more detail.

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 which was passed during task construction is run when the task is ready (e.g. `SimpleBLEPeripheral_taskFxn`). The first thing this function should do is 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 as well as sets several hardware and software configuration settings and parameters. Some common examples are: 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, etc. See their respective sections in this guide for more information on all of these examples.

***Note that in the application initialization function, `ICall_registerApp()` must be called before any stack API is called.

4.3.2 Event Processing in the Task Function

After the initialization function, as shown in the code snippet above, the task function will enter an infinite loop as so that it will continuously process as an independent task and not run to completion. In this infinite loop, it will remain blocked and wait until a semaphore signals a new reason for processing:

```
    ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

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

Once an event or other stimulus occurs and is processed, the task will again wait for the semaphore and remain in a blocked state until there is another reason to process. This is represented in the following flow diagram:

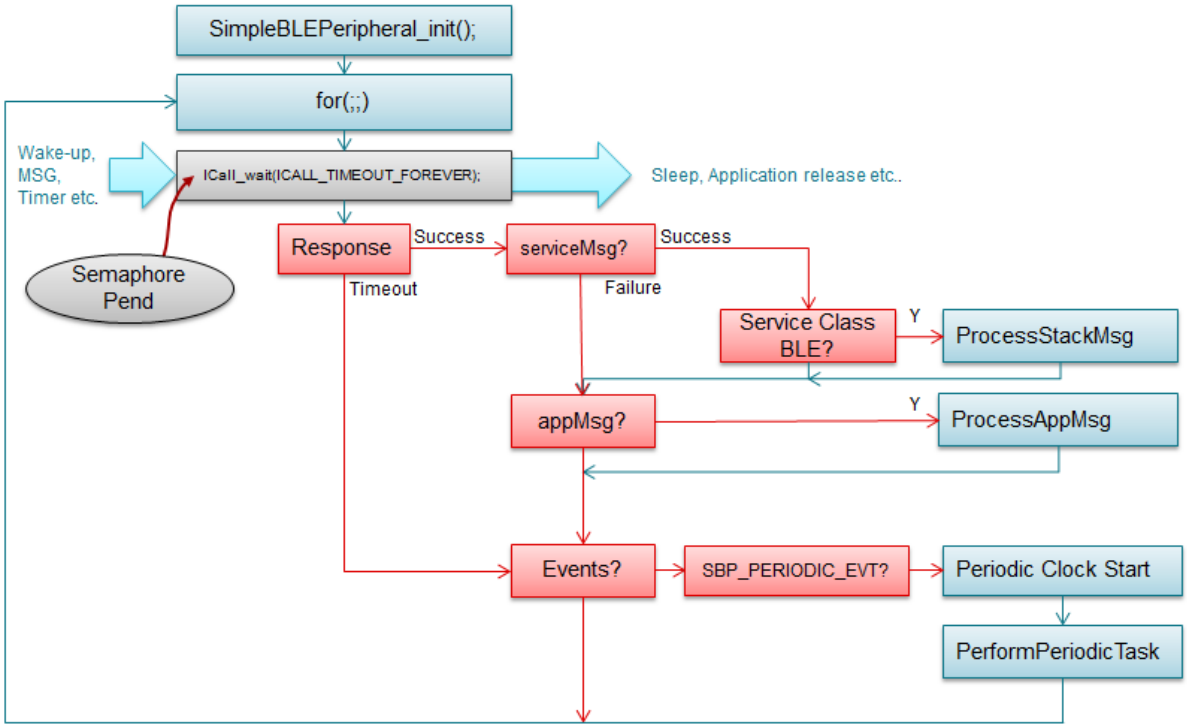


Figure 22: SBP Task Flow Chart

As seen in Figure 22 there are various reasons which will cause the semaphore to be posted to and the task to become active to process. These include:

4.3.2.1 Task Events

This involves the BLE Protocol Stack setting, through ICall, an event in the application task. An example of this is when the `HCI_EXT_ConnEventNoticeCmd()` is called (see Appendix VIII.1) to indicate the end of a connection event. An example of this is seen in SimpleBLEPeripheral's task function:

```
    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 that in the code above, the `pEvt->signature` will always be equal to `0xFFFF` if the event is coming from the BLE Protocol Stack.

When selecting an event value for an inter-task event, it must be unique for the given task and must be a power of 2 (so that only one bit is set). Since the `pEvt->event` variable is initialized as `uint16_t`, this allows for a maximum of 16 events. The only event values that can not be used are those that are already used for BLE 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 that these inter-task events are a different set of events than the intra-task events mentioned in Section 4.3.2.4.

4.3.2.2 Inter-task Messages

These are messages that are passed from another task (such as the BLE protocol stack) through ICall to the application task. Some possible examples are:

- a confirmation sent from the protocol stack in acknowledgement 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 5.3.3.1for an example of this operation.

Here is an example of this from SimpleBLEPeripheral's main task loop.

```
    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 Application Task's RTOS Queue

These are messages which have been enqueued using the `SimpleBLEPeripheral_enqueueMsg()` function. Because they are posted to a queue, they will be 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);
        }
      }
```

4.3.2.4 *Events Signaled Via the Internal Event Variable*

These are asynchronous events which are signalled to the application task to process by setting the appropriate bit in the application task's `events` variable, where each bit corresponds to a defined event, i.e.:

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

Whichever function sets this bit in the events variable must also ensure to post to the semaphore to wake up the application for processing. An example of this is the clock handler which handles clock timeouts. This was described in Section 3.4.2.

Here is an example of processing the periodic event from SimpleBLEPeripheral's main task function:

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

      Util_startClock(&periodicClock);

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

Note that when adding an event, it must be unique for the given task and must be a power of 2 (so that only one bit is set). Since the `events` variable is initialized as `uint16_t`, this allows for a maximum of 16 internal events.

4.3.3 Callbacks

The application code will also likely include various callbacks to protocol stack layers, profiles, and RTOS modules. In order to ensure thread safety, processing should be minimized in the actual callback and the bulk of the processing should be done in the application context. Therefore, there are generally two functions defined per callback (consider the GAPRole state change callback):

- **The actual callback**: This function is called in the context of the calling task / module (i.e. the GAPRole task). In order to minimize processing in the calling context, all this function should do is enqueue an event to the application's queue 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 will be 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.

5 The BLE Protocol Stack

This section will describe the functionality of the BLE protocol stack and provide a list of API's to interface with the protocol stack.

The Stack project and its associated files serve to implement the BLE protocol stack task. This is the highest priority task in the system and it implements the BLE protocol stack as shown in Figure 2.

Most of the BLE protocol stack is provided as object code in a single library file (Texas Instruments does not provide the protocol stack source code as a matter of policy). Therefore, it is important to understand the functionality of the various protocol stack layers and how they interact with the application and profiles. This section will strive to explain these layers.

5.1 Generic Access Profile (GAP)

The GAP layer of the BLE protocol stack is responsible for connection functionality: it handles the device's access modes and procedures including device discovery, link establishment, link termination, initiation of security features, and device configuration.

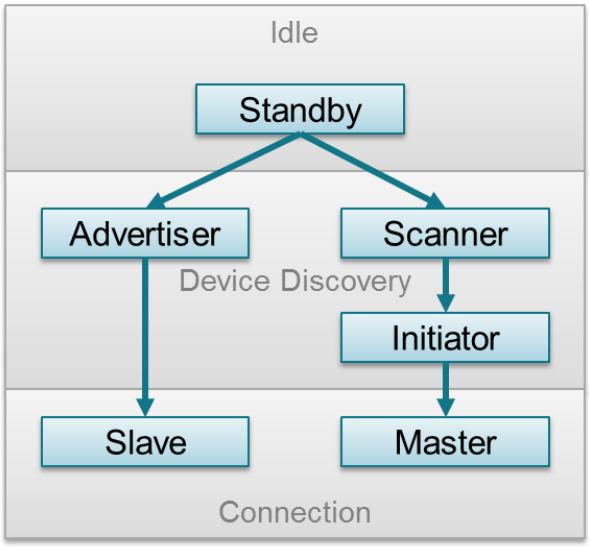


Figure 23: GAP State Diagram

Based on which role the device is configured for, the various states that a device can be in are depicted in Figure 23 and described below:

- **Standby**: Initial idle state entered upon reset.
- **Advertiser**: 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**: The scanning device, upon receiving the advertisement, sends a "scan request" to the advertiser. The advertiser responds with a "scan response". This is the process of device discovery, in that the scanning device is now aware of the advertising device, and knows that it can initiate a connection with it.
- **Initiator:** When initiating, the imitator must specify a peer device address to connect to. If an Advertisement is received matching that peer device's address, the initiating device will then send out a request to establish a connection (link) with the advertising device with the connection parameters described in Section 5.1.1
- **Slave/Master:** Once a connection is formed, the device will function as a slave if it was the advertiser and a master if it was the initiator.

5.1.1 Connection Parameters

This section will describe the various connection parameters which are sent by the initiating device with the connection request and can be modified by either device once the connection is established. These parameters are:

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

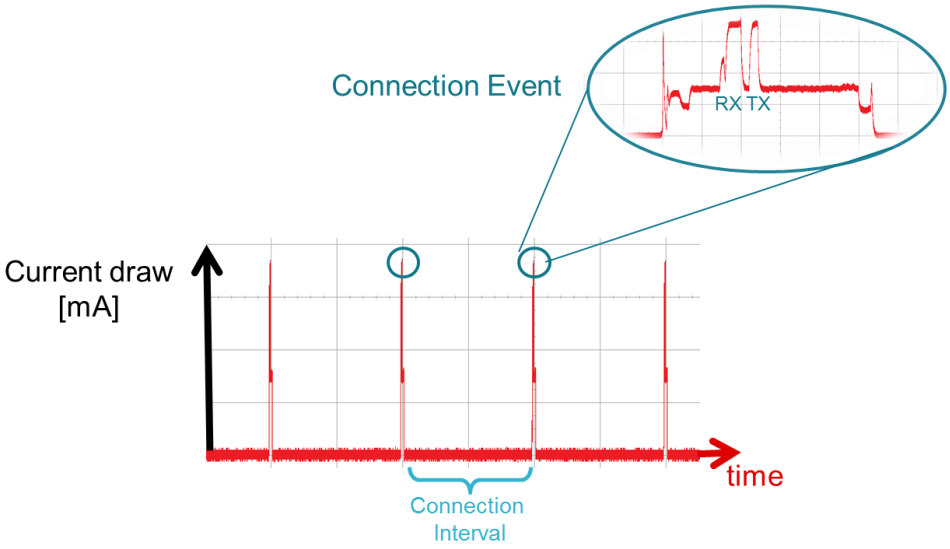


Figure 24: Connection Event and Interval

Different applications may require different connection intervals. As described in Section 5.1.3, this will affect the device's power consumption. For more detailed information on power consumption, please see the power consumption application note [3].

- **Slave Latency** – This parameter gives the slave (peripheral) device the option of skipping a number of connection events. This gives the peripheral device some flexibility, in that if it does not have any data to send it can choose to skip connection events and stay asleep, thus providing some power savings. The decision is up to the peripheral device.

  The slave latency value represents the maximum number of events that can be skipped. It can range from a minimum value of 0 (meaning that no connection events can be skipped) to a maximum of 499; however the maximum value must not make the effective connection interval (see below) greater than 16.0s.

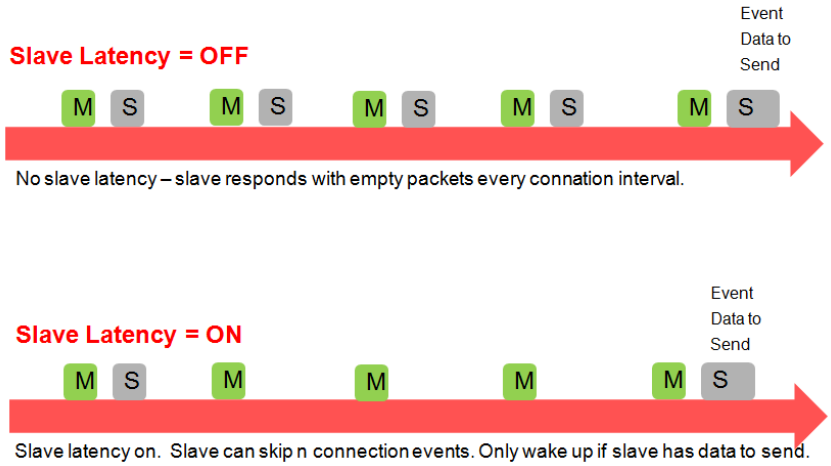


Figure 25: Slave Latency

- **Supervision Timeout** – This is the maximum amount of time between two successful connection events. If this amount of time passes without a successful connection event, the device is to consider the connection lost, and return to an unconnected state. This parameter value is represented in units of 10ms. The supervision timeout value can range from a minimum of 10 (100ms) to 3200 (32.0s). In addition, the timeout must be larger than the effective connection interval (explained below).

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 zero). It can be calculated using the formula:

Effective Connection Interval = (Connection Interval) * (1 + (Slave Latency))

Consider the following example:

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

This tells us that in a situation in which no data is being sent from the slave to the master, the slave will only transmit during a connection event once every 500ms.

5.1.3 Connection Parameter Considerations

In many applications, the slave will skip the maximum number of connection events. Therefore it is useful to 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 BLE device. The following list gives a general summary of the trade-offs in connection parameter settings:

Reducing the connection interval will:

- Increase the power consumption for both devices
- Increase the throughput in both directions
- Reduce the amount of time that it takes for data to be sent in either direction

Increasing the connection interval will:

- Reduce the power consumption for both devices
- Reduce the throughput in both directions
- Increase the amount of time that it takes for data to be sent in either direction

Reducing the slave latency (or setting it to zero) will:

- Increase the power consumption for the peripheral device
- Reduce the amount of time that it takes for data sent from the central device to be received by the peripheral device

Increasing the slave latency will:

- Reduce power consumption for the peripheral during periods when the peripheral has no data to send to the central device
- Increase the amount of time that it takes for data sent from the central device to be received by the peripheral device

5.1.4 Connection Parameter Limitations with Multiple Connections

Due to controller processing limitations in the stack, there are additional rules, beyond what the spec defines, that must be followed when multiple simultaneous connections exist. These are:

- All connection's connection intervals must be multiples of each other. Otherwise, the connection will not be formed or the parameters will not be updated
- If additional connections are desired, the controller must have enough processing time to scan for a new connection. Multiple connections at short connection intervals will limit and possibly remove opportunities for scanning. Therefore, it may be necessary to update connection parameters to a longer connection interval in order to add a new connection. The parameters can then be updated to shorter parameters once the connection is formed.

5.1.5 Connection Parameter Update

In some cases, the central device will request 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 BT4.1 capable devices, this request is handled directly by the Link Layer. For BT4.0 devices, the L2CAP layer of the protocol stack handles the request. Note that the BLE Stack automatically selects the update method.

This request contains four parameters: minimum connection interval, maximum connection interval, slave latency, and timeout. These values represent the parameters that the peripheral device desires for the connection (the connection interval is given as a range). When the central device receives this request, it has the option of accepting or rejecting the new parameters.

Note that sending a Connection Parameter Update Request is optional, and it is not required for the Central device to accept or apply the requested parameters. Typically, applications want to establish a connection at a faster Connection Interval to allow for a faster Service Discovery and initial setup. The application will then 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 Appendix II.1 and Appendix III.1, respectively. Furthermore, the peripheral GAPRole can be configured to automatically send a parameter update a certain amount of time after a connection is established. The SimpleBLEPeripheral application contains an example of this where it uses the following defines:

```
#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
```

This will, 6 seconds after a connection is established, automatically send a connection parameter update. See Section 5.2.1 for an explanation of how the parameters are configured and Appendix II.2 for a more detailed description of these parameters. This can be disabled by setting `DEFAULT_ENABLE_UPDATE_REQUEST` to `FALSE`.

5.1.6 Connection Termination

A connection can be voluntarily terminated by either the master or the slave for any reason. One side initiates termination, and the other side must respond accordingly before both devices exit the connected state.

5.1.7 Connection Security

GAP also handles the initiation of security features during a BLE connection. Certain data may be readable or writeable only in an authenticated connection. Once a connection is formed, two devices can go through a process called pairing. When pairing is performed, keys are established which encrypt and authenticate the link. In a typical case, the peripheral device will require that the central device provide a passkey in order to complete the pairing process. This could be a fixed value, such as "000000", or could be a randomly generated value that gets 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.

In many cases, the same central and peripheral devices will be regularly connecting and disconnecting from each other. BLE has a security feature that allows two devices, when pairing, to

give each other a long-term set of security keys. This feature, called bonding, allows the two devices to quickly re-establish encryption and authentication after re-connecting without going through the full pairing process every time that they connect, as long as 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.

5.1.8 GAP abstraction

It is possible for the application and profiles to directly call GAP API functions to perform BLE-related functions such as advertising or connecting. However, most of the GAP functionality is handled by the GAPRole Task. This abstraction hierarchy is depicted in the figure below:

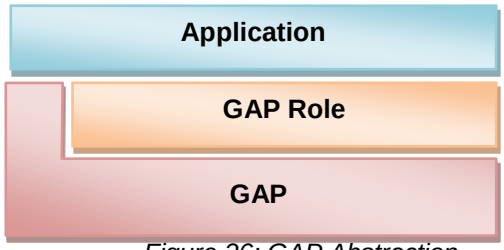


Figure 26: GAP Abstraction

Therefore, it is preferable and easier to configure the GAPRole module and use its API's to interface with the GAP layer. This is described in Section 5.2. For this reason, the following GAP Layer section will describe the functions and parameters that are not handled or configured through the GAPRole task and therefore 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. As stated above, most of these functions are used by the GAPRole and will not need to be called directly. For reference, the GAP API is defined in Appendix IV. However, there are several parameters which may be desirable to modify before starting the GAPRole. These can be set / get via the `GAP_SetParamValue()` and `GAP_GetParamValue()` functions and include advertising / scanning intervals, windows, etc (see the API for more information). As an example, here is the configuration of the GAP layer done 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);
}
```

5.2 GAPRole Task

As mentioned above, the GAPRole task is a separate task which offloads the application by handling most of the GAP layer functionality. It is enabled and configured by the application upon initialization. Based on this configuration, many BLE protocol stack events are handled directly by the GAPRole task and never passed to the application. However, there are callbacks which the application can register with the GAPRole task so that it can be notified of certain events and proceed accordingly.

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

- **Broadcaster** – an advertiser that is non-connectable
- **Observer** – scans for advertisements, but cannot initiate connections
- **Peripheral** – an advertiser that is connectable, and operates as a slave in a single link-layer connection.
- **Central** – scans for advertisements and initiates connections; operates as a master in a single or multiple link-layer connections. Currently, the BLE central protocol stack supports up to three simultaneous connections.

Furthermore, the BLE specification allows for certain combinations of multiple-roles, all of which are supported by the BLE protocol stack. Refer to Section 10.4 for configure BLE-Stack features.

5.2.1 Peripheral Role

The peripheral GAPRole Task is defined in *peripheral.c* and *peripheral.h*. The full API including commands, configurable parameters, events, and callbacks is described in Appendix I.3. The general steps to use this module are:

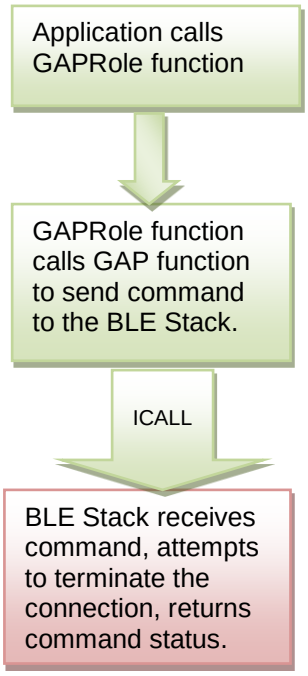
1. Initialize the GAPRole Parameters (see Appendix II). This should be done in the application initialization function, (i.e. `SimpleBLEPeripheral_init()`):

```
{
    // 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. This should also be done in the application initialization function. This involves passing function pointers to application callback functions. These callbacks are defined in Appendix II.3.

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

3. Send GAPRole commands as desired from the application. Here is an example of the application using `GAPRole_TerminateConnection()`. Green corresponds to the app context and red corresponds to the BLE protocol stack context.



simpleBLEPeripheral.c:

```
GAPRole_TerminateConnection();
```

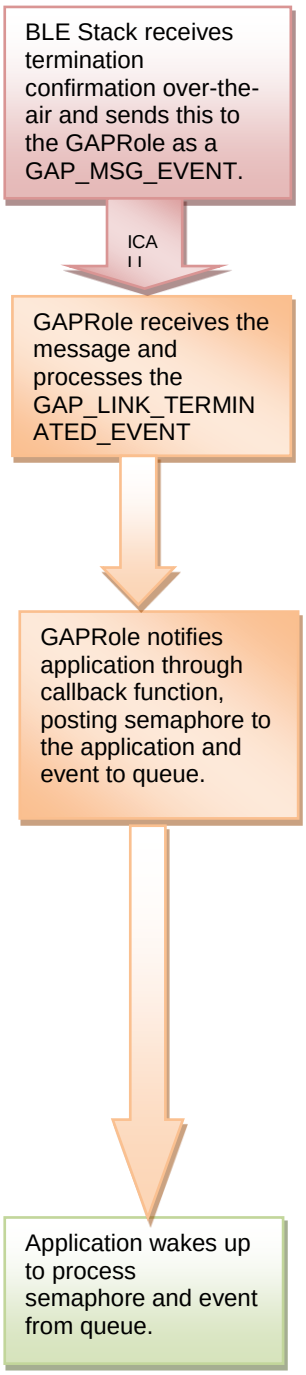
peripheral.c

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

Library Code

***Note that the return value from the BLE protocol stack only indicates whether the attempt to terminate the connection was initiated successfully. The actual "termination of connection" event will be returned asynchronously and is described next. The API in Appendix II.3 lists the return parameters for each command and associated callback function events.

4. The GAPRole task will process most of the GAP-related events passed to it from the BLE protocol stack. However, there are some events which it will also forward to the application. Here is an example tracing the `GAP_LINK_TERMINATED_EVENT` from the BLE protocol stack to the application. Green corresponds to the app context, orange to the GAPRole context, and red to the protocol stack context.



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:
...
```

5.2.2 Central Role

The central GAPRole Task is defined in *central.c* and *central.h*. The full API including commands, configurable parameters, events, and callbacks is described in Appendix III.

The general steps to use this module are:

1. Initialize the GAPRole Parameters. These parameters are defined in Appendix III. This should be done in the application initialization function, (i.e. `SimpleBLECentral_init()`):

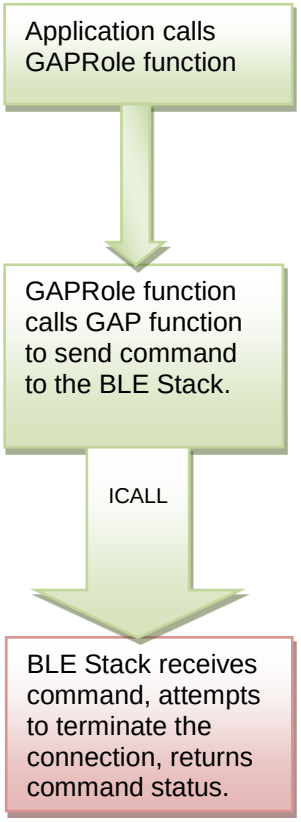
```
// Setup GAP
GAP_SetParamValue(TGAP_GEN_DISC_SCAN, DEFAULT_SCAN_DURATION);
GAP_SetParamValue(TGAP_LIM_DISC_SCAN, DEFAULT_SCAN_DURATION);
GGS_SetParameter(GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN,
```

```
                (void *)attDeviceName);
```

2. Start the GAPRole task. This should also be done in the application initialization function. This involves passing function pointers to application callback functions. These callbacks are defined in Appendix III.3 .

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

3. Send GAPRole commands as desired from the application. Here is an example of the application using `GAPCentralRole_StartDiscovery()`. Green corresponds to the app context and red corresponds to the BLE protocol stack context.



simpleBLEcentral.c

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

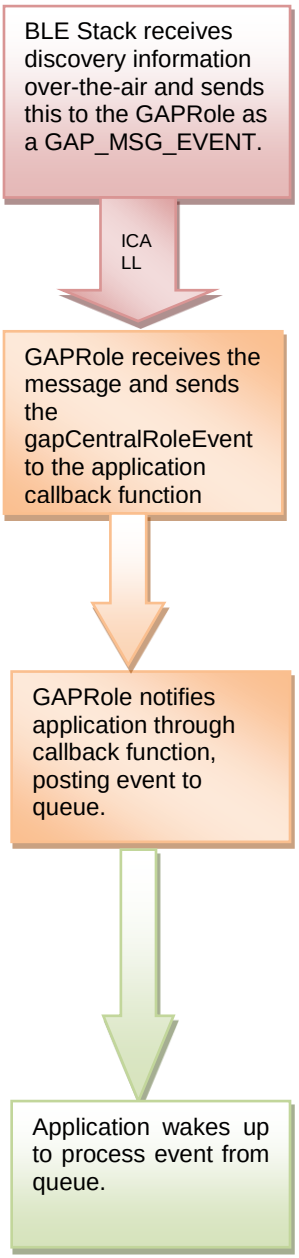
central.c

```
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 that the return value from the BLE protocol stack only indicates whether the attempt to perform device discovery was initiated or not. The actual termination of connection event will be returned asynchronously and is described next. The API lists the return parameters for each command and associated callback events.

4. The GAPRole task will process most of the GAP-related events passed to it from the BLE protocol stack. However, there are some events which it will also forward to the application. Here is an example tracing the `GAP_DEVICE_DISCOVERY_EVENT` from the BLE protocol stack to the application. Green corresponds to the app context, orange to the GAPRole context, and red to the protocol stack context.



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:
      {
...
```

5.3 Generic Attribute Profile (GATT)

Whereas the GAP layer handles most of the connection-related functionality, the GATT layer of the BLE Protocol Stack is designed to be 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 BLE device. From a GATT standpoint, when two devices are connected they are each in one of two roles:

- **GATT Server** – This is the device containing the characteristic database that is being read/written by a GATT Client.
- **GATT Client** – This is the device that is reading/writing data from/to the GATT Server.

The following figure depicts this relationship in a sample BLE connection where the peripheral device, i.e. a SensorTag, is acting as the GATT server and the central device, i.e. a smart phone, is acting as the GATT client.

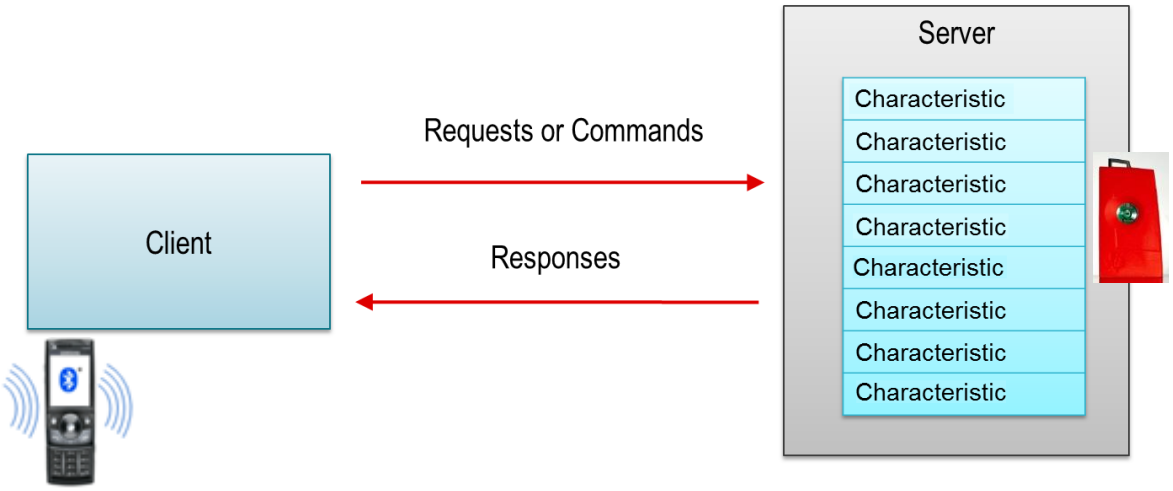


Figure 27: GATT Client and Server

While this is the typical case, it is important to note that the GATT roles of Client and Server are completely 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. It is also possible to act as both a GATT Client and a GATT Server.

5.3.1 GATT Characteristics / Attributes

While they are sometimes used interchangeably when referring to BLE, it is helpful to consider "characteristics" as being composed of groups of information called "attributes." Attributes are the information that is actually transferred between devices. Characteristics organize and use attributes as data values, properties & 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:** this allows the GATT server to configure the characteristic to be notified (sent to GATT server) or indicated (sent to GATT server and expect an ACK).
- **Characteristic User Description:** this is an ASCII string describing the characteristic.

All of these various attributes are stored in the GATT server in an attribute table. In addition to the value itself, each attribute has the following properties associated with it:

- **Handle** – this is essentially the attribute's index in the table. Every attribute has a unique handle.
- **Type** – this indicates what the attribute data represents. It is often referred to as a "UUID" (universal unique identifier). Some of these are Bluetooth-SIG defined and some are custom.
- **Permissions** – this enforces if and how a GATT client device can access the attribute's value.

5.3.2 GATT Services / 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. In reality, many profiles only implement one service so the two terms are sometimes used interchangeably.

In the case of the SimpleBLEPeripheral application, there are four GATT profiles:

- **Mandatory GAP Service** – This service contains device and access information, such as the device name, vendor identification, and product identification, and is a part of the BLE protocol

stack. It is required for every BLE device as per the BLE 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 and is a part of the BLE protocol stack. It is required for every GATT server device as per the BLE 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 and firmware version, regulatory & compliance info, and manufacturer name. The Device Info Service is provided as part of the BLE protocol stack and configured by the Application.
- **SimpleGATTProfile Service** – This service is a sample profile that is provided for testing and for demonstration. The full source code is provided in the files *simpleGATTProfile.c* and *simpleGATTProfile.h*.

The portion of the attribute table in the SimpleBLEPeripheral project corresponding to the simpleGATTProfile service is shown and described below. This section is meant 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	Uuid	Uuid Description	Value	Properties
0x001F	0x2800	GATT Primary Service Declaration	F0: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 28: Simple GATT Profile Characteristic Table from BTool

The simpleGATTProfile contains five characteristics:

1. **SIMPLEPROFILE_CHAR1** – a one-byte value that can be read or written from a GATT client device.
2. **SIMPLEPROFILE_CHAR2** – a one-byte value that can be read from a GATT client device, but cannot be written.
3. **SIMPLEPROFILE_CHAR3** – a one-byte value that can be written from a GATT client device, but cannot be read.
4. **SIMPLEPROFILE_CHAR4** – a one-byte value that cannot be directly read or written from a GATT client device. It is notifiable: it can be configured for notifications to be sent to a GATT client device.
5. **SIMPLEPROFILE_CHAR5** – a five-byte value that can be read (but not written) from a GATT client device.

Here is a line-by-line description of this attribute table, referenced by the handle:

- **0x001F**: this is the simpleGATTprofile service declaration. It has a UUID of 0x2800 (Bluetooth-defined GATT_PRIMARY_SERVICE_UUID). Its value is the UUID of the simpleGATTprofile (custom-defined).
- **0x0020:** this is the SimpleProfileChar1 characteristic declaration. This can be thought of as a pointer to the SimpleProfileChar1 value. It has a UUID of 0x2803 (Bluetooth-defined GATT_CHARACTER_UUID). Its value, as well as all other characteristic declarations, is a five-byte value explained here (from MSB to LSB):

  - **Byte 0:** the properties of the SimpleProfileChar1. These are defined in the Bluetooth spec Here 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 acknowledgement)
    - **0x20:** permits notifications of the characteristic value (with acknowledgement)

    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 located (handle 0x0021).
  - **Bytes 3-4:** the UUID of the SimpleProfileChar1 value (custom defined 0xFFF1).
- **0x0021:** this is the SimpleProfileChar1 value. It has a UUID of 0xFFF1 (custom-defined). Its value is the actual payload data of the characteristic. As indicated by its characteristic declaration (handle 0x0020), it is readable and writeable.
- **0x0022:** this is the SimpleProfileChar1 user description. It has a UUID of 0x2901 (Bluetooth-defined). Its value is a user-readable string describing the characteristic.
- **0x0023** – **0x002F:** These attributes follow the same structure as the simpleProfileChar1 described above with regard to the remaining four characteristics. The only different attribute, handle 0x002B, is described below
- **0x002B:** this is the SimpleProfileChar4 client characteristic configuration. It 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 a 0x0000 to this attribute will disable notifications / indications.

5.3.3 GATT Client abstraction

Similar to the GAP layer, the GATT layer is also abstracted. However, this abstraction will depend on whether the device is acting as a GATT Client or a GATT server. Furthermore, as defined by the Bluetooth Spec, the GATT layer itself is an abstraction of the ATT layer.

GATT clients do not have attribute tables or profiles as they are gathering, not serving, information. Therefore, most of the interfacing with the GATT layer will occur directly from the application. In this case, the direct GATT API described in Appendix IV should be used. The abstraction can be visualized as:

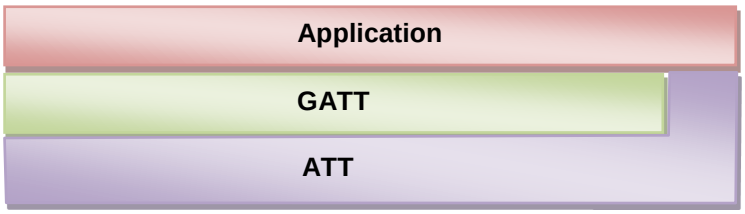


Figure 29: GATT Client Abstraction

5.3.3.1 Using the GATT Layer Directly

This section will describe how to directly 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 gatt.h. The complete API for the GATT layer can be found in Appendix IV. More information on the functionality of these commands can be found in the Bluetooth spec [15]. As described above, these functions will be used primarily for GATT client applications. There are a few server-specific functions which are described in the API and not considered here. Note that most of the GATT functions will return ATT events to the application so it also necessary to consider the ATT API in Appendix IV. The general procedure to use the GATT layer when functioning as a GATT Client (i.e. in the SimpleBLECentral project) is as follows:

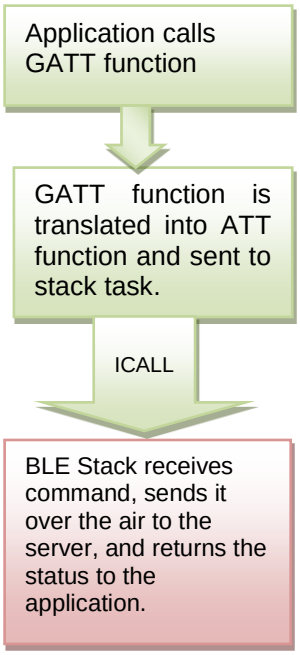
1) Initialize the GATT Client

```
VOID GATT_InitClient();
```

2) Register to receive incoming ATT Indications / Notifications

```
GATT_RegisterForInd(selfEntity);
```

3) Perform a GATT Client procedure. The example here will use `GATT_WriteCharValue()`, which is triggered by a left key press in the SimpleBLECentral application. Green corresponds to the app context and red to the protocol stack context.



simpleBLECentral.c:

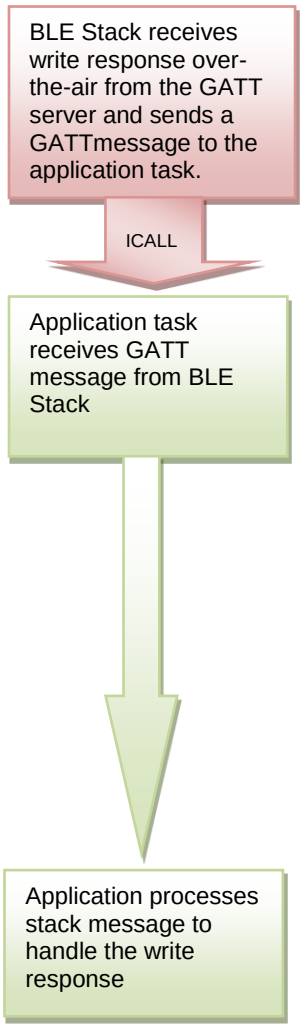
```
status = GATT_WriteCharValue(connHandle, &req, selfEntity);
```

ICallBLEApi.c

```
bStatus_t GATT_WriteCharValue(uint16 connHandle, attWriteReq_t
*pReq,uint8 taskId)
{
  return gattRequest(connHandle, (attMsg_t *)pReq, taskId,
  ATT_WRITE_REQ);
}
```

Library Code

4) Receive and handle the response to the GATT Client procedure in the application. In this example, the application will be receiving an `ATT_WRITE_RSP` event. See Appendix IV.6 for a list of GATT commands and their corresponding ATT events. Green corresponds to the app context and red 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 that even though the event sent to the application is an ATT event, it is still sent as a GATT protocol stack message (`GATT_MSG_EVENT`)

5) Besides receiving responses to its own commands, a GATT Client may also receive asynchronous data from the GATT Server in the form of indications or notifications. Note that is necessary to have registered to receive these as was done in Step 2. These will also be sent as ATT events in GATT messages to the application and should be handled in the same manner as described above.

5.3.4 GATT Server Abstraction

As a GATT server, most of the GATT functionality is handled by the individual GATT profiles. These profiles make use of the GattServApp, a configurable module which stores and manages the attribute table. This abstraction hierarchy is depicted in the figure below:

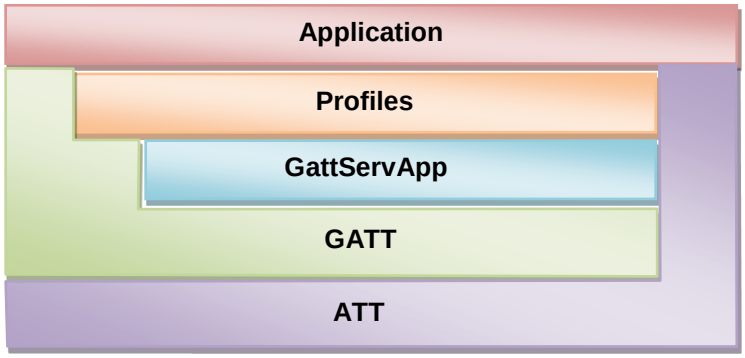


Figure 30: GATT Server Abstraction

The general design process is to create GATT profiles which configure the GATTServApp module and use its API to interface with the GATT layer. In this case of a GATT server, it is unlikely that direct calls to GATT layer functions will be needed. The application will then interface with the Profiles.

5.3.4.1 GATTServApp Module

The GATTServApp stores and manages the application-wide attribute table. Various profiles will use it to add their characteristics to the attribute table. The BLE Stack will use it to respond to discovery requests from a GATT client. For example, a GATT client may send a "Discover all Primary Characteristics" message. The BLE Stack on the GATT server will receive this message and use the GATTServApp to find and send over-the-air all of the primary characteristics stored in the attribute table. This type of functionality is out of the scope of this document and is implemented in the library code. The GATTServApp functions which are accessible from the profiles are defined in *gattservapp_util.c* and described in the API in Appendix V. These functions include finding specific attributes and reading / modifying client characteristic configurations.

5.3.4.1.1 **Building up the Attribute Table**

Upon power-on / 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 r/w call-backs. As shown in Figure 31, all of this information is passed through the GATTServApp to GATT and stored in the stack.

This should be done in the application initialization function, i.e. `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
```

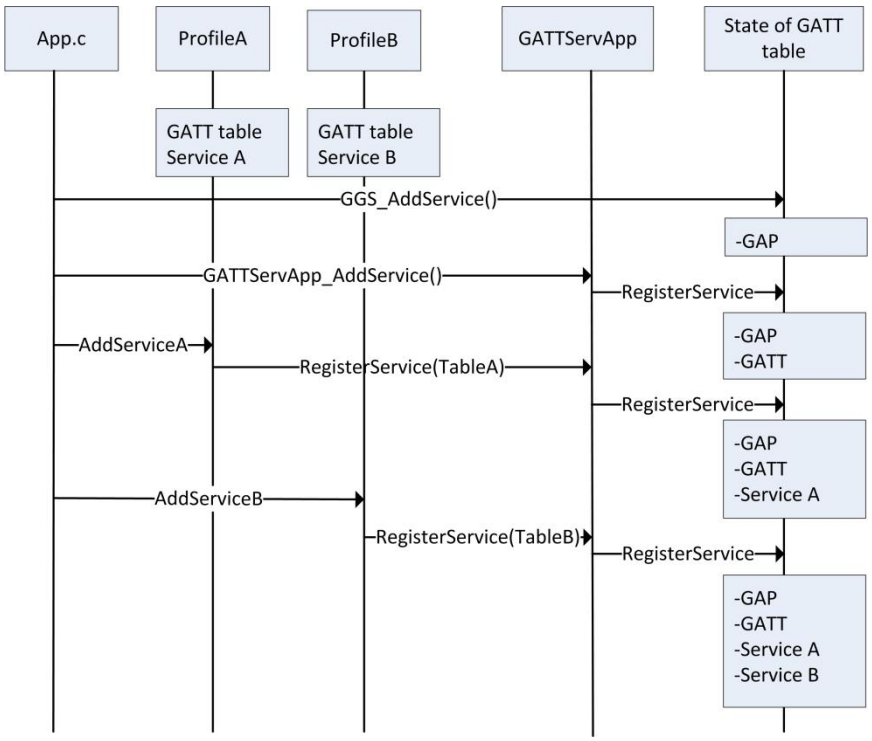


Figure 31: Attribute Table Initialization

5.3.4.2 Profile Architecture

This section will describe the general architecture for all profiles and provide specific functional examples in relation to the simpleGATTProfile in the SimpleBLEPeripheral project. Please refer to Section 5.3.2 for an overview of the simpleGATTProfile.

At minimum, in order to interface with the application and BLE protocol stack, each profile must contain the following:

5.3.4.2.1 **Attribute Table Definition**

Each service or group of GATT attributes must define a fixed size attribute table which gets passed into GATT. This table, in *simpleGATTProfile.c*, is defined as:

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

Each attribute in this table is of the 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 here:

- `type`**:** This is the UUID associated with the attribute and is defined as:

```
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` – this enforces how/if a GATT client device can access the attribute's value. Possible permissions are defined in *gatt.h* as:
  - `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

  Authentication, authorization, and encryption are further described in Section 5.3.5.
- `handle` – This is a placeholder in the table where GATTServApp will assign a handle. This is not settable by the user. Handles will be assigned sequentially.
- `pValue` – This is a pointer to the attribute value. Note that the size can't be changed after initialization. Max size is 512 octets.

The following sections will give examples of attribute definitions for common attribute types.

5.3.4.2.1.1 Service Declaration

Consider the 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 will need to read this so the `permission` is set to `GATT_PERMIT_READ`.

The `pValue` is a pointer to the service's UUID, 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 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 will need to read this so the `permission` is set to `GATT_PERMIT_READ`.

The value of a characteristic declaration was described in Section 5.3.1. For functional purposes here, the only information that needs to be passed to the GATTServApp in `pValue` is a pointer to the

characteristic value's properties. The GATTServApp will take care of adding the UUID and the handle of the value. These properties are defined as:

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

***Note that there is an important distinction 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. However, it is the GATT permissions of the characteristic value which actually affect its functionality in the protocol stack. Therefore, it is important for these properties to match that of the GATT permissions of the characteristic value. This will be expanded on in the following section.

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
      },
```

The `type` is set to the custom-defined simpleProfilechar1 UUID (0xFFF1).

Because, in the characteristic declaration, it is stated that this characteristic value's properties are readable and writeable, it is necessary to actually set the GATT `permissions` to readable and writeable.

The `pValue` is a pointer to the location of the actual value, statically defined in the profile as:

```
// 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 will need to read and write to this 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:

```
static gattCharCfg_t *simpleProfileChar4Config;
```

***Note that this is an array because this value must be cached for each connection. This is described in more detail in the following section.

5.3.4.2.2 Add Service Function

As described in Section 5.3.4.1, when an application starts up it will need to add the GATT services it supports. Therefore, each profile needs a global AddService function which 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 have to expose their own AddService function which the application can call for profile initialization. Using `SimpleProfile_AddService()` as an example, these functions should:

- **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. Here, in the AddService function, is where the number of supported connections is declared

and memory is allocated for each array. Note that there is only one CCC defined in the simpleGATTProfile but it is possible for there to be multiple CCC's.

```
  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 will attempt to initialize the CCC's with information from a previously bonded connection and, if not found, set the initial values to default values.

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

- **Register the Profile with the GATTServApp.** This function will pass the profile's attribute table to the GATTServApp so that the profile's attributes are added to the application-wide attribute table managed by the protocol stack and handles are assigned for each attribute. This also passes pointers the profile's callbacks to the stack to initiate communication between the GATTServApp and the Profile.

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

5.3.4.2.3 Register Application Callback Function

Profiles can relay messages to the application using callbacks. For example, in the SimpleBLEPeripheral project, the simpleGATTProfile calls an application callback whenever the GATT client writes a characteristic value. In order for these application callbacks to be used, the profile must define a "Register Application Callback" function which the application will use to setup callbacks during its initialization. Here is the simpleGATTProfile's "register application callback" function:

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

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

Where the callback typedef is defined as:

```
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 is done in *simpleBLEPeripheral.c* as:

```
// Simple GATT Profile Callbacks
static simpleProfileCBs_t SimpleBLEPeripheral_simpleProfileCBs =
{
  SimpleBLEPeripheral_charValueChangeCB // Characteristic value change callback
};
…
```

```
// Register callback with SimpleGATTprofile
SimpleProfile_RegisterAppCBs(&SimpleBLEPeripheral_simpleProfileCBs);
```

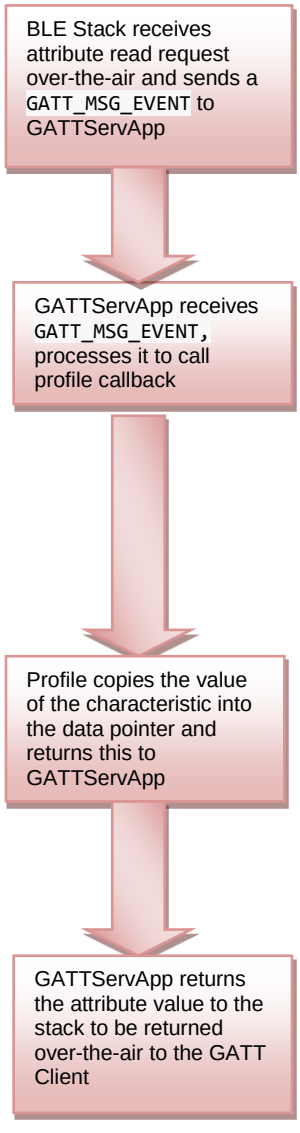
See the next section for the mechanism of how this callback is used.

5.3.4.2.4 Read/Write Callback Functions

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

Read Request from Client

When a read request from a GATT Client is received for a given attribute, the protocol stack will check the attribute's permissions and, if the attribute is readable, call the profile's read call-back. It is up to the profile to copy in the value, perform any profile-specific processing, and optionally notify the application. This 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

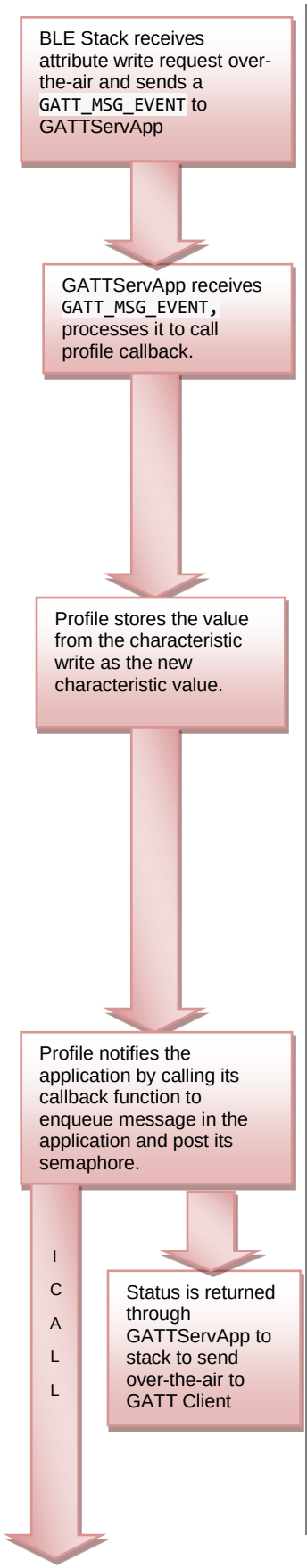
```
static bStatus_t simpleProfile_ReadAttrCB(......)
{
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    // 16-bit UUID
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
      >type.uuid[1]);
    switch ( uuid )
    {
      case SIMPLEPROFILE_CHAR1_UUID:
        *pLen = 1;
        pValue[0]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

Library Code

Note that all of the processing here is in the context of the protocol stack. If there is any intensive profile related processing that needs to 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.

Write Request from Client

When a write request from a GATT Client is received for a given attribute, the protocol stack will check the attribute's permissions and, if the attribute is write, call the profile's write call-back. It is up to the profile to store the value to be written, perform any profile-specific processing, and optionally notify the application. This is illustrated in the following flow diagram 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.



Library Code

gattservapp.c:

```
// Find the owner of the attribute
pAttr = GATT_FindHandle( handle, &service );
if ( pAttr != NULL )
{
  // Find out the owner's callback functions
  pfnGATTWriteAttrCB_t pfnCB = gattServApp_FindWriteAttrCB( service);
  if ( pfnCB != NULL )
  {
    // Try to write the new value
    status = (*pfnCB)( connHandle, pAttr, pValue, len, offset,
    method);
...
```

simpleGATTProfile.c

```
static bStatus_t simpleProfile_WriteAttrCB(......)
{
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
    >type.uuid[1]);
    switch ( uuid )
    {
      //Write the value
      if ( status == SUCCESS )
      {
        uint8 *pCurValue = (uint8 *)pAttr->pValue;
        *pCurValue = pValue[0];
...
```

```
notifyApp = SIMPLEPROFILE_CHAR3;

// If a charactersitic value changed then callback function to notify
application of change
if ( (notifyApp != 0xFF ) && simpleProfile_AppCBs &&
simpleProfile_AppCBs->pfnSimpleProfileChange )
{
  simpleProfile_AppCBs->pfnSimpleProfileChange( notifyApp );
}
```

simpleBLEPeripheral.c

```
static void SimpleBLEPeripheral_charValueChangeCB(uint8_t paramID)
{
  SimpleBLEPeripheral_enqueueMsg(SBP_CHAR_CHANGE_EVT, paramID);
}
```

Library Code

simpleBLEPeripheral.c

```
static void SimpleBLEPeripheral_processAppMsg(sbpEvt_t *pMsg)
{
```

Application wakes up to do additional write-related processing

```
  switch (pMsg->event)
  {
    case SBP_STATE_CHANGE_EVT:
    SimpleBLEPeripheral_processStateChangeEvt((gaprole_States_t)pMsg-
      >status);
...
```

```
static void SimpleBLEPeripheral_processCharValueChangeEvt(uint8_t
paramID)
{
  switch(paramID)
  {
    case SIMPLEPROFILE_CHAR3:
      SimpleProfile_GetParameter(SIMPLEPROFILE_CHAR3, &newValue);

      LCD_WRITE_STRING_VALUE("Char 3:", (uint16_t)newValue, 10,
LCD_PAGE4);
```

As stated previously, it is important to minimize the processing done in protocol stack context. Therefore, in this example, additional processing beyond storing the attribute write value in the profile (i.e. writing to the LCD) was done in the application context by enqueing a message in the application's queue.

5.3.4.2.5 Get / Set Functions

The profile containing the characteristics shall provide set and get abstraction functions for the application to read / write a profile's characteristic. The "set parameter" function should also include logic to check for and implement notifications/indications if the relevant characteristic has notify/indicate properties. The following flow chart and code depict this example for setting simpleProfileChacteristic4 in the simpleGATTProfile.

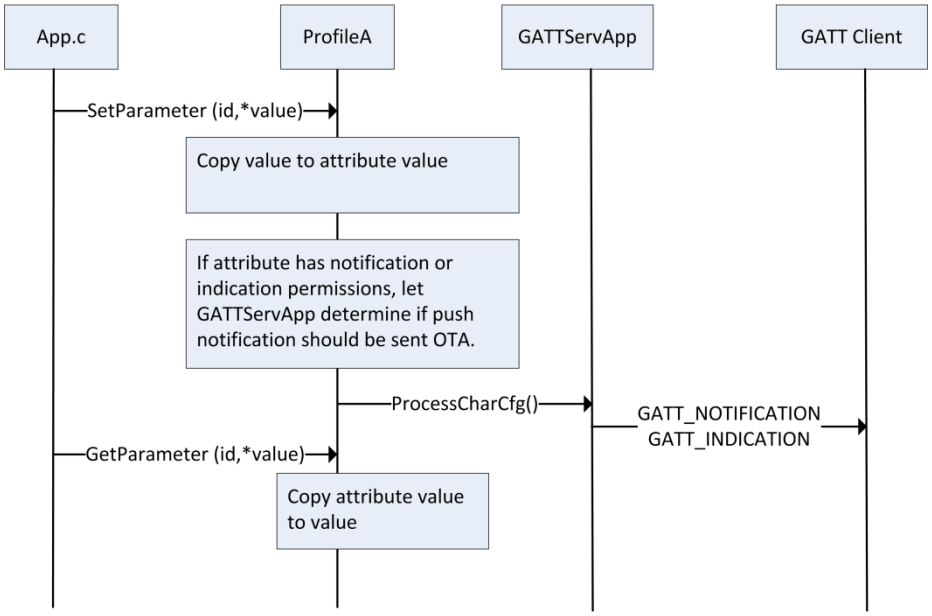


Figure 32: Get / Set Profile Parameter

For example, the application initializes simpleProfileCharacteristic4 to 0 in *SimpleBLEPeripheral.c* via:

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

The code for this function is displayed below (from *simpleGATTProfile.c)*. Besides setting the value of the static `simpleProfileChar4`, this function should also call `GATTServApp_ProcessCharCfg` because it has notify properties. This will force GATTServApp to check if notifications have been enabled by the GATT Client and, if so, to send 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

In order to support fragmentation, GATT / ATT payload structures must be dynamically allocated for commands sent over-the-air. For example, a buffer must be allocated when sending a `GATT_Notification`. Note that this is done by the stack if the preferred method to send a GATT notification / indication to used: calling a profile's SetParameter function (i.e. `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 will need to be added:

1. Attempt to allocate memory for the notification / indication payload using `GATT_bm_alloc()`.

2. If allocation succeeds, send notification / indication using `GATT_Notification()` / `GATT_Indication()`.

3. If the return value of the notification / indication is `SUCCESS` (0x00), this means the memory was freed by the stack. If the return value is something other than SUCCESS (i.e. `blePending`), free the memory using `GATT_bm_free()`. There is an example of this in `the gattServApp_SendNotiInd()` function in gattservapp_util.c:

```
  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;
  }
```

Similar steps will need to be taken for other GATT procedures. This will be noted in the API in Appendix IV.

5.3.6 Registering to Receive Additional GATT Events in the Application

Using `GATT_RegisterForMsgs()` (see Appendix IV), it is possible to receive additional GATT messages to handle certain corner cases. This can be seen in `SimpleBLEPeripheral_processGATTMsg()`. The three cases currently handled here are:

- GATT server in the stack was unable to send an ATT response (due to lack of available HCI buffers). Therefore, 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);
  }
}
```

- ATT flow control violation. The application is being notified that the connected device has violated the ATT flow control specification. Therefore, no more ATT requests or indications can be sent over-the-air 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);
}
```

- ATT MTU size is update. The application is notified in case this affects its processing in any way. See Section 5.5.2 for 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 which offloads most of the security mechanisms from the application. It is important to define the terminology used here:

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 non-volatile memory)
Authentication	The pairing process completed with MITM (Man in the Middle) protection.
Bonding	Storing the keys in non-volatile 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:

1) Pair: exchange keys via
   a. Just Works
   b. MITM
2) Encrypt the link with keys from step 1
3) Bond: store keys in secure flash (SNV)
4) Upon reconnection, use the keys stored in SNV to encrypt the link

Note that it is not necessary to perform all of these steps. For example, it is possible to skip bonding and just re-pair upon each reconnection.

Also note that the GAPBondMgr makes use of the SNV flash area for storing bond information. For more information on SNV, see Section 3.10.4.

5.4.1 Using GAPBondMgr

This section will describe what the application needs to do in order to configure, start, and use the GAPBondMgr. Note that the GAPRole will handle some of the GAPBondMgr functionality. The GAPBondMgr is defined in *gapbondmgr.c* and *gapbondmgr.h*. The full API including commands, configurable parameters, events, and callbacks is described in Appendix VI. The general steps to use this module are as follows. Note that the SimpleBLECentral project is being used as the example here since it makes use of the callback functions from the GAPBondMgr.

1. Initialize the GAPBondMgr parameters (see Appendix VI.2) as desired. This should be done in the application initialization function, (i.e. `SimpleBLECentral_init()`). Consider the following parameters. For the sake of the example, the pairMode has been changed in order to initiate pairing.

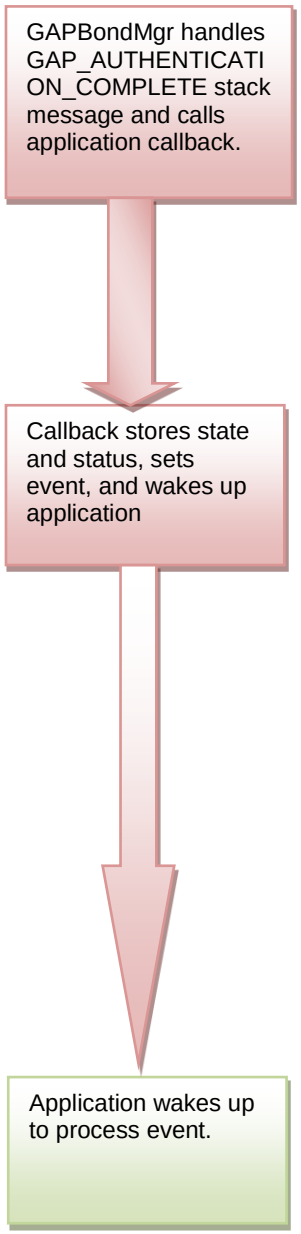
```
// Setup the GAP Bond Manager
{
  uint32_t passkey = DEFAULT_PASSCODE;
  uint8_t pairMode = GAPBOND_PAIRING_MODE_INITIATE;
  uint8_t mitm = DEFAULT_MITM_MODE;
  uint8_t ioCap = DEFAULT_IO_CAPABILITIES;
  uint8_t bonding = DEFAULT_BONDING_MODE;

  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. This should also be done in the application initialization function after the GAPRole profile has been started. These callbacks are defined in Appendix VI.3.

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

3. At this point, the GAPBondMgr is configured and, for the most part, will operate autonomously from the Application's perspective. For example, once a connection is established the GAPBondMgr will initiate pairing and bonding depending on the configuration parameters from step 1. There are a few parameters which can be set asynchronously at this point such as GAPBOND_ERASE_ALLBONDS. See Appendix VI.2 for more information. However, for the most part all communication between the GAPBondMgr and the Application at this point will occur 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. These callbacks will be expanded upon in the following sections. Red corresponds to processing in the protocol stack context and green to the application context.



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 various security modes

This section will provide message diagrams for the various types of security that can be implemented. Note that these all assume acceptable I/O capabilities are present for the given security mode. See the Core Spec for more information [15] on how I/O capabilities affect pairing.

5.4.2.1 Pairing Disabled

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

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

5.4.2.2 Just Works Pairing without Bonding

Just works pairing allows encryption without MITM authentication and is thus vulnerable to MITM attacks. Configure the GAPBondMgr as such for "just works" pairing:

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

The following message sequence chart gives an overview of this process for peripheral device.

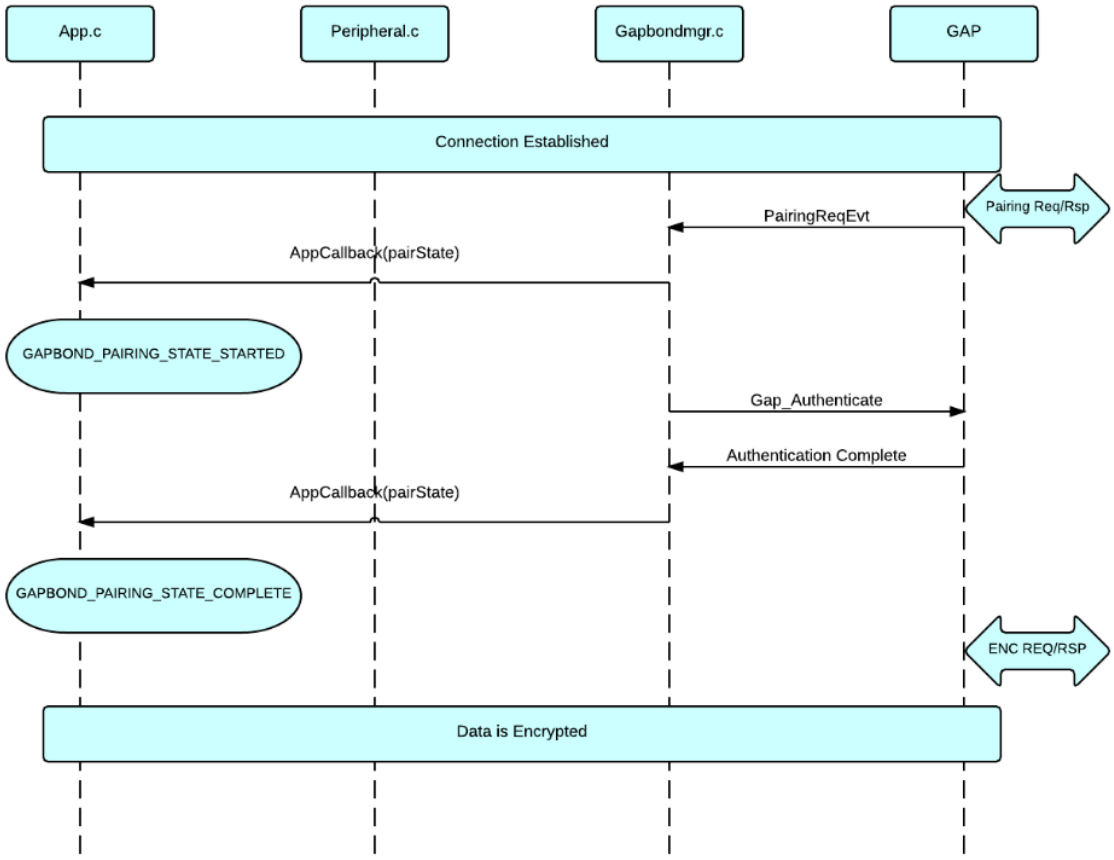


Figure 33: Just Works Pairing

As shown above, the GAPBondMgr pairing states are passed to the application callback at the appropriate time. GAPBOND_PAIRING_STATE_STARTED is passed when the pairing request / response is initially sent / received. GAPBOND_PAIRING_STATE_COMPLETE is sent when the pairing has completed. Therefore, "just works" pairing requires the pair state callback. See Appendix VI.3 for more information.

5.4.2.3 Just Works Pairing with Bonding Enabled

In order to enable bonding with "just works" pairing, the following settings should be used:

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

The following message sequence chart gives an overview of this process for peripheral device.

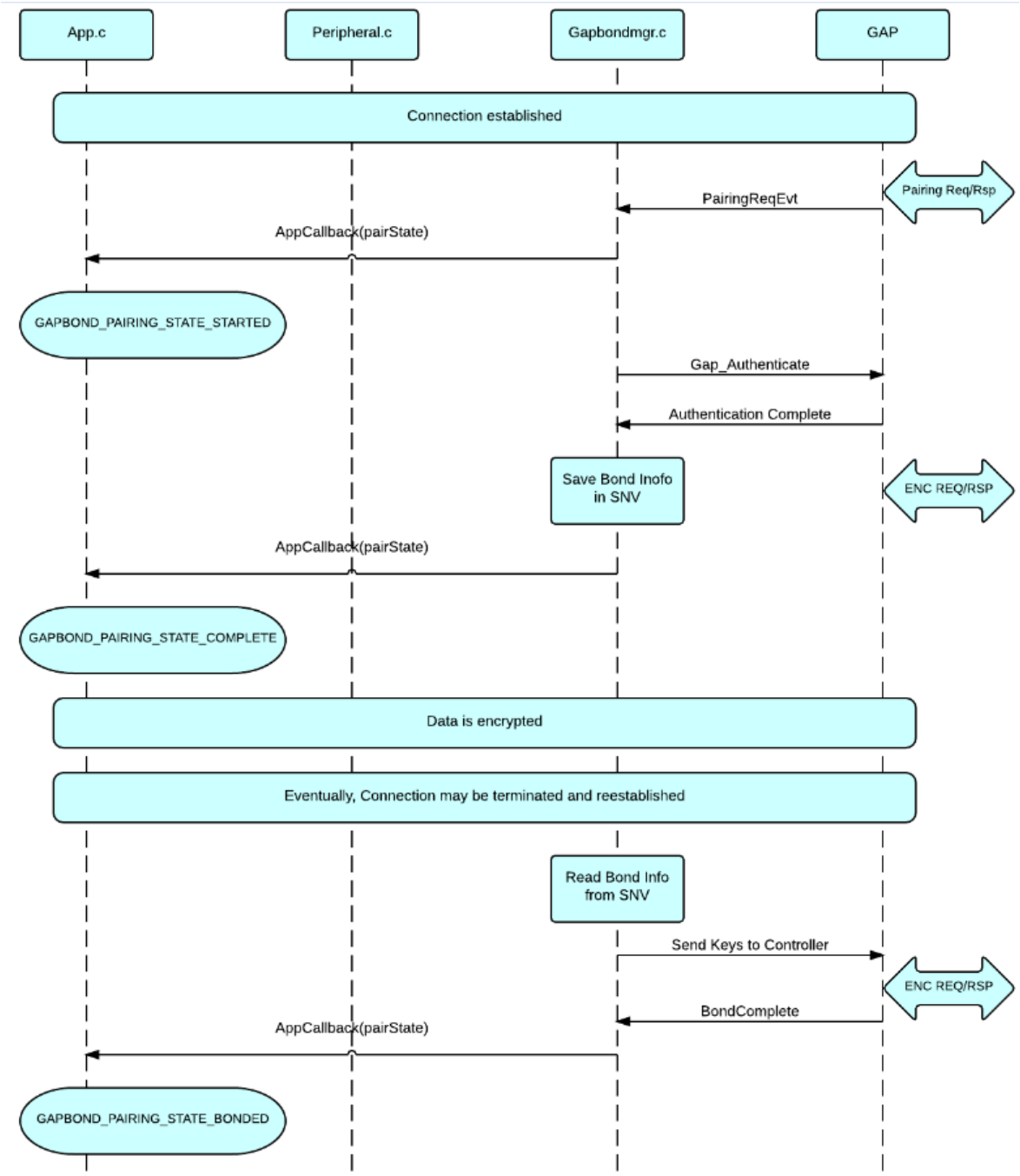


Figure 34: Bonding after Just Works Pairing

Note that GAPBOND_PAIRING_STATE_COMPLETE will only be passed to the application pair state callback upon the initial connection, pairing, and bond. Upon subsequent connections, the security keys will be loaded from flash, thus skipping the pairing process. In this case, only PAIRING_STATE_BONDED will be passed to the application pair state callback.

5.4.2.4 Authenticated Pairing

Authenticated pairing requires MITM protection – some method of transferring a passcode between the devices. The passcode can not go over the air and is generally displayed on one device (using an LCD screen or a serial number on the device) and entered on the other device. One more option is OOB (out-of-band) passcode transfer using NFC but that is out of the scope of this document.

In order to pair with MITM authentication, the following settings should be used:

```
uint8 mitm = TRUE;
```

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

This requires an additional step in the security process as shown in the message sequence chart below: entering a passcode. After pairing is started, the GAPBondMgr will notify the application that a passcode is needed. Then, depending on the I/O capabilities of the device which determines its role in the passcode display / entering process, it must display / enter the passcode and, if displaying, send this passcode back to the GAPBondMgr.

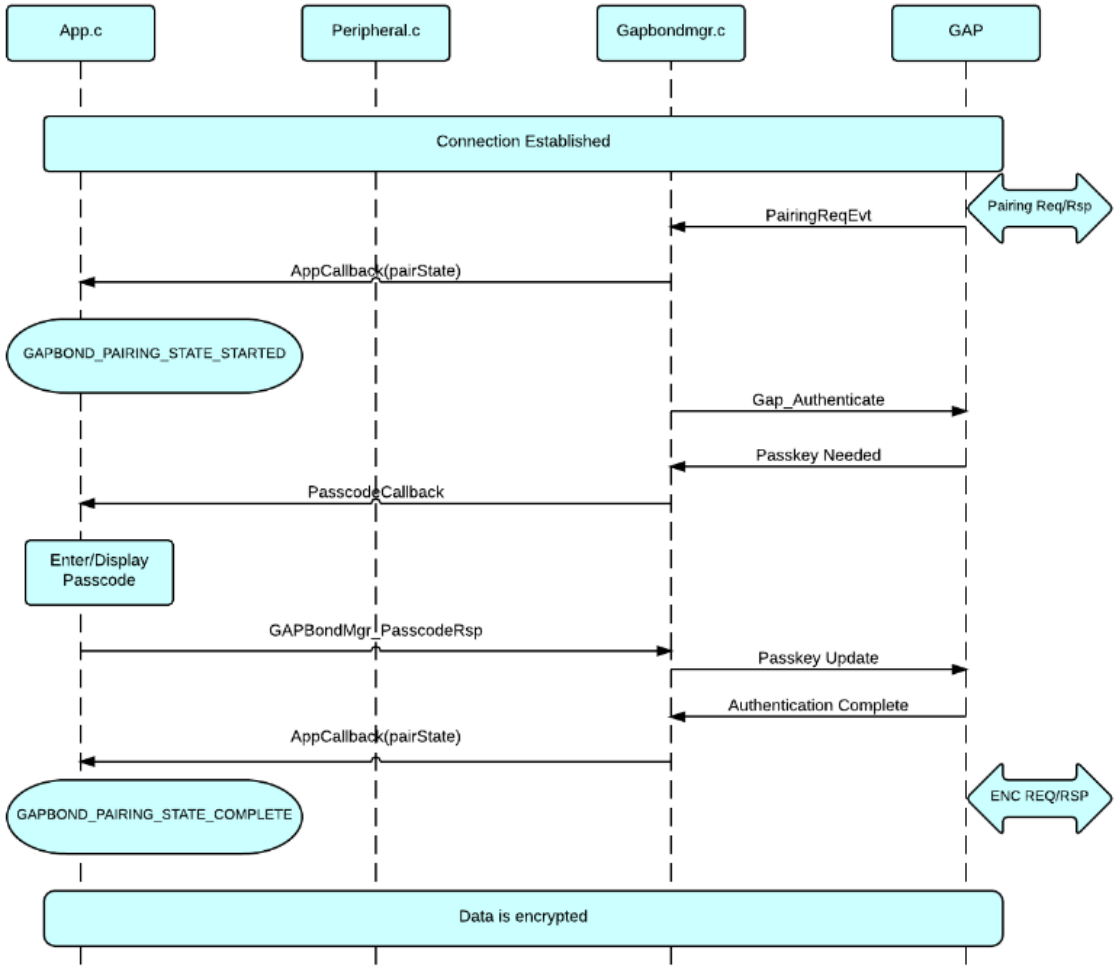


Figure 35: Pairing with MITM Authentication

This passcode communication with the GAPBondMgr is accomplished using a passcode callback function when registering with GAPBondMgr. In order to do this, a passcode function must be added to the GAPBondMgr application callbacks, i.e.:

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

Then, when the GAPBondMgr requires a passcode as shown above, it will use this callback to request a passcode from the application. Depending on the devices I/O capabilities, it should either display a passcode or read in an entered passcode. In both cases, this passcode must be sent down to the GAPBondMgr using the `GAPBondMgr_PasscodeRsp()` function. Here is the SimpleBLECentral example:

```
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 password is created and displayed the password on an LCD screen. The other connected device must then enter this passcode.

5.4.2.1 Authenticated Pairing with Bonding Enabled

After pairing and encrypting with MITM authentication, bonding occurs in the same manner as 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 64 kilobytes in length (Note: The actual size is limited by amount of memory available on the specific device being implemented). L2CAP also permits per-channel flow control and retransmission.

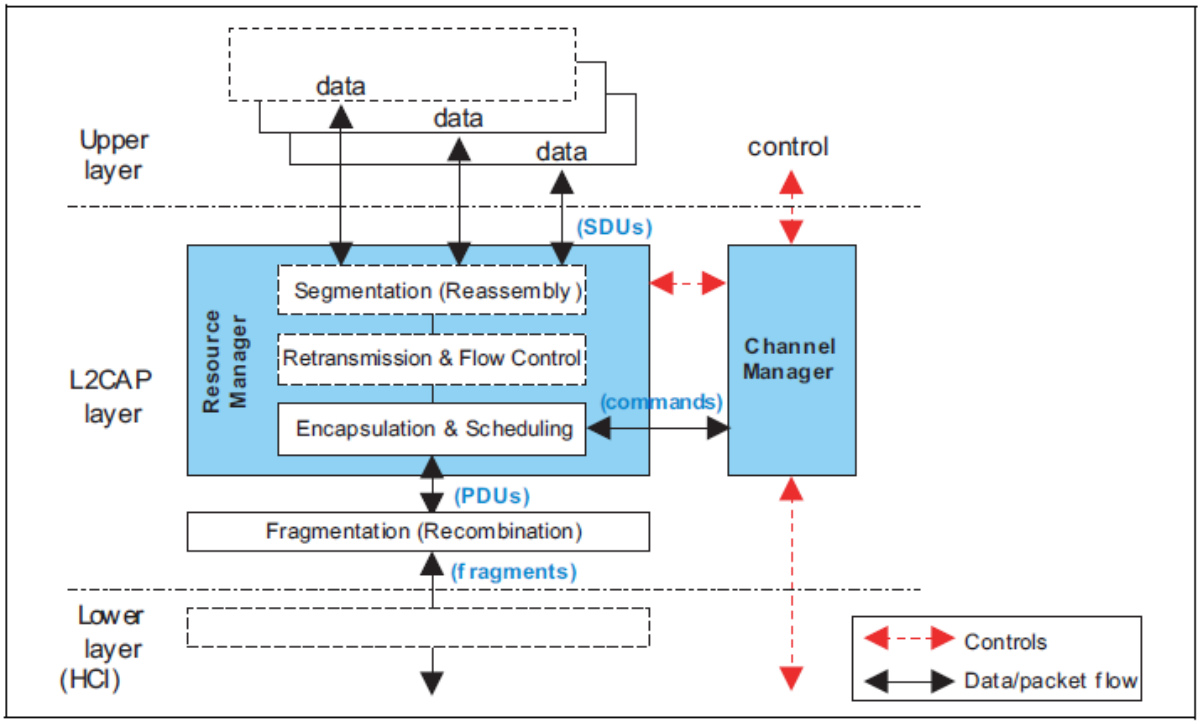


Figure 36 L2CAP Architectural Blocks

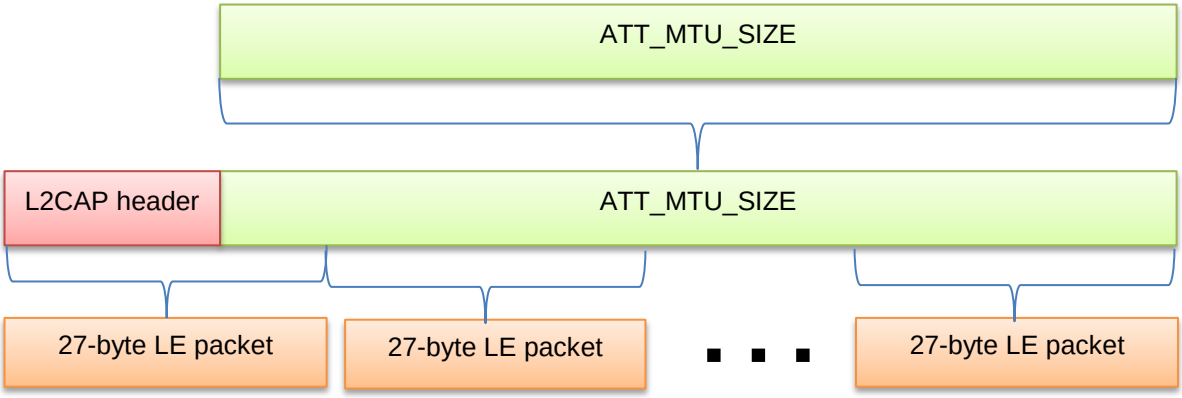
5.5.1 General L2CAP Terminology

Term	Description
L2CAP channel	The logical connection between two endpoints in peer devices, characterized by their Channel Identifiers (CID)
SDU,	Service Data Unit: a packet of data that L2CAP exchanges

or L2CAP SDU	with the upper layer and transports transparently over an L2CAP channel using the procedures specified here.
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 is capable of accepting, i.e. 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 is capable of accepting, i.e. the MPS corresponds to the maximum PDU payload size.

5.5.2 Maximum Transmission Unit (MTU)

The BLE 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 thus reduce the overhead associated with larger data transactions. When fragmentation is used, larger packets are split into multiple Link Layer packets and then reassembled by the peer device's Link Layer. This relationship is depicted here:



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

5.5.2.1 Configuring for Larger MTU Values

It is possible for a Client device to request a larger ATT_MTU during a connection by using the `GATT_ExchangeMTU()` command (see the API defined in Appendix IV.3). During this procedure, the Client (i.e., Central) informs the server of its maximum supported receive MTU size, and the Server (i.e., Peripheral) will respond with its maximum supported receive MTU size. Note that only the client can initiate this procedure. Once the messages have been exchanged, the ATT_MTU for the duration of the connection will be the minimum of the Client MTU and Server MTU values. For example, if the Client indicates it can support an MTU of 200 bytes, and the Server responds with a max size of 150 bytes, the ATT_MTU size will be 150 for that connection. For more information see the "MTU Exchange" section of the Bluetooth Core Spec [15].

The following steps should be taken 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 L2CAP PDU size. The maximum ATT_MTU size will always be four bytes less than the `MAX_PDU_SIZE` value.

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 the previous step.
3. Receive the `ATT_MTU_UPDATED_EVENT` in the calling task to verify that the MTU was successfully updated. Note that this requires the calling task to have registered for GATT messages. See Section 5.3.6 for more information.

Note that even though the Stack can be configured to support a MAX_PDU_SIZE up to 255 bytes, each BLE connection will initially use the default 27 bytes (ATT_MTU=23 bytes) value until the Exchange MTU procedure results in a larger MTU size. Therefore, the Exchange MTU procedure must be performed on each BLE connection and must be initiated by the Client.

Increasing the ATT_MTU size has the effect of increasing 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. However, some procedures, such as Notifications, have additional length restrictions. For example, if an attribute value has a length of 100 bytes, a read of this entire attribute will require 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/Response procedures are needed, 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/Response procedure.

Note: Due to memory & processing limitations, not all BLE systems support larger MTU sizes. It is therefore important to know the capabilities of expected peer devices when defining the behavior of the system. If the capability of peer devices is not known, the system should be designed 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 will result in truncation of data on devices that do not support larger MTU sizes.

5.5.3 L2CAP Channels

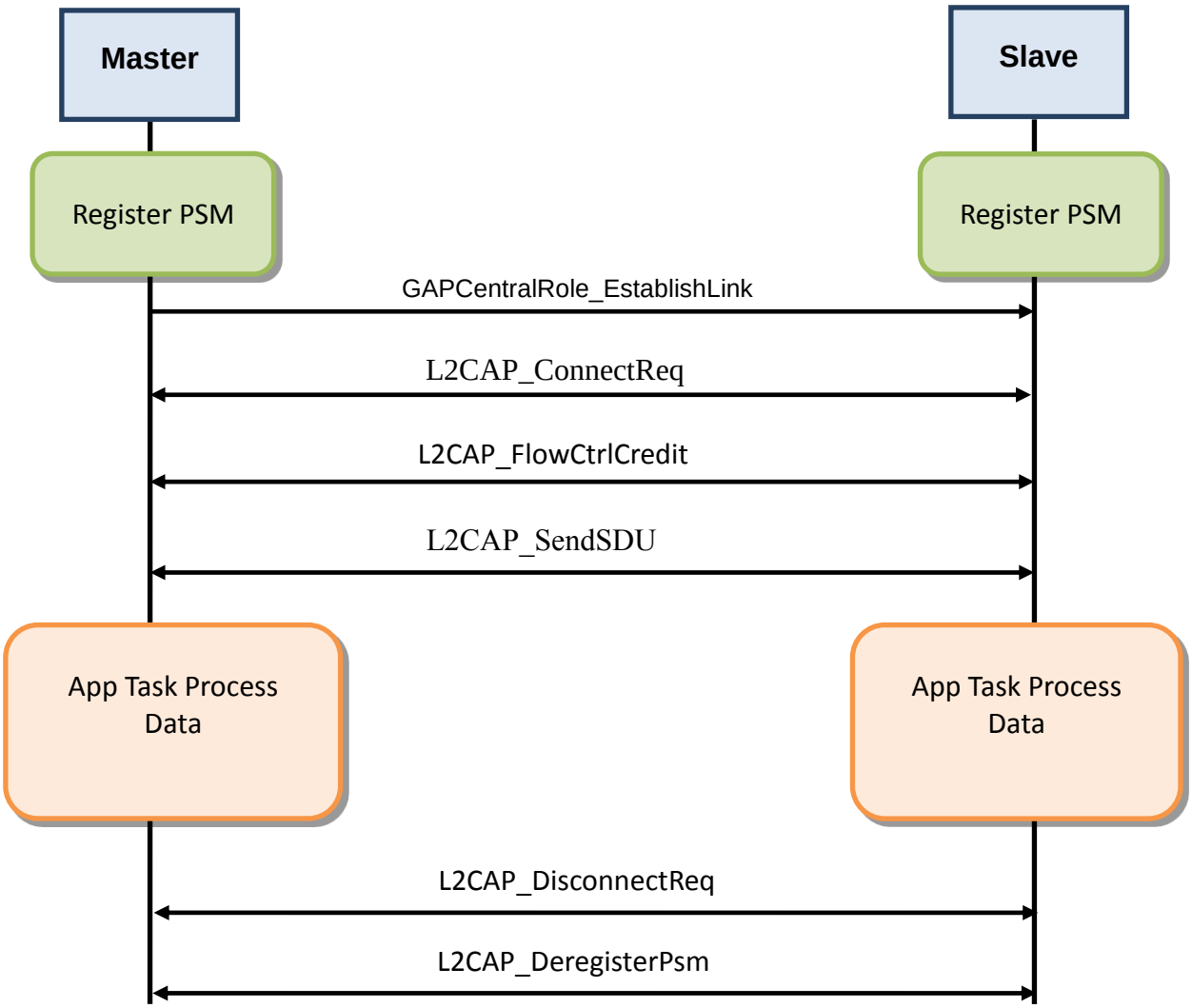
L2CAP is based around the concept of 'channels'. Each one of the endpoints of an L2CAP channel is referred to by a channel identifier (CID). Refer to Volume 3, Part A, Section 2.1 of the Bluetooth Core Specification [15] for more details on L2CAP Channel Identifiers.

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, along with the logical link, the local endpoint and shall be in the range 0x0040 to 0xFFFF. This is used in the connection orientated L2CAP channels described in the following section.

5.5.4 L2CAP Connection oriented connection (CoC) Example

The BLE-Stack SDK provides APIs to create L2CAP CoC channels to transfer bidirectional data between two BLE devices supporting this feature. This feature is enabled by default in the protocol stack. The following diagram shows a sample connection/data exchange process between master and slave device using 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 VII.

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 (i.e. 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, etc., pass from the given layer through the HCI layer to the controller and vice versa.

5.6.1 HCI Extension Vendor Specific Commands

There are a number of HCI Extension Vendor Specific Commands which extend some of the functionality of the controller for use by the application / host. Refer to Appendix VIII 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 will result in HCI Extension Events being passed from the controller to the host. In order to receive these messages, a task needs to register with the stack using the `GAP_RegisterForMsgs()` function. There is an example of this in SimpleBLEPeripheral.c:

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

HCI events will be passed as stack messages to the calling task. An example of receiving these events can be seen 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 Runtime BLE Protocol Stack Configuration

The BLE protocol stack can be configured with various parameters that control its runtime behavior & RF antenna layout. The available configuration parameters are described in *bleUserConfig.h* located in the Application's "ICallBLE" IDE folder. During initialization, these parameters are supplied to the BLE protocol stack by the `user0Cfg` structure, declared in *main.c*:

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

Since the *bleUserConfig.h* file is shared with all projects within the SDK, it is recommended to define configuration parameters in the Application's preprocessor symbols when using a non-default 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. Note that increasing certain parameters may increase heap memory utilization by the protocol stack; adjust the HEAPMGR_SIZE as required. See Section 9.2. The available configuration parameters are listed in Figure 37.

Parameter	Description
MAX_NUM_BLE_CONNS	Maximum number of simultaneous BLE connections. Default is 1 for Peripheral & Central roles. Maximum value is based on GAP Role.
MAX_NUM_PDU	Maximum number of BLE 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 BLE HCI PDU. Default is 27. Valid range: 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 (us) the system will wake up prior to 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. Note that this value must be set directly in *bleUserConfig.h* by adding a board-type define. Default values are based on Evaluation Module (EM) boards.

Figure 37: BLE Stack Configuration Parameters

5.8 Configuring BLE Protocol Stack Features

The BLE protocol stack can be configured to include or exclude certain BLE 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, it may be desirable to exclude certain BLE protocol stack features that may not be required.

The available BLE features are defined in the *buildConfig.opt* file located in the Stack project's "TOOLS" IDE folder. Based on the features selected in the buildConfig.opt file, the LibSearch.exe tool will select the respective precompiled library during the build process of the Stack project. A summary of configurable features is shown below, refer to the *buildConfig.opt* file for additional details.

Feature	Description
HOST_CONFIG	BLE 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	BLE Core Spec V4.1 Controller Feature Partition Build Configuration

6 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 on-board 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 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 actual installation location and can be viewed in IAR's 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 pre-compiled driver, include the respective driver's C include file in the application file(s) where driver APIs are referenced.

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

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

To override a specific prebuilt version of a driver, include the respective C source/include file(s) to the project within the IDE. The IDE will use the source version(s) included in the project in lieu 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(s).

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

For FlashOnly project configurations needed for Over-the-Air (OAD) download, the driver & DriverLib source must be added to the project directly.

For a description of available features and driver APIs, refer to the included TI-RTOS Driver documentation [12].

6.2 Board File

The Board file is used to set fixed driver configuration parameters 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.

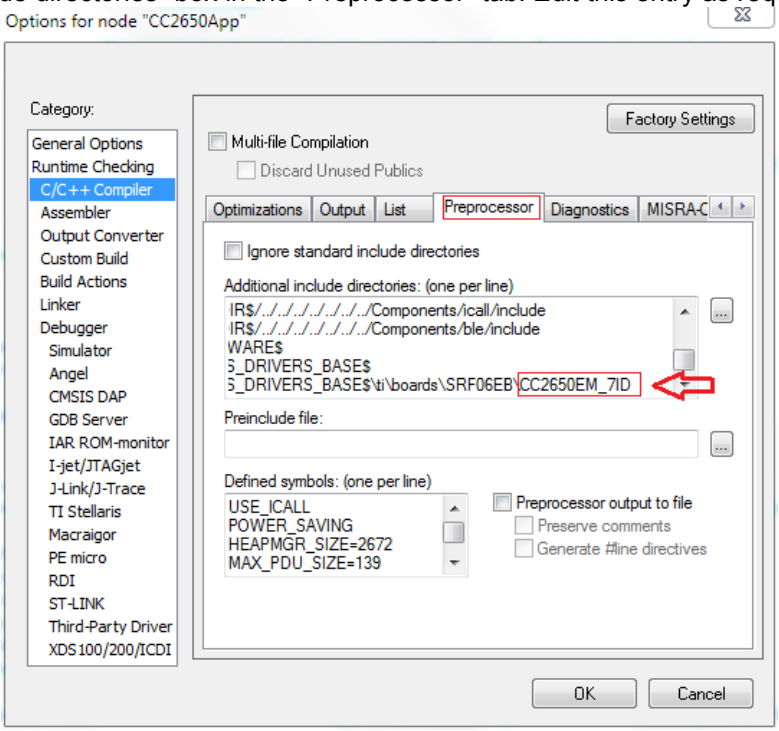
The board files for the SmartRF06 Evaluation Board are located in the following path: $TI_RTOS_DRIVERS_BASE$\ti\boards\SRF06EB\<Board_Type>

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

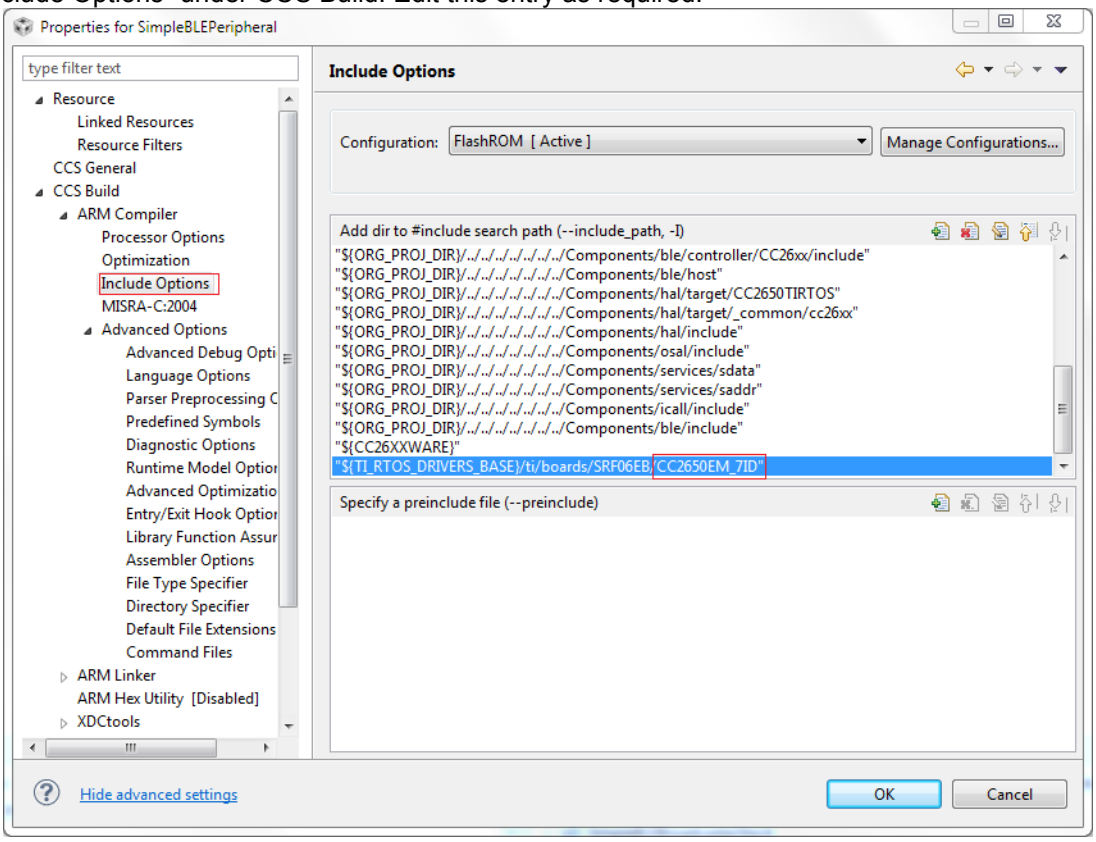
IAR: Tools → Configure Custom Argument Variables
CCS: Project Options → Resources → Linked Resources, "Path Variables" tab

The <Board_Type> used is based on the preprocessor symbol and search path setting in the Application project. To change EM board type, update the Search Path to point to the desired board type's source and include file. Available CC2640 EM options include 'CC2650EM_7ID', 'CC2650EM_5XD', or 'CC2650EM_4XS':

In IAR, open the Project Options for the Application. The board type is on the last line of the "Additional include directories" box in the "Preprocessor" tab. Edit this entry as required:



In CCS, open the Project Properties for the Application. The board type is on the last line of the "Include Options" under CCS Build. Edit this entry as required:



At a minimum, the board file should contain a PIN_Config structure that places all configured & 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 will then be used to initialize the pins in `main()` as seen in Section 4.1:

```
PIN_init(BoardGpioInitTable);
```

6.3 Available Drivers

This section will describe 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 RTOS Driver API [12]

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. Example projects that use the PIN driver are the SimpleBLECentral or SensorTag.

As stated in Section 6.2, the pins should first be initialized to a safe state in `main()`. After this occurs, any module can use the PIN driver to configure a set of pins for use as desired. 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.

Note that "IOID_x" pin numbers directly map to "DIO" pin numbers as referenced in the TRM [2]. The pins used are stated here, as well as 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 / 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 that all this is doing is setting an event in the application task and waking it up. This is done 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);
}
```

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. In `SimpleBLEPeripheral_init()`, open the pins for use and configure the interrupt:

```
// 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, download, and run. Pushing the Up button on the SmartRF06 will toggle the LED1. Note that there is no debouncing implemented here.

6.3.2 UART

There are many possible ways to configure the UART driver. See the RTOS Driver API [12] for more information. An example project that uses the UART driver is HostTest. Note that the HostTest project includes, in addition to a UART driver, additional GPIO's for power management, a packet parser, and other items that are out of the scope of this documentation. In this section, an example will be provided for using the UART driver with the default settings from `UART_Params_init()`: blocking mode, baud rate 115200, etc.

This example will use the UART peripheral already 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)

1. Include the UART driver:

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

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

```
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 where desired:

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

6.3.3 SPI

There are many possible ways to configure the SPI driver. See the RTOS Driver API [12] for more information. An example project that uses the SPI driver is the LCD in the simpleBLEPeripheral project. In this section, an example will be provided for using the SPI driver with the default settings from `SPI_Params_init()`: as a master, blocking mode, etc.

The board file used for simpleBLEPeripheral declares two SPI peripheral where the LCD peripheral already uses `Board_SPI0`. Therefore, this example will use `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 that the jumpers on these pins should be removed to disconnect them from any SmartRF06 peripherals.

The board file (`board.c`) needs to be modified as follows:

1. While the `Board_SPI1` peripheral is declared in the board file, its pins are not set to a safe initialization state in `BoardGpioInitTable`. Therefore, the following line must be added to this table:

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

2. The board file assumes that the CSN pin will be controlled by the application. For simplicity in this example, this pin shall be controlled by the SPI driver. So assign the CSN pin:

```
    {   /* 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,
    }
```

The following steps must then be taken 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 where desired:

```
  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 I2C

There are many possible ways to configure the I2C driver. See the RTOS Driver API [12] for more information. An example project that uses the I2C driver is the SensorTag. Here, an example will

be provided for using the I2C driver with the default settings from `I2C_Params_init()`: as a master, in blocking mode.

Because there is no I2C peripheral on the SmartRF06 board, the LED1 and LED2 pins will be 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 that the jumpers on these pins should be removed to disconnect them from the LED's.

This will require a modification of the board file as follows:

1. Comment out the LED pins and define the I2C pins in *Board.h*:

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

2. Replace the LED configuration in the initial GPIO config table with the I2C pin configuration in *Board.c*

```
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,
```

3. Also in *Board.c*, declare the necessary I2C structures:

```
/* 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 }
};
```

Now that the board file has been modified to add an I2C peripheral, the steps to initialize I2C and perform a sample transaction from the SimpleBLEPeripheral task are as follows. Note that all 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 where desired:

```
  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, download, and run. This will perform a 5-byte I2C write / read.

7 Sensor Controller

The Sensor Controller Engine (SCE) is an autonomous processor within the CC2640 that can control the peripherals in the Sensor Controller independently of the main CPU. This means that the main CPU does not have to wake up to for example execute an ADC sample or poll a digital sensor over SPI, and saves both current and wake-up time that would otherwise be wasted. A PC tool enables the user to configure the Sensor Controller and choose what peripherals are controlled and what conditions will wake up the main CPU.

The Sensor Controller Studio (SCS) is a standalone IDE used to develop and compile microcode for execution on the SCE. Refer to [11] for more details on the SCS, including documentation embedded within the SCS IDE.

8 Startup Sequence

For a complete description of the CC2640 reset sequence, refer to the TRM [2]

8.1 Programming internal Flash with the ROM Bootloader

The CC2640 internal Flash memory can be programmed using the bootloader located in device ROM. Both UART & SPI protocols are supported. Refer to Ch 9 of the TRM [2] for more details on the programming protocol & requirements.

Important: Since the ROM bootloader uses pre-defined DIO pins for internal Flash programming, be sure to allocate these pins in your board's layout. The TRM [2] has details on the pins allocated to the bootloader based on the chip package type.

8.2 Resets

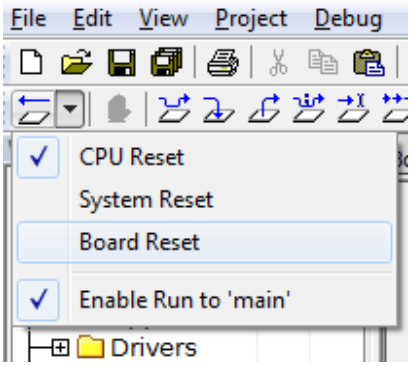
Only hard resets should be used to reset the device. From software, this can be accomplished via:

```
HCI_EXT_ResetSystemCmd(HCI_EXT_RESET_SYSTEM_HARD);
```

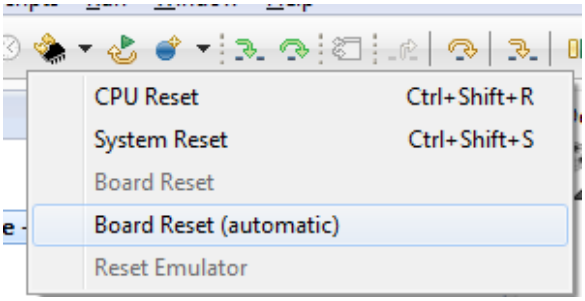
Or...

```
HAL_SYSTEM_RESET();
```

In IAR, this is accomplished by selecting the "Board Reset" option from the Reset (back arrow) Debug Menu dropdown box:



In CCS, select "Board Reset" from the reset menu:



9 Development and Debugging

9.1 Debug Interfaces

The CC2640 platform supports the cJTAG (two-wire) and JTAG (four-wire) interfaces. Any debuggers that support cJTAG, like the TI XDS100v3 and XDS200, will work natively. Others, like the IAR I-Jet and Segger J-Link can only be used in JTAG mode, but their drivers will inject a cJTAG sequence which enables JTAG mode when connecting.

The hardware resources included on the devices that can be used for debugging are listed below. Note that 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) – 32x 32bit stimulus registers
- Trace Port Interface Unit (TPIU) – Serialization and time stamping of DWT and ITM events

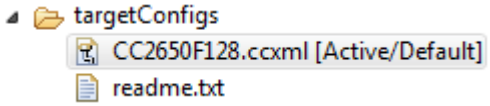
The SmartRF06 Board contains and XDS100v3 debug probe so this is the debugger used by default in the sample projects.

9.1.1 Connecting to the XDS Debugger

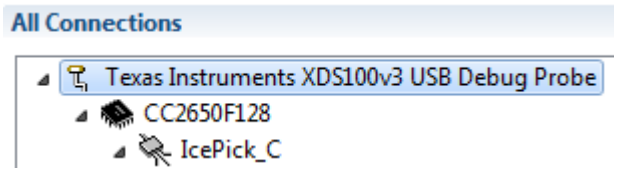
If only one debugger is attached, the IDE will use this automatically. If there are multiple debuggers connected, the individual debugger must be chosen manually. The following steps will detail how to select a debugger in CCS and IAR:

9.1.1.1 Debugging Using CCS:

1. Open the target configuration file and open the '*Advanced'* pane.



2. Choose the top-level debugger entry



3. Choose to select by serial number, and then write in the serial number.

To find this for XDS100v3 debuggers open a command prompt and run C:\ti\ccsv6\ccs_base\common\uscif\xds100serial.exe to get a list of connected debuggers' serial numbers.

9.1.1.2 Debugging Using IAR

1. Open the project options (Project → Options)
2. Go to the Debugger entry
3. Go to 'Extra options' and add command line option: --drv_communication=USB:#select
4. This will make IAR ask on every connection which debugger to use

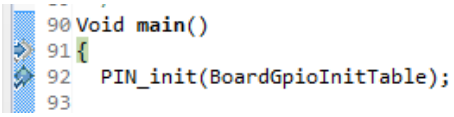
9.2 Breakpoints

Both IAR and CCS reserve one of the instruction comparators, leaving five hardware breakpoints available for debugging. This section will describe setting breakpoints in IAR and CCS:

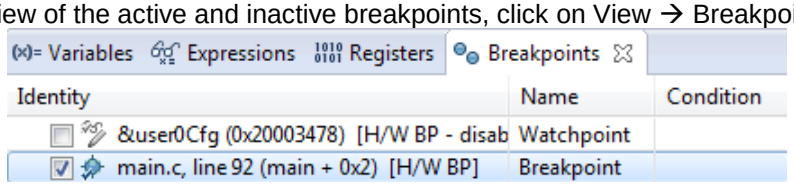
9.2.1 Breakpoints in CCS:
To toggle a breakpoint, either

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

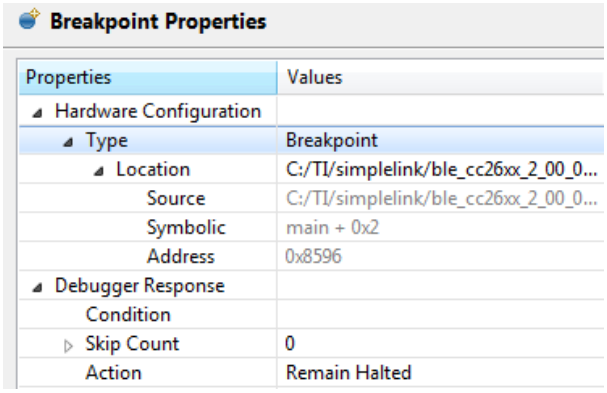
For example, a breakpoint set on line 92 will look like:



To get an overview of the active and inactive breakpoints, click on View → Breakpoints:



To set a conditional break, right click the breakpoint in the overview, and choose Properties:



Skip Count and Condition can be useful when debugging, to skip a number of breaks or only break if a variable is a certain value.

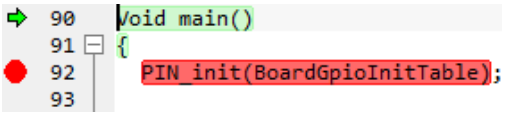
Note: Conditional breaks require a debugger response and, although unlikely, may halt the processor long enough to break e.g. an active connection even if the condition is false or the skip count hasn't been reached yet.

9.2.2 Breakpoints in IAR

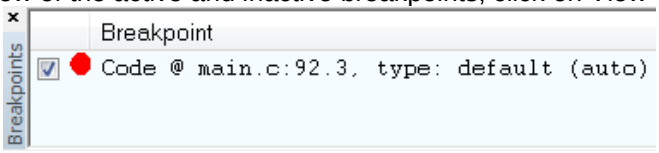
To toggle a breakpoint, either

- Single click the area to the left of the line number
- Go to the line and press F9
- Right click on the line, select Toggle Breakpoint (Code)

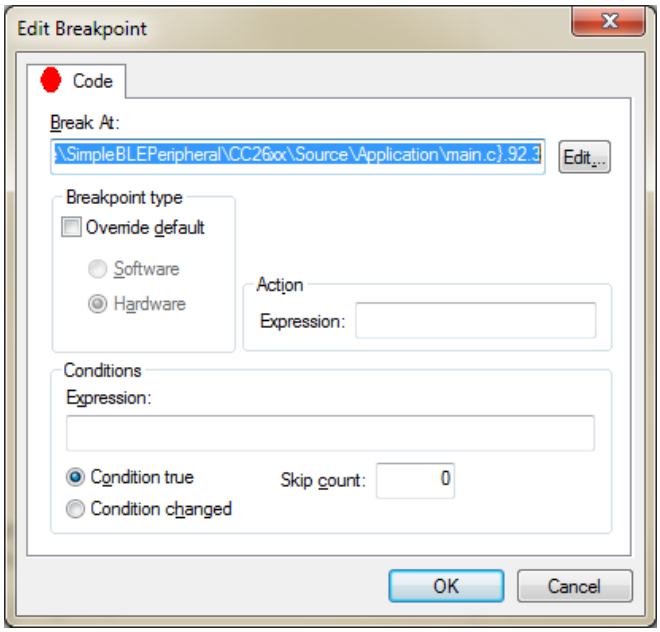
For example, a breakpoint set on line 92 will look like:



To get an overview of the active and inactive breakpoints, click on View → Breakpoints



To set a conditional break, right click the breakpoint in the overview, and choose *Edit...*.



Skip Count and Condition can be useful when debugging, to skip a number of breaks or only break if a variable is a certain value.

Note: Conditional breaks require a debugger response and, although unlikely, may halt the processor long enough to break e.g. an active BLE connection even if the condition is false or the skip count hasn't been reached yet.

9.2.3 Considerations when using breakpoints with an active BLE connection

As the BLE protocol is very timing sensitive, any breakpoints will likely break the execution sufficiently long that network timing is lost and the link will break.

Therefore, it is necessary to place breakpoints as close as possible to where the relevant debug information can be read, or offending code be stepped through, and to potentially experiment on breakpoint placements by restarting debugging and repeating the conditions that lead to hitting the breakpoint.

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 behaviour. Some examples include:

- Code is removed/not compiled in: Toggling a breakpoint in the IDE will result in a breakpoint some other unintended place and not on the selected line. Some IDEs will disable breakpoints on non-existing code.
- Code block is part of a common subexpression: Toggling a breakpoint will work, but code will also break on another execution path than the intended
- If-clause is represented by a conditional branch in assembly: A breakpoint inside an if-clause will always break on the conditional statement, even if not executed.

Because of this, it is in general recommended to select as low optimization level as possible when debugging. See Section 9.4 for information on modifying optimization levels.

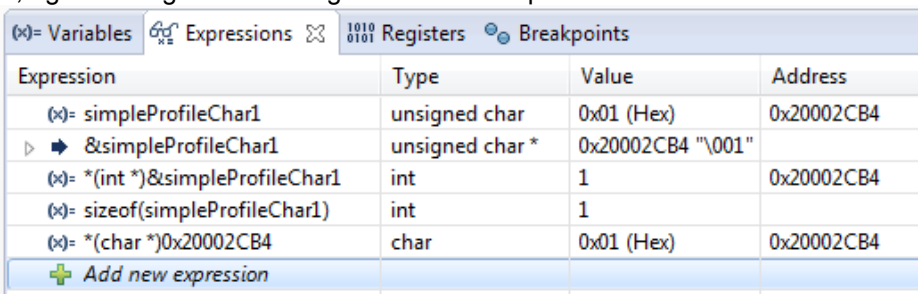
9.3 Watching Variables and Registers

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 can be accessed at any time via the Watch and Expression windows.

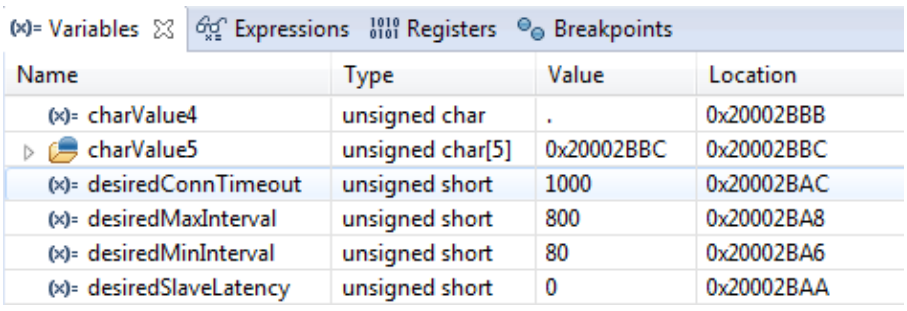
Unless removed due to optimizations, global variables are always available in these views. Local variables, variables that are only valid inside a limited scope, are placed on the active task's stack. Such variables can also be viewed with the Watch/Expression views, but can also be automatically displayed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

9.3.1 Variables in CCS

Global Variables can be viewed by selecting View → Expressions or by selecting a variable name in code, right clicking and selecting "Add Watch Expression:"

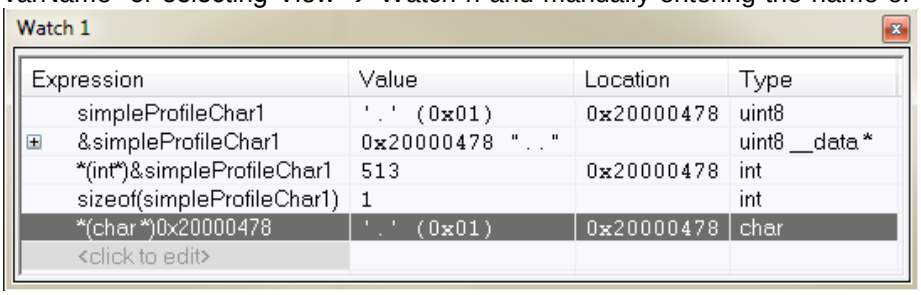


Local Variables can be automatically viewed by selecting View → Variables

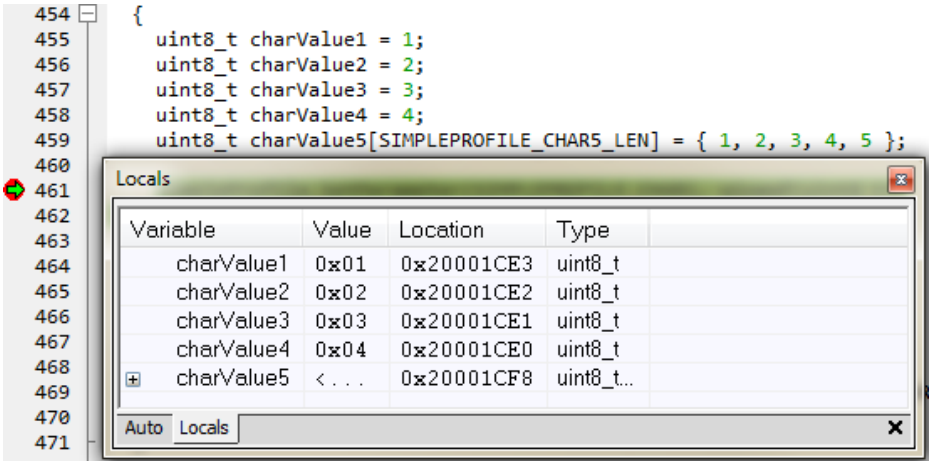


9.3.2 Variables in IAR

Global Variables can be viewed by either right-clicking on the variable and selecting "Add to Watch: varName" or selecting View → Watch n and manually entering the name of the variable:



View → Locals will show the local variables in IAR:



9.3.3 Considerations when Viewing Variables

Local variables will often be placed in CPU registers and not on the stack. They will also typically have a very limited lifetime even within the scope they are valid in, depending on the optimization performed. Therefore, both CCS and IAR may struggle to show a particularly interesting variable. The solution to this when debugging is to:

- Move the variable to global scope, so it's always accessible in RAM.
- Make the variable volatile, so the compiler doesn't use a limited scope even if it could.
- Alternatively make a shadow copy of the variable that is global and volatile. Note: IAR may still remove the variable during optimization, in which case the `__root` directive can be added to `volatile`.

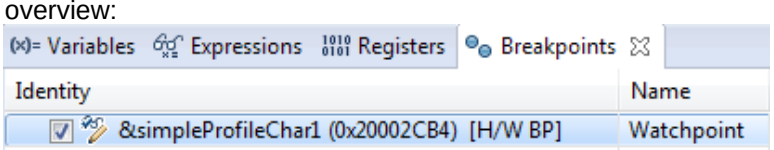
9.4 Memory Watchpoints

As mentioned in Section 9, the DWT module contains 4 memory watchpoints which allow breakpoints on memory access. The hardware match functionality only looks at the address, so if this is intended for use on a variable, the variable must be global. This will be described for IAR and CCS as follows:

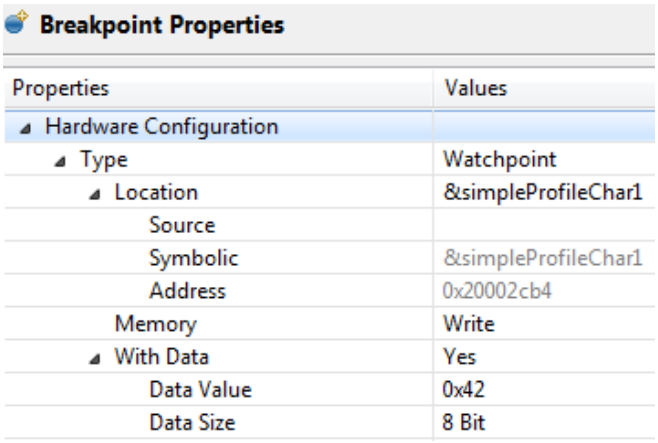
Note: If data watchpoint with value match is used, two of the four watchpoints are used.

9.4.1 Watchpoints in CCS

Right click on a global variable and select Breakpoint → Hardware Watchpoint to add it to the breakpoint overview:



Similar to code breakpoints, right click and edit the "Breakpoint Properties" to configure the watchpoint.

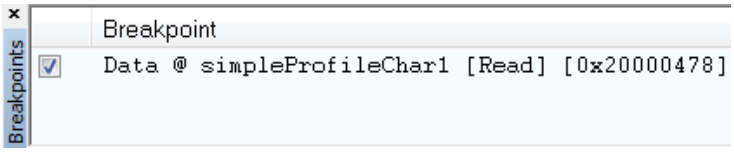


This example configuration will make sure that if 0x42 is written to the memory location for "Characteristic 1" in the BLE SimpleBLEPeripheral example project, the device will halt execution.

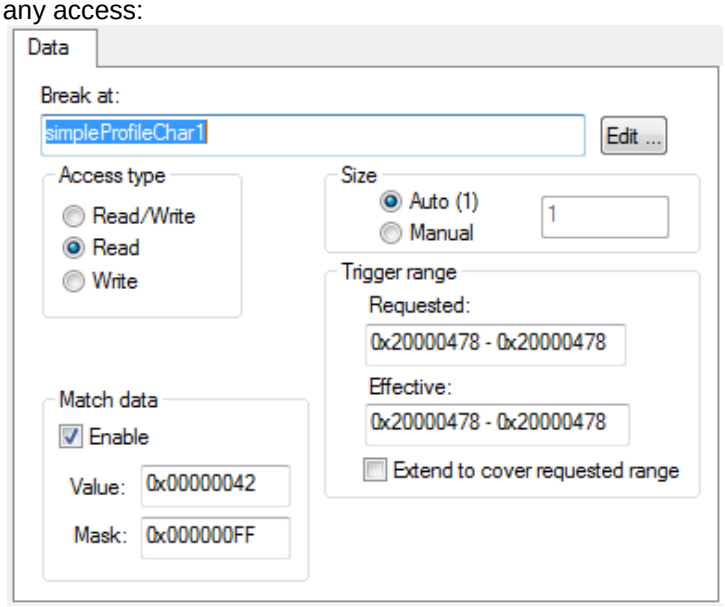
9.4.2 Watchpoints in IAR

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

Right-click on a variable and select "Set Data Breakpoint for 'myVar'" to add it to the active breakpoints.



From the breakpoints view, right click and choose 'Edit...' to set up whether it should match on read, write or any access:



The above example shows a break on read access when the value matches 0x42.

9.5 TI-RTOS Object Viewer

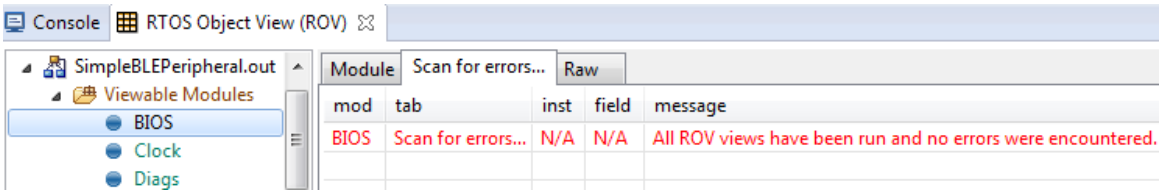
Both IAR and CCS include the RTOS Object Viewer (ROV) plug-in which provides insight into the current state of TI-RTOS, including Task states, stacks, etc. Both CCS and IAR have a similar interface, so these examples only discuss CCS.

To access the ROV in IAR, use the TI-RTOS menu on the menu-bar, then select a sub-view. To access the ROV in CCS, click the Tools menu, then RTOS Object View.

This section will discuss some ROV views that are useful for debugging and profiling

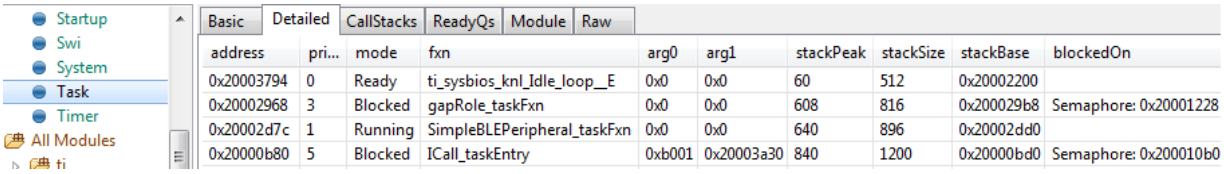
9.5.1 Scanning the BIOS for Errors

The BIOS → Scan for errors view will sweep through the available ROV modules and report any errors found. This can be a good starting point if anything has gone unpredictably wrong. It only shows errors related to TI-RTOS modules and only errors that it's able to catch.



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 run-time stack usage. This example shows the state the first time the user-thread is called. The image shows the Idle task, the GAPRole task, the SimpleBLEPeripheral task, and the BLE stack task, represented by its ICall proxy:

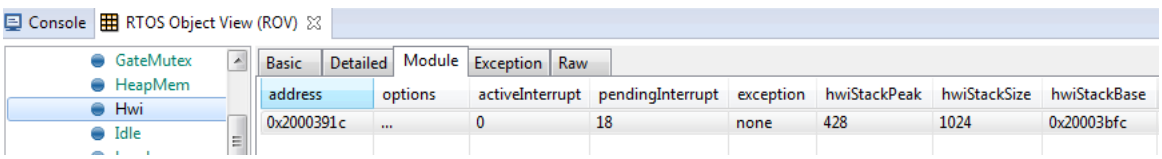


The columns are explained here (see Section 3.3 for more information on the various run-time stacks):

- *address*: Memory location of the `Task_Struct` instance for each task.
- *priority*: The TI-RTOS priority for the task.
- *mode*: The current state of the task
- *fxn*: The name of the task's entry function.
- arg0, arg1: Arbitrary values that can be given to task's entry function. In the image, the ICall proxy is given 0xb001 which is the flash location of the RF stack image's entry function, and 0x20003a30 which is the location of *bleUserCfg_t user0Cfg*, defined in main().
- *stackPeak*: Maximum run-time stack memory used based on watermark in RAM, where the stacks are pre-filled 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. Therefore, a stack peak near stackSize but not exceeding it may still indicate stack overflow.
- *stackSize*: The size of the run-time stack, configured when instantiating a task.
- *stackBase*: Logical top of the task's run-time stack. Usage starts at stackBase + stackSize and grows down to this address.
- *blockedOn*: 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, for `main()`, Hwi execution and SWI execution. See Section 3.11.3 for more information on the System Stack.



The hwiStackPeak, hwiStackSize, and hwiStackBase can be used to check for System Stack overflow.

9.5.4 Viewing Power Manager Information

The Power → Module view shows a logical OR between all the constraints currently enforced via the Power API. The numeric defines are subject to change, but at the time of writing, the value in

the example (0x06) indicates Standby Disallow (0x4) and Shutdown Disallow (0x02). See the Power Management Guide [8] 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 is used to allocate messages between the BLE Stack Task and the Application Task, as well as dynamic memory allocations in the various 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 both 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 below become available:

Global variables:

- heapmgrBlkMax: Maximum amount of simultaneous allocated blocks.
- heapmgrBlkCnt: Current amount of allocated blocks.
- heapmgrBlkFree: Current amount of free blocks.
- heapmgrMemAlo: Current total memory allocated in bytes.
- heapmgrMemMax: Maximum amount of simultaneous allocated memory in blocks. ***This 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: 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 above described variables in the pointers passed in as parameters
- `int heapmgrSanityCheck(void)`
  - returns 0 if the heap is ok, non-zero otherwise (i.e. an array access has overwritten a header in the heap)

9.7 Optimizations

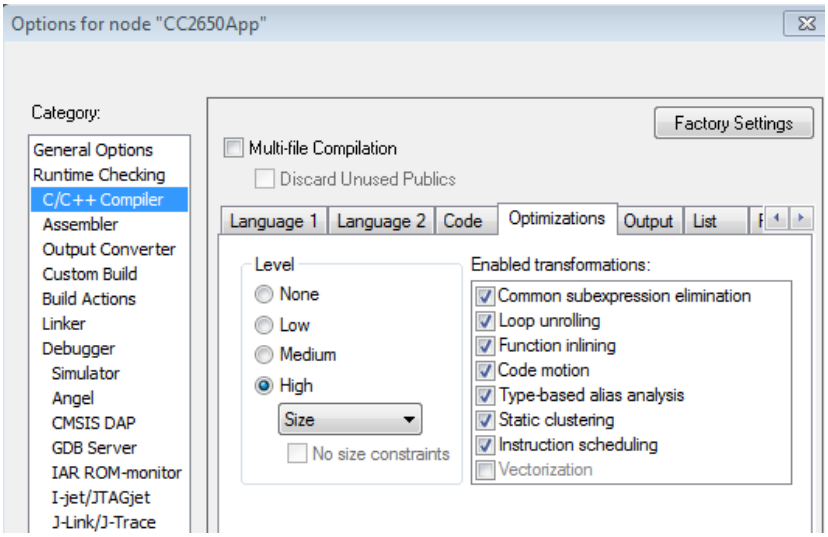
During debugging, it is sometimes useful to turn off or lower optimizations to ease single-stepping through code. This is possible at the following levels.

9.7.1 Project-wide optimizations

Note that there may not be enough available flash to do this.

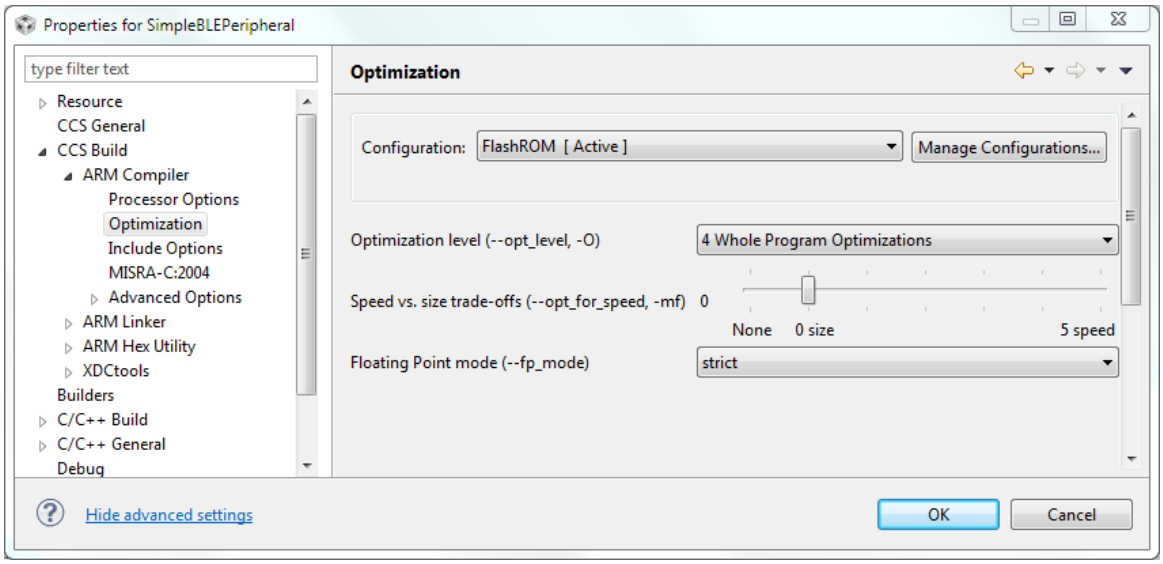
In IAR:

Project Options → C/C++ Compiler → Optimizations



In CCS:

Project Properties → CCS Build → ARM Compiler → Optimization

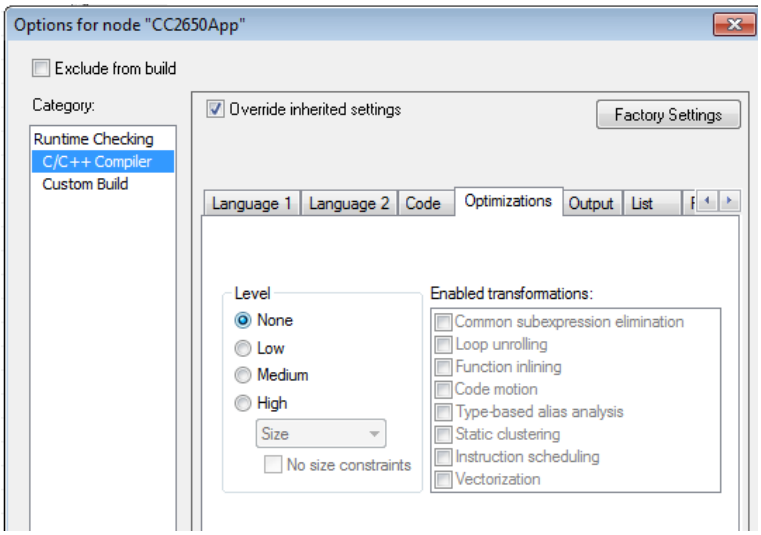


9.7.2 Single file optimizations

In IAR:

Right-click on the file in the "Workspace" pane and choose "Options." Then check "Override inherited Settings" and choose the desired optimization level:

***Do this with care as this also will override the project-wide preprocessor symbols.



In CCS:

Right-click on the file in the "Workspace" pane and choose "Properties". Change the file's optimization level using the same menu as shown in the CCS project-wide optimization menu.

9.7.3 Single function optimizations

Using compiler directives, it is possible to control the optimization level of a single function.

In IAR:

Use `#pragma optimize=none` before the function definition to unoptimize the entire function, i.e.:

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

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

9.8 Deciphering CPU Exceptions

There are several possible exception causes, and if an exception is caught, an exception handler function can be called. Depending on the project settings this 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, depending on the debugger, it may be caught immediately and the execution halted in debug mode, or if halted manually later via debugger 'Break', the execution 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*). This register is described in detail in the ARM Cortex-M3 User Guide [14]. Most exception causes fall into 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, etc) is accessed without being powered:
  - Typically (IM)PRECISERR exceptions

Note that the CFSR is available in View → Registers in IAR and CCS.

Most often when an access violation occurs, the exception type will be IMPRECISERR because writes to Flash and peripheral memory regions are mostly buffered writes.

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

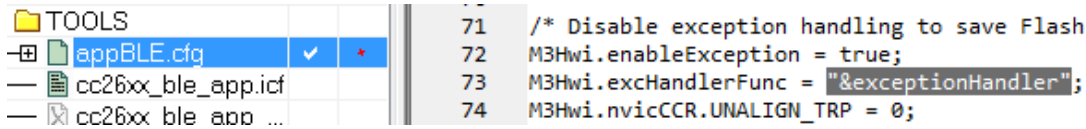
Tip: If the exception is IMPRECISERR, PRECISERR can be forced by manually disabling buffered writes. Set [CPU_SCS:ACTRL:DISDEFWBUF] to 1, either by manually setting the bit in the register view in IAR/CCS, or by including `<inc/hw_cpu_scs.h>` from Driverlib and calling:

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

Note that this will negatively affect performance.

9.8.2 Using a Custom Exception Handler

It is possible to use a custom exception handler instead of the default exception handler from ROM. In the sample projects, this is configured in *appBle.cfg*, via the `M3Hwi.excHandlerFunc` property:



When this function is called, the Core-M3 has already, at the time the exception was registered, pushed the core registers R0-3, R12, PC, LR and xPSR on the active task's run-time stack, and the TI-RTOS exception handler has pushed R4-11 onto the run-time stack as well.

9.8.3 Parsing the Exception Frame

The custom exception handler must be of the type:

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

where `execLr` is the LR value set by the Core-M3 and `e` points to the following structure which will describe the CPU state (core registers) at the time the exception happened:

```
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;
};
```

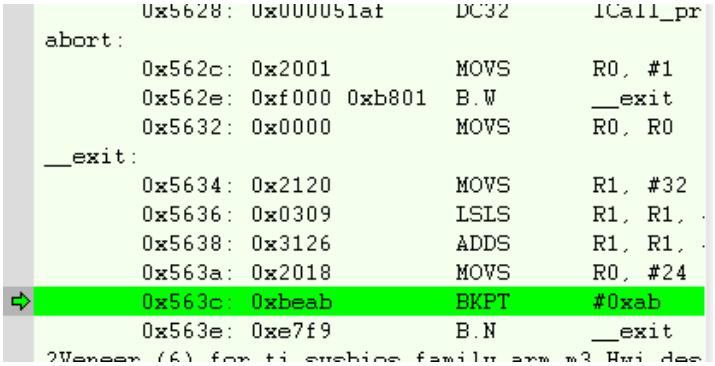
Note that, due to optimization, these variables will often not be shown properly in the IDE's watch windows. Therefore, a sample IAR implementation is shown here:

```
#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);
}
```

9.9 Debugging a Program Exit

The program should never exit the `main()` function. If this occurs, the disassembly will look something like this:



This is likely because the ICall_abort() function was called which can be caused, among other things, by:

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

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

9.10 Debugging Memory Problems

This section will describe how to debug a situation where the program runs out of memory, either on the heap or on the run-time stack for the individual thread contexts. Also, exceeding array bounds or dynamically allocating too little memory for a structure will corrupt the memory. If an exception like INVPC, INVSTATE, IBUSERR shows up in the CFSR register, this is a likely cause.

9.10.1 Task / System Stack Overflow

If there is an overflow on a task's run-time stack or the System Stack (as can be found using the ROV plug-in as described in Section 9.5.2 and 9.5.3), the following steps should be taken:

1. Note the current size of each task's run-time stack and increase by a few 100 bytes as described in Sections 3.3.1 and 3.11.3.
2. Check the stackPeaks using the ROV as described in Sections 9.5.2 and 9.5.3. If the peak is higher than the previous run-time stack size, the issue has been found.
3. If desired, reduce the run-time stack sizes so that they are still larger than their respective stackPeaks in order to save some memory.

9.10.2 Dynamic Allocation Errors

Using the ICall Heap profiling functionality described in Section 9.6,

- See if `memAlo` or `memMax` approach the preprocessor defined HEAPMGR_SIZE
- Check `memFail` to see if allocation failures have occurred
- Call the sanity check function.

If the heap is 'sane', but there are allocation errors, try to increase HEAPMGR_SIZE and see if the problem goes away.

It is also possible to 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 are used to configure system behaviour, features & resource usage at compile time. Some symbols are required as part of the BLE system, while others are configurable. Refer to 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. For example, to disable power management, change POWER_SAVING to xPOWER_SAVING.

9.11.2 Options

The preprocessor symbols used by the Application in the SimpleBLEPeripheral project are shown in Figure 38. Symbols that should never be modified are marked with an 'N' in the "Modify" column while modifiable/configurable symbols are marked with a 'Y'.

Preprocessor Symbol	Description	Modify
USE_ICALL	Required to use ICall BLE & primitive services.	N
POWER_SAVING	Power management is enabled when defined, 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 number of ICall aware tasks. Only modify 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. Only modify 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 define is required when using the LCD driver on the CC2650EM 7x7 Evaluation Module.	Y
TI_DRIVERS_LCD_INCLUDED	Includes SmartRF06 LCD Driver. This define 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 max number of simultaneous BLE collections allowed. Adding more connections uses more RAM and may require increasing HEAPMGR_SIZE. Be sure to profile heap usage accordingly.	Y
CC26XXWARE	This includes DriverLib.	N
CC26XX	This selects the chipset.	N
xdc_runtime_Assert_DISABLE_ALL	Disables XDC runtime assert	N
xdc_runtime_Log_DISABLE_ALL	Disables XDC runtime logging	N
HEAPMGR_METRICS	Enables collection of ICall heap metrics. See Section 9.6 for details on how to profile heap usage.	Y

Figure 38 Application Preprocessor Symbols

The following are the only stack preprocessor options that may be modified:

Preprocessor Symbol	Description	Modify
POWER_SAVING	Power management is enabled when defined, disabled when not defined. Requires same option in Application project.	Y
GATT_NO_CLIENT	When defined, the GATT client is not included in order to save flash. GATT Client excluded in most Peripheral projects, included in Central & certain Peripheral projects (e.g., TimeApp).	Y
OSAL_SNV	Select the number of NV pages to use for SNV. Each page is 4kB of flash. Minimum one page required when GAP_BOND_MANAGER is defined. See Section 3.10.4.	Y

Figure 39 Stack Preprocessor Symbols

9.12 Check System Flash/RAM usage with Map file

Both Application & Stack projects produce a map file which can be used to compute the combined Flash & RAM system memory usage. Both projects have their own memory space, therefore both map files must be analysed to determine the total system memory usage. The map file is located in the respective project's Output folder in IAR. To compute the total memory usage:

1. Open the Application map file (i.e., SimpleBLEPeripheralApp.map)
2. At the end of the file, there will be three lines containing a breakdown of memory usage for *readonly code*, *readonly data*, and *readwrite data*.
3. Add the two values for *readonly code* and *readonly data* memory. This sum is the total Flash memory usage for the Application project. The *readwrite data* memory is the total RAM usage by the Application project. Note these values.
4. Open the Stack map file and compute the same Flash & RAM values for the Stack project.
5. Add the total Flash memory value from the Application with the Stack's total Flash usage to determine the total system Flash usage. Similarly, add the total RAM usage from the Application with the Stack to get the total system RAM usage.

For CCS, the respective project's map file gives a summary of Flash & RAM usage.

To determine the remaining available memory for each project, refer to sections 3.10 (Flash) and 3.11 (RAM).

Note that due to section placement & alignment requirements, some remaining memory may not be available. The map file memory usage is only valid if the project builds & links successfully.

10 Creating a Custom BLE Application

By now the BLE system designer should have a firm grasp on the general system architecture, Application & BLE Stack framework required to implement a custom Bluetooth Smart application. This section provides indications on where and how to start writing a custom Application. First, it's required to decide what role and purpose the custom Application should have. If it's an application that is tied to a specific Service or Profile, it makes sense to start there. One example is the **Heart Rate** sensor project. It's good to start with one of the SimpleBLE sample projects:

- SimpleBLECentral
- SimpleBLEPeripheral
- SimpleBLEBroadcaster

- SimpleBLEObserver

10.1 Adding a Board File

Once the reference application has been selected, the next step is to define 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. It is recommended to use Evaluation Module (EM) board file as a starting reference. Modify the PIN structure and add the necessary peripheral driver initialization objects according to the board design.
2. Remove the existing EM board C & include files from the "Startup" Application's folder.
3. Add the custom board file to the Application project. Update the IDE's C compiler search path to point to the new board file's header file.
4. Define a new board file identifier and add the RF Front End and Bias configuration to the *bleUserConfig.h* file. Refer to the direction in this file for guidance on adding a new custom board RF configuration.
5. Rebuild the Application project.

10.2 Configuring Parameters for Custom Hardware

Set parameters, such as Sleep Clock Accuracy of the 32.768kHz crystal, and define CCFG parameters as needed. Refer to the TRM [2] for a description of CCFG configuration parameters.

10.3 Creating Additional Tasks

Many implementations can utilize the RTOS environment to operate in the Application task framework. If the system design requires the addition of an additional RTOS task, refer to Section 3.3.1 for guidance on adding a task.

10.4 Configuring the BLE Stack

Configure the BLE protocol stack with parameter & features. Available options are described in Sections 5.7 & 5.8.

10.4.1 Minimizing Flash Usage

To increase the available flash memory allocated to the Application project, it is necessary to minimize the stack's flash usage by only including BLE features that are needed to implement the defined role of the device. Note that this comes at the tradeoff of some features:

- Verify your application uses the "optimize for flash size" compiler optimization settings (default for TI projects).
- Do not use any NV pages, or use only 1 page of SNV. Set the Stack preprocessor option NO_OSAL_SNV and do not include GAP_BOND_MGR in *buildConfig.opt (*This will remove pairing/bonding capability). See section 3.10.4 for a description of SNV.
- For Peripheral devices, exclude the GATT client functionality by defining the GATT_NO_CLIENT predefined symbol in the Stack project. Peripheral devices generally do not implement the GATT client role.
- Do not include the GATT service changed characteristic by defining GATT_NO_SERVICE_CHANGED in *buildConfig.opt*.
- For devices that only use BT4.0 features, exclude Bluetooth 4.1 specific features from the BLE-Stack. For example, the BT4.1 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 Define BLE behavior

This step involves utilizing BLE protocol stack APIs to define the system behavior. This involves adding Profiles, defining the GATT database, and configuring the security model, etc. Use the concepts explained in Section 5 as well as the BLE API reference located in the Appendix of this guide.

11 Porting from CC254x to CC2640

11.1 Introduction

TI-RTOS is the new operating environment for BLE projects on CC26xx devices. It is a multithreaded environment where the protocol stack and application, plus its profiles, exist on different threads. It 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 BLE protocol stack. Although the incorporation of the RTOS is a major architecture change, BLE APIs, and related procedures, are largely similar to CC254x.

The following differences will be explained:

- OSAL
- Application and Stack separation with ICall
- Threads
- Semaphores
- Queues
- Peripheral Drivers
- Event Processing

Most of these differences are unique to TI-RTOS and will be explained further in detail and how it relates to OSAL.

11.2 OSAL

One of the big changes in moving to TI-RTOS is the application's complete removal from the OSAL environment. While the stack code still uses OSAL within its own thread, the application thread can no longer use OSAL's APIs, except for those defined in *ICallBleAPI.c*. Many functions, such as `osal_memcpy()`, `osal_memcmp()` and `osal_mem_alloc()` are now unavailable. In turn, they have been replaced by TI-RTOS, C runtime and ICall APIs.

11.3 Application and Stack Separation with ICall

In the CC2640 BLE 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. The main reason is to allow upgrade of application and stack images independently. The address of the startup entry for the stack image is known at build time so the Application image knows where the Stack image starts. Messages between the Application and Stack are passed through a framework developed called ICall short for indirect function calls. This allows the application to call the same APIs used in OSAL but is parsed by the ICall and sent to the stack for processing. Many of these stack function are defined in `ICallBleAPI.c` for the Application to use transparently while ICall handles the sending and receiving from the stack transparently.

11.4 Threads, Semaphores & Queues

Unlike single threaded operating systems such as OSAL, TI-RTOS is multithreaded with custom priorities for each thread. Thread synchronization is handled by the TIRTOS 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. They are used to pass event messages to the application's event processor. Profile callbacks which run in the context of the BLE protocol stack thread – and in the case of key presses and Clock events which are more

like ISRs – are made re-entrant by storing event data and posting an application's semaphore to process in the application's context.

Unique to TI-RTOS, queues are how applications process events in an ordered manner and make callback functions from profiles and the stack re-entrant. They also provide a FIFO ordering for event processing. An example project may use a queue to manage internal events from an Application profile, if needed, or a GAP profile role (e.g Peripheral or Central). ICall uses a queue as well and it is accessed via the ICall API.

Refer to Section 3 for a description of the TI-RTOS objects used by the BLE-Stack SDK.

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 will need to be ported to the respective TI-RTOS driver interfaces. Refer to Section 6 for details on adding and using a CC26xx peripheral driver.

11.6 Event Processing

Similar to OSAL, each RTOS task has two functions which implement the fundamental tasks for an application: `SimpleBLEPeripheral_init()` and `SimpleBLEPeripheral_taskFxn()`. The former contains ICall registration routines and initialization functions for the application profiles and the GAP and GATT roles. Function calls that would normally be found in the `START_DEVICE_EVT` event of the CC254x application are also made here. This includes setting up callbacks that the application should receive from the profile and stack layers. See Section 4.3.3 for more details on callbacks.

The latter function contains an infinite loop in which event are processed. Following entry of the loop and having just finished initialization, the application task calls ICall_wait() to block on its semaphore until an event occurs. See Section 4.3.2.2 for more information on how the application processes different events.

Similar to osal_set_event() in a CC254x application, the application task can post the application's semaphore 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. Also similar to `osal_start_timerEx()` in a CC254x application, the user can use a clock to set an event after a predetermined amount of time using `Util_constructClock()`.This function can also be used to set a periodic event as shown in the SimpleBLEPeripheral project.

Events come from internally (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 here too. In many cases such as in GAP role profile callbacks, it is necessary to 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.

12 Sample Applications

The purpose of this section is to give an overview of the sample applications that are included in the Texas Instruments BLE-Stack software development kit.

Some of these implementations are based on specifications that have been adopted by the Bluetooth Special Interest Group (BT SIG), while others are based on specifications that are a work-in-progress and have not been finalized. In addition, some applications are not based on any standardized profile being developed by the BT SIG, but rather are custom implementations developed by Texas Instruments.

Note that 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, most sample applications described in this section are intended 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 BLE peripheral device to provide an example blood pressure monitor 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 User Interface

There are two button inputs for this application.

SmarfRF Button Right

When not connected, this button is used to toggle advertising on and off. When in a connection, this increases the value of various measurements.

SmartRF Button Up

This button cycles through different optional measurement formats.

12.1.2 Basic Operation

Power up the device and press the right button to enable advertising. From a blood pressure collector peer device, initiate a device discovery and connection procedure to discover and connect to the blood pressure sensor. The peer device should discover the blood pressure service and configure 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 mfg and serial number.

Once blood pressure measurements have been enabled the application will begin sending data to the peer containing simulated measurement values. Pressing the up button cycles through different data formats as follows:

- **MMHG | TIMESTAMP | PULSE | USER | STATUS**
- **MMHG | TIMESTAMP**
- **MMHG**
- **KPA**
- **KPA | TIMESTAMP**
- **KPA |TIMESTAMP | PULSE**

If the peer device initiates pairing, the blood pressure sensor will request a passcode. The default passcode is “000000”.

Upon connection termination, the BPM will not begin to advertising again until the button is pressed.

The peer device may also query the blood pressure for read only device information. Further details on the supported items are listed in the GATT_DB excel sheet for this project. Examples are model number, serial number, etc.

12.2 Heart Rate Sensor

This sample project implements the Heart Rate and Battery profiles in a BLE 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 User Interface

When not connected, the SmarfRF's left button is used to toggle advertising on and off. When in a connection, the SmartRF's up button cycles through different heart rate sensor data formats. When in a connection and the battery characteristic is enabled for notification the battery level will be periodically notified.

12.2.2 Basic Operation

Power up the device and press the left button to enable advertising. From a heart rate collector peer device, initiate a device discovery and connection procedure to discover and connect to the heart rate sensor. The peer device should discover 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.

Once heart rate measurement notifications have been enabled the application will begin sending data to the peer containing simulated measurement values. Pressing the up button cycles 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 then the devices will 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 will start advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason the application will start 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 Cycling Speed and Cadence (CSC) profile in a BLE 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 Cycling Speed and Cadence Profile. This profile also makes use of the optional Device Info Service, which has default values that may be altered at compile or runtime to aid in identifying a specific BLE device. This project is configured to run on the SmartRF06 board.

12.3.1 User Interface

When not connected, the SmartRF's right button is used to toggle advertising on and off. When in a connection, the SmartRF's up button cycles through different cycling speed and cadence sensor data formats.

Pressing the select key initiates a "soft reset." This includes:

- Terminating all current connections
- Clearing all bond data
- Clearing white list of all peer addresses

12.3.2 Basic Operation

Power up the device and press the right button to enable advertising. From a CSC collector peer device, initiate a device discovery and connection procedure to discover and connect to the cycling sensor. The peer device should receive a slave security request and initiate a bond. Once bonded, the collector should discover the CSC service and configure it to enable cycling speed and cadence measurements.

Once CSC measurement notifications have been enabled the application will begin sending data to the peer containing simulated measurement values. Pressing the up button cycles 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 will start advertising at the fast interval for 30 seconds. If the sensor has successfully bonded to a peer device and stored the devices address in its white list, then for the first 10 seconds of advertising the sensor will only attempt to connect to any device addresses stored in its white list. After 10 seconds, the sensor will attempt to connect to any peer device that wishes to connect. Independent of white list use, after 30 seconds of fast interval connection, a 30 second period of slow interval advertising passes. After this, the device sleeps and waits for the right button to be pressed before resuming advertising again.

If the device terminates connection for any other reason, the sensor will advertise for 60 seconds at a slow interval and then sleep if no connection is made. It will begin advertising again only if the right button is pressed.

12.3.3 Neglect Timer

This device has a compile time option that allows the sensor to terminate a connection if it sees no user input for 15 seconds. In the context of this application, this means that 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, then this timer is permanently disabled at runtime.

12.4 Running Speed and Cadence (RSC) Sensor

This sample project implements the Running Speed and Cadence (RSC) profile in a BLE 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 Running Speed and Cadence 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.

12.4.1 User Interface

When not connected, the SmartRF's right button is used to toggle advertising on and off. When in a connection, the SmartRF's up button cycles through different running speed and cadence sensor data formats.

Pressing the select key initiates a "soft reset." This includes:

- Terminate all current connections
- Clearing all bond data
- Clearing white list of all peer addresses

12.4.2 Basic Operation

Power up the device and press the right button to enable advertising. From a RSC collector peer device, initiate a device discovery and connection procedure to discover and connect to the cycling sensor. The peer device should receive a slave security request and initiate a bond.

Once bonded, the collector should discover the RSC service and configure it to enable running speed and cadence measurements.

Once RSC measurement notifications have been enabled the application will begin sending data to the peer containing simulated measurement values. Pressing the up button cycles 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 will start advertising at the fast interval for 30 seconds. If the sensor has successfully bonded to a peer device and stored the devices address in its white list, then for the first 10 seconds of advertising the sensor will only attempt to connect to any device addresses stored in its white list. After 10 seconds, the sensor will attempt to connect to any peer device that wishes to connect. Independent of white list use, after 30 seconds of fast interval connection, a 30 second period of slow interval advertising passes. After this, the device sleeps and waits for the right button to be pressed before resuming advertising again.

If the device terminates connection for any other reason, the sensor will advertise for 60 seconds at a slow interval and then sleep if no connection is made. It will begin advertising again only if the right button is pressed.

12.4.3 Neglect Timer

This device has a compile time option that allows the sensor to terminate a connection if it sees no user input for 15 seconds. In the context of this application, this means that 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, then this timer is permanently disabled at runtime.

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.

12.5.1 User Interface

The SmartRF buttons and display provide a user interface for the application. The buttons used 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/from the currently 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 is used to display the following information:

- Devices BD address.
- 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 collector sends control point messages to a sensor by using write requests, while the sensor sends control point messages to the collector by using indications. When records are retrieved, the glucose measurement and glucose context are sent via notifications on their respective characteristics.

If an expected response is not received, the operation will time out after 30 seconds and the collector will close the connection.

12.6 Glucose Sensor

This sample project implements the Glucose profile in a BLE peripheral device to provide an example glucose sensor using simulated measurement data. The application implements the "Sensor" role of the Glucose Profile. It is compiled to run on a SmartRF06 board.

12.6.1 User Interface

When not connected, the right button is used to toggle advertising on and off. When in a connection, the up button sends a glucose measurement and glucose context.

12.6.2 Basic Operation

Power up the device and press the right button to enable advertising. From a glucose collector peer device, initiate a device discovery and connection procedure to discover and connect to the glucose sensor. The peer device should discover the glucose service and configure it to enable notifications of the glucose measurement. It may also enable notifications of the glucose measurement context.

Once 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 will be 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 will 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 BLE 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. It is configured to run on the SmartRF06 board

12.7.1 User Interface

When in a connection, the left button sends a “left arrow” key, the right button sends a “right arrow” key, the up button sends an “up arrow” key, and the down button sends a “down arrow” key.

Note that a secure connection must be established before key presses will be sent to the peer device.

12.7.2 Basic Operation

Power up the device and it will begin advertising by default. From a HID Host peer device, initiate a device discovery and connection procedure to discover and connect to the HID device. The peer device should discover the HID service and recognize 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 will disconnect. When the connection is closed the HID device will advertise again.

12.8 HostTest- BLE Network Processor

The HostTest project implements a pure BLE network processor, for use with an external microcontroller or a PC software application such as BTool. Communication is accomplished via the HCI interface.

12.9 KeyFob

The KeyFob application will demonstrate 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 Proximity Profile, Reporter role and Find Me Profile, Target role)
- Tx Power (for Proximity Profile, Reporter role)
- Battery
- Accelerometer
- SimpleKeys

The accelerometer and simple keys profiles are not aligned to official SIG profiles, but rather serve as an example of profile service implementation. The device information service and proximity-related services are based on adopted specifications.

12.9.1 User Interface

There are two button inputs for this application, an LED, and buzzer.

Right Button

When not connected, this button is used to toggle advertising on and off. When in a connection, this will register a key press which may be enabled to notify a peer device or may be read by a peer device.

Up Button

When in a connection, this will register a key press which may be enabled to notify a peer device or may be read by a peer device.

Buzzer

The buzzer turns on 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 actually 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". Once 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>. In addition, the values can be read by reading <ACCEL_X_UUID>, <ACCEL_Y_UUID>,<ACCEL_Z_UUID>.

12.9.4 Keys

The simple keys service on the SmartRF allows the device to 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 may read the value of the keys by reading <SK_KEYPRESSED_UUID>.

The peer device may enable key press notifications by writing a "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 allows the proximity reporter to begin an alert in the event 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 timeout until 20 seconds have gone by without receiving a packet. This "Supervision Timeout" value can be changed in the "Connection Services" tab; however the timeout 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 the link drops. Alternatively, the USB Dongle can be disconnected from the PC, effectively dropping the connection. Once the timeout expires, the alarm will be triggered. If a low alert was set, the SmartRF will make a low pitched beep. If a high alert was set, the SmartRF will make a high pitched beep. In either case, the SmartRF will beep ten times and then stop. Alternatively to stop the beeping, either a new connection can be formed with the SmartRF, or the button can be pressed.

12.10 SensorTag

The SimpleLink Bluetooth Smart CC2650 SensorTag 2.0 is a BLE peripheral slave device which runs on the CC2650 SensorTag reference hardware platform. Note that the SimpleLink CC2650 is a multi-standard wireless MCU which supports Bluetooth low energy as well as other wireless protocols. Software developed with the BLE-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 BLE protocol stack. The GATT server contains a primary service for each sensor for configuration and data collection. For a description of the available sensors, refer to http://www.ti.com/sensortag.

12.10.1 Operation

On start-up, the SensorTag is advertising with a 100ms 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 BLE compliant device and the main focus is on BLE compliant mobile phones, running either Android or iOS. The central device should be able to

- Scan and discover the Sensor Tag. (Scan response contain name "SensorTag")
- Establish connection based on user defined Connection Parameters
- Perform Service Discovery – Discover Characteristic by UUID
- Operate as a GATT Client - Write to and read from Characteristic Value

The Central Device shall initiate the connection and thereby become the Master.

To obtain the data, the corresponding sensor must first be activated, which is done via a Characteristic Value write to appropriate service.

The most power efficient way to obtain measurements for a sensor is to

- Enable notification
- Enable Sensor
- When notification with data is obtained at the Master side, disable the sensor (notification still on though)

Alternative do not use notifications at all, then simply

- Enable sensor
- Read data and verify
- Disable sensor

For the latter alternative please keep in mind that sensor take different amount of time to perform measurement. Depending on the connection interval (~10 – 4000 ms) set by the Central Device, the time for achieving measurement data can vary. The individual sensors require varying delays to complete measurements. Recommended setting is 100ms but for fast accelerometer and Magnetometer data updates, a lower is necessary. Notifications can be stopped and the sensors turned on/off

12.10.2 Sensors

The SensorTag has support for the following sensors:

- 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 very simple BLE central device with GATT client functionality. It makes use of the SmartRF05 + CC2650EM hardware platform. This project can be used as a framework for developing many different 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 SimpleProfile UUID. To modify this behaviour, set DEFAULT_DEV_DISC_BY_SVC_UUID to FALSE in *SimpleBLECentral.c*.

12.11.1 User Interface

The SmartRF buttons and display provide a user interface for the application. The buttons used are as follows:

- Up: If not connected, 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/from the currently selected device.
- Right: If connected, send a parameter update request.
- Down: If connected, start or cancel RSSI polling.

The LCD display is used to display the following information:

- Devices BD address.
- Device discovery results.
- Connection state.
- Pairing and bonding status.
- Attribute read / write value after parameter update.

12.12 SimpleBLEPeripheral

The SimpleBLEPeripheral project implements a very simple BLE peripheral device with GATT services and demonstrates the TI Simple Profile. This project can be used as a framework for developing many different peripheral-role applications. This project is thoroughly explained in the Software Developer's Guide.

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, via SPI or UART, running the SimpleNP network processor application. Project build configurations for the SimpleAP running on a SensorTag and SmartRF06 + CC2650EM evaluation module are provided. With SimpleAP project configuration, processing of GATT profile and service data is handled on the SimpleAP, while the GATT database & BLE-Stack reside on the SimpleNP network processor.

12.14 SimpleNP

The SimpleNP project implements the TI Simple Network Processor BLE device configuration. In this configuration, the CC2640 operates as a BLE network processor with the application & profiles executing off-chip on a host MCU. The SimpleNP network processor is ideal for designs where it is desired to add BLE capability to an existing embedded system where a host MCU or Application Processor is present. More details on how to interface to the SimpleNP, including the API interface, can be found in the Simple Network Processor API Guide [13].

12.15 TimeApp

This sample project implements time and alert-related profiles in a BLE peripheral device to provide an example of how Bluetooth LE 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. In addition, 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 User Interface

The SmartRF06 buttons and display provide a user interface for the application. The buttons are used as follows:

- Up: Start or stop advertising.
- Left: If connected, send a command to the Alert Notification control point.
- Center: If connected, disconnect. If held down on power-up, erase all bonds.
- Right: If connected, initiate a Reference Time update.
- Down: If connected, initiates a Ringer Control Point update

The LCD display is used to display the following information:

- Devices BD address.
- 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 Basic Operation

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". To connect, press Up to start advertising then

initiate a connection from a peer device. The connection status will be displayed. Once connected, the application will attempt 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 will cache handles of interest. When bonded to a peer device, the handles are saved so that the discovery procedure is not performed on subsequent connections.

If a service is discovered certain service characteristics are read and displayed. The network availability status and battery level will be displayed and the current time will be updated.

The application also enables notification or indication for characteristics that support these operations. This 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 will be displayed on the LCD.

The peer device may initiate pairing. If a passcode is required the application will generate and display 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 will start advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason the application will start advertising at the slow interval. The advertising intervals and durations are configurable in file *timeapp.c*.

12.16 Thermometer

This sample project implements a Health Thermometer and Device Information profile in a BLE 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 User Interface

There are two button inputs for this application.

Button Right

When not connected and not configured to take measurements, this button is used to toggle advertising on and off.

When in a connection or configured to take measurements, this increases the temperature by 1 degree Celsius. After 3 degrees in temperature rise, the interval will be set to 30 seconds and if configured, this will indicate to the peer an interval change initiated at the thermometer.

Button Up

This button cycle will through different measurement formats.

12.16.2 Basic Operation

Power up the device and press the right button to enable advertising. From a thermometer collector peer device, initiate a device discovery and connection procedure to discover and connect to the thermometer sensor. The peer device should discover the thermometer service and configure 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 mfg and serial number.

Once thermometer measurements have been enabled the application will begin sending data to the peer containing simulated measurement values. Pressing the up button cycles through different data formats as follows:

- **CELCIUS | TIMESTAMP | TYPE**
- **CELCIUS | TIMESTAMP**
- **CELCIUS**
- **FARENHEIT**
- **FARENHEIT | TIMESTAMP**
- **FARENHEIT | TIMESTAMP | TYPE**

If the peer device initiates pairing, the HT will request a passcode. The passcode is "000000".

The thermometer operates in the following states:

- **Idle –** In this state, the thermometer will wait for the right button to be pressed 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 will advertise to allow connection. The thermometer will periodically advertise in this state.
- **Connected Not Configured** - The thermometer may be configured to enable measurement reports. The thermometer will not send stored measurements until the CCC is enabled. Once connection is established, the thermometer sets a timer to disconnect in 20 seconds.
- **Connected Configured** - The thermometer will send any stored measurements if CCC is set to send measurement indications.
- **Connected Bonded** - The thermometer will send 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, etc.

I. *GAP API*

I.1 Commands

This section will detail the GAP commands from *gap.h* which the application will use. All other GAP commands are abstracted through the GAPRole or the GAPBondMgr.

Note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

uint16 GAP_GetParamValue (gapParamIDs_t paramID)

Description: Get a GAP parameter.

Parameters:
paramID – parameter ID (Appendix 0)

Returns:
GAP Parameter Value if successful
0xFFFF if *paramID* invalid

bStatus_t GAP_SetParamValue (gapParamIDs_t paramID, uint16 paramValue)

Description: Set a GAP parameter.

Parameters:
paramID – parameter ID (Appendix 0))
paramValue – new param value

Returns:
SUCCESS (0x00)
INVALIDPARAMETER (0x02): *paramID* is invalid

bStatus_t bStatus_t GAP_ConfigDeviceAddr(uint8 addrType, uint8 *pStaticAddr);

Description: Setup the device's address type. If ADDRTYPE_PRIVATE_RESOLVE is specified, the stack will change 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 can't be done during a connection.
INVALIDPARAMETER (0x02): invalid address type passed into function

void GAP_RegisterForMsgs(uint8 taskID);

Description: 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

I.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, thus advertising infinitely. Default is 0 (continue indefinitely)
TGAP_LIM_ADV_TIMEOUT	Time (sec) to remain advertising in Limited Discovery mode. Default is 180 seconds.
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 mSec)
TGAP_CONN_SCAN_WIND	Scan window used during Link Layer Initiating state, when in Connectable mode (n * 0.625 mSec)
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 mSec)
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 mSec)
TGAP_GEN_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in General Discovery proc (n * 0.625 mSec).
TGAP_GEN_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in General Discovery proc (n * 0.625 mSec)
TGAP_LIM_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in Limited Discovery proc (n * 0.625 mSec)
TGAP_LIM_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in Limited Discovery proc (n * 0.625 mSec)
TGAP_CONN_EST_INT_MIN	Minimum Link Layer connection interval, when using Connection Establishment proc (n * 1.25 mSec)
TGAP_CONN_EST_INT_MAX	Maximum Link Layer connection interval, when using Connection Establishment proc (n * 1.25 mSec)
TGAP_CONN_EST_SCAN_INT	Scan interval used during Link Layer Initiating state, when using Connection Establishment proc (n * 0.625 mSec)

TGAP_CONN_EST_SCAN_WIND	Scan window used during Link Layer Initiating state, when using Connection Establishment proc (n * 0.625 mSec)
TGAP_CONN_EST_SUPERV_TIMEOUT	Link Layer connection supervision timeout, when using Connection Establishment proc (n * 10 mSec)
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 min len of connection needed, when using Connection Establishment proc (n * 0.625 mSec)
TGAP_CONN_EST_MAX_CE_LEN	Local informational parameter about max len of connection needed, when using Connection Establishment proc (n * 0.625 mSec).
TGAP_PRIVATE_ADDR_INT	Minimum Time Interval between private (resolvable) address changes. In minutes (default 15 minutes)
TGAP_CONN_PAUSE_CENTRAL	Central idle timer. In seconds (default 1 second)
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 seconds.
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.

I.3 Events

This section will detail the events relating to the GAP layer that can be returned to the application from the BLE stack. Some of these events will be passed directly to the application and some will be handled by the GAPRole or GAPBondMgr layers. Regardless, they will be 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;
```

- 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 was 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;
```

- 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 needed. This is 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;
```

- 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;
```

II. *GAPRole peripheral Role API*

Note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

II.1 Commands

bStatus_t GAPRole_SetParameter(uint16_t param, uint8_t len, void *pValue)
Description: Set a GAP Role parameter. Parameters: *param* – Profile parameter ID (see Appendix II.2) *len* – length of data to write *pValue* – pointer to value to set parameter. This is dependent on the parameter ID and will be

cast to the appropriate data type

Returns:
SUCCESS (0x00)
INVALIDPARAMETER (0x02): *param* was not valid
bleInvalidRange (0x18): *len* is not valid for the given *param*
blePending (0x16)*:* previous param update has not been completed
bleIncorrectMode (0x12): can not start connectable advertising because non-connectable advertising is enabled

bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue)

Description: Set a GAP Role parameter.

Parameters:
param – Profile parameter ID (Appendix 0)
pValue – pointer to location to get parameter. This is dependent on the parameter ID and will be cast to the appropriate data type

Returns:
SUCCESS (0x00)
INVALIDPARAMETER (0x02): *param* was not valid

bStatus_t GAPRole_StartDevice(gapRolesCBs_t *pAppCallbacks)

Description: Initializes the device as a peripheral and configures the application callback function.

Parameters:
pAppCallbacks – pointer to application callbacks (Appendix II.3)

Returns:
SUCCESS (0x00)
bleAlreadyInRequestedMode (0x11): device was already initialized

bStatus_t GAPRole_TerminateConnection(void)

Description: 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)

Description: Update the parameters of an existing connection. See Section 5.1.5.

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

II.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 / 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 seconds. 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 0's.
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.
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 sec. 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 sec. 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 timeout to use for a param update (n * 10 ms). Range: 100 ms to 32 sec. 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 timeout.

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).

II.3 Callbacks

These 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.

II.3.1 State Change Callback (pfnStateChange)

This callback will pass the current GAPRole state to the application whenever the state changes. This function is of the type:

```
typedef void (*gapRolesStateNotify_t)(gaprole_States_t newState);
```

The various GAPRole states (`newState`) are enumerated as:

- 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 but not advertising, is in waiting period before advertising again
- GAPROLE_WAITING_AFTER_TIMEOUT //!< Device just timed out from a connection but is not yet advertising, 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

III. *GAPRole Central Role API*

Note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

III.1 Commands

bStatus_t GAPCentralRole_StartDevice(gapCentralRoleCB_t *pAppCallbacks)

Description: 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)

Description: Set a GAP Role parameter.

Parameters:
param – Profile parameter ID (Appendix **Error! Reference source not found.**)
len – length of data to write
pValue – pointer to value to set parameter. This is dependent on the parameter ID and will be 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)

Description: Set a GAP Role parameter.

Parameters:
param – Profile parameter ID (section **Error! Reference source not found.**)
pValue – pointer to buffer to contain the read data

Returns:
SUCCESS (0x00)
INVALIDPARAMETER (0x02): *param* was not valid

bStatus_t GAPCentralRole_TerminateLink (uint16_t connHandle);

Description: 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)

Description: 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): can't process now.
bleNoResources (0x15): too many links.

bStatus_t GAPCentralRole_UpdateLink(uint16_t connHandle, uint16_t connIntervalMin, uint16_t connIntervalMax, uint16_t connLatency, uint16_t connTimeout)

Description: Update the link connection parameters.

Parameters:
connHandle - connection handle
connIntervalMin - minimum connection interval in 1.25ms units
connIntervalMax - maximum connection interval in 1.25ms units
connLatency - number of LL latency connection events
connTimeout - connection timeout in 10ms units

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 BLE spec requirements: STO > (1 + Slave Latency) * (Connection Interval * 2)
LL_STATUS_ERROR_INACTIVE_CONNECTION (0x02): *connHandle* is not active
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 will not work because it is not a multiple / divisor of other simultaneous connection's intervals, or the connection's interval 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)

Description: 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)

Description: 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.

III.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 will be randomly generated.
GAPCENTRALROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0, which means the SRK will be 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.

III.3 Callbacks

These 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
{
  pfnGapCentralRoleEventCB_t  eventCB;  //!< Event callback.
} gapCentralRoleCB_t;
```

See the SimpleBLECentral application for an example.

III.3.1 Central event callback (eventCB)

This callback is used to pass GAP state change events to the application. It is of the type:

```
typedef uint8_t (*pfnGapCentralRoleEventCB_t)
(
  gapCentralRoleEvent_t *pEvent         //!< Pointer to event structure.
);
```

***NOTE! TRUE should be returned from this function if the GAPRole is to deallocate the event message. FALSE should be returned if the deallocation will be done by the application. Consider the 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 will deallocate 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.

As long as there is enough room in the heap, the message should always be successfully enqueued.

The possible GAPRole central states are listed here. See Appendix I.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

IV. *GATT / ATT API*

This section will describe 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 Appendix IV.4.

The possible return values are similar for all of these commands so they are described in Section IV.4. Note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

IV.1 General Commands

void GATT_RegisterForMsgs(uint8 taskID);

Description: Register a task ID to receive GATT local events and ATT response messages pending for transmission.

Parameters:
taskID – task ID to send events to

IV.2 Server Commands

bStatus_t GATT_Indication(uint16 connHandle, attHandleValueInd_t *pInd, uint8 authenticated, uint8 taskId);

Description: Indicates a characteristic value to a client and expect an acknowledgement.

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 acknowledgement

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 will receive a GATT_MSG_EVENT message with type ATT_HANDLE_VALUE_CFM upon an acknowledgement. It is only at this point that this subprocedure is considered complete.

bStatus_t GATT_Notification(uint16 connHandle, attHandleValueNoti_t *pNoti, uint8 authenticated)

Description: Notifies a characteristic value to a client.

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.

IV.3 Client Commands

bStatus_t GATT_InitClient(void)

Description: Initialize the GATT client in the BLE Stack.

Notes:
GATT clients should call this from the application init function.

bStatus_t GATT_RegisterForInd (uint8 taskId)

Description: Register to receive incoming ATT Indications or Notifications of attribute values.

Parameters:
taskId: task to forward indications or notifications to

Notes:
GATT clients should call this from the application initialization function.

bStatus_t GATT_ExchangeMTU(uint16 connHandle, attExchangeMTUReq_t *pReq, uint8 taskId);

Description: 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

Notes:
This 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 will receive an OSAL GATT_MSG_EVENT message. The type of the message will be 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.

bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId)

Description: 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 will receive 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 sub-procedure 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)

Description: 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) to look for
len: length of value
taskId: task to be notified of response

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive 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 sub-procedure 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_DiscPrimaryServiceByUUID(uint16 connHandle, uint8 *pValue, uint8 len, uint8 taskId)

Description: 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) to look for
len: length of value
taskId: task to be notified of response

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive 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 sub-procedure 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)

Description: 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 to search in
endHandle: end handle of primary service to search in
taskId: task to be notified of response

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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 GAPRole_GetParameter(uint16_t param, void *pValue)

Description: Get a GAP Role parameter.

Parameters:
param – Profile parameter ID (See Appendix VI.2)
pValue – pointer to a location to get the value. This 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)

Description: 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 to search in
endHandle: end handle of service to search in

taskId: task to be notified of response

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: 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 to search for.
taskId: task to be notified of response

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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_DiscAllCharDescs (uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Description: used by a client to find all the characteristic descriptor's Attribute Handles and AttributeTypes 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

Notes:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_FIND_INFO_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: Used to read a Characteristic Value 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

Notes:
If the return status is SUCCESS, the calling application task will receive an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: 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

Notes:
If the return status is SUCCESS, the calling application task will receive 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 sub-procedure 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.

bStatus_t GATT_ReadLongCharValue (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

Description: 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

Notes:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: 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

Notes:
If the return status from this function is SUCCESS, the calling application task will receive an OSAL GATT_MSG_EVENT message with type ATT_READ_MULTI_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: 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

Notes:
No response will be sent to the calling application task for this sub-procedure. If the Characteristic Value write request is the wrong size, or has an invalid value as defined by the profile, then the write will not succeed and no error will be generated by the server.

The payload must be dynamically allocated as described in Section 5.3.5.

bStatus_t GATT_SignedWriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Description: 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 sub-procedure shall 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

Notes:
No response will be sent to the calling application task for this sub-procedure. 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 will not succeed and no error will be 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)

Description: 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

Notes:
If the return status from this function is SUCCESS, the calling application task will receive an OSAL GATT_MSG_EVENT message with type ATT_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: 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

Notes:
If the return status from this function is SUCCESS, the calling application task will receive 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 sub-procedure 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_ReliableWrites (uint16 connHandle, attPrepareWriteReq_t *pReq, uint8 numReqs, uint8 flags, uint8 taskId)

Description: 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

Notes:
If the return status is SUCCESS, the calling application task will receive 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 sub-procedure 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.

Note: The payload must be dynamically allocated as described in Section 5.3.5.

bStatus_t GATT_ReadCharDesc (uint16 connHandle, attReadReq_t *pReq, uint8 taskId)

Description: Used to read a characteristic descriptor from a server when the client knows the characteristic descriptor declaration's Attribute handle.

Parameters:
connHandle: connection to use
pReq: pointer to request to be sent
taskId: task to be notified of response

Notes:
If the return status from this function is SUCCESS, the calling application task will receive an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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_ReadLongCharDesc (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

Description: Used to read a characteristic descriptor from a server when the client knows the characteristic descriptor declaration's Attribute handle 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

Notes:
If the return status is SUCCESS, the calling application task will receive multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure 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)

Description: Used to write a characteristic descriptor value to a server when the client knows the characteristic descriptor handle.

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)

Description: 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

Notes:
If the return status is SUCCESS, the calling application task will receive 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 sub-procedure 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.

IV.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 currently in a connection.
- blePending (0x17):
    - when returned to a client function: a response is still pending with the server or the GATT sub-procedure is still in progress
    - when returned to server function: confirmation from a client is still pending
- bleTimeout (0x16): the previous transaction timed out. No more ATT/GATT messages can be sent until the connection is re-established.
- bleMemAllocError (0x13): memory allocation error occurred
- bleLinkEncrypted (0x19): link is already encrypted. An Attribute PDU that includes an Authentication Signature should not be sent on an encrypted link

IV.5 Events

These will be received from the BLE stack in the application as a GATT_MSG_EVENT stack message sent through ICall. They will be received as the following structure where the `method` signifies the ATT Event and the `msg` is a union 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 will list the various ATT Events by their `method` and display their structure that is used in the `msg` payload. These 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;
```

- 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.

- 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)
  - 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;
```

IV.6 GATT commands and corresponding ATT events

This table will list all of the possible commands which may have caused a given 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

IV.7 ATT_ERROR_RSP errCode's

This section will list the possible error codes that can be present in the `ATT_ERROR_RSP` event and their possible causes.

- ATT_ERR_INVALID_HANDLE (0x01): Attribute handle value given was not valid on this attribute server
- ATT_ERR_READ_NOT_PERMITTED (0x02): Attribute cannot be read
- ATT_ERR_WRITE_NOT_PERMITTED (0x03): Attribute cannot be written
- ATT_ERR_INVALID_PDU (0x04): The attribute PDU was invalid
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): The attribute requires authentication before it can be read or written
- ATT_ERR_UNSUPPORTED_REQ (0x06): Attribute server doesn't support the request received from the attribute client
- ATT_ERR_INVALID_OFFSET (0x07): Offset specified was 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 found within the given attribute handle range
- ATT_ERR_ATTR_NOT_LONG (0x0B): Attribute cannot be read or written using the Read Blob Request or Prepare Write Request
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The Encryption Key Size used 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 that was requested has encountered an error that was very unlikely, and therefore 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 not a supported grouping attribute as defined by a higher layer specification
- ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient Resources to complete the request

V. GATTServApp API

This section will detail the API of the GATTServApp which is defined in *gattservapp_util.c*.

Note that these are only the public commands which should be called by the profile and / or application.

Also note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

V.1 Commands

void GATTServApp_InitCharCfg(uint16 connHandle, gattCharCfg_t *charCfgTbl)

Description: Initialize the client characteristic configuration table for a given connection. This should be used whenever 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)

Description: 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

gattAttribute_t *GATTServApp_FindAttr(gattAttribute_t *pAttrTbl, uint16 numAttrs, uint8 *pValue)

Description: 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)

Description: 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): not a valid value for a CCC
ATT_ERR_INVALID_VALUE_SIZE (0x0D): not a valid size for a CCC
ATT_ERR_ATTR_NOT_LONG (0x0B): offset needs to be 0
ATT_ERR_INSUFFICIENT_RESOURCES (0x11): CCC not found

VI. *GAPBondMgr API*

This section will detail the API of the GAPBondMgr which is defined in *gapbondmgr.c*.

Note that many of these commands do not need to be called from the application as they are called by the GAPRole or the BLE Stack.

Also note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

VI.1 Commands

bStatus_t GAPBondMgr_SetParameter(uint16_t param, void *pValue)

Description: Set a GAP Bond Manager parameter.

Parameters:
param – Profile parameter ID (see Appendix VI.2)
len – length of data to write
pValue – pointer to value to set parameter. This is dependent on the parameter ID and will be cast to the appropriate data type

Returns:
SUCCESS (0x00): parameter was set
INVALIDPARAMETER (0x02): *param* was not valid
bleInvalidRange (0x18): *len* is not valid for the given *param*

bStatus_t GAPBondMgr_GetParameter(uint16_t param, void *pValue)

Description: Get a GAP Bond Manager parameter.

Parameters:
param – Profile parameter ID (see Appendix VI.2)
pValue – pointer to a location to get the value. This 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)

Description: 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

void GAPBondMgr_LinkTerm(uint16_t connHandle)

Description: Notify the Bond Manager that a connection has been terminated.

Parameters:
connHandle - connection handle

void GAPBondMgr_SlaveReqSecurity(uint16_t connHandle)

Description: Notify the Bond Manager that a Slave Security Request is received.

Parameters:
connHandle - connection handle

uint8 GAPBondMgr_ResolveAddr(uint8 addrType, uint8 *pDevAddr, uint8 *pResolvedAddr)

Description: 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)

Description: Set/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)

Description: 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)

Description: Register callback functions with the bond manager.

Parameters:
pCB - pointer to callback function structure (See Appendix VI.3)

bStatus_t GAPBondMgr_PasscodeRsp(uint16 connectionHandle, uint8 status, uint32 passcode)

Description: 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)

Description: This is a bypass mechanism to allow the bond manager to process GAP messages.

Note: This is an advanced feature and shouldn't be called unless the normal GAP Bond Manager task ID registration is overridden

Parameters:
pMsg - GAP event message

Returns:
TRUE: safe to deallocate incoming GAP message,
FALSE: otherwise.

uint8 GAPBondMgr_CheckNVLen(uint8 id, uint8 len)

Description: This function will check 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

VI.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(in milliseconds)
GAPBOND_MITM_PROTECTION	R/W	uint8	Man-In-The-Middle (MITM) basically 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 0's.
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 - 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.

VI.3 Callbacks

These are functions whose pointers are passed from the application to the GAPBondMgr so that it can return events to the application as needed. They are passed as the following structure:

```
typedef struct
{
  pfnPasscodeCB_t     passcodeCB;       //!< Passcode callback
  pfnPairStateCB_t    pairStateCB;      //!< Pairing state callback
} gapBondCBs_t;
```

VI.3.1 Passcode Callback (passcodeCB)

This callback will return to the application the peer device info whenever a passcode is requested during the paring process. This function is defined as:

```
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 / outputs, the application should act accordingly by displaying the passcode or initiating the entrance of a passcode.

VI.3.2 Pairing State callback (pairStateCB)

This callback will return the current pairing state to the application whenever the state changes as well as the current status of the pairing / bonding process associated with the current state. This function is defined as:

```
typedef void (*pfnPairStateCB_t)
(
  uint16 connectionHandle,   //!< Connection handle
  uint8  state,              //!< Pairing state @ref GAPBOND_PAIRING_STATE_DEFINES
  uint8  status              //!< Pairing status
);
```

The pairing states (`state`) are enumerated as:

- GAPBOND_PAIRING_STATE_STARTED
  - The following `status` are possible for this `state`
    - SUCCESS (0x00): pairing has been initiated.
- GAPBOND_PAIRING_STATE_COMPLETE
  - The following `status` 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): IO 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 not supported
    - SMP_PAIRING_FAILED_ENC_KEY_SIZE (0x06): encryption key size is insufficient
    - SMP_PAIRING_FAILED_CMD_NOT_SUPPORTED (0x07): The SMP command received is not supported 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 `status` are possible for this `state`
    - LL_ENC_KEY_REQ_REJECTED (0x06): encryption key is missing
    - LL_ENC_KEY_REQ_UNSUPPORTED_FEATURE (0x1A): feature is not supported by the remote device
    - LL_CTRL_PKT_TIMEOUT_TERM (0x22): Timeout waiting for response
    - bleGAPBondRejected (0x32): this is received due to one of the above three errors

VII. L2CAP API

VII.1 Commands

This section will describe the API related to setting up bidirectional communication between two BLE devices using L2CAP connection orientated channels.

Also note that the return values described here 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 will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

bStatus_t L2CAP_RegisterPsm(l2capPsm_t *pPsm)

Description: Register a Protocol/Service Multiplexer with L2CAP.

Parameters:
pPsm: PSM to deregister

Returns:
SUCCESS (0x00): Registration was successful.
INVALIDPARAMETER (0x02): Max 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)

Description: Deregister a Protocol/Service Multiplexer with L2CAP.

Parameters:
taskId - task PSM belongs to.
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)

Description: Get information about a given registered PSM.

Parameters:
pPsm: PSM Id.
pInfo - structure to copy PSM info into.

Returns:
SUCCESS (0x00): Operation was successful.
INVALIDPARAMETER (0x02): PSM is not registered.

bStatus_t L2CAP_PsmChannels(uint16 psm, uint8 numCIDs, uint16 *pCIDs)

Description: Get all active channels for a given registered PSM.

Parameters:
pPsm: PSM Id.
numCIDs - number of CIDs can be copied.
pCIDs - structure to copy CIDs into.

Returns:
SUCCESS (0x00): Operation was successful.
INVALIDPARAMETER (0x02): PSM is not registered.

bStatus_t L2CAP_ChannelInfo(uint16 CID, l2capChannelInfo_t *pInfo)

Description: Get information about a given active Connection Oriented Channnel.

Parameters:

CID - local channel id.
pInfo - structure to copy channel info into.

Returns:
SUCCESS (0x00): Registration was successful.
INVALIDPARAMETER (0x02): No such a channel.

bStatus_t L2CAP_ConnectReq(uint16 connHandle, uint16 psm, uint16 peerPsm)

Description: 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.

bStatus_t L2CAP_ConnectRsp(uint16 connHandle, uint8 id, uint16 result)

Description: Send Connection Response.

Parameters:
connHandle - connection to create channel on
psm - local PSM
peerPsm - peer PSM

Returns:
SUCCESS: (0x00) Request was sent successfully.
INVALIDPARAMETER (0x02): PSM is not registered or Channel is not open.
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)

Description: 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.

bStatus_t L2CAP_FlowCtrlCredit(uint16 CID, uint16 peerCredits)

Description: 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)

Description: 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): In the middle of another transmit.
bleInvalidMtuSize (0x1B): SDU size is larger than peer MTU.

VIII. *HCI API*

This section will describe the vendor specific HCI Extension API, the HCI LE API, and the HCI Support API. In the case where more detail is needed, an example will be provided.

Note that, unless stated otherwise, the return values for all of these commands will always be SUCCESS. However, this does not indicate successful completion of the command. These commands will result in corresponding events that should be checked by the calling application.

If ICall is incorrectly configured or does not have enough memory to allocate a message, the command will never get processed by the BLE stack. In this case, one of the ICall return values from Appendix IX will be returned.

VIII.1 Commands

hciStatus_t HCI_EXT_AdvEventNoticeCmd (uint8 taskID, uint16 taskEvent)

Description: This command is used to configure the device to set an event in the user task after each advertisement event completes. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent is taken to be "disable".

Note: This command will not return any events but it does have a meaningful return status.

Note: This command requires additional checks in the task function as described in Section 4.3.2.1

Parameters:
taskID– User's task ID.
taskEvent- User's task event. This must be a single bit value.

Returns:
SUCCESS: event configured correctly
LL_STATUS_ERROR_BAD_PARAMETER: there is more than one bit set.

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 BLE Protocol Stack to return the event (in `simpleBLEPeripheral_init()`)

```
HCI_EXT_AdvEventNoticeCmd( selfEntity, SBP_ADV_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_ADV_CB_EVT)
         {
            // Advertisement ended. Process as desired.
         }
…
```

hciStatus_t HCI_EXT_BuildRevision(uint8 mode, uint16 userRevNum)

Description: This command is used to a) allow the embedded user code to set their own 16 bit revision number, or b) read the build revision number of the BLE Stack library software. The default value of the user revision number is *zero*.

When the user updates a BLE project by adding their own code, they may use this API to set their own revision number. When called with mode set to HCI_EXT_SET_APP_REVISION, the stack will save this value. No event will be returned from this API when used this way as it is intended to be called from within the target itself. Note however that this does not preclude this command from being received via the HCI. However, no event will be 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 succesfully
LL_STATUS_ERROR_BAD_PARAMETER: not a valid mode

Corresponding Events (only when mode == HCI_EXT_SET_USER_REVISION):
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ConnEventNoticeCmd (uint8 taskID, uint16 taskEvent)

Description: This command is used to configure the device to set an event in the user task after each connection event completes. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent is taken to be "disable".

Note: Only a Slave with one connection is supported (this API will only work while device is configured as a slave and connected to one master). This command should be sent AFTER a connection is established.

Note: This command will not return any events but it does have a meaningful return status.

Note: This command requires additional checks in the task function as described in Section 4.3.2.1

Parameters:
taskID– User's task ID.
taskEvent- User's task event.

Returns:
SUCCESS or FAILURE
LL_STATUS_ERROR_BAD_PARAMETER: there is more than one bit set.

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 BLE Protocol Stack to return the event (in `SimpleBLEPeripheral_processStateChangeEvt())` AFTER the connection is established.

```
    case GAPROLE_CONNECTED:
      {
        HCI_EXT_ConnEventNoticeCmd ( selfEntity, SBP_CONN_EVT_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)*

Description: This command is used to decrypt encrypted data using the AES128 .

Note: This should only be used by the application. Incoming encrypted BLE data is automatically decrypted by the stack and does not require the use of this API.

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)

Description: This command is used to disconnect a connection immediately. This command can be useful for when a connection needs to be ended without the latency associated with the normal BLE Controller Terminate control procedure.

Note that the Host issuing the command will still receive the HCI Disconnection Complete event with a Reason status of 0x16 (i.e. 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

hciStatus_t HCI_EXT_EnablePTMCmd (void)

Description: This command is used to enable Production Test Mode (PTM). This mode is used by the customer during assembly of their product to allow limited access to the BLE Controller for testing and configuration. This mode will remain enabled until the device is reset. Please see the related application note for additional details.

Note: This commands will cause a reset of the controller so in order to re-enter the application, the device should be reset.

Note: This command will not return any events.

Return Values:
HCI_SUCCESS: Successfully entered PTM

hciStatus_t HCI_EXT_EndModemTestCmd (void)

Description: This command is used to shutdown a modem test. A complete link layer reset will take place.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_GetConnInfoCmd(uint8 *numAllocConns, uint8 *numActiveConns, hciConnInfo_t *activeConnInfo)

Description: This command is used to get 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)

Note: If all the parameters are NULL, then the command is assumed to have originated from the transport layer. Otherwise, they are assumed to have originated from a direct call by the application and any non-NULL pointer will be used.

Parameters:
numAllocConns – pointer to number of build time connections allowed
numActiveConns - pointer to number of active BLE connections
activeConnInfo - pointer for active connection information

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

Description: 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..39) every 625us. Use the HCI_EXT_EndModemTest command to end the test.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Note: The device will transmit at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemTestRxCmd(uint8 rxFreq)

Description: This API is used to start 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 may 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.

Note: The RF channel, not the BLE frequency, is specified! The RF channel can be obtained from the BLE frequency as follows: RF Channel = (BLE Frequency – 2402) / 2.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Parameters:
rxFreq- selects which channel [0 to 39] to receive on

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemTestTxCmd(uint8 cwMode, uint8 txFreq)

Description: This API is used to start 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.

Note: The RF channel, not the BLE frequency, is specified by txFreq. The RF channel can be obtained from the BLE frequency as follows: RF Channel = (BLE Frequency – 2402) / 2.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Note: The device will transmit at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Parameters:
cwMode - HCI_EXT_TX_MODULATED_CARRIER, HCI_EXT_TX_UNMODULATED_CARRIER
txFreq - Transmit RF channel k=0..39, where BLE F=2402+(k*2MHz)

Corresponding Event:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_NumComplPktsLimitCmd (uint8 limit, uint8 flushOnEvt)

Description: This command is used to set 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 will be returned (if non-zero) based on the **flushOnEvt** flag. The limit can be set from one to the maximum number of HCI buffers (please see the LE Read Buffer Size command in the Bluetooth Core specification). The default limit is *one*; the default **flushOnEvt** flag is *FALSE*.

Note: The purpose of this command is to minimize the overhead of sending multiple Number of Completed Packet events, thus maximizing the processing available to increase over-the-air throughput. This is often used in conjunction with HCI_EXT_OverlappedProcessingCmd.

Parameters:
limit– From 1 to HCI_MAX_NUM_DATA_BUFFERS (returned by HCI_LE_ReadBufSizeCmd).
flushOnEvt-
- HCI_EXT_DISABLE_NUM_COMPL_PKTS_ON_EVENT: only return 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

hciStatus_t HCI_EXT_OnePacketPerEventCmd (uint8 control)

Description: This command is used to configure the Link Layer to only allow 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.

Note: A thorough power analysis of the system needs to be performed before it is certain that this command will save power. It may be more power efficient to transfer multiple packets 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 will only be returned if the setting is changing from enable to disable or vice versa

hciStatus_t HCI_EXT_PacketErrorRateCmd (uint16 connHandle, uint8 command)

Description: 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.

Note: The counters are only 16 bits. At the shortest connection interval, this provides a little over 8 minutes of data.

Parameters:
connId– The connection ID on which to perform the command
command- HCI_EXT_PER_RESET, HCI_EXT_PER_READ

Corresponding Event:
HCI_VendorSpecifcCommandCompleteEvent

*hciStatus_t HCI_EXT_PERbyChanCmd (uint16 connHandle, perByChan_t *perByChan)*

Description: This command is used to start or end 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 will be discontinued. If ***perByChan** is not NULL, then it is assumed that there is sufficient memory at this location for the PER data, 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: **It is the user's responsibility to ensure there is sufficient memory allocated in the perByChan structure!** The user is also responsible for maintaining the counters, clearing them if required before starting accumulation.

Note: The counters are 16 bits. At the shortest connection interval, this provides a bit over 8 minutes of data.

Note: 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

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

Description: 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..39) every 625us. Use the HCI_EXT_EndModemTest command to end the test.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Note: The device will transmit at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_ReadBDADDRCmd(void)

Description: This command is used to read the device's BLE address (BDADDR).

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent where the BDADDR will be a parameter

hciStatus_t HCI_ReadTransmitPowerLevelCmd(uint16 connHandle, uint8 txPwrType)

Description: This command is used to read 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)

Description: This command is used to issue a hard or soft system reset. A hard reset is caused by a watchdog timer timeout, while a soft reset is caused by jumping to the reset ISR.

Note that the reset occurs after a 100 ms delay in order to allow the correspond event to be returned to the application.

Note: only hard reset is allowed. A soft reset will cause the command to fail. See Section 8.2.

Parameters:
mode – HCI_EXT_RESET_SYSTEM_HARD

Corresponding event:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetBDADDRCmd(uint8 *bdAddr)

Description: This command is used to set the device's BLE address (BDADDR). This address will override the device's address determined when the device is reset). To restore the device's initialized address stored in flash, issue this command with an invalid address (0xFFFFFFFFFFFF).

Note: This command is only allowed when the Controller is in the Standby state. This command is intended to only be used during initialization. Changing the device's BDADDR after various BLE operations have already taken place may cause unexpected problems.

Parameters:
bdAddr – A pointer to a buffer to hold this device's address. An invalid address (i.e. all FF's) will restore this device's address to the address set at initialization.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd (uint8 control)

Description: This command is used to configure the Link Layer fast transmit response time feature. The default system value for this feature is *enabled*.

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 LL connection with as little delay as possible, even when the connection is configured to use slave latency. That is, the transmit response time will tend to be no longer than the connection interval (instead of waiting for the next

effective connection interval due to slave latency). This results in lower power savings since the LL may need to wake to transmit during connection events that would normally have been skipped due to slave latency. If saving power is more critical than fast transmit response time, then this feature can be disabled using this command. When disabled, the transmit response time will be no longer than the effective connection interval (slave latency + 1 times the connection interval).

Parameters:
control – HCI_EXT_ENABLE_FAST_TX_RESP_TIME, HCI_EXT_DISABLE_FAST_TX_RESP_TIME

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetLocalSupportedFeaturesCmd (uint8 * localFeatures)

Description: This command is used to set the Controller's Local Supported Features.

Note: This command can be issued either before or after one or more connections are formed. However, the local features set in this manner are only effective if performed *before* a Feature Exchange Procedure has been initiated by the Master. Once this control procedure has been completed for a particular connection, only the exchanged feature set for that connection will be used. Since the Link Layer may initiate the feature exchange procedure autonomously, it is best to use this command before the connection is formed.

Note that the features are initialized by the controller upon start up. The need for this command is very unlikely. The defines for the feature values are in ll.h.

Parameters:
localFeatures – A pointer to the Feature Set where each bit where each bit corresponds to a feature
0: Feature shall not be used.
1: Feature can be used.

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd (uint8 txPower)

Description: This command is used to override the RF transmitter output power used by the Direct Test Mode (DTM). Normally, the maximum transmitter output power setting used by DTM is the maximum transmitter output power setting for the device (i.e. 5 dBm). This command will change the value used by DTM.

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.
Parameters:
txPower – one of:
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

hciStatus_t HCI_EXT_SetSCACmd (uint16 scaInPPM)

Description: This command is used to set this device's Sleep Clock Accuracy (SCA) value, 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 the Bluetooth Spec [15], which will be 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 50ppm and 40ppm, respectively.

Note: This command is only allowed when the device is *not* in a connection.

Note: The device's SCA value remains unaffected by an HCI Reset.

Parameters:
scaInPPM – This device's SCA in PPM from 0..500.

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd (uint8 mode)

Description: This command is used to enable or disable the Slave Latency Override, allowing the user temporarily suspend Slave Latency even though it is still active for the connection. That is, once enabled, the device will wake up for every connection until Slave Latency Override is disabled again. The default value is *Disable.*

Note: This only applies to devices acting in the Slave role.

This can be helpful when the Slave application knows it will soon receive something that needs to be handled without delay. Note that this does not actually change the slave latency connection parameter: the device will simply wake up for each connection event.

Parameters:
control – HCI_EXT_ENABLE_SL_OVERRIDE, HCI_EXT_DISABLE_SL_OVERRIDE

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetTxPowerCmd(uint8 txPower)

Description: This command is used to set the RF transmitter output power. The default system value for this feature is *0 dBm*.

Parameters:
txPower– Device's transmit power, one of:

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)

Description: This command is used to read 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 SimpleBLECentral project's application task.

Note: If the Receiver Modem test is running, then the RF RSSI for the last received data will be returned. If there is no RSSI value, then HCI_RSSI_NOT_AVAILABLE will be returned

Parameters:

connHandle – connection handle of connection to read the RSSI from

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

VIII.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 error code 'SUCCESS' 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 table below provides 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

Table 1: List of Possible Host Error Codes

IX. ICall API

IX.1 Commands

The ICall commands which are useful from the application task are defined in Section 4.2.

IX.2 Error Codes

This section will list the error codes associated with ICall failures. Note that it is possible for these to 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