CC2540 and CC2541 *Bluetooth*® low energy Software Developer's

Reference Guide

Literature Number: SWRU271G
October 2010–Revised September 2015

Contents

List of Figures

List of Tables

References

The following references are included with the TI *Bluetooth*® low energy v1.4.1 stack release.

NOTE: Path and file references in this document assume you have installed the BLE development kit software to the path: *C:\Texas Instruments\BLE-CC254X-1.4.1*. This path is referred to as $INSTALL$.

1. *TI BLE Vendor-Specific HCI Reference Guide*, $INSTALL$\Documents\TI_BLE_Vendor_Specific_HCI_Guide.pdf
2. *TI CC2540 Bluetooth low energy API Guide*, $INSTALL$\Documents\BLE_API_Guide_main.htm
3. *Advanced Remote Control Quick Start Guide*, $INSTALL$\Documents\TI_CC2541_ARC_Quick_Start_Guide.pdf
4. *Advanced Remote Control User's Guide*, $INSTALL$\Documents\TI_CC2541_ARC_User_Guide.pdf
5. *TI CC2540 Bluetooth low energy Sample Applications Guide*, $INSTALL$\Documents\TI_BLE_Sample_Applications_Guide.pdf
6. *Universal Bootloader (UBL) Guide*, $INSTALL$\Documents\Universal Boot Loader for SOC-8051 by USB-MSD Developer's Guide.pdf
7. *OSAL API Guide*, $INSTALL$\Documents\osal\OSAL API.pdf
8. *HAL API Guide*, $INSTALL$\Documents\hal\HAL API.pdf

   Also available for download from TI.com:

9. *TI CC2540DK-MINI Bluetooth low energy User Guide v1.1* (SWRU270)
10. *Measuring Power Consumption Application Note* (SWRA347)
11. *CC2541/43/44/45 Peripherals Software Examples* (SWRC257)
12. *CC254x Chip User's Guide* (SWRU191)

    Available for download from the *Bluetooth* Special Interest Group (SIG) website:

13. *Specification of the Bluetooth System, Covered Core Package version: 4.0 (30-June-2010)*, https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737
14. *Device Information Service (Bluetooth Specification), version 1.0 (24-May-2011)*, https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=238689

    Links

15. TI *Bluetooth* low energy Wiki-page: www.ti.com/ble-wiki
16. Latest stack download: www.ti.com/ble-stack
17. Support forum: www.ti.com/ble-forum

Overview

The purpose of this document is to give an overview of the TI CC2540 and CC2541 *Bluetooth* low energy (BLE) software development kit. This document also introduces the BLE standard but should not be used as a substitute for the complete specification. For more details, see the *Specification of the Bluetooth System*, Covered Core Package Version: 4.0 (30-June-2010).

IAR Overview has the release history of the BLE software development kit, including detailed information on changes, enhancements, bug fixes, and issues.

NOTE: The TI BLE Stack™ v1.4.1 supports *Bluetooth* 4.0. For *Bluetooth* 4.1 support, see the TI BLE Stack for the SimpleLink™ *Bluetooth* low energy CC2640 Wireless MCU.

1.1 Introduction

Version 4.0 of the *Bluetooth* standard supports the following two systems of wireless technology:

- Basic Rate (BR, often referred to as BR/EDR for Basic Rate/Enhanced Data Rate)
- *Bluetooth* low energy (BLE)

The BLE protocol was created to transmit 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 should be branded as *Bluetooth* Smart Ready. Typically in a *Bluetooth* system, a mobile phone or laptop computer will be a dual-mode device. Devices that only support BLE are referred to as single-mode devices and should be branded as *Bluetooth* Smart. These single-mode devices are used for application in which low power consumption is a primary concern, such as those that run on coin-cell batteries. Figure 1-1 shows the *Bluetooth* Smart and *Bluetooth* Smart Ready branding marks.

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

1.2 BLE Protocol Stack Basics

Figure 1-2 shows the BLE protocol stack architecture.

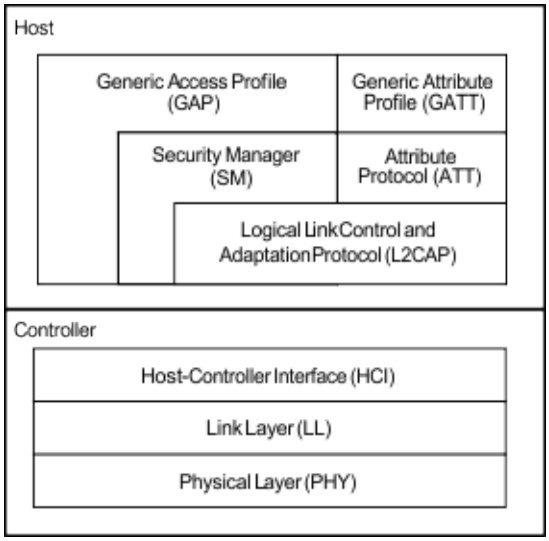


Figure 1-2. BLE Protocol Stack

The BLE protocol stack (or protocol stack) consists of two sections: the controller and the host. This separation of controller and host derives from standard *Bluetooth* BR/EDR devices, where the two sections were often implemented separately. Profiles and applications are implemented in Figure 1-2 of the generic access protocol (GAP) and generic attribute protocol (GATT) layers of the protocol stack.

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

The link layer (LL) controls the RF state of the device.

The device has five possible states:

- standby
- advertising
- scanning
- initiating
- connected

Advertisers transmit data without forming a BLE connection, while scanners receive the data broadcasted by advertisers. An initiator is a device that responds to an advertiser by requesting to connect. If the advertiser accepts, both the advertiser and initiator connect. When a device is in a connected state, it is either a master or slave. The device that initiated the connection becomes the master and the device that accepted the request becomes the slave. This layer is implemented in the library code in the TI 1.4.1 BLE stack, .

The host control interface (HCI) layer provides a means of communication between the host and controller through a standardized interface. This layer can be implemented either through a software API, or by a hardware interface such as UART, SPI, or USB. Device Information Service (Bluetooth Specification), Version 1.0 (24-May-2011) specifies the standard HCI commands and events. The TI BLE Vendor Specific HCI Reference Guide specifies the TI proprietary commands and events.

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. For more information on TI's implementation of the L2CAP layer, see Section 5.5.

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 with and exchange data with another device. For more information on TI's implementation of the SM layer, see Section 5.3.

The GAP layer directly interfaces with the application and/or profiles to handle device discovery and connection-related services for the device. GAP also handles the initiation of security features. For more information on the GAP layer, see Section 5.1.

The attribute protocol (ATT) layer protocol lets a device expose certain pieces of data (attributes) to another device.

The GATT layer is a service framework that defines the subprocedures for using ATT. GATT subprocedures handle data communications between two devices in a BLE connection. The application and/or profiles use GATT directly. For more information on the ATT and GATT layers, see Section 5.4.

The TI BLE Software Development Platform

The TI royalty-free, BLE software development kit is a complete software platform for developing single-mode BLE applications. The kit is based on the CC2540/41 complete System-on-Chip (SoC) solution. The solution combines a 2.4-GHz RF transceiver, microcontroller, up to 256KB of in-system programmable memory, 8KB of RAM, and a full range of peripherals.

2.1 Configurations

The platform supports two different stack and application configurations:

- Single-Device: The controller, host, profiles, and application are implemented on the CC2540/41 as a true single-chip solution. This configuration is the simplest and most common when using the CC2540/41 devices. TI uses this configuration in most sample projects. The configuration is the most cost effective and provides the lowest-power performance. The SimpleBLEPeripheral and SimpleBLECentral projects are examples of applications built using the single-device configuration. For more information on these projects, see Chapter 3.

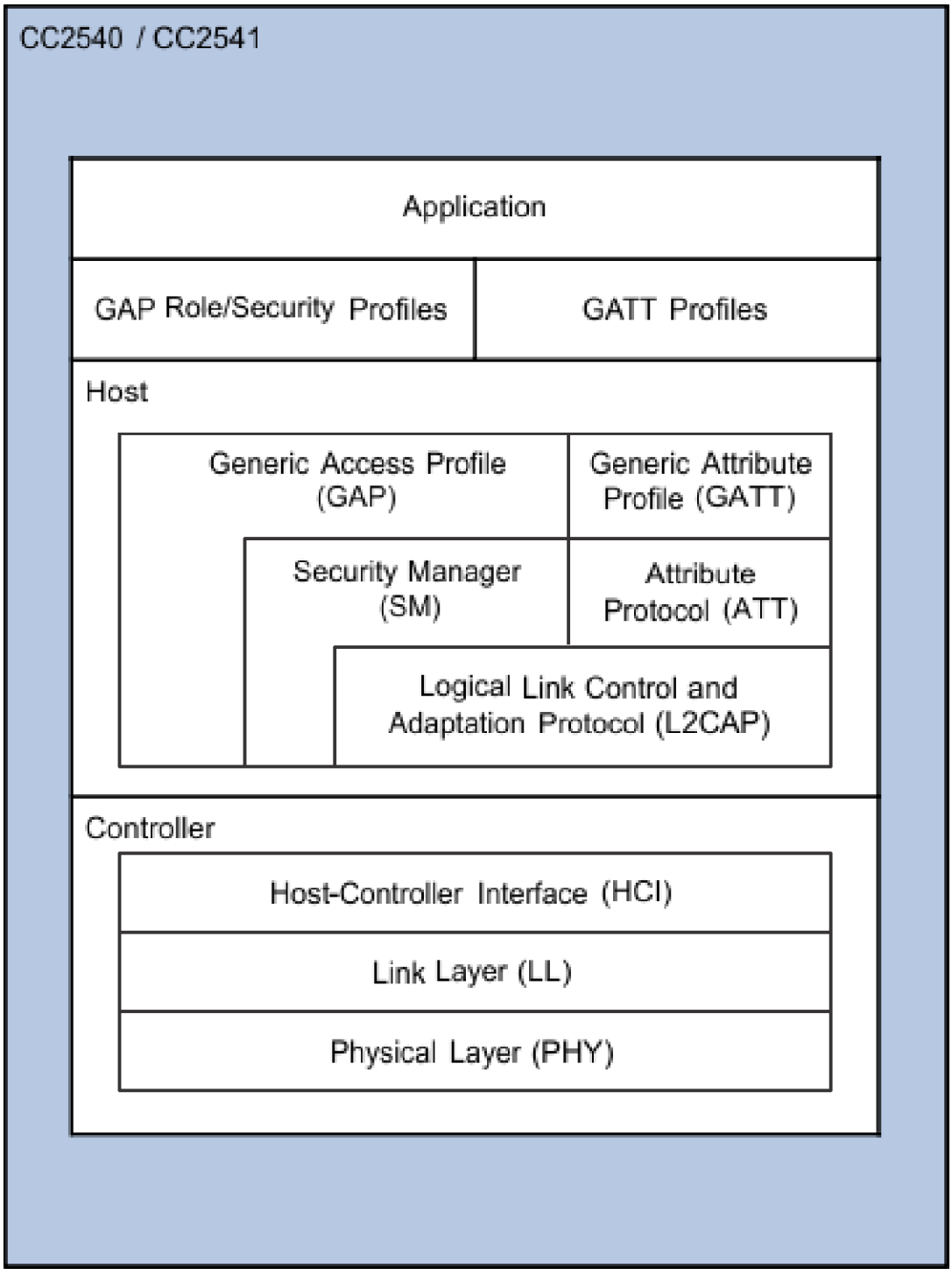


Figure 2-1. Single-Device Configuration

- Network Processor: The controller and host layers are implemented together on the CC2540/41, while the profiles and the application are implemented separately on an external host processor. The application and profiles communicate with the CC2540/41 through vendor-specific HCI commands using an SPI, a UART interface, or a virtual UART interface over USB. This configuration is optimal for applications that execute on another device such as an external microcontroller or a PC. When using this type of application, you can develop it externally while running the BLE stack on the CC2540/41. To use the network processor, you must use the HostTestRelease project.

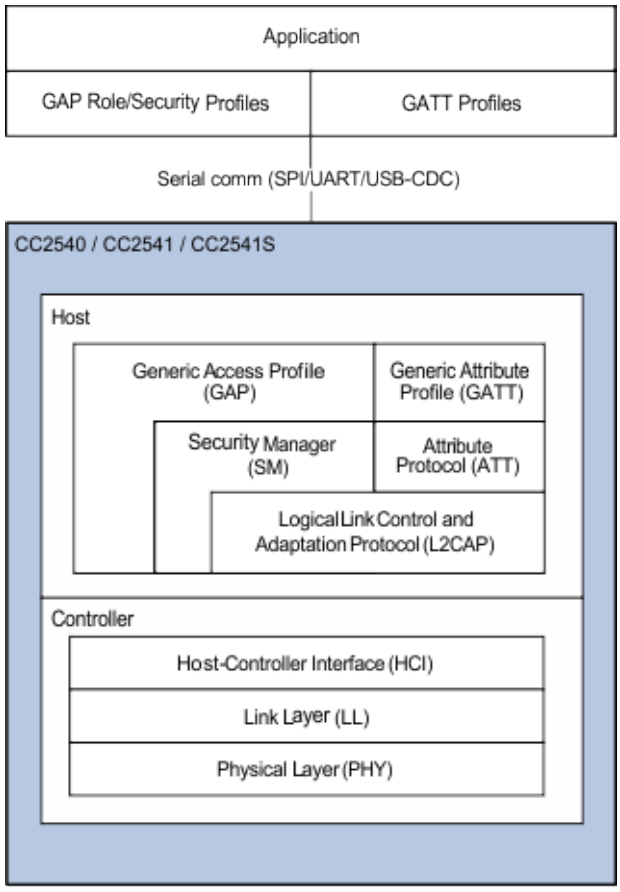


Figure 2-2. Network Processor Configuration

2.2 Projects

The SimpleBLEPeripheral project consists of sample code that demonstrates a simple application in the single-device configuration. You can use this project as a reference for developing a slave/peripheral application.

The SimpleBLECentral project is similar because it demonstrates a simple master/central application in the single-device configuration, and can be a reference for developing master/central applications.

The HostTestRelease project is used to build the BLE network processor software for the CC2540/41. This project contains configurations for both master and slave roles.

The BLE development kit includes other sample projects. These projects implement various profiles and demonstration applications. For more information on these other projects, see the TI CC2540 Bluetooth low energy Sample Applications Guide.

2.3 Software Overview

Software developed using the BLE software development kit consists of following five major components:

- OSAL
- HAL
- The BLE Protocol Stack
- Profiles
- Application

The kit provides the BLE protocol stack as object code and the OSAL and HAL components in source form.

The kit provides three GAP profiles:

- Peripheral role
- Central role
- Peripheral bond manager

The kit also provides several sample GATT profiles and applications.

Path and file references in this document assume that you have installed the BLE development kit software to the path: *C:\Texas Instruments\BLE-CC254X-1.4.1*.

NOTE: This guide references the SimpleBLEPeripheral project. The BLE projects in the development kit follow a similar structure.

The Operating System Abstraction Layer (OSAL)

The BLE protocol stack, profiles, and applications are built around the OSAL. The OSAL is a control loop that lets software set up how events should execute. Each layer of software functions as a task and requires a task identifier (ID), a task initialization routine, and an event processing routine. You can also define a message processing routine. These layers must adhere to a priority scheme with the LL as the highest priority because of its timing requirements. The following is the hierarchy from the SimpleBLEPeripheral project:

```
// The order in this table must be identical to the task initialization calls below in osalInitTask.
const pTaskEventHandlerFn tasksArr[] =
{
  LL_ProcessEvent,                                      // task 0
  Hal_ProcessEvent,                                     // task 1
  HCI_ProcessEvent,                                     // task 2
#if defined ( OSAL_CBTIMER_NUM_TASKS )
  OSAL_CBTIMER_PROCESS_EVENT( osal_CbTimerProcessEvent ),         // task 3
#endif
  L2CAP_ProcessEvent,                                   // task 4
  GAP_ProcessEvent,                                     // task 5
  SM_ProcessEvent,                                      // task 6
  GATT_ProcessEvent,                                    // task 7
  GAPRole_ProcessEvent,                                 // task 8
  GAPBondMgr_ProcessEvent,                              // task 9
  GATTServApp_ProcessEvent,                             // task 10
  SimpleBLEPeripheral_ProcessEvent                      // task 11
};
```

The OSAL also provides services such as message passing, heap management, and timers. OSAL code is provided in source form. For more information on the OSAL functions, see the OSAL API Guide.

3.1 Task Initialization

To use OSAL, locate a call to osal_start_system() at the end of the main() function. This function call is the OSAL routine that starts the system. This routine starts the system and calls the osalInitTasks() function defined by the application. In the SimpleBLEPeripheral project, you can find this function in OSAL_SimpleBLEPeripheral.c:

```
void osalInitTasks( void )
{
  uint8 taskID = 0;
  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  /* LL Task */
  LL_Init( taskID++ );

  /* Hal Task */
  Hal_Init( taskID++ );

  /* HCI Task */
  HCI_Init( taskID++ );

#if defined ( OSAL_CBTIMER_NUM_TASKS )
  /* Callback Timer Tasks */
  osal_CbTimerInit( taskID );
  taskID += OSAL_CBTIMER_NUM_TASKS;
#endif

  /* L2CAP Task */
  L2CAP_Init( taskID++ );

  /* GAP Task */
  GAP_Init( taskID++ );

  /* SM Task */
  SM_Init( taskID++ );

  /* GATT Task */
  GATT_Init( taskID++ );

  /* Profiles */ GAPRole_Init(
  taskID++ ); GAPBondMgr_Init(
  taskID++ );

  GATTServApp_Init( taskID++ );

  /* Application */
  SimpleBLEPeripheral_Init( taskID );
}
```

Each layer of software using OSAL must have an initialization routine called from the function osalInitTasks(). Within this function, the initialization routine for every layer of software is called within the osalInitTasks(). As each task initialization routine is called, an 8-bit task ID value is assigned to the task.

NOTE: When creating an application, add this 8-bit task ID value to the end of the list and ensure that the task ID is greater than the other task ID values. OSAL and HAL components are provided in source form.

The task ID determines the priority of the tasks. The task ID gives lower values higher priority. The protocol stack tasks must have the highest priority. The initialization function of the SimpleBLEPeripheral application, SimpleBLEPeripheral_Init(), has the highest task ID and the lowest priority.

3.2 Task Events and Event Processing

After the OSAL initializes, it runs in an infinite loop checking for task events. You can find this loop in the osal_start_system() function in the OSAL.c file. Task events are stored as unique bits in a 16-bit variable where each bit corresponds to a unique event. The application determines the definition and use of these event flags. Figure 3-1 shows a flow diagram of the OSAL processing scheme.

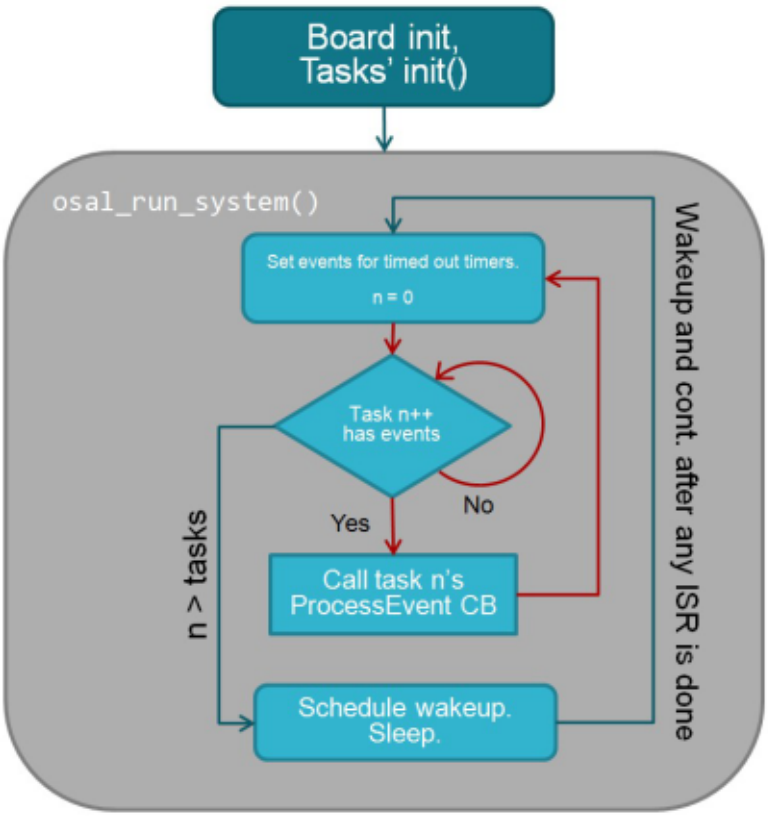


Figure 3-1. OSAL Task Loop

If the SimpleBLEPeripheral application defines a flag in simpleBLEPeripheral.h: SBP_START_DEVICE_EVT (0x0001) indicating the initial device start is complete, the application processing begins. The application cannot define one reserved flag value (0x8000). This value corresponds to the event SYS_EVENT_MSG for messaging between tasks (see Section 3.4 for more information).

When the OSAL detects an event set for a task, it calls the event processing routine of that task to process the event. The task layer must add its own event processing routine to the table formed by the array of function pointers called tasksArr (defined in OSAL_SimpleBLEPeripheral.c for the SimpleBLEPeripheral example project). The order of the event processing routines in tasksArr is the same as the order of task IDs in the osalInitTasks() function. Maintaining this task order is required for the correct software layer to process events.

In the SimpleBLEPeripheral application, the event processing function is called SimpleBLEPeripheral_ProcessEvent() to handle all active events associated with the task. After processing, the events must be cleared to prevent duplicate processing of the same event. The SimpleBLEPeripheral_ProcessEvent() application function shows that after the START_DEVICE_EVT event occurs it returns the 16-bit events variable with the SBP_START_DEVICE_EVT flag cleared.

```
  if ( events & SBP_START_DEVICE_EVT )
  {
…
    return ( events ^ SBP_START_DEVICE_EVT );
  }
```

Any layer of the software can set an OSAL event for any layer. Use the osal_set_event() function (prototype in OSAL.h) to immediately schedule a new OSAL event. With this function, you specify the task ID (of the task that will process the event) and the event flag as parameters.

An alternate method to set an OSAL event for any layer is to use the osal_start_timerEx() function (prototype in OSAL_Timers.h). This function operates similarly to the osal_set_event() function. You select the task ID of the task that will process the event and the event flag as parameters. The osal_start_timerEx() function has a third parameter that you must use to input a time-out value in milliseconds. This timeout parameter causes the OSAL to set a timer and set the specified event when the timer expires.

3.3 Heap Manager

Th eOSAL provides basic memory management functions. The osal_mem_alloc() function can allocate memory similarly to the standard C malloc function. The OSAL function takes a single parameter specifying the number of bytes to allocate and returns a void pointer if successful. If memory is unavailable, a NULL pointer is returned.

The osal_mem_free() function frees memory allocated using osal_mem_alloc() similarly to the standard C free function.

The INT_HEAP_LEN preprocessor symbol reserves memory for dynamic allocation.

To profile dynamic memory usage, do the following:

1. Set the preprocessor symbol as OSALMEM_METRICS=TRUE in the project options.
2. Exercise the system in stress conditions that replicate the worst-case expected system load. (This may involve having the maximum connected clients with maximum throughput while the application is operating at maximum capactity.)
3. If configured, enable Pairing/Bonding.
4. Perform the test with encryption enabled.
5. Review the value of the variable memMax in OSAL_Memory.c to see the maximum amount of memory allocated.
6. Use this value as a guideline for lowering INT_HEAP_LEN.

Because the BLE stack also uses the heap, you must test it with both components in the maximum-expected operating conditions.

3.4 OSAL Messages

OSAL provides a scheme for different subsystems of the software to communicate by sending or receiving messages. Messages can contain any type of data and can be any size (assuming enough memory is available).

To send an OSAL message, do the following:

1. Use the osal_msg_allocate() function to allocate the memory to store the messages by supplying a length parameter that specifies the length of the message.

NOTE: A pointer to a buffer containing the allocated space is returned (you do not need to use osal_mem_alloc() when using osal_msg_allocate()).

If no memory is available, a NULL pointer is returned.

2. Copy the data into the buffer.
3. Call osal_msg_send() specifying the destination task ID and pointer to the message to be sent.

The following code shows an example from OnBoard.c:

```
// Send the address to the task
msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
if ( msgPtr )
{
  msgPtr->hdr.event = KEY_CHANGE;
  msgPtr->state = state;
  msgPtr->keys = keys;

  osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
}
```

The OSAL sets the SYS_EVENT_MSG flag for the receiving task that a message indicating that an incoming message is available..This flag results in the event handler being invoked for the receiving task.. The receiving task retrieves the data by calling osal_msg_receive() and processes the message based on the data contents. TI recommends every OSAL task have a local message processing function (themessage processing function of the SimpleBLEPeripheral application is simpleBLEPeripheral_ProcessOSALMsg()). The processing function chooses what action to take based on the type of message received. When the receiving task processes the message, it must deallocate the memory using the function osal_msg_deallocate() (you do not need to use osal_mem_free() when using osal_msg_deallocate()). Examples of receiving OSAL messages will be depicted in the event processing functions of the various layers.

The Application and Profiles

The BLE software development kit contains a sample project, SimpleBLEPeripheral, that implements a basic BLE peripheral device. This project is built using the single-device stack configuration, with the stack, profiles, and application running on the CC2540/41.

4.1 Project Overview

On the left side of the IAR window, the Workspace section lists the files used by the project.

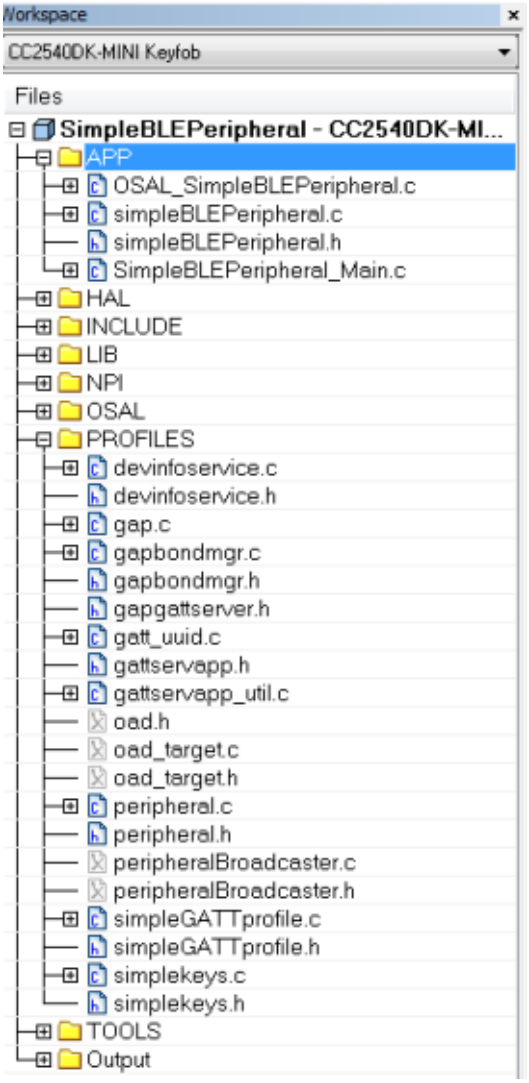


Figure 4-1. Project Files

The file list is divided into the following groups:

- APP – These are the application source code and header files. More information on these files can be found later in this section.
- HAL – This group contains the HAL source code and header files. See Chapter 6 for more information on the HAL group.
- INCLUDE – This group includes all of the necessary header files for the BLE protocol stack API (see appendices A through G and CC2540 Bluetooth low energy API Guide for details).
- LIB – This group contains the protocol stack library file CC2540_BLE_peri.lib. For more information on the protocol stack libraries, see Section 5.7.
- NPI – Network processor interface, a transport layer that allows you to route HCI data to a serial interface. CC254X_BLE_HCI_TL_Full.lib must be included for this capability (see the HostTest project in the SDK). If not used, the CC254X_BLE_HCI_TL_None.lib should be used (see SimpleBLEPeripheral in the SDK) when developing a single-chip application.
- OSAL – This group contains the OSAL source code and header files. See OSAL API Guide and Chapter 3 for more information on the OSAL.
- PROFILES – This group contains the source code and header files for the GAP role profile, GAP security profile, and the sample GATT profile. In addition, this section contains the necessary header files for the GATT server application. See Chapter 5 for more information on these modules.
- TOOLS – This group contains the configuration files buildComponents.cfg and buildConfig.cfg. Section 5.7 describes these files and contains the files OnBoard.c and OnBoard.h, which handle interface functions.
- OUTPUT – This group contains files that are generated by IAR during the build process, including binaries and the map file (see Section 8.2.4).

4.2 Start-up in main()

The main() function in SimpleBLEPeripheral_Main.c is the starting point at runtime. This function brings up the board and initializes the OSAL and SNV drivers. Next, this function initializes power management and creates the tasks. Finally, the function calls the osal_start_system(), which starts the processing loop (OSAL) and does not return.

```
int main(void)
{
  /* Initialize hardware */
  HAL_BOARD_INIT();

  // Initialize board I/O
  InitBoard( OB_COLD );

  /* Initialze the HAL driver */
  HalDriverInit();

  /* Initialize NV system */
  osal_snv_init();

  /* Initialize the operating system */
  osal_init_system();

  /* Enable interrupts */
  HAL_ENABLE_INTERRUPTS();

  // Final board initialization
  InitBoard( OB_READY );

  #if defined ( POWER_SAVING )
    osal_pwrmgr_device( PWRMGR_BATTERY );
  #endif

  /* Start OSAL */
  osal_start_system(); // No Return from here

  return 0;
}
```

4.3 Application Initialization

The initialization of the application occurs in two phases. OSAL calls the SimpleBLEPeripheral_Init() function. This function sets up the GAP role profile parameters, GAP characteristics, the GAP bond manager parameters, and simpleGATTprofile parameters. This function also sets an OSAL SBP_START_DEVICE_EVT event.

The processing in this event triggers the second phase of the initialization, which is in the SimpleBLEPeripheral_ProcessEvent() function. During this phase, the GAPRole_StartDevice() function is called to set up the GAP functions. This function sets up the GAP functions of the application. Connectable undirected advertisements make the device discoverable (for CC2540/41DK-MINI keyfob builds, the device becomes discoverable when you press the button on the right). A central device can discover the peripheral device by scanning. If a central device sends a request to connect to the peripheral device, the peripheral device accepts the request and goes into a connected state as a slave. If the peripheral device recieves no connection request, the device remains discoverable for 30.72 seconds before going into a standby state.

The project also includes the simpleGATTProfile service. A connected central device operating as a GATT client can perform characteristic reads and writes on simpleGATTProfile characteristic values. The device can also enable notifications of one of the characteristics.

4.4 Event Processing

After initialization, the application task processes events in SimpleBLEPeripheral_ProcessEvent when a bit is set in its events variable. Possible sources of events are described in the following subsections

4.4.1 Periodic Event

The application contains an OSAL event called SBP_PERIODIC_EVT. An OSAL timer sets SBP_PERIODIC_EVT to occur periodically. After the SBP_START_DEVICE_EVT processing has completed, the timer is set with a time-out value of PERIODIC_EVT_PERIOD (the default value is 5000 milliseconds). Every 5 seconds the periodic event occurs and the function performPeriodicTask() is called.

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id, uint16 events )
{
…
  if ( events & SBP_PERIODIC_EVT )
  {
    // Restart timer
    if ( SBP_PERIODIC_EVT_PERIOD )
    {
      osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
    }

    // Perform periodic application task
    performPeriodicTask();

    return (events ^ SBP_PERIODIC_EVT);
  }
```

The performPeriodicTask() function retrieves the value of the third characteristic in the simpleGATTProfile and copies that value into the fourth characteristic. This periodic event processing is an example for demonstration only but highlights how a custom operation can be performed in a periodic task. Before processing the periodic event, a new OSAL timer is started, which sets up the next periodic task.

4.4.2 OSAL Messages

OSAL messages can come from various layers of the BLE stack. For example, these messages can be from key presses sent by the HAL. The application has code specific to the Keyfob reference hardware in the CC2540/41DK-MINI development kit. This code is surrounded by the preprocessor directive #if defined (CC2540_MINIDK) and gets compiled when using the CC2540/41DK-MINI Keyfob configuration. This code adds the TI-proprietary simple keys service to the GATT server and handles key presses through the simple keys profile.

Each time you press or release a key on the keyfob, HAL sends an OSAL message to the application. As Section 3.4 describes, this action causes a SYS_EVENT_MSG event to occur. This event is handled in the application by the function simpleBLEperipheral_ProcessOSALMsg(). In the current SimpleBLEPeripheral application, the KEY_CHANGE message is the only recognized OSAL message type. You can define additional message types. The KEY_CHANGE message event processing calls the calls the simpleBLEPeripheral_HandleKeys() function, which checks the state of the keys.

4.5 Callbacks

Other than processing events, application code can also within the callback functions defined by the application such as simpleProfileChangeCB() and peripheralStateNotificationCB(). These callbacks process in the context of the task that called them. Processing should be limited in these callbacks. If any intensive processing must be done, send an event from the callback to the application so that processing can occur. For more information, see Section 4.4.

4.6 Complete Attribute Table

Section 5.4.4.1 describes the process for adding profiles and services to the application. Figure 4-2 shows the complete attributes of the SimpleBLEPeripheral and can be a reference when communicating wirelessly with the device. Services are red. Characteristic descriptors are yellow. General attributes are white. See Section 5.4 for further details. When working with the SimpleBLEPeripheral application, print Figure 4-2 as a reference.

ConHnd	Handle	Uuid	Uuid Description	Value	Properties
0x0000	0x0001	0x2800	GATT Primary Service Declaration	00:18	
0x0000	0x0002	0x2803	GATT Characteristic Declaration	02:03:00:00:2A	
0x0000	0x0003	0x2A00	Device Name	Simple BLE Peripheral	Rd 0x02
0x0000	0x0004	0x2803	GATT Characteristic Declaration	02:05:00:01:2A	
0x0000	0x0005	0x2A01	Appearance	00:00	Rd 0x02
0x0000	0x0006	0x2803	GATT Characteristic Declaration	0A:07:00:02:2A	
0x0000	0x0007	0x2A02	Peripheral Privacy Flag	00	Rd Wr 0x0A
0x0000	0x0008	0x2803	GATT Characteristic Declaration	08:09:00:03:2A	
0x0000	0x0009	0x2A03	Reconnection Address		Wr 0x08
0x0000	0x000A	0x2803	GATT Characteristic Declaration	02:0B:00:04:2A	
0x0000	0x000B	0x2A04	Peripheral Preferred Connection Parameters	50:00:A0:00:00:00:E8:03	Rd 0x02
0x0000	0x000C	0x2800	GATT Primary Service Declaration	01:18	
0x0000	0x000D	0x2803	GATT Characteristic Declaration	20:0E:00:05:2A	
0x0000	0x000E	0x2A05	Service Changed		Ind 0x20
0x0000	0x000F	0x2902	Client Characteristic Configuration	00:00	
0x0000	0x0010	0x2800	GATT Primary Service Declaration	0A:18	
0x0000	0x0011	0x2803	GATT Characteristic Declaration	02:12:00:23:2A	
0x0000	0x0012	0x2A23	System ID	2D:5B:6E:00:00:E5:C5:78	Rd 0x02
0x0000	0x0013	0x2803	GATT Characteristic Declaration	02:14:00:24:2A	
0x0000	0x0014	0x2A24	Model Number String	Model Number	Rd 0x02
0x0000	0x0015	0x2803	GATT Characteristic Declaration	02:16:00:25:2A	
0x0000	0x0016	0x2A25	Serial Number String	Serial Number	Rd 0x02
0x0000	0x0017	0x2803	GATT Characteristic Declaration	02:18:00:26:2A	
0x0000	0x0018	0x2A26	Firmware Revision String	Firmware Revision	Rd 0x02
0x0000	0x0019	0x2803	GATT Characteristic Declaration	02:1A:00:27:2A	
0x0000	0x001A	0x2A27	Hardware Revision String	Hardware Revision	Rd 0x02
0x0000	0x001B	0x2803	GATT Characteristic Declaration	02:1C:00:28:2A	
0x0000	0x001C	0x2A28	Software Revision String	Software Revision	Rd 0x02
0x0000	0x001D	0x2803	GATT Characteristic Declaration	02:1E:00:29:2A	
0x0000	0x001E	0x2A29	Manufacturer Name String	Manufacturer Name	Rd 0x02
0x0000	0x001F	0x2803	GATT Characteristic Declaration	02:20:00:2A:2A	
0x0000	0x0020	0x2A2A	IEEE 11073-20601 Regulatory Certification ...	FE:00:65:78:70:65:72:69:6D:...	Rd 0x02
0x0000	0x0021	0x2803	GATT Characteristic Declaration	02:22:00:50:2A	
0x0000	0x0022	0x2A50	PnP ID	01:0D:00:00:00:10:01	Rd 0x02
0x0000	0x0023	0x2800	GATT Primary Service Declaration	F0:FF	
0x0000	0x0024	0x2803	GATT Characteristic Declaration	0A:25:00:F1:FF	
0x0000	0x0025	0xFFF1	Simple Profile Char 1	01	Rd Wr 0x0A
0x0000	0x0026	0x2901	Characteristic User Description	Characteristic 1	
0x0000	0x0027	0x2803	GATT Characteristic Declaration	02:28:00:F2:FF	
0x0000	0x0028	0xFFF2	Simple Profile Char 2	02	Rd 0x02
0x0000	0x0029	0x2901	Characteristic User Description	Characteristic 2	
0x0000	0x002A	0x2803	GATT Characteristic Declaration	08:2B:00:F3:FF	
0x0000	0x002B	0xFFF3	Simple Profile Char 3		Wr 0x08
0x0000	0x002C	0x2901	Characteristic User Description	Characteristic 3	
0x0000	0x002D	0x2803	GATT Characteristic Declaration	10:2E:00:F4:FF	
0x0000	0x002E	0xFFF4	Simple Profile Char 4		Nfy 0x10
0x0000	0x002F	0x2902	Client Characteristic Configuration	00:00	
0x0000	0x0030	0x2901	Characteristic User Description	Characteristic 4	
0x0000	0x0031	0x2803	GATT Characteristic Declaration	02:32:00:F5:FF	
0x0000	0x0032	0xFFF5	Simple Profile Char 5		Rd 0x02
0x0000	0x0033	0x2901	Characteristic User Description	Characteristic 5	

Figure 4-2. SimpleBLEPeripheral Complete Attribute Table

4.7 Additional Sample Projects

The BLE development kit includes several sample projects implementing profiles such as the following:

- A heart rate monitor
- A health thermometer
- A proximity keyfob

See Chapter 5 for more information on these projects.

The BLE Protocol Stack

The BLE protocol stack is object code in the library files. TI does not provide the protocol stack source code. TI intends the functionality of these layers to be understood as they interact directly with the application and profiles.

5.1 Generic Access Profile (GAP)

5.1.1 Overview

The GAP layer of the BLE protocol stack defines the behavior of devices performing the following actions:

- Device discovery
- Link establishment
- Link termination
- Initiation of security features
- Device configuration

See Figure 5-1 for an overview of possible device states.

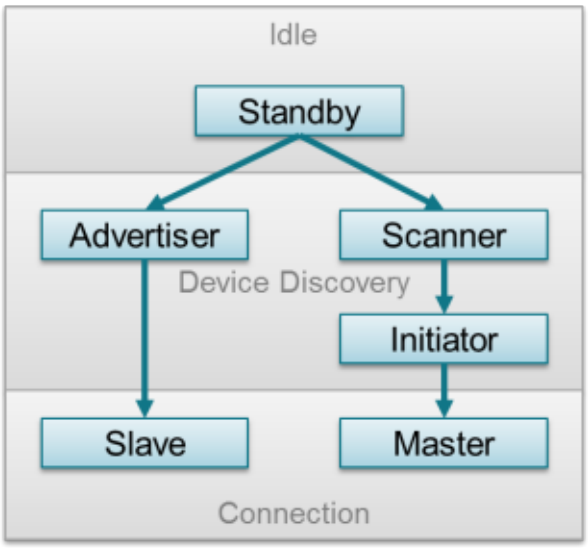


Figure 5-1. GAP State Diagram

The following describes the possible device states:

- Standby: the initial idle state following reset or when the BLE stack is not active
- Advertiser: The device advertises with specific data that signals it can connect to any initiating devices. This advertisement contains the device address and additional data such as the device name.
- Scanner: When receiving the advertisement, the scanning device sends a request to scan the advertiser. The advertiser responds with a scan response. This process outlines how the device discovers other devices. The scanning device reads the advertising device and determines whether or not they can connect.
- Initiator: When initiating, the initiator must specify a peer device address with which to connect. If the initiator receives an advertisement that matches the address of the peer device, the initiator will request to establish a connection with the advertising device. The initiator specifies the initial connection parameters when the connection is formed.
- Master or Slave: If the device was the advertiser , it becomes a slave after connecting. If the device was the initiator, it becomes a master after connecting.

5.1.1.1 Connection Parameters

This section describes the connection parameters sent by the initiating device with the connection request. These parameters can be modified by either device when the connection is established.

These parameters are the following:

- Connection Interval – BLE connections use a frequency-hopping scheme. The devices send and receive data on a specific channel at a specific time and meet at a new channel later. The link layer of the BLE protocol stack handles the channel switching. This meeting, where the two devices send and receive data, is a connection event. If there is no application data sent or received, the devices exchange link layer data to maintain the connection. The connection interval is the time between two connection events in units of 1.25 ms. The connection interval can range from a minimum value of 6 (7.5 ms) to a maximum of 3200 (4.0 seconds).

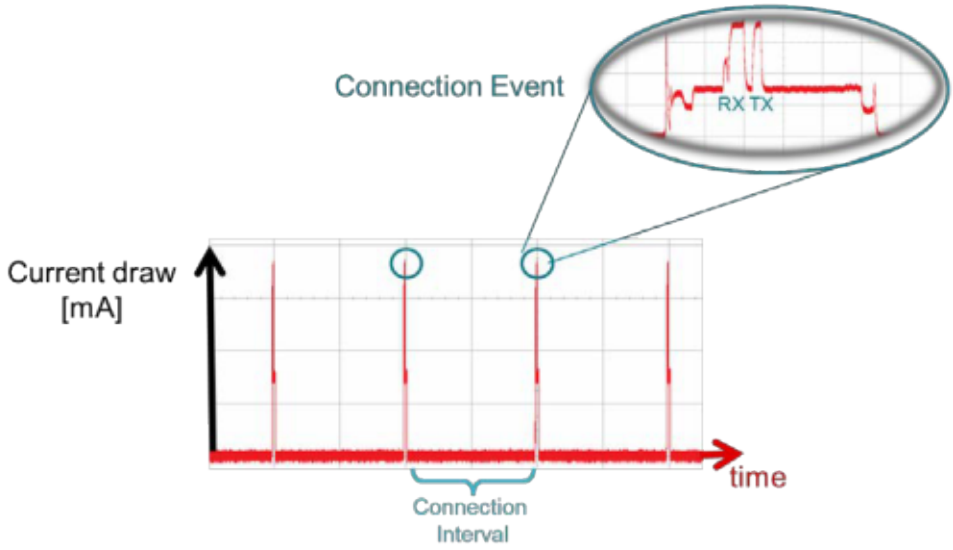


Figure 5-2. Connection Event and Interval

Applications may require different connection intervals. This difference affects the power consumption of the device. See the *Measuring Power Consumption Application Note* (SWRA347) for more detailed information on power consumption.

- Slave Latency – This parameter lets the slave (peripheral) device skip several connection events. If the device has no data to send, it can skip connection events and deactivate its radio during the connection event, which saves power. The slave latency value represents the maximum number of events that can be skipped. This value ranges from a minimum value of 0 (no connection events)) to a maximum of 499. The maximum value must create an effective connection interval less than 16 seconds. See Figure 5-3 for an overview of this parameter.

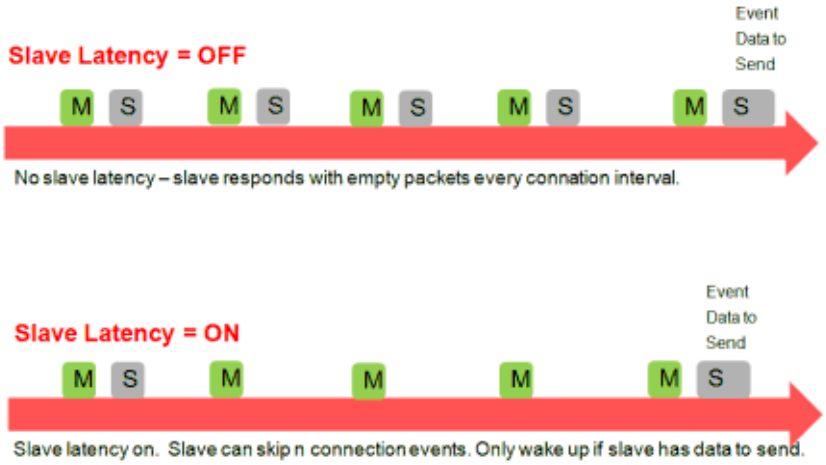


Figure 5-3. Slave Latency

- Supervision Time-out – This parameter is the maximum time period between two successful connection events. If this time period passes without a successful connection event, the device considers the connection lost and returns to an unconnected state. This parameter value is represented in units of 10 ms. The supervision time-out value can range from a minimum of 10 (100 ms) to 3200 (32 seconds). The time-out must be larger than the effective connection interval. For more details, see Section 5.1.1.2.

5.1.1.2 Effective Connection Interval

The effective connection interval is equal to the amount of time between two connection events, assuming 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.

Calculate this interval using the following formula:

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

Where:

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

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

5.1.1.3 Connection Parameter Considerations

In many applications, the slave skips the maximum number of connection events. Consider the effective connection interval when selecting or requesting connection parameters. Selecting the correct group of connection parameters helps optimize the power of the BLE device. The following list is a summary of the trade-offs in connection-parameter settings:

Reducing the connection interval will do the following:

- Increase the power consumed by both devices
- Increase the throughput to and from both devices
- Reduce the amount of time required to send data to and from both devices

Increasing the connection interval will do the following:

- Reduce the power consumed by both devices
- Reduce the throughput to and from both devices
- Increase the amount of time required to send data to and from both devices

Reducing the slave latency or setting it to zero will do the following:

- Increase the power consumed by the peripheral device
- Reduce the amount of time required to send data from the central device to the peripheral device

Increasing the slave latency will do the following:

- Reduce the power consumed by the peripheral device when it has no data to send to the central device
- Increase the amount of time required to send data from the central device to the peripheral device

5.1.1.4 Connection Parameter Update

Sometimes the central device will request a connection with a peripheral device containing connection parameters unfavorable to the peripheral device. Other times, a peripheral device might change parameters based on the peripheral application during a connection. The peripheral device can send a Connection Parameter Update Request to the central device to change the connection settings. For *Bluetooth* 4.0 devices, the L2CAP layer of the protocol stack handles this request.

This request contains the following four parameters:

- A minimum connection interval
- A maximum connection interval
- A slave latency
- A timeout

These values represent the parameters the peripheral device requires for the connection (the connection interval is given as a range). When the central device receives this request, it can accept or reject the parameters.

5.1.1.5 Connection Termination

The master or slave can terminate a connection for any reason. When either device initiates termination, the other must respond by acknowledging the termination indication before both devices disconnect.

5.1.2 GAP Abstraction

The application and profiles can call GAP API functions to perform BLE-related functions such as advertising or connecting. Most of the GAP functionality is handled by the GAPRole Task. For more information on this abstraction hierarchy, see Figure 5-4 .

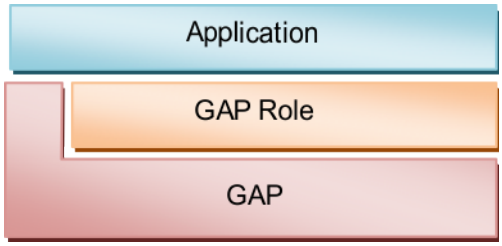


Figure 5-4. GAP Abstraction

Configure the GAPRole module and use its APIs to interface with the GAP layer. Section 5.1.3 describes the functions and parameters not handled or configured through the GAPRole task. These functions and parameters must be modified directly through the GAP layer.

5.1.3 Configuring the GAP Layer

The GAP layer functionality is defined mostly in library code. You can find the function headers in gap.h. Most of these functions are used by the GAPRole task and do not need to be called directly. Appendix A defines the GAP API. You may want to modify several parameters before starting the GAPRole task. These parameters can be set or retrieved through the GAP_SetParamValue() and GAP_GetParamValue() functions. These parameters include advertising and scanning intervals, windows, and so forth (see B and C [GAPRole xxx API). A configuration of the GAP layer in SimpleBLEPeripheral_init() folows.

```
// Set advertising interval
{
  uint16 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 in Section 3.1, the GAPRole task is a separate task (GAPRole_ProcessEvent) that simplifies the application by handling most of the GAP layer functionality. This task is enabled and configured by the application during initialization. Based on this configuration, many BLE protocol stack events are handled directly by the GAPRole task and never passed to the application. Callbacks exist that the application can register with the GAPRole task. This registration notifies the GAPRole task of certain events. See Section B.3.1 for peripheral events or Section C.3.2 for central events.

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

- Broadcaster – an advertiser that is nonconnectable
- 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 and operates as a master in a single or multiple link-layer connections (The BLE central protocol stack supports up to three simultaneous connections.)

The BLE specification supports certain combinations of roles supported by the BLE protocol stack. See the for sample projects of combination roles. The CC254x does not support simultaneous peripheral and central device roles. This functionality is supported by the CC2640. The peripheral and central roles are described in the following sections.

5.2.1 Peripheral Role

The peripheral GAPRole task is defined in peripheral.c and peripheral.h. See Appendix B for descriptions of the full API including commands, configurable parameters, events, and callbacks.

The general steps to use this module are the following:

1. Initialize the GAPRole parameters (see Section B.2). Do this initialization in the application initialization function (that is, SimpleBLEPeripheral_init()).

```
{
    // For all hardware platforms, device starts advertising upon initialization
    uint8 initialAdvertEnable = TRUE;

    uint16 advertOffTime =0;

    uint8 enableUpdateRequest = DEFAULT_ENABLE_UPDATE_REQUEST;
    uint16 desiredMinInterval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
    uint16 desiredMaxInterval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
    uint16 desiredSlaveLatency = DEFAULT_DESIRED_SLAVE_LATENCY;
    uint16 desiredConnTimeout = DEFAULT_DESIRED_CONN_TIMEOUT;

    // Set the GAP Role Parameters
    GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8),
                         &initialAdvertEnable);
    GAPRole_SetParameter(GAPROLE_ADVERT_OFF_TIME, sizeof(uint16),
                         &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),
                         &enableUpdateRequest);
    GAPRole_SetParameter(GAPROLE_MIN_CONN_INTERVAL, sizeof(uint16),
                         &desiredMinInterval);
    GAPRole_SetParameter(GAPROLE_MAX_CONN_INTERVAL, sizeof(uint16),
                         &desiredMaxInterval);
    GAPRole_SetParameter(GAPROLE_SLAVE_LATENCY, sizeof(uint16),
                         &desiredSlaveLatency);
    GAPRole_SetParameter(GAPROLE_TIMEOUT_MULTIPLIER, sizeof(uint16),
                         &desiredConnTimeout);
}
```

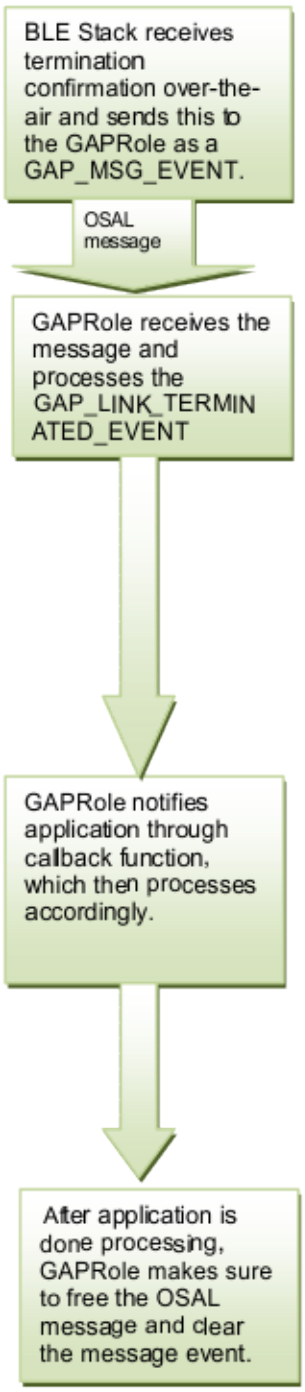
2. Initialize the GAPRole task. Do this initialization when processing START_DEVICE_EVT. This initialization involves passing function pointers to application callback functions. Section B.3 defines these callbacks.

```
if ( events & START_DEVICE_EVT )
  {
    // Start the Device
    VOID GAPRole_StartDevice( &simpleBLEPeripheral_PeripheralCBs );
…
```

3. Send GAPRole commands from the application. The following is an example of the application using GAPRole_TerminateConnection().

NOTE: The return value from the BLE protocol stack only indicates whether the attempt to terminate the connection was initiated successfully. The termination of connection event sent to the application asynchronously and is described in the following example. The API in Section B.3 lists the return parameters for each command and associated callback function events.

4. The GAPRole task processes most of the GAP-related events passed to it from the BLE protocol stack. The task forwards some events to the application. The following is an example tracing the GAP_LINK_TERMINATED_EVENT from the BLE protocol stack to the application.



Library Code

peripheral.c:

```
static void gapRole_ProcessOSALMsg( osal_event_hdr_t *pMsg )
{
…
    case GAP_MSG_EVENT:
      gapRole_ProcessGAPMsg( (gapEventHdr_t *)pMsg );
      break;
```

```
static void gapRole_ProcessGAPMsg( gapEventHdr_t *pMsg )
{
…
    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_processStateChangeEvt (gaprole_State s_t
newState)
{
  switch ( newState )
  {
    case GAP_LINK_TERMINATED_EVENT:
...
```

peripheral.c:

```
…
      // Release the OSAL message
      VOID osal_msg_deallocate( pMsg );
    }

    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }
```

5.2.2 Central Role

The central GAPRole task is defined in central.c and central.h. For the full API including commands, configurable parameters, events, and callbacks, see Appendix C.

To use this module, do the following:

1. Initialize the GAPRole parameters. Section B.2 defines these parameters. Define the parameters in the application initialization function (that is, SimpleBLECentral_init()).

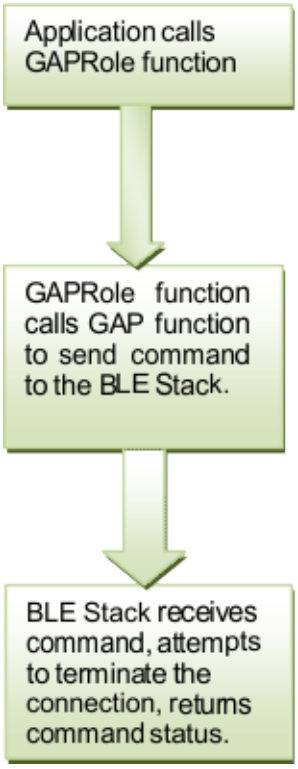
```
  // Setup GAP
uint8 scanRes = DEFAULT_MAX_SCAN_RES;
```

```
    GAPCentralRole_SetParameter ( GAPCENTRALROLE_MAX_SCAN_RES, sizeof( uint8 ), &scanRes );
```

2. Initialize the GAPRole task. Do this initialization when processing START DEVICE EVT. This initialization involves passing function pointers to application callback functions. Section C.3 defines these callbacks.

```
  if ( events & START_DEVICE_EVT )
  {
    // Start the Device
    VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *) &simpleBLERoleCB );
…
```

3. Send GAPRole commands from the application. The following is an example of the application using GAPCentralRole_StartDiscovery().



simpleBLEcentral.c

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

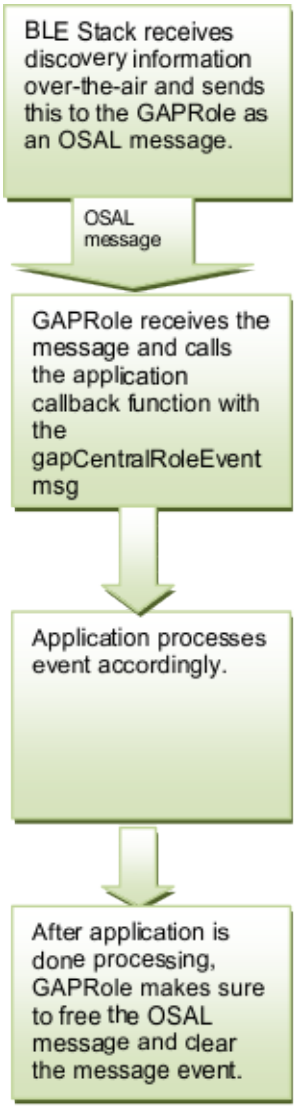
central.c

```
bStatus_t GAPCentralRole_StartDiscovery(uint8 mode, uint8 activeScan,
uint8 whiteList)
{
  gapDevDiscReq_t params;        .task

  params     ID = gapCentralRoleTaskId;
  params.mode = mode;
  params.activeScan = activeScan;
  params.whiteList = whiteList;

  return GAP_DeviceDiscoveryRequest(&params);
}
```

Library Code

NOTE: The return value from the BLE protocol stack only indicates whether the attempt to perform device discovery was initiated or not. The termination of connection event is returned asynchronously and is described in the following step. See appendices A through G for the list of return parameter associated with each API.

4. The GAPRole task processes some of the GAP-related events passed to it from the BLE protocol stack. The task forwards some events to the application. The following is an example tracing the GAP_DEVICE_DISCOVERY_EVENT from the BLE protocol stack to the application.



Library Code

central.c:

```
static void gapCentralRole_ProcessOSALMsg ( osal_event_hdr_t *pMsg )
{
…
  // Pass event to app
  if ( pGapCentralRoleCB && pGapCentralRoleCB->eventCB )
  {
    pGapCentralRoleCB->eventCB( (gapCentralRoleEvent_t *) pMsg );
  }
```

simpleBLECentral.c:

```
static void simpleBLECentralEventCB( gapCentralRoleEvent_t *pEvent )
{
…
    case GAP_DEVICE_DISCOVERY_EVENT:
      {
…
```

central.c:

```
…
      // Release the OSAL message
      VOID osal_msg_deallocate( pMsg );
    }

    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }
```

5.3 Gap Bond Manager (GAPBondMgr)

The GAPBondMgr profile handles the initiation of security features during a BLE connection. Some data may be readable or writeable only in an authenticated connection. Table 5-1 defines the terminology used in BLE security.

Table 5-1. GAP Bond Manager Security Terms

Term	Description
Pairing	The process of exchanging keys
Encryption	Data is encrypted after pairing, or re-encryption (a subsequent connection where keys are looked up from nonvolatile memory)
Authentication	The pairing process completed with MITM (Man in the Middle) protection (passcode, NFC, and so forth).
Bonding	Storing the encryption keys in nonvolatile memory to use for the next encryption sequence.
Authorization	An additional application level key exchange in addition to authentication
OOB	Out of Band. Keys are not exchanged wirelessly, 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 wirelessly to break the encryption.
Just Works	Pairing method where keys are transferred wirelessly without MITM.

The general process to establish security is:

1. Pair the keys (exchanging keys through the following methods).
   (a) Just Works (to send the keys wirelessly)
   (b) MITM (to use a passcode to create a key)
2. Encrypt the link with keys from step 1.
3. Bond the keys (store keys in secure flash [SNV]).
4. When reconnected, use the keys stored in SNV to encrypt the link.

NOTE: You can skip steps. For example, you can to skip bonding and just re-pair after reconnecting. The GAPBondMgr uses the SNV flash area to store bond information. For more information on SNV, see Section 6.9

5.3.1 Overview of BLE Security

This section describes BLE security methods. For more information, see Device Information Service (Bluetooth Specification), Version 1.0 (24-May-2011).

When connected, the devices can go through a process called pairing. When paired, keys are established that encrypt and can authenticate the link. Either device may require a passkey to complete the pairing process. This process is called man in the middle (MITM) protection. You could create this passcode with a value such as 000000. Alternatively, the passcode can be a predetermined randomly-generated value displayed on the device. After the correct passkey is displayed and entered, the devices exchange security keys to encrypt and authenticate the link. The input and output capabilities of the devices in the pairing request must match to make authentication is possible.

In many cases, the same central and peripheral devices often connect and disconnect from each other. BLE has a security feature that lets the devices exchange a long-term set of security keys when pairing. With this long-term set of security keys, re-pairing is unnecessary when reconnecting in the future. This feature is called bonding and it lets the devices store the security keys and quickly reestablish encryption and authentication after reconnecting without going through the pairing process.

5.3.2 Using the GapBondMgr Profile

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

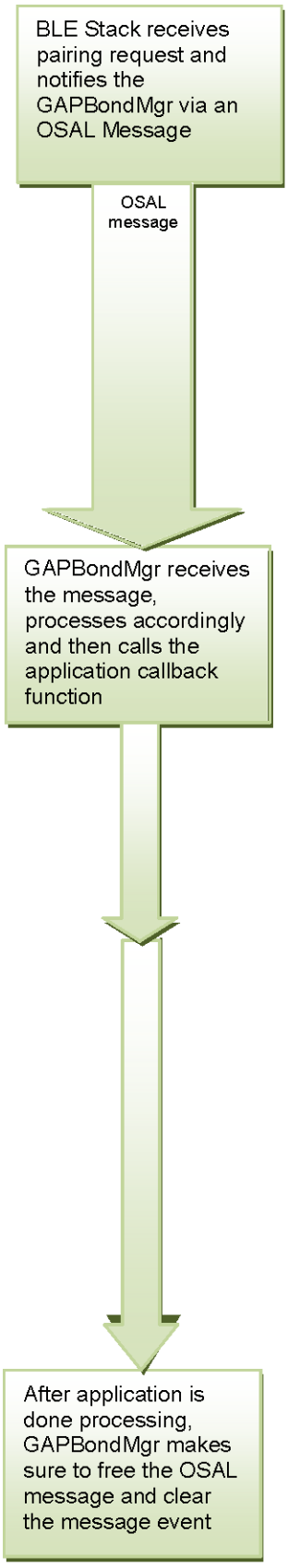
1. Initialize the GAPBondMgr parameters. Do this in the application initialization function (that is, SimpleBLECentral_init()). Consider the following parameters. For the example, the pairMode has been changed to initiate pairing.

```
// Setup the GAP Bond Manager
{
  uint32 passkey = DEFAULT_PASSCODE;
  uint8 pairMode = GAPBOND_PAIRING_MODE_INITIATE;
  uint8 mitm = DEFAULT_MITM_MODE;
  uint8 ioCap = DEFAULT_IO_CAPABILITIES;
  uint8 bonding = DEFAULT_BONDING_MODE;

  GAPBondMgr_SetParameter(GAPBOND_DEFAULT_PASSCODE, sizeof(uint32),
                          &passkey);
  GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8), &pairMode);
  GAPBondMgr_SetParameter(GAPBOND_MITM_PROTECTION, sizeof(uint8), &mitm);
  GAPBondMgr_SetParameter(GAPBOND_IO_CAPABILITIES, sizeof(uint8), &ioCap);
  GAPBondMgr_SetParameter(GAPBOND_BONDING_ENABLED, sizeof(uint8), &bonding);
}
```

2. Register application callbacks with the GAPBondMgr. Do this registration after the GAPRole starts in the START_DEVICE_EVT processing:

```
if ( events & START_DEVICE_EVT )
{
  // Start the Device
  VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *) &simpleBLERoleCB );

  // Register with bond manager after starting device
  GAPBondMgr_Register( (gapBondCBs_t *) &simpleBLEBondCB );
…
```

The GAPBondMgr is configured and operates autonomously. When a connection is established, the GAPBondMgr initiates pairing and bonding depending on the configuration parameters from Step 1. You can set a few parameters asynchronously such as GAPBOND_ERASE_ALLBONDS. All communication between the GAPBondMgr and the application occurs through the callbacks that were registered in Step 2. The following is a flow diagram example from SimpleBLECentral of the GAPBondMgr, notifying the application that pairing has started. The following sections expand on these callbacks.



Library Code

gapbondmgr.c:

```
uint16 GAPBondMgr_ProcessEvent( uint8 task_id, uint16 events )
{
  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;

    if ( (pMsg = osal_msg_receive( gapBondMgr_TaskID )) != NULL )
    {
      if ( gapBondMgr_ProcessOSALMsg( (osal_event_hdr_t *)pMsg ) )
      {
```

```
static uint8 gapBondMgr_ProcessOSALMsg( osal_event_hdr_t *pMsg )
{ switch ( pMsg->event )

  {
    case GAP_MSG_EVENT:
      safeToDealloc = GAPBondMgr_ProcessGAPMsg( (gapEventHdr_t *)pMsg
);
      break;
```

```
uint8 GAPBondMgr_ProcessGAPMsg( gapEventHdr_t *pMsg )
{
…
    case GAP_PAIRING_REQ_EVENT:
      {
      …
          // Call app state callback
          if ( pGapBondCB && pGapBondCB->pairStateCB )
          {
            pGapBondCB->pairStateCB(          pPkt->connectionHandle,
GAPBOND_PAIRING_STATE_STARTED, SUCCESS );
```

```
        }
```

SimpleBLECentral.c

```
static  void   SimpleBLECentral_processPairState(uint8   state    uint8
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);
    }
  }
...
```

```
        // Release the OSAL message
        VOID osal_msg_deallocate( pMsg );
      }
    }

    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }
```

5.3.3 GAPBondMgr Examples for Various Security Modes

This section provides message diagrams for the types of security to implement. These security types assume acceptable input and output capabilities are present for the security mode. See the Specification of the Bluetooth System, Covered Core Package version: 4.0 (30-June-2010) on how input and output capabilities affect pairing.

5.3.3.1 Pairing Disabled

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

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

5.3.3.2 Just Works Pairing Without Bonding

Just Works pairing encrypts without MITM authentication and is vulnerable to MITM attacks. For Just Works pairing without bonding, configure the GAPBondMgr as follows:

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

For an overview of this process for peripheral device, see Figure 5-5.

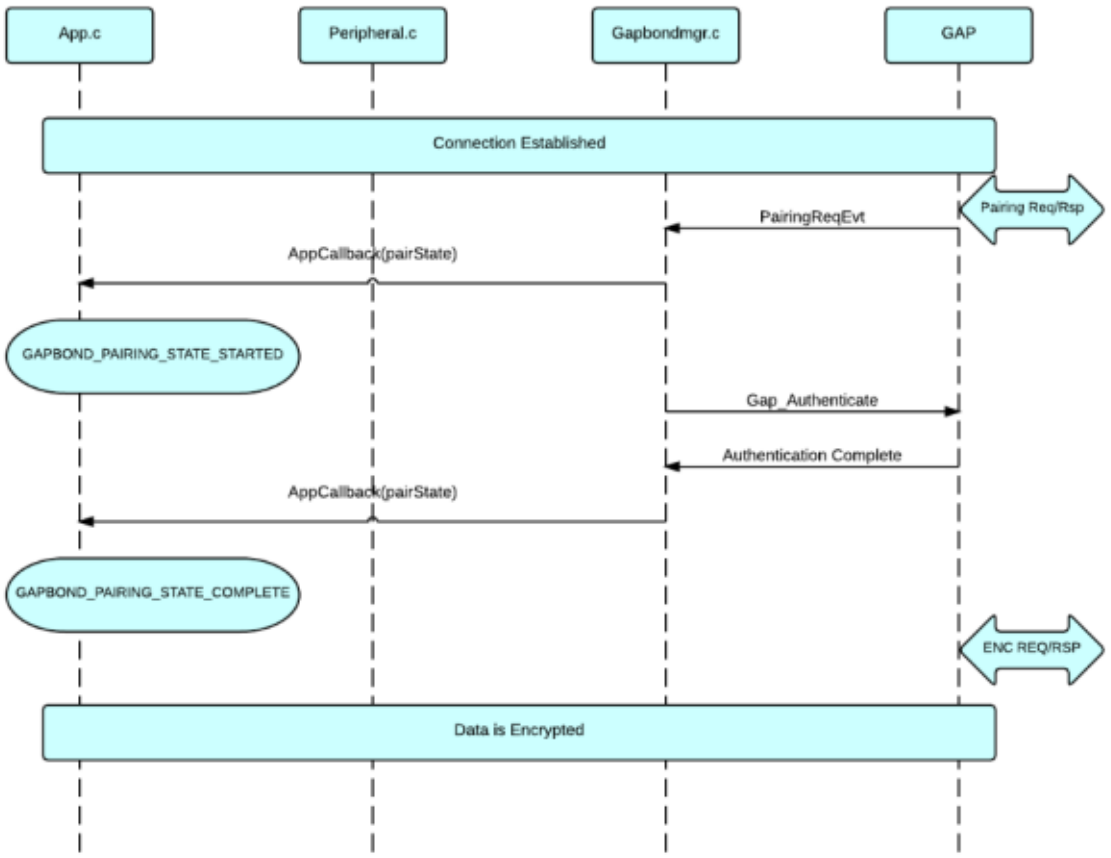


Figure 5-5. Just Works Pairing

The GAPBondMgr pairing states are passed to the application callback when required during the pairing process. GAPBOND_PAIRING_STATE_STARTED is passed when sent or received by the stack. GAPBOND_PAIRING_STATE_COMPLETE is sent when the pairing completes. A Just Works pairing requires the pair-state callback. For more information, see Section F.3.

5.3.3.3 Just Works Pairing With Bonding Enabled

To enable bonding with a Just Works pairing, use the following settings:

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

For an overview of this process for peripheral device, see Figure 5-6.

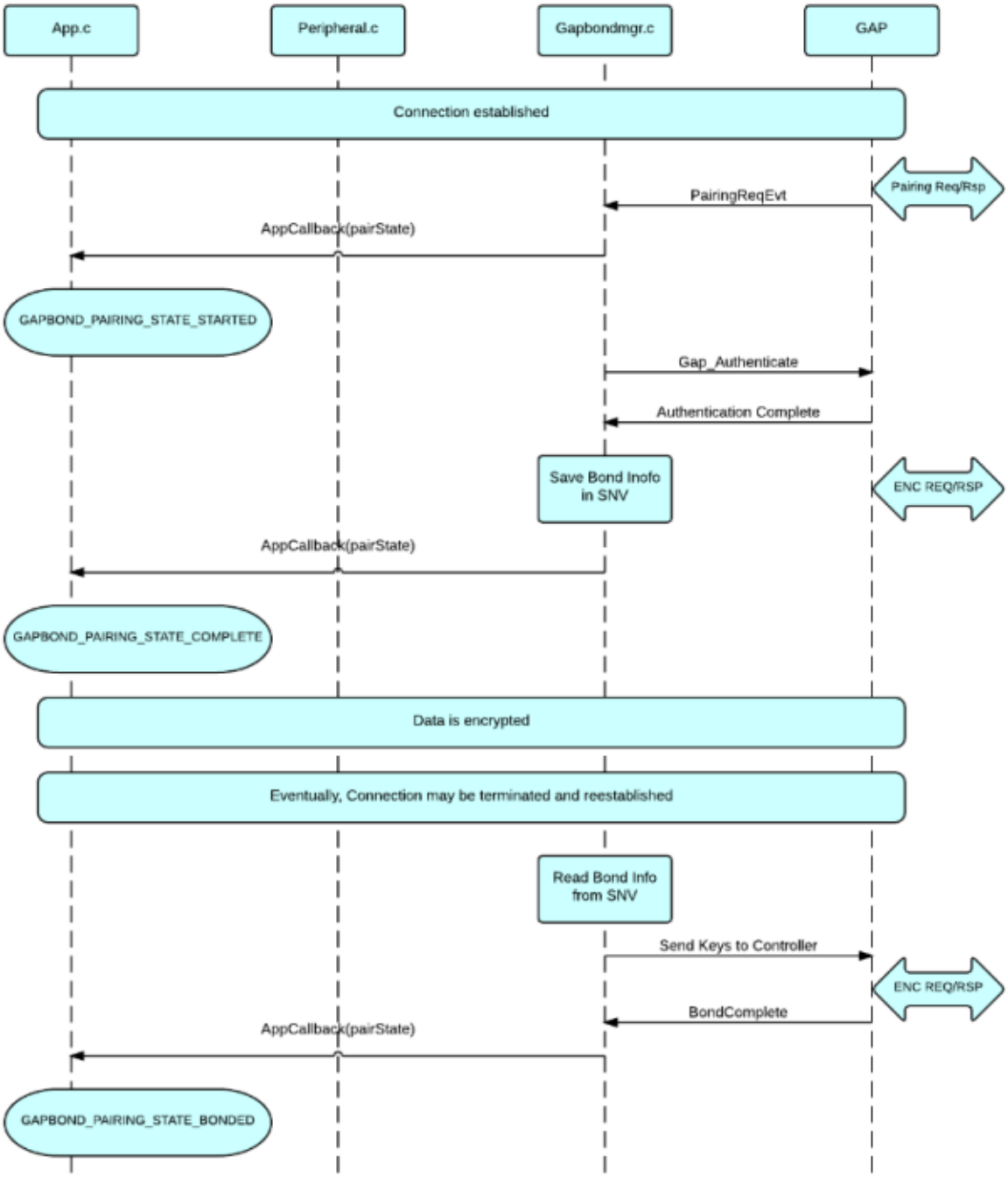


Figure 5-6. Bonding After Just Works Pairing

NOTE: GAPBOND_PAIRING_STATE_COMPLETE is only passed to the application pair state callback after the initial connection, pairing, and bond. For future connections, the security keys loads from flash. This capability skips the pairing process. In this case, only PAIRING_STATE_BONDED is passed to the application pair state callback.

5.3.3.4 Authenticated Pairing

Authenticated pairing requires MITM protection. This method is a way of transferring a passcode between the devices. The passcode cannot transmit wirelessly and is displayed on one device (typically on an LCD screen or a serial number on the device) and entered on the other device.

To pair with MITM authentication, use the following settings:

```
uint8 mitm = TRUE;
```

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

This method requires an additional step in the security process in Figure 5-7. After pairing is started, the GAPBondMgr notifies the application that a passcode is required through a passcode callback. Depending on the input and output capabilities of the device, the device must display and/or enter the passcode. If entering a passcode, the application sends this passcode to the GAPBondMgr.

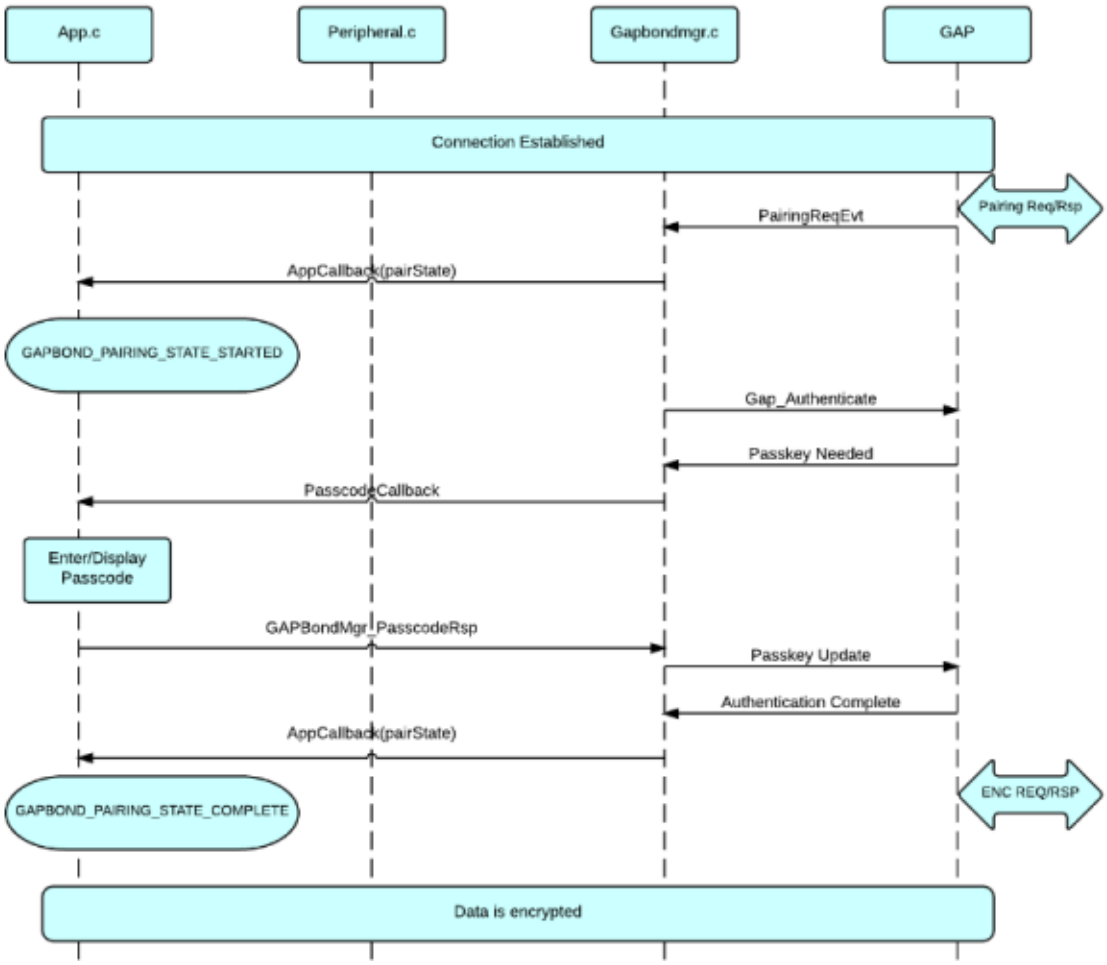


Figure 5-7. Pairing With MITM Authentication

This passcode communication with the GAPBondMgr uses a passcode callback function when registering with GAPBondMgr. You must add a passcode function to the GAPBondMgr application callbacks. The following is an example of a passcode function.

```
static const gapBondCBs_t simpleBLEBondCB =
{
  simpleBLECentralPasscodeCB,
  simpleBLECentralPairStateCB
};
```

When the GAPBondMgr requires a passcode, the GAPBondMgr use the following callback to request a passcode from the application. Depending on the input and output capabilities of the devices, the callback function should either display a passcode or read in an entered passcode. This passcode must be sent by the application to the GAPBondMgr using the GAPBondMgr_PasscodeRsp() function. The following is an example of the SimpleBLECentral.

```
static void simpleBLECentralPasscodeCB( uint8 *deviceAddr, uint16 connectionHandle,
                                        uint8 uiInputs, uint8 uiOutputs )
{
#if (HAL_LCD == TRUE)

  uint32  passcode;
  uint8   str[7];

  // Create random passcode
  LL_Rand( ((uint8 *) &passcode), sizeof( uint32 ));
  passcode %= 1000000;

  // Display passcode to user
  if ( uiOutputs != 0 )
  {
    LCD_WRITE_STRING( "Passcode:",  HAL_LCD_LINE_1 );
    LCD_WRITE_STRING( (char *) _ltoa(passcode, str, 10),  HAL_LCD_LINE_2 );
  }

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

In the previous example, a random password is generated and displayed on an LCD screen by the passcode callback function. The other connected device must then enter this passcode.

5.3.3.5 Authenticated Pairing with Bonding Enabled

After pairing and encrypting with MITM authentication, bonding occurs similarly as described in Section 5.3.3.3.

5.4 Generic Attribute Profile (GATT)

TI designed the GATT layer of the BLE protocol stack for use by the application for data communication between two connected devices. Data are passed and stored in the form of characteristics, which are stored in memory on the BLE device. In GATT when two devices are connected, they each fill one of two roles:

- GATT Server — This device contains the characteristic database being read or written by a GATT client.
- GATT Client — This device reads or writes data from or to the GATT server. The Figure 5-8 shows this relationship in a sample BLE connection where the peripheral device (a SensorTag) is the GATT server and the central device (a smart phone) is the GATT client.

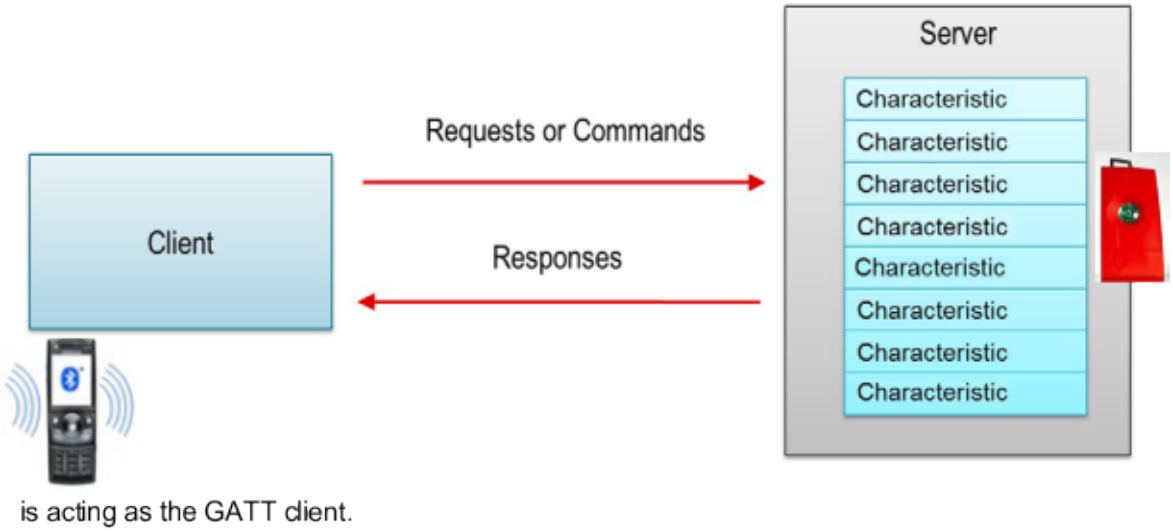


Figure 5-8. GATT Client and Server

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

5.4.1 GATT Characteristics and Attributes

While characteristics are sometimes interchangeable when referring to BLE, consider them as groups of information called attributes. Attributes are the base groups of information transferred between devices. Characteristics organize and use attributes as data values, properties, and configuration information.

A typical characteristic is composed of the following attributes:

- Characteristic Value: This value is the data value of the characteristic.
- Characteristic Declaration: A descriptor stores the properties, location, and type of the characteristic value.
- Client Characteristic Configuration: This configuration lets the GATT server configure the characteristic to be sent to the GATT server (notified) or sent to the GATT server and expect an acknowledgment (indicated).
- Characteristic User Description: This description is an ASCII string describing the characteristic.

These attributes are stored in the GATT server in an attribute table. The following properties are associated with each attribute:

- Handle – This property is the attribute's index in the table. Every attribute has a unique handle.
- Type – This attribute indicates what the attribute data represents. This attribute is called a universal unique identifier (UUID). Some of these UUIDs are defined by *Bluetooth* SIG and others are user-definable.
- Permissions – This attribute enforces whether and how a GATT client device can access the value of

the attribute.

5.4.2 GATT Services and Profile

A GATT service is a collection of characteristics. For example, the heart rate service contains a heart rate measurement characteristic and a body location characteristic. You can group services together to form a profile. Many profiles implement only one service; so the two terms are used interchangeably.

The SimpleBLEPeripheral application has the following four GATT profiles:

- Mandatory GAP Service – This service contains device and access information such as the device name, vendor identification, and product identification. This service is a part of the BLE protocol stack and is required for every BLE device per the BLE specification. The source code for this service is not provided but 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. This service is required for every GATT server device per the BLE specification. The source code for this service is not provided but is built into the stack library.
- Device Information Service – This service exposes information about the device such as the hardware version, software version, firmware version, regulatory information, compliance information, and the name of the manufacturer. The Device Information Service is part of the BLE protocol stack and is configured by the application. See Device Information Service (Bluetooth Specification), version 1.0 (24-May-2011) for more information.
- simpleGATTProfile Service – This service is a sample profile for testing and demonstration. The full source code is in the files simpleGATTProfile.c and simpleGATTProfile.h.

Figure 5-9 shows and describes the portion of the attribute table in the SimpleBLEPeripheral project corresponding to the simpleGATTProfile service. This section is an introduction to the attribute table. See Section 5.4.4.2 for information on how this profile is implemented in the code.

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 5-9. simpleGATTProfile Characteristic Table from BTool

The simpleGATTProfile contains the following five characteristics:

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

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

- 0x001F: This attribute is the simpleGATTprofile service declaration. This declaration has a UUID of 0x2800 (*Bluetooth*-defined GATT_PRIMARY_SERVICE_UUID). The value of this declaration is the UUID of the simpleGATTprofile (custom-defined by TI).
- 0x0020: This attribute is the SIMPLEPROFILE_CHAR1 characteristic declaration. This declaration can be thought of as a pointer to the value of SIMPLEPROFILE_CHAR1. This declaration has a UUID of 0x2803 (*Bluetooth*-defined GATT_CHARACTER_UUID). The value of this declaration and all other characteristic declarations is a five-byte value explained as follows (from MSB to LSB):
  - Byte 0: the properties of the SIMPLEPROFILE_CHAR1. These properties are defined in Specification of the Bluetooth System, Covered Core Package version: 4.0 (30-June-2010). The following are a few 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 SIMPLEPROFILE_CHAR1 value is located (handle 0x0021)
  - Bytes 3-4: the UUID of the SimpleProfileChar1 value (custom-defined 0xFFF1)
- 0x0021: This attribute is the SIMPLEPROFILE_CHAR1 value. The attribute has a UUID of 0xFFF1 (custom-defined). Its value is the actual payload data of the characteristic. Indicated by its characteristic declaration (handle 0x0020), this value is readable and writeable.
- 0x0022: This attribute is the SIMPLEPROFILE_CHAR1 user description. The attribute 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 SIMPLEPROFILE_CHAR1 with regard to the remaining four characteristics. The only different attribute, handle 0x002B, is described in the following bullet.
- 0x002B: This attribute is the SIMPLEPROFILE_CHAR4 client characteristic configuration. This configuration has a UUID of 0x2902 (*Bluetooth*-defined). By writing to this attribute, a GATT server can configure the SIMPLEPROFILE_CHAR4 for notifications (writing 0x0001) or indications (writing 0x0002). Writing a 0x0000 to this attribute will disable notifications and indications.

5.4.3 GATT Client Abstraction

Like the GAP layer, the GATT layer is also abstracted. This abstraction depend on whether the device is a GATT Client or a GATT server. According to Device Information Service (Bluetooth Specification), version 1.0 (24-May-2011), the GATT layer is an abstraction of the ATT layer.

GATT clients do not have attribute tables or profiles because they gather information rather than serving it. Most interfacing with the GATT layer occurs directly from the application. Use the direct GATT API described in Appendix D. Figure 5-10 shows the abstraction.

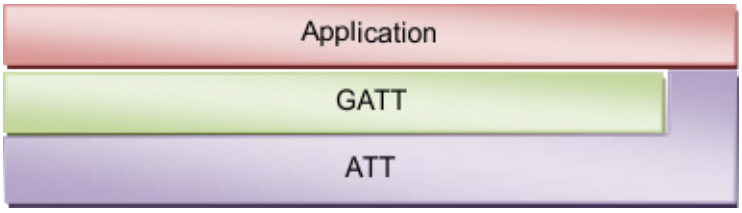


Figure 5-10. GATT Client Abstraction

5.4.3.1 Using the GATT Layer Directly

This section describes how to use the GATT layer directly in the application. The functionality of the GATT layer is implemented in the library code but you can find the header functions can in gatt.h. You can find the complete API for the GATT layer in Appendix D. You can find more information on the functionality of these commands in the Device Information Service (Bluetooth Specification), version 1.0 (24-May-2011). GATT client applications uses these functions primarily. A few server-specific functions exist which are described in the API and not considered here. Most GATT functions return ATT events to the application, so consider the ATT API in Appendix D. Perform the following procedure to use the GATT layer when functioning as a GATT client (that is, in the SimpleBLECentral project):

1. Initialize the GATT client. Do this in the application initialization function.

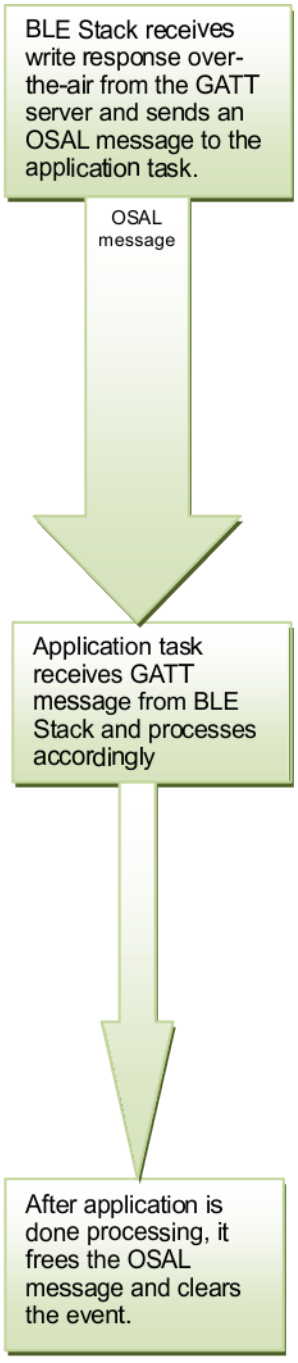
```
VOID GATT_InitClient();
```

2. Register to receive incoming ATT indications and notifications. Do this in the application initialization function.

```
GATT_RegisterForInd(selfEntity);
```

3. Perform a GATT client procedure. The following example uses GATT_WriteCharValue(), which is triggered by pressing the key on the left in the SimpleBLECentral application.

4. Receive and handle the response to the GATT client procedure in the application. In the following example, the application receives an ATT_WRITE_RSP event. See Section D.5 for a list of GATT commands and their corresponding ATT events.



Library Code

simpleBLECentral.c:

```
uint16 SimpleBLECentral_ProcessEvent( uint8 task_id, uint16 events )
{
…
  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;

    if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
    {
      simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)pMsg );
…
```

```
static void simpleBLECentral_ProcessOSALMsg( osal_event_hdr_t *pMsg )
{
...
    case GATT_MSG_EVENT:
      simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *) pMsg );
      …
```

```
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 ) ) )
  {
  …
```

simpleBLECentral.c:

```
…
      // Release the OSAL message
      VOID osal_msg_deallocate( pMsg );
    }

    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }
```

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

Besides receiving responses to its own commands, a GATT client may also receive asynchronous data from the GATT server as indications or notifications.

5. Ensure the GATT client is registered to receive these ATT events in step 2.

   These events will also be sent as ATT events in GATT messages to the application and should be handled as described in this procedure.

5.4.4 GATT Server Abstraction

As a GATT server, most of the GATT functionality is handled by the individual GATT profiles. These profiles use of the GattServApp, a configurable module which stores and manages the attribute table. Figure 5-11 shows the abstraction hierarchy:

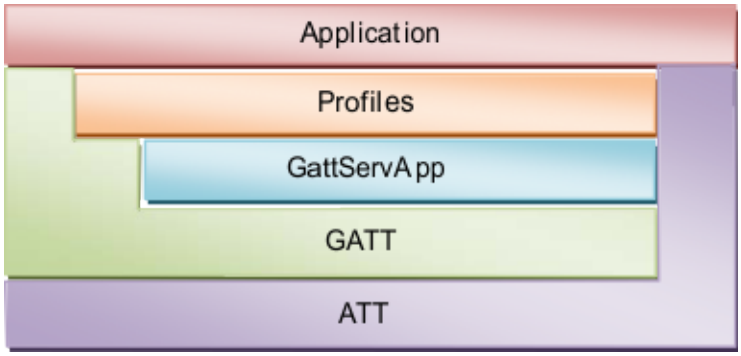


Figure 5-11. GATT Server Abstraction

The design process is as follows:

1. Create GATT profiles that configure the GATTServApp module.
2. Use API of the module to interface with the GATT layer.

With a GATT server, direct calls to GATT layer functions are unnecessary. The application interfaces with the profiles.

5.4.4.1 GATTServApp Module

The GATTServApp module stores and manages the application-wide attribute table. Various profiles use the table to add their characteristics to the attribute table. The BLE stack uses the table 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 receives this message and uses the GATTServApp module to find and send the primary characteristics in the attribute table wirelessly. This type of functionality is beyond the scope of this document and is implemented in the library code. The functions of the GATTServApp are accessible from the profiles and defined in gattservapp_util.c and in the API in Appendix E. These functions include finding specific attributes, reading client characteristic configurations, and modifying client characteristic configurations.

5.4.4.1.1 Building Up the Attribute Table

When powering on or resetting the device, the application builds the GATT table by using the GATTServApp module to add services. Each service is a list of attributes with UUIDs, values, permissions, and read/write call-backs. Figure 5-12 shows that this information is passed through the GATTServApp to GATT and stored in the stack. Do this in the application initialization function, that is, simpleBLEPeripheral_init():

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

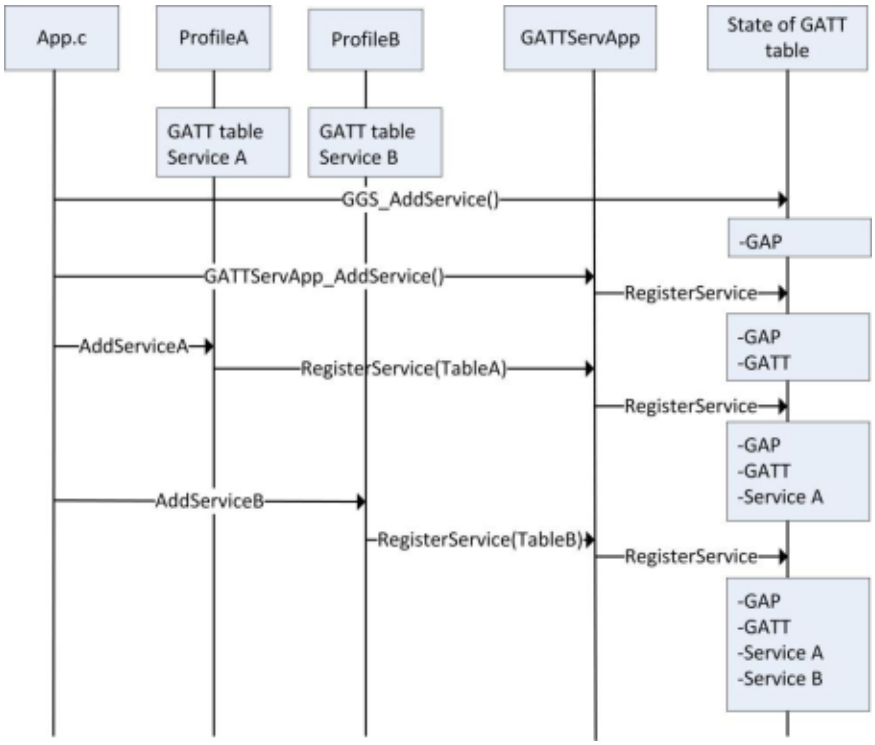


Figure 5-12. Attribute Table Initialization

5.4.4.2 Profile Architecture

This section describes the architecture for profiles and provides functional examples of the simpleGATTProfile in the SimpleBLEPeripheral project. See Section 5.4.2 for an overview of the simpleGATTProfile.

To interface with the application and BLE protocol stack, each profile must contain the elements in the following sections

5.4.4.2.1 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 follows:

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

Each attribute in the following 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 elements of this attribute type are as follows:

- type: This is the UUID associated with the attribute. This UUID is defined as the following:

```
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 in the profile.
- permissions – This element enforces how and if a GATT client device can access the value of the attribute. Possible permissions are defined in gatt.h as the following:
  - 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_ENCRYPT_READ // Read requires Encryption
  - GATT_PERMIT_ENCRYPT_WRITE // Write requires Encryption

  Section 5.3 describes authentication, authorization, and encryption further.
- handle – This is a placeholder in the table where GATTServApp assigns a handle. Handles are assigned sequentially.
- pValue – This is a pointer to the attribute value. The size cannot be changed after initialization. The maximum size is 512 octets.

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

5.4.4.2.1.1 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 */
},
```

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

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

5.4.4.2.1.2 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 must read the UUID, so the permission must be set to GATT_PERMIT_READ.

Section 5.4.1 describes the value of a characteristic declaration. The pointer to the properties of the characteristic value is passed to the GATTServApp in pValue. The GATTServApp adds the UUID and the handle of the value. These properties are defined as the following:

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

NOTE: These properties are visible to the GATT client stating the properties of the characteristic value but the GATT permissions of the characteristic value affect its functionality in the protocol stack. These properties must match the GATT permissions of the characteristic value. The following section expands on this.

5.4.4.2.1.3 Characteristic Value

Consider the value of the simpleGATTProfile SIMPLEPROFILE_CHAR1.

```
// 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 the properties of this characteristic value are readable and writeable, set the GATT permissions to readable and writable.

NOTE: If the GATT permissions are not set to readable and writable, errors occur.

The pValue is a pointer to the location of the actual value. This value is statically defined in the profile as follows:

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

5.4.4.2.1.4 Client Characteristic Configuration

Consider the simpleGATTProfile simpleProfileCharacteristic4 configuration.

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

The type is set to the *Bluetooth* SIG-defined client characteristic configuration UUID (0x2902).

GATT clients must read and write to this so the GATT permissions are set to readable and writable.

The pValue is a pointer to the location of the client characteristic configuration array, defined in the profile as the following:

```
static gattCharCfg_t *simpleProfileChar4Config;
```

NOTE: Because this value must be cached for each connection, this is an array ion.Tthe following section describes this ion.

5.4.4.2.2 Add Service Function

As Section 5.4.4.1 describes, when an application starts up it must add the GATT services it supports. Each profile needs a global AddService function that can be called by the application. Some of these services are defined in the protocol stack, such as GGS_AddService and GATTServApp_AddService. User-defined services must expose their own AddService function that the application can call for profile initialization. Using SimpleProfile_AddService() as an example, these functions should do the following:

- Allocate space for the client characteristic configuration (CCC) arrays.

  As described in Section 5.4.4.2.1.4, a pointer to one of these arrays initialized in the profile. In the AddService function, several supported connections are declared and memory is allocated for each array. One CCC is defined in the simpleGATTProfile but the profile may contain additional CCCs.

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

- Initialize the CCC arrays.

  CCC values do not change between power downs and bonded device connections because they are stored in NV. For each CCC in the profile, the GATTServApp_InitCharCfg() function must be called. This function initializes the CCCs with information from a previously bonded connection. If the function cannot find the information, set the initial values to default values.

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

- Register the profile with the GATTServApp.

  This function passes the attribute table of the profile to the GATTServApp so the attributes of the profile are added to the application-wide attribute table, which is managed by the protocol stack. The GATTServApp function assigns handles for each attribute. This function also passes pointers the callback of the profile to the stack to initiate communication between the GATTServApp and the profile.

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

5.4.4.2.3 Register Application Callback Function

Profiles can relay messages to the application using callbacks. In the SimpleBLEPeripheral project, the simpleGATTProfile calls an application callback whenever the GATT client writes a characteristic value. To use these application callbacks, the profile must define a register application callback function that the application uses to set up callbacks during initialization. The following is the register application callback function of the simpleGATTProfile:

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

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

Where the typedef callback is defined as follows:

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

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

The application must define a callback of this type and pass it to the simpleGATTProfile with the SimpleProfile_RegisterAppCBs() function. The application does this in simpleBLEPeripheral.c through the following:

```
// Simple GATT Profile Callbacks
static simpleProfileCBs_t simpleBLEPeripheral_SimpleProfileCBs =
{
  simpleProfileChangeCB    // Charactersitic value change callback
};
…
// Register callback with SimpleGATTprofile
VOID SimpleProfile_RegisterAppCBs( &simpleBLEPeripheral_SimpleProfileCBs );
```

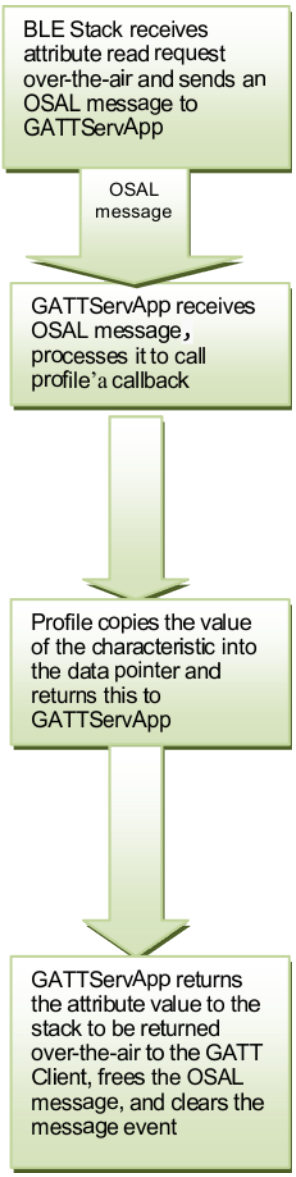
See the following section for the mechanism of how this callback is used.

5.4.4.2.4 Read and Write Callback Functions

The profile must define read and write callback functions that the protocol stack will call when one of the attributes of the profile is written to and/or read from. The callbacks must be registered with GATTServApp function as mentioned in Section 5.4.4.2.2. These callbacks perform the characteristic reads, writes, and other processing like possibly calling an application callback defined by the profile.

5.4.4.2.4.1 Read Request from A GATT Client

When a read request from a GATT client is received for a given attribute, the protocol stack checks the permissions of the attribute. If the attribute is readable, the protocol stack calls the read call-back of the profile. The profile must copy the value, perform any profile-specific processing, and notify the application if applicable. The following flow diagram shows the processing workflow for a read of SIMPLEPROFILE_CHAR1 in the simpleGATTProfile.



Library Code

Library Code

simpleGATTProfile.c

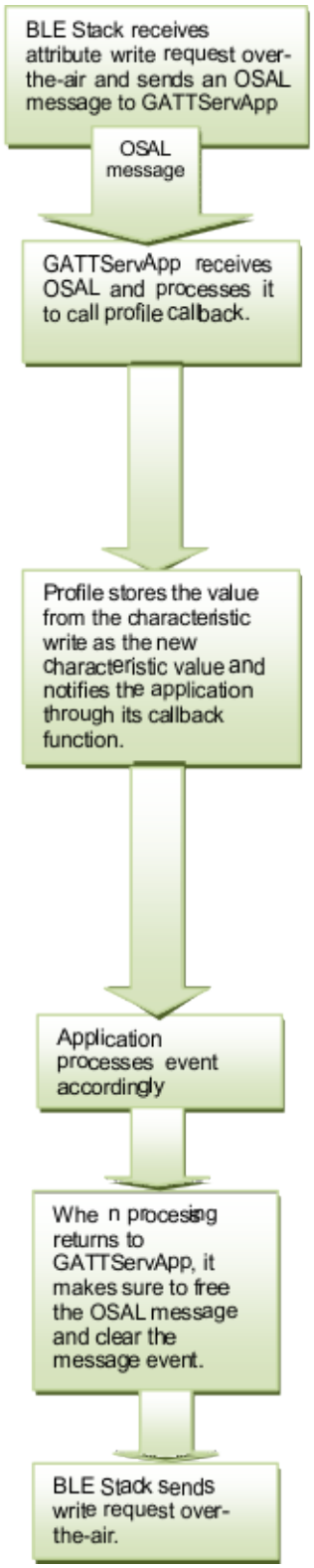
```
static bStatus_t simpleProfile_ReadAttrCB(......)
{
…
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    // 16-bit UUID
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
      >type.uuid[1]);
    switch ( uuid )
    {
      case SIMPLEPROFILE_CHAR1_UUID:
        *pLen = 1;
        pValue[0]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

Library Code

NOTE: Consider the processing in this section in the context of the protocol stack. If any intensive profile related processing must be completed for an attribute read, this processing should be split up and completed in the context of the application task. See the following write request for more information.

5.4.4.2.4.2 Write Request from Client

When a write request from a GATT Client is received for a given attribute, the protocol stack will check the permissions of the attribute. If the attribute is write, call the write callback of the profile. The profile stores the value to be written, performs any profile-specific processing, and notifies the application if applicable. The following flow diagram illustrates a write of simpleprofileChar3 in the simpleGATTProfile. In the diagram, red corresponds to processing in the protocol stack context and green corresponds to processing in the application context.



Library Code

Library Code

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 simpleProfileChangeCB( uint8 paramID )
{
…
```

Library Code

Library Code

NOTE: Minimize the processing done in the stack task. Set an application so processing can complete in the application task if extensive additional processing beyond storing the attribute write value in the profile is required.

5.4.4.2.5 Get and Set Functions

The profile containing the characteristics provides set and get abstraction functions for the application to read and/or write a characteristic. The set parameter function should include logic to check for and implement notifications and/or indications if the relevant characteristic has notify and/or indicate properties. The following flow chart and code depict how to set the simpleProfileChacteristic4 in the simpleGATTProfile.

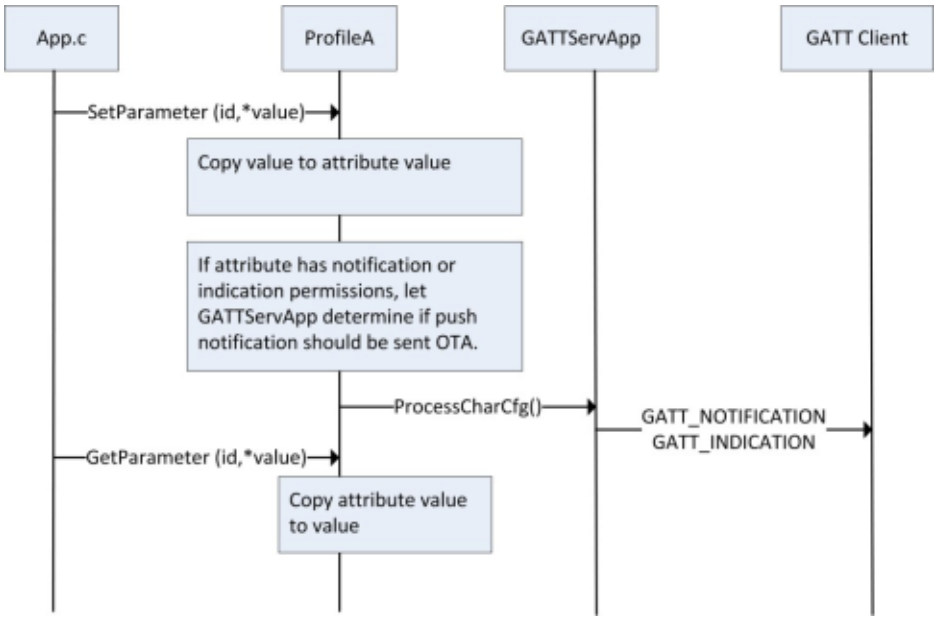


Figure 5-13. Get and Set Profile Parameter Usage

The application initializes simpleProfileCharacteristic4 to 0 in SimpleBLEPeripheral.c through the following:

```
uint8 charValue4 = 4;
SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8), &charValue4);
```

The code for this function is displayed in the following code from simpleGATTProfile.c. Other than setting the value of the static simpleProfileChar4, this function also calls GATTServApp_ProcessCharCfg because it has GATT_PROP_NOTIFY properties. This call forces GATTServApp to check if notifications have been enabled by the GATT client. If so, the GATTServApp sends a notification of this attribute to the GATT client.

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

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

5.5 L2CAP

The L2CAP layer sits on top of the HCI layer and transfers data between the upper layers of the host (GAP, GATT, application, and so forth) and the lower-layer protocol stack. This layer multiplexes higher-level protocol and reassembles 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, [SDUs]) up to 64KB long. The amount of memory available on the specific device implementation limits the size. The CC254x and the 1.4.1 stack support an effective MTU size of 23, set by L2CAP_MTU_SIZE in l2cap.h. Changing the L2CAP_MTU_SIZE results in incorrect stack behavior.

5.6 HCI

The HCI layer is a thin layer that transports commands and events between the host and controller. In a network processor application, the HCI layer is implemented by a transport protocol such as SPI or UART. In embedded SoC projects, the HCI layer is implemented through function calls and callbacks. The commands and events discussed previously in this guide pass from the given layer through the HCI layer to the controller and through the controller to the HCI layer.

5.6.1 HCI Extension Vendor-Specific Commands

Some HCI extension vendor-specific commands extend the functionality of the controller for the application and host. See Appendix G for a description of HCI extension commands and examples for an SoC project.

5.6.2 Receiving HCI Extension Events in the Application

Like the GAP and ATT layers, the HCI extension commands result in HCI extension events being passed passing from the controller to the host. Additional steps are required to receive these HCI extension events at the application.

The GAPRole task registers receive HCI Extension events in gapRole_init() by default:

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

The GAPRole task receives events related to HCI extension commands, even if they are called from the application. Implement a callback function to pass these events from the GAPRole task back to the application if needed in the application.

5.7 Library Files

Each project must include the following two library files:

- BLE Stack Library: This library includes the lower-layer stack functionality and varies based on the GAP role. You can include the full library, but you can use a smaller subset to conserve code space typically. This library is different for a CC2540 and CC2541 project.
- HCI Transport Layer Library: This library includes transport layer functionality for a network processor. This library is the same for CC2540 and CC2541.

The library files are at $INSTALL$\Projects\ble\Libraries. Use Table 5-2 to determine the correct library file to use in the project.

Table 5-2. Supported BLE-Stack Library Configurations

Configuration	GAP Roles Supported				Chipset	Library
	Broadcaster	Observer	Peripheral	Central		
Network processor	X	X	X	X	CC2540	CC2540_BLE.lib CC254X_BLE_HCI_TL_Full.lib
Single-device	X	X	X	X	CC2540	CC2540_BLE.lib
Single-device	X				CC2540	CC2540_BLE_bcast.lib
Single-device	X	X			CC2540	CC2540_BLE_bcast_observ.lib
Single-device		X		X	CC2540	CC2540_BLE_cent.lib
Single-device	X	X		X	CC2540	CC2540_BLE_cent_bcast.lib
Single-device		X			CC2540	CC2540_BLE_observ.lib
Single-device	X		X		CC2540	CC2540_BLE_peri.lib
Single-device	X	X	X		CC2540	CC2540_BLE_peri_observ.lib
Network processor	X	X	X	X	CC2541	CC2541_BLE.lib CC254X_BLE_HCI_TL_Full.lib
Single-device	X	X	X	X	CC2541	CC2541_BLE.lib
Single-device	X				CC2541	CC2541_BLE_bcast.lib
Single-device	X	X			CC2541	CC2541_BLE_bcast_observ.lib
Single-device		X		X	CC2541	CC2541_BLE_cent.lib
Single-device	X	X		X	CC2541	CC2541_BLE_cent_bcast.lib
Single-device		X			CC2541	CC2541_BLE_observ.lib
Single-device	X		X		CC2541	CC2541_BLE_peri.lib
Single-device	X	X	X		CC2541	CC2541_BLE_peri_observ.lib

Drivers

The hardware abstraction layer (HAL) of the CC254x software provides an interface of abstraction between the physical hardware and the application and/or protocol stack. This HAL allows for the development of new hardware (such as a new PCB) without making changes to the protocol stack or application source code. The HAL includes software for the SPI and UART communication interfaces, AES, keys, LCD, and LEDs. The HAL drivers that support the following hardware platforms include the following:

- SmartRF05EB + CC2540EM
- SmartRF05EB + CC2541EM
- CC2540 Keyfob
- CC2541 Keyfob
- CC2541 SensorTag
- CC2540 USB Dongle

When developing with a different hardware platform, you might need to modify the HAL source for compatablility.

Find the HAL files in the sample projects by doing the following:

1. Click HAL.
2. Click Target.
3. Click CC2540EB.
4. Click Drivers. See Figure 6-1.

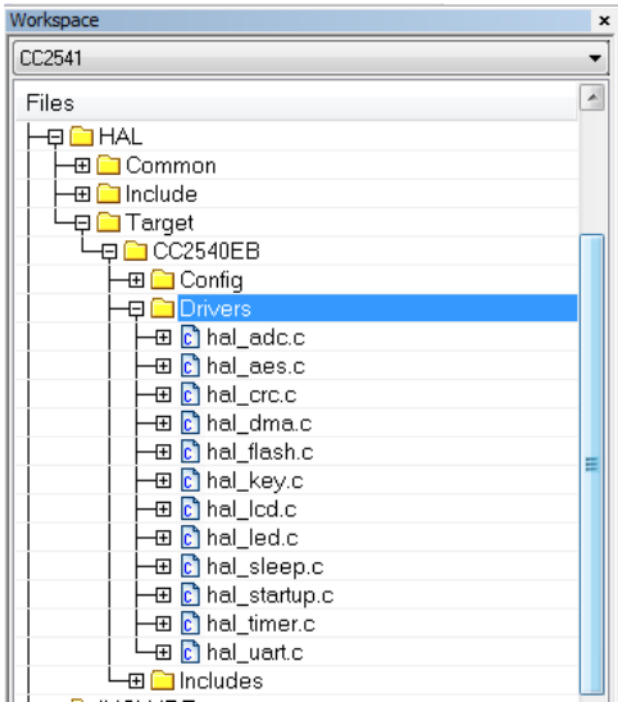


Figure 6-1. HAL Drivers

The following drivers are supported:

6.1 ADC

To include the ADC driver, define HAL_ADC=TRUE in the preprocessor definitions. See the hal_adc.h header file for the ADC API. *CC2541/43/44/45 Peripherals Software Examples* (SWRC257) has several ADC examples.

6.2 AES

To include the AES driver, define HAL_AES=TRUE. The stack requires AES for encryption. Always set this symbol set to TRUE. You cannot reuse DMA channels 1 and 2 because the AES driver uses them. Set the following in hal_aes.h:

```
/* Used by DMA macros to shift 1 to create a mask for DMA registers. */
#define  HAL_DMA_AES_IN    1
#define  HAL_DMA_AES_OUT   2
```

You can find the AES API in hal_ase.h. TI recommends letting the application use the HCI encrypt and decrypt functions (from hci.h):

```
hciStatus_t HCI_LE_EncryptCmd( uint8 *key, uint8 *plainText )
```

```
hciStatus_t HCI_EXT_DecryptCmd( uint8 *key, uint8 *encText )
```

6.3 LCD

To include the LCD driver, define HAL_LCD=TRUE in the preprocessor definitions. TI designed the driver to function with the LCD on the SmartRF05 boards. You must modify parts of the driver such as the pin and port definitions in hal_lcd.c to use it with custom hardware. See the CC2540/1 configuration in the SimpleBLEPeripheral project for an example using the LCD driver.

6.4 LED

To include the LED driver, define HAL_LED=TRUE in the preprocessor definitions. TI designed the driver to function with the LEDs on the Keyfob. You must modify parts of the driver such as the LED bit definitions in hal_led.c to use it with custom hardware. See the CC2540/1 DK-miniKeyfob in the SimpleBLEPeripheral project for an example using the LED driver.

6.5 KEY

The KEY driver handles button inputs. To include the KEY driver, define HAL_KEY=TRUE in the preprocessor definitions. TI designed the driver to function with the buttons on the keyfob or SmartRF05 Board depending on whether CC2540_MINIDK is defined. You must modify parts of the driver such as the port and pin definitions in hal_key.c to use it with custom hardware. See the SimpleBLEPeripheral project for an example using the KEY driver.

6.6 DMA

To include the DMA driver, define HAL_DMA=TRUE in the preprocessor definitions. Because the AES driver uses the DMA driver, always include the DMA driver in the project. Channels 1 and 2 are reserved for the AES driver. If the UART DMA or SPI driver is used, other channels are used. You can find the DMA API in hal_dma.h. See the SPI driver for an example of using the DMA driver.

6.7 UART and SPI

Describing the UART and SPI drivers is beyond the scope of this document. See the Device Information Service (Bluetooth Specification), version 1.0 (24-May-2011) for several UART and SPI examples, specifically the BLE serial bridge.

6.8 Other Peripherals

See *CC2541/43/44/45 Peripherals Software Examples* (SWRC257) for examples of hardware peripherals without drivers, such as the timers. See *CC254x Chip User's Guide* (SWRU191) for explanation of the hardware peripherals.

6.9 Simple NV (SNV)

The SNV area of flash securely stores persistent data, such as encryption keys from bonding or custom parameters. The protocol stack reserves two 2-kB flash pages for SNV. These pages are the last two pages of flash by default. To minimize the number of erase cycles on the flash, the SNV manager performs compactions on the flash sector when the sector has 80% invalidated data. A compaction is copying valid data to a temporary area then erasing the sector where the data was stored. The SNV driver uses the hal_flash driver.

SNV can be read from or written to using the following APIs:

uint8 osal_snv_read(osalSnvId_t id, osalSnvLen_t len, void *pBuf) *Read data from NV*

Parameters id – valid NV item

len – Length of data to read

pBuf – pointer to buffer to store data read

Returns SUCCESS: NV item read successfully

NV_OPER_FAILED: failure reading NV item

uint8 osal_snv_write(osalSnvId_t id, osalSnvLen_t len, void *pBuf) *Write data to NV*

Parameters id – valid NV item

len – Length of data to write

pBuf – pointer to buffer containing data to be written

Returns SUCCESS: NV item read successfully

NV_OPER_FAILED: failure reading NV item

Because SNV is shared with other modules in the BLE SDK such as the GAPBondMgr, carefully manage the IDs of the NV item. The available IDs are defined in bcomdef.h by default:

```
// 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
```

Creating a Custom BLE Application

After reading the preceding sections, you should understand the general system architecture, application, BLE stack framework to implement a custom *Bluetooth* Smart™ application. This following section provides guidance on where and how to start writing a custom application and some considerations.

7.1 Configuring the BLE Stack

You must decide which role and purpose the custom application should have. If the application is related to a specific service or profile, start with one of those. An example application includes the heart rate sensor project. TI recommends starting with one of the following SimpleBLE sample projects:

- SimpleBLECentral
- SimpleBLEPeripheral
- SimpleBLEBroadcaster
- SimpleBLEObserver

When deciding the role and purpose of the application, choose from the appropriate libraries in Section 5.7.

7.2 Define BLE Behavior

Use the BLE protocol stack APIs to define the system behavior such as adding profiles, defining the GATT database, configuring the security model, and so forth. Use the concepts in Chapter 5 and the BLE API reference in the appendices A through G of this guide.

7.3 Define Application Tasks

Ensure the application contains callbacks from the various stack layers and event handlers to process OSAL messages. You can add other tasks by following the guidelines in Chapter 3.

7.4 Configure Hardware Peripherals

Add drivers as specified in Chapter 6. If drivers do not exist for a given peripheral, create a custom driver.

7.5 Configuring Parameters for Custom Hardware

You must adjust several software parameters when working with custom hardware.

7.5.1 Board File

You can find the board file (hal_board_cfg.h) in the sample projects by doing the following:

1. Click HAL.
2. Click Target.
3. Click CC2540EB.
4. Click Config.

Depending on the hardware platform (keyfob, EM, and so forth), the project contains a different board file. These board files are specific to the given platform and must be adjusted for custom hardware. Some modifications may include the following:

- Modifying the symbols used by drivers for specific pins (that is, LED1_SBIT)
- Selecting the 32-kHz oscillator source (OSC_32KHZ)
- Initializing input and output pins to safe initialization levels to prevent current leakage

7.5.2 Adjusting for 32-MHz Crystal Stabilization Time

Before entering sleep, the stack sets the sleep timer to wake before the next BLE event. The closer the wakeup is to the event, less power is wasted. If the wakeup is too close to the even, the sleep time might miss the event.

When the timer wakes, it must wait for the 32-MHz external crystal to stabilize. This stabilization time is affected by the inherent stabilization time of the crystal, how long the crystal has been off, the temperature, the voltage, and so forth. You must add a buffer to the wakeup time (that is, start earlier) to handle this variability in stabilization.

This buffer time is implemented using the HAL_SLEEP_ADJ_TICKS definition where the value of the definition corresponds to the number of 32-MHz ticks. This definition is set in hal_sleep.c to 25 for the CC2541 EM and 35 for the CC2540 EM by default. If the value of the definition is larger, the buffer time is longer and more power is wasted. Calculate this value empirically. If HAL_SLEEP_ADJ_TICKS is set too low, false advertisement restarts and connection drops occur. If these restarts and drops occur, increase the definition until they stop.

7.5.3 Setting the Sleep Clock Accuracy

If you must modify the sleep clock accuracy from the default (50 ppm for a master and 40 ppm for a slave), use the HCI_EXT_SetSCACmd(). See Appendix G for more information.

7.6 Software Considerations

7.6.1 Memory Management for GATT Notifications and Indications

TI recommends using the SetParameter function (that is, SimpleProfile_SetParameter()) and call GATTServApp_ProcessCharCfg() to send a GATT notification or indication. If using GATT_Notification() or GATT_Indication() directly, you require additional memory management. For additional memory management, do the following:

1. Attempt to allocate memory for the notification or indication using GATT_bm_alloc().
2. If allocation succeeds, send the notification or indication using GATT_Notification()/GATT_Indication().

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

If the return value is something other than SUCCESS (that is, blePending), free the memory using GATT_bm_free().

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

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

    if            & 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;
}
```

7.6.2 Limit Application Processing During BLE Activity

Because of the time-dependent nature of the BLE protocol, the controller (LL_ProcessEvent()) must process before each connection event or advertising event. If the controller does not get process, advertising restarts or the connection drops. Because OSAL is not multithreaded, each task must stop processing to let the controller process. The stack layers do not have this issue. Ensure that the application processes less than the following:

(connection/advertising interval) – 2 ms

The 2 ms are added as buffer to account for controller processing time. If extensive processing is required in the application task, split it up using OSAL events in Section 3.2.

7.6.3 Global Interrupts

During BLE activity, the controller must process radio and MAC timer interrupts quickly to set up the BLE event postprocessing. Never globally disable interrupts during BLE activity.

Development and Debugging

Embedded software for the CC2540/41 is developed using Embedded Workbench for 8051 9.10.3 from IAR software. This section provides where to find this software and contains some basics on the use of IAR, such as opening and building projects, as well as information on the configuration of projects using the BLE protocol stack. IAR contains many features beyond the scope of this document. More information and documentation is available on the IAR website: www.iar.com.

8.1 IAR Overview

Two options are available for developing software on the CC2540/41:

- Download IAR Embedded Workbench 30-day Evaluation Edition – This version of IAR is free and fully functional for 30 days. This version includes the standard features. Download the IAR 30-day Evaluation Edition from the following URL: http://supp.iar.com/Download/SW/?item=EW8051-EVAL
- Purchase the full-featured version of IAR Embedded Workbench – For complete BLE application development using the CC2540/41, TI recommends the complete version of IAR without restrictions. Information on purchasing the complete version of IAR is available at the following URL: http://www.iar.com/en/Products/IAR-Embedded-Workbench/8051/

8.2 Using IAR Embedded Workbench

After installing IAR Embedded Workbench, download the latest patches from IAR. These patches are required to build and debug projects with the CC2540/41. When the patches have been installed, you can develop software for the CC2540/41. This section describes how to open and build an existing project for a CC2540. Similar steps apply for a CC2541. This section uses the SimpleBLEPeripheral project as an example. The TI BLE software development kit includes the SimpleBLEPeripheral project.

8.2.1 Open an Existing Project

To open an existing project when using Windows, do the following:

1. Click Start.
2. Click Programs.
3. Click IAR Systems.
4. Click IAR Embedded Workbench for 8051 9.10.
5. Click IAR Embedded Workbench.

   When IAR opens up, do the following:
6. Click File.
7. Click Open.
8. Click Workspace.
9. Select the following file:
   $INSTALL$\Projects\ble\SimpleBLEPeripheral\CC2540DB\SimpleBLEPeripheral.eww

This file is the workspace for the SimpleBLEPeripheral project. When you select this file, the files associated with the workspace should also open with a list of files on the left side. See Figure 8-1 for the IAR Embedded Workbench.

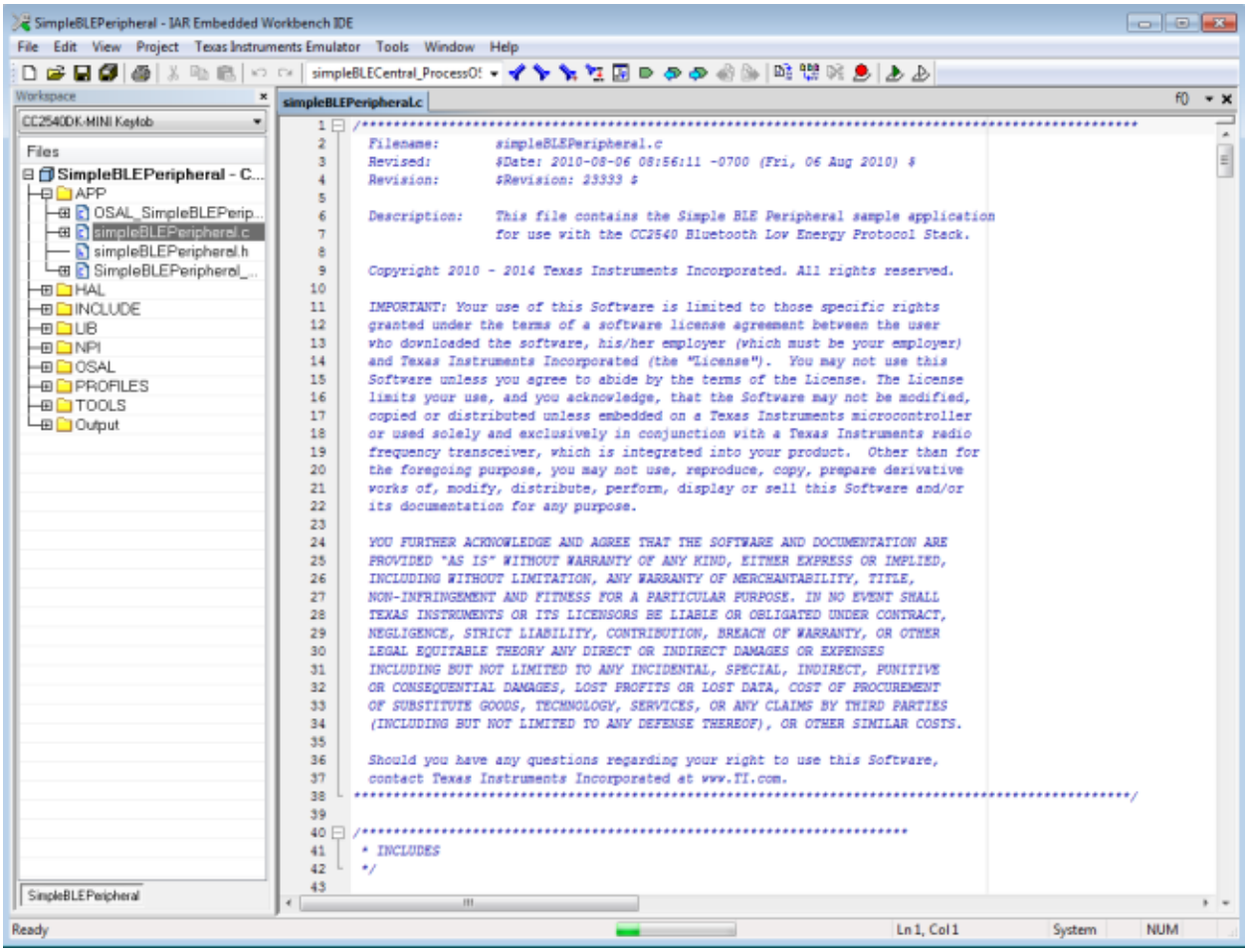


Figure 8-1. IAR Embedded Workbench

8.2.2 Project Options, Configurations, and Defined Symbols

Every project has a set of options, including settings for the compiler, linker, debugger, and so forth.

To view the project options, do the following:

1. Right-click on the project name at the top of the file list.

2. Select Options… (See Figure 8-2.)

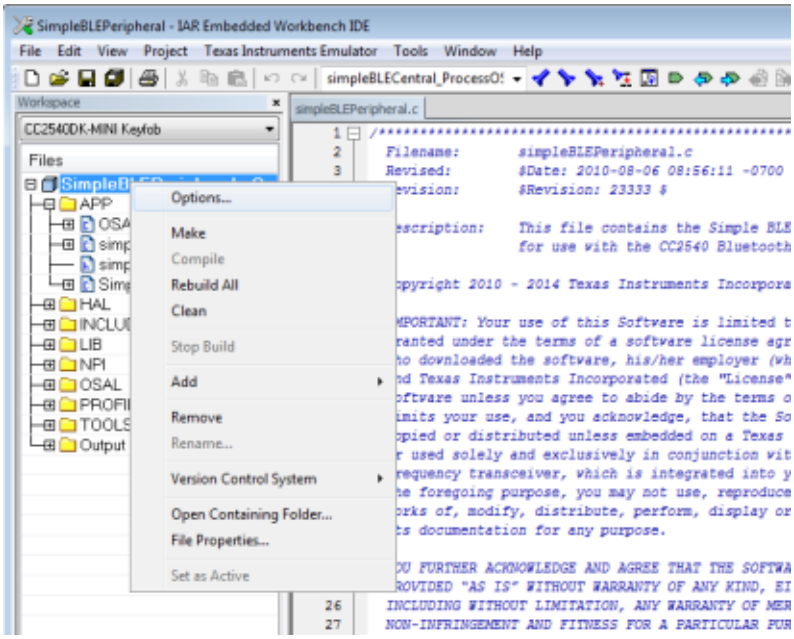


Figure 8-2. Project Configurations and Options

After clicking Options…, a window will pop-up displaying the project options. You might need to have a few different configurations of options for different setups, like when using multiple hardware platforms. The IAR lets you create configurations. You can select these configurations through the drop-down menu in the top of the Workspace pane (see Figure 8-2).

The default configuration in the SimpleBLEPeripheral project is the CC2540DK-MINI Keyfob configuration, which is targeted toward the keyfob hardware platform included with the CC2540/41DK mini development kit. Alternatively, CC2540 is optimized for the SmartRF05 + CC2540 EM included with the full development kit. Other configurations include a 128-KB part, an OAD, and so forth:

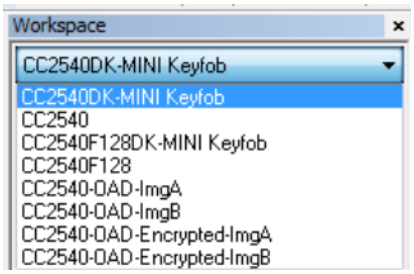


Figure 8-3. Project Configurations

Compiler preprocessor definitions or symbols are important settings when building a project (see Figure 8-4).

You can find and set these values by doing the following:

1. Click the C/C++ Compiler category on the left.
2. Click the Preprocessor tab on the right:

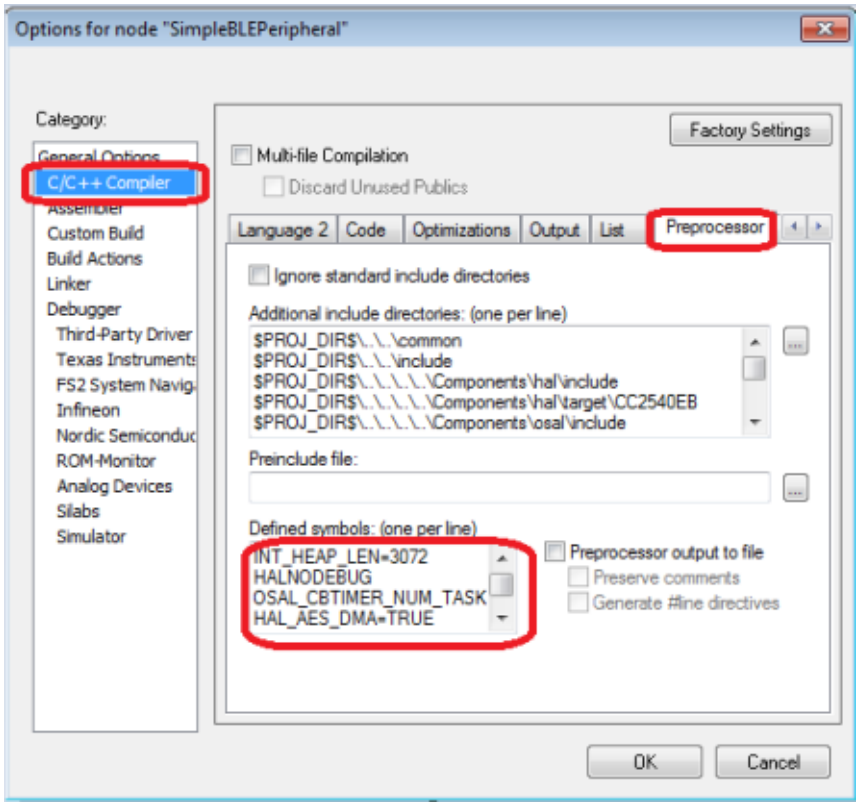


Figure 8-4. Preprocessor Defined Symbols Settings

When preceded by an x, the symbol has no valid definition and can be considered disabled. Removing the preceding x to restore the proper name of the symbol reenables the feature or definition.

Symbols can be defined in configuration files, which are included when compiling. The Extra Options tab under the compiler settings let you set up the configuration files to be included. You must include the config.cfg file with every build because it defines some required universal constants. The buildConfig.h file included with the software development kit defines the appropriate symbols for the project (see Figure 8-5).

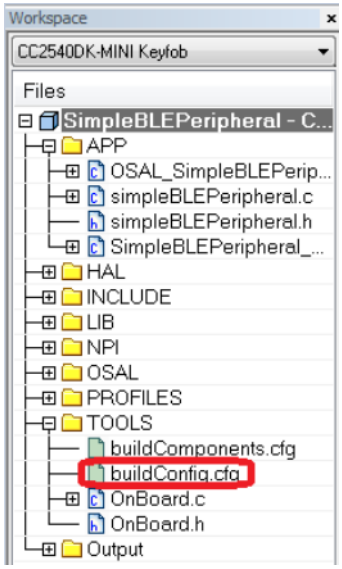


Figure 8-5. The buildConfig.h File

Select the *Use command line option* box through the *Extra Options* tab shown in Figure 8-6 to supply the compiler with additional options.

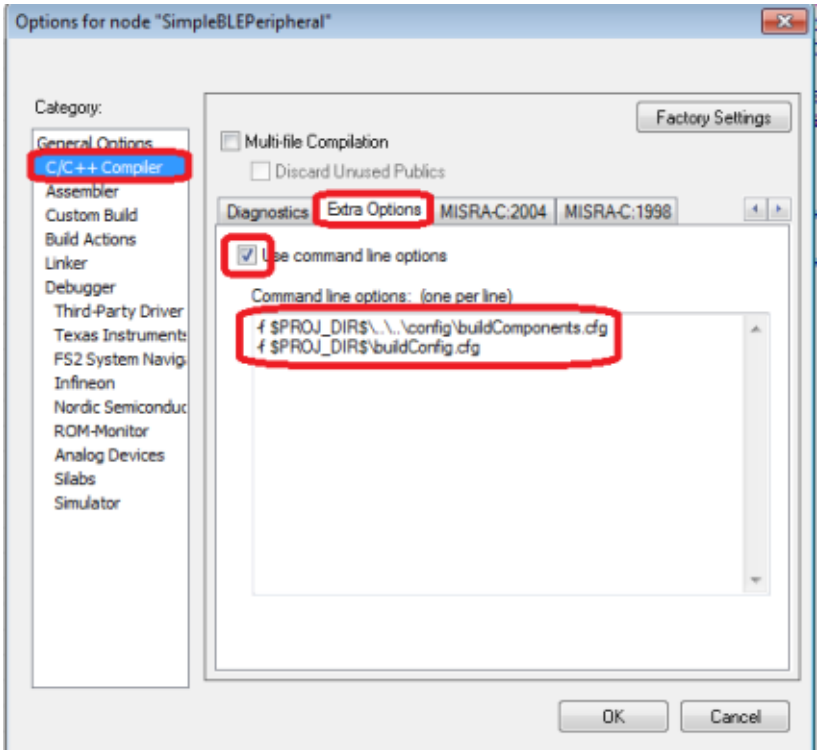


Figure 8-6. Configuration File Setup

The BLE protocol stack and software uses the following symbols. You can find them in the sample project:

Symbols Mandatory for BLE Stack

- **INT_HEAP_LEN** – This symbol defines the size of the heap used by the OSAL Memory Manager (see Section 3.3) in bytes. The default value in the sample project is 3072. You can increase this value if the application requires additional heap memory. If this value is increased too much, you may exceed the RAM limit. If the application requires additional memory for local variables, you may need to increase this value. The memory set aside for the heap shows up in the map file under the OSAL_Memory module. For more information on the map file, see section Section 8.2.4.
- **HALNODEBUG** – Define this symbol for all projects to disable HAL assertions.
- **OSAL_CBTIMER_NUM_TASKS** – This symbol defines the number of OSAL callback timers that you can use. The BLE protocol stack uses the OSAL callback timer. You must define this value as either 1 or 2 (a maximum of two callback timers are allowed). For applications without any callback timers such as the sample application, define this value as 1.
- **HAL_AES_DMA** – Define this symbol as TRUE because the BLE stack uses DMA for AES encryption.
- **HAL_DMA** – This value must be defined as TRUE for all BLE projects, as the DMA controller is used by the stack when reading and writing to flash.

Optional Symbols

- **POWER_SAVING** – When defined, this symbol configures the system to go into sleep mode when free of any pending tasks.
- **PLUS_BROADCASTER** – This symbol indicates that the device is using the GAP Peripheral/Broadcaster multirole profile rather than the single GAP Peripheral role profile. The default option in the simpleBLEPeripheral project is undefined.
- **HAL_LCD** – This symbol indicates whether to include and use the LCD driver when set to TRUE. If not defined, it is set to TRUE.
- **HAL_LED** – This symbol indicates whether to include the LED driver when set to TRUE. If not defined, it is set to TRUE.
- **HAL_KEY** – This symbol indicates whether to include the KEY driver when set to TRUE. If not defined, it is set to TRUE.
- **HAL_UART** – This symbol indicates whether to include the UART driver when set to TRUE. If not defined, it is set to FALSE.
- **CC2540_MINIDK** – Define this symbol when using the keyfob board in the CC2540/41DK-MINI development kit. This symbol configures the hardware based on the keyfob board layout.
- **HAL_UART_DMA** – This symbol sets the UART interface to use DMA mode when set to 1. When HAL_UART is defined, set either HAL_UART_DMA or HAL_UART_ISR to 1.
- **HAL_UART_ISR** – This symbol sets the UART interface to use ISR mode when set to 1. When HAL_UART is defined, set either HAL_UART_DMA or HAL_UART_ISR to 1.
- **HAL_UART_SPI** – This symbol indicates whether to include the SPI driver.
- **GAP_BOND_MGR** – The HostTestRelease network processor project uses this symbol. When this symbol is defined for slave and peripheral configurations, use the GAP peripheral bond manager security profile to manage bonds and handle keys. See Section 5.3 for more information on the peripheral bond manager.
- **GATT_DB_OFF_CHIP** – The HostTestRelease network processor project uses this symbol. This symbol sets a GATT client in a network processor configuration to manage the attributes in the application processor instead of the CC2540/41.

Other definitions relating to specific use cases such as the serial bootloader, OAD, and so forth are defined in the documentation.

8.2.3 Building and Debugging a Project

To build a project, do the following:

1. Right-click on the workspace name. See Figure 8-7.

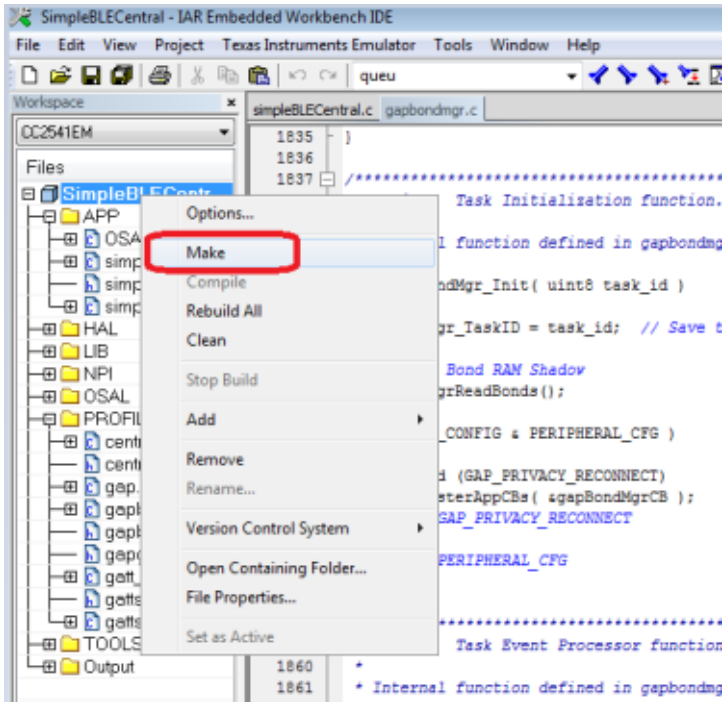


Figure 8-7. Building a Project

2. Click Make or press F7.

NOTE: This action compiles the source code, links the files, and builds the project. Any compiler errors or warnings appear in the Build window.

3. To download the compiled code onto a CC2540/41 device and debug, connect the keyfob using a hardware debugger (such as the CC Debugger included with the CC2540/41DK-MINI development kit) connected to the PC over USB.

4. Find the Debug button on the upper-right side of the IAR window (see Figure 8-8).

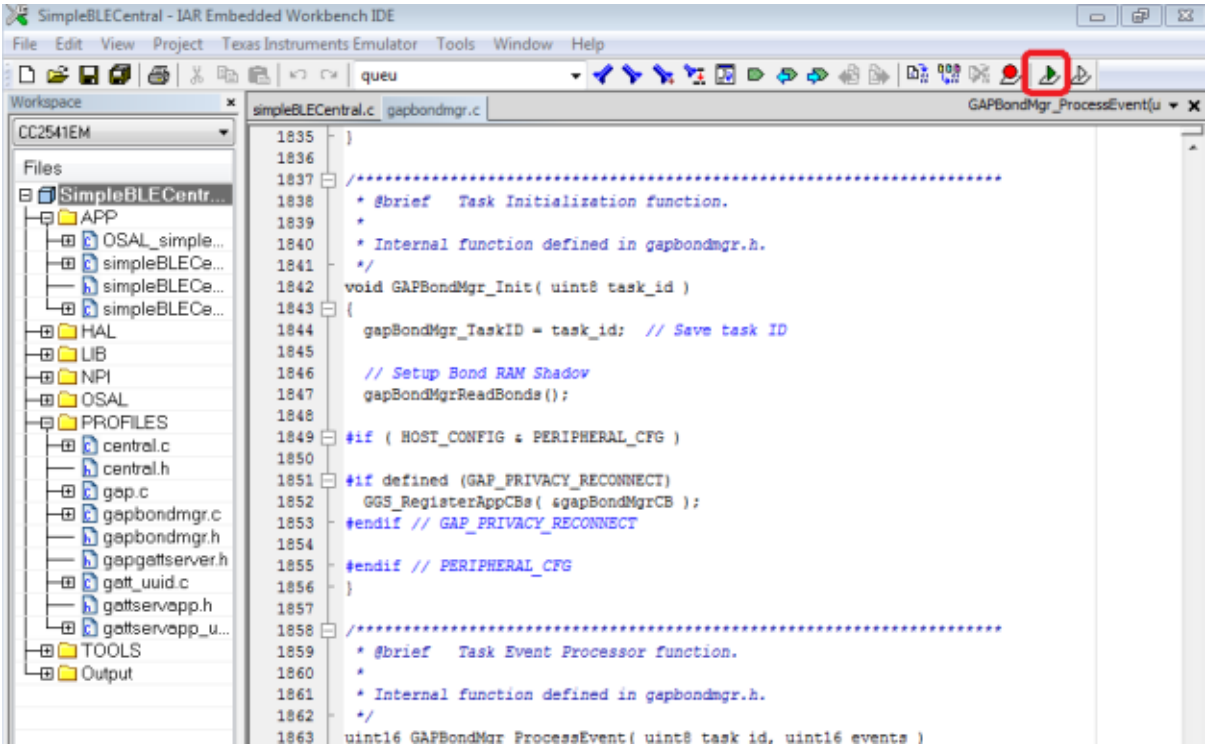


Figure 8-8. Debug Button in IAR

NOTE: If there are multiple debug devices connected, Figure 8-9 appears to select a device

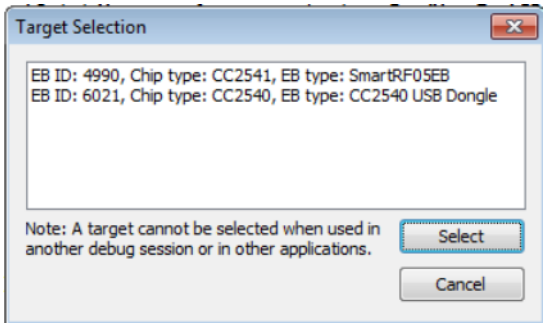


Figure 8-9. Target Selection

5. Select a device.

NOTE: After selecting a device, the code downloads. When the code is downloaded, a toolbar with the debug commands appears in the upper-left corner of the screen.

6. Click the Go button on the toolbar to execute the program (see Figure 8-10).

7. Click the Stop Debugging button to leave debugging mode.
   Figure 8-10 shows both of these buttons:

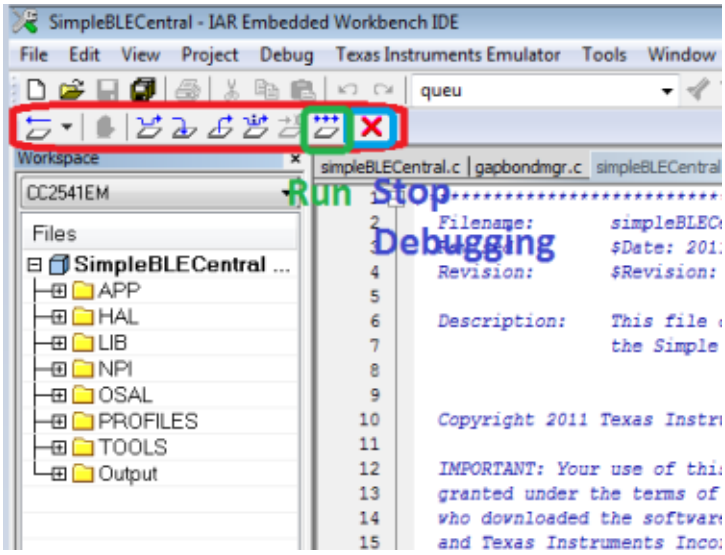


Figure 8-10. IAR Debug Toolbar

8. While the program executes, disconnect the hardware debugger from the CC2540/41.

The debugger runs while the device remains powered.

8.2.4 Linker Map File

After building a project, IAR generates a linker map file that you can find in the Output group in the file list.

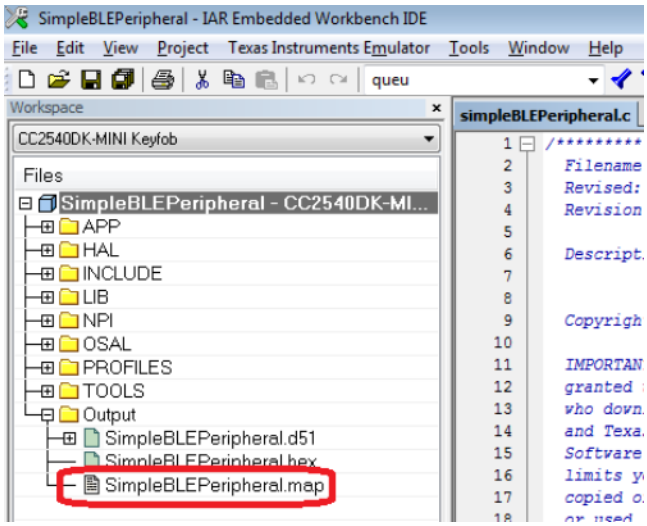


Figure 8-11. Map File in File List

The map file contains low-level information about the build. Lines of text similar to the following are at the end of the map file:

```
118 544 bytes of CODE   memory

35 bytes of DATA memory (+ 73 absolute )

6 242 bytes of XDATA memory

194 bytes of IDATA memory

8 bits  of BIT memory

4 149 bytes of CONST memory

Errors: none
Warnings: none
```

This text shows the total code space (CODE memory) and RAM (XDATA memory) the project uses. Ensure the sum of the CODE memory plus CONST memory does not exceed the maximum flash size of the device (either 128KB or 256KB, depending on the version of the CC2540/41). Ensure the size of the XDATA memory does not exceed 7936 bytes, as the CC2540/41 contains 8KB of SRAM (256 bytes are reserved).

For more specific information, the map file contains a section called MODULE SUMMARY. This section is approximately 200 to 300 lines before the end of the file (the exact location varies from build-to-build). Within this section, you can see the exact amount of flash and memory for every module in the project.

General Information

The release notes also can be found in the installer at: $INSTALL$\README.txt.

9.1 Release Notes History

Texas Instruments, Inc.

CC2540/41 *Bluetooth* low energy Software Development Kit Release Notes

Version 1.4.1 May 18, 2015

Notices:

- This version of the Texas Instruments BLE stack and software is a maintenance update to the v1.4 release. It contains several bug fixes and enhancements.
- The BLE protocol stack, including both the controller and host, was completely retested for v1.4.1.

Changes and minor enhancements:

- All projects have been migrated from IAR v8.20.2 to IAR 9.10.3. To build all projects, upgrade to IAR v9.10.3.
- Smarter handling of connection parameter updates with multiple connections
- GAPRole_SetParameter(GAPROLE_ADVERT_DATA) changes the advertising data
- Allows removal of Service Changed Characteristic
- HAL components set to TRUE if not defined
- Added HCI Vendor Specific Guide revision history
- Several bug fixes

Bug Fixes:

- Fix for RSSI value does not change in V1.4 stack
- Fix for Number HCI Commands parameter not updated in Command Complete Event
- Fixed CC254x UART DMA reception discontinuity
- Fix for updating advertisement data while simultaneously connected as peripheral and advertising
- Fix for filtering duplicate ADV reports even when the filter is FALSE
- Fix for possible race condition T2ISR vs T2E1 on slow wakeups
- Fix for HAL_DMA_CLEAR_IRQ() can be interrupted causing missed ISR cause
- Fixed HCI_LERemoveDeviceFromWhiteList Fails after Scan
- Watchdog Kick Macro Affected by Interrupts
- Fixed HCI_EXT_ResetSystem soft reset to work as expected on CC254x
- Fixed White List Irregularities During Scan / Connect
- Fixed Overlap processing that causes Slave task to last too long for next event setup
- Add BTool Support for new field 'encKeySize' added to GATT_AddService command
- Fixed CC254x UART DMA reception discontinuity
- Fixed CC254x unresponsive when resetting in initiating state
- Fix for after successful reconnect using private non-resolvable address, rebond fails with "Key Req Rejected"

- Fix for CC254x host Bond Manager setParam configuration does not support M/S LinkKey enc exchange
- Fixed TICKSPD, CLKSPD is overwritten on X/HS-OSC change
- Fixed Device Fails to Return to Sleep After Last BLE Task

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. TI recommends using the NV memory sparingly or only when a connection is not active.
- HCI packet size of 128 bytes or more will be disregarded by the stack and no HCI event will be returned.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum: http://www.ti.com/ble-forum

For additional sample applications, guides, and documentation, visit the Texas Instruments *Bluetooth* low energy wiki page at: http://processors.wiki.ti.com/index.php/Category:BluetoothLE

Texas Instruments, Inc.

CC2540/41 *Bluetooth* low energy Software Development Kit Release Notes

Version 1.4.0 November 8, 2013

Notices:

- This version of the Texas Instruments BLE stack and software is a minor update to the v1.3.2 release. It contains some minor bug fixes and a few functional changes.
- The BLE protocol stack, including both the controller and host, was completely retested for v1.4.0.

Changes and Enhancements:

- All projects have been migrated from IAR v8.10.4 to IAR 8.20.2.To build all projects, upgrade to IAR v8.20.2.
- Updated SPI and UART_DMA drivers for improved robustness and throughput.
- Added an overlapped processing feature to improve throughput and reduce power consumption in devices where peak power consumption isn't an issue. Overlapped processing allows the stack to concurrently process while the radio is active. Since the stack is concurrently processing, it is able to insert new data in the Tx buffer during the connection event, causing additional packets to be sent before the end of the event.
- Added a Number of Completed Packets HCI command which offers the possibility of waiting for a certain number of completed packets before reporting to the host. This allows higher throughput when used with overlapped processing.
- Added an HCI Extension command HCI_EXT_DelaySleepCmd which provides the user control of the system initialization sleep delay (wake time from PM3/boot before going back to sleep). The default sleep delay is based on the reference design 32 kHz XOSC stabilization time.
- Added a low duty cycle directed advertising option.
- Added support for deleting a single bond with the GAP_BondSetParam command.
- Decreased CRC calculation time during OAD by using DMA.

Bug Fixes:

- Using a short connection interval and exercising high throughput, there was some loss of packets. This was fixed by adding host to application flow control support.
- Bonding was unstable at short connection intervals. This is now fixed.
- Fixed USB CDC Drivers to work with Windows 8.
- OAD sample project would fail if long connection interval was used. This was fixed by not allowing parameter updates to the central device.
- Fixed linking errors in UBL project.
- Fixed minor issues in sample apps to work with PTS dongle.
- Fixed USB descriptors in HostTestRelease to display correct string.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- HCI packet size of 128 bytes or more will be disregarded by the stack, and as such, no HCI event will be returned.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum: http://e2e.ti.com/support/low_power_rf/f/538.aspx

For additional sample applications, guides, and documentation, visit the Texas Instruments *Bluetooth* low energy wiki page at: http://processors.wiki.ti.com/index.php/Category:BluetoothLE

Version 1.3.2 June 13, 2013

Notices:

- This version of the Texas Instruments BLE stack and software is a minor update to the v1.3.1 release. It contains some minor bug fixes and a few functional changes.
- The BLE protocol stack, including both the controller and host, was completely retested for v1.3.2. The profiles Running Speed and Cadence, Cycling Speed and Cadence, and Glucose were fully tested and passed certification. Other profiles with no code changes since 1.3.1 were sanity tested only.

Changes and Enhancements:

- Added Running Speed and Cadence profile and service. An example application demonstrating running speed and cadence is provided.
- Added Cycling Speed and Cadence profile and service. An example application demonstrating cycling speed and cadence is provided.
- Added delay before performing Connection Parameter changes. Implemented conn_pause_peripheral) and TGAP(conn_pause_central) timers as described in CSA 3 rev 2, Gap Connection Parameters Changes, Section 1.12. Updated HIDAdvRemote, HIDEmuKbd, KeyFob, SensorTag, and SimpleBLEPeripheral applications.
- Update Privacy Flag and Reconnection Address characteristics permissions (Erratum 4202)
- A new Windows USB CDC driver has been included in the installer. This new driver is signed and is functional on Windows 8 systems.

Bug Fixes:

- Some minor updates to glucose sensor and collector were made.
- The gyroscope would draw continuous 6mA when enabled. The updated code now performs a read and turns off the gyro after 60ms.
- The master's host would accept invalid connection parameters requested by the Slave, and would send back the Connection Parameter Update Response with 'parameters accepted'. The host now performs validation on these parameters.
- When coming out of sleep, the HCI_EXT_ExtendRfRangeCmd would override HCI_EXT_SetRxGainCmd setting and set it to default gain. This has been fixed.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- HCI packet size of 128 bytes or more will be disregarded by the stack, and as such, no HCI event will be returned.
- The HAL SPI driver that was implemented since the v1.3 release can sometimes hang, particularly in cases in which power management is used and when there is heavy traffic on the SPI bus.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.3.1 April 18, 2013

Notices:

- This version of the Texas Instruments BLE stack and software is a minor update to the v1.3 release. It contains some minor bug fixes, with no major functional changes. It also contains two additional projects for the CC2541 Advanced Remote Control Kit.
- Since none of the profile source code was significantly changed since the v1.3 release, no additional re-testing of the profiles and sample application were done for v1.3.1. The only exception is the HID-over-GATT profile, which was fully re-tested for this release. The BLE protocol stack, including both the controller and host, was completely retested for v1.3.1.

Major Changes and Enhancements:

- The GAP parameter TGAP_LIM_ADV_TIMEOUT now uses units of seconds instead of milliseconds.
- The HidAdvRemote Project has been added. This implements a full mouse-like pointing functionality using motion and gesture control. The project runs on the CC2541 BLE Advanced Control included as part of the CC2541DK-REMOTE kit. The application implements the HID-over-GATT (HOGP) profile with a report descriptor supporting the keyboard, mouse, and consumer control classes of HID devices.
- The HidAdvRemoteDongle project has been added. This application runs on the CC2540USB dongle, and implements partial functionality of HID-over-GATT (HOGP) host with a fixed report descriptor to match that of the descriptor of the HidAdvRemote Project. This means that the HidAdvRemoteDongle was designed only to work with the HidAdvRemote, and will not be compatible with any other HOGP devices. This project was created to allow users who are using a host device that does not have native Bluetooth Smart Ready support and/or does not have HOGP support to use the BLE Advanced Remote Control with their system.
- For GAP central role applications, the bond manager now properly handles cases in which the peripheral device has erased previously stored bonding information
- A new HCI extension API has been added to allow peripheral/slave devices to temporarily ignore any nonzero slave latency value, and explicitly wake up at every connection event regardless of whether it has any data to send. The prototype for the API function HCI_EXT_SetSlaveLatencyOverrideCmd can be found in hci.h, including the description of the function.

- A new HCI extension API has been added to allow the application layer to get or set a build revision number.

Bug Fixes:

- In some cases L2CAP Peripheral Connection Parameter Update requests failed due to a zero value in the transmitWindowOffset parameter when the connection was initially established. This has been fixed and updates should now work successfully.
- During bonding, connection failures would occasionally occur due to the OSAL Simple NV driver performing a page compaction and halting the CPU for longer than the time required for the link layer to maintain proper connection timing. To prevent this from occurring, the simple NV driver now has any API to force a page compaction if the page is full beyond a specified threshold. The bond manager calls this API every time a connection is terminated to ensure that compaction occurs before the next connection is set up.
- Occasional slave connection failures would previously occur in cases in which the master device sends Update Channel Map requests while a large slave latency value is in use. This has been fixed.
- The SensorTag application now properly supports storage of GATT Client Characteristic Configuration Descriptor values with bonded devices.
- After disabling advertising, the CC254x would unnecessarily wake up for a short period of time 500ms later. This unecessary wake-up has been removed.
- Upon Power-On Reset or after wake-up from PM3, a 400ms delay has been implemented, during which time the CC254x will not go into PM2 sleep. This allows time for the 32kHz crystal to stabilize. Previously, in rare cases with certain hardware configurations the CC254x could have timing issues due to the crystal not having time to stabilize.
- Minor bug fixes to GlucoseSensor and GlucoseCollector projects.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- HCI packet size of 128 bytes or more will be disregarded by the stack, and as such, no HCI event will be returned.
- The HAL SPI driver that was implemented since the v1.3 release can sometimes hang, particularly in cases in which power management is used and when there is heavy traffic on the SPI bus.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.3 Dec 12, 2012

Notices:

- This version of the Texas Instruments BLE stack and software features several changes, enhancements, and bug fixes from v1.2.1. Details of these can be found below.

Changes and Enhancements:

- A new sample project, SensorTag, has been added. This application runs on the CC2541 Sensor Tag board, which is included as part of the CC2541DK-SENSOR development kit. The application includes custom services for an accelerometer, barometer, gyro, humidity sensor, IR temperature sensor, and magnetometer.
- A new Boot Image Manager (BIM) is included. This allows one CC2540 or CC2541 device to contain two separate software images (an "A" image and a "B" image) stored in flash. Upon power-up, the BIM selects which image to boot into. This can be based on criteria such as the state of a GPIO pin, or based on a selection from the previously running application upon reset.

- A new Over-the-air firmware download (OAD) feature is included. The feature allows a peer device (which could be a central BT Smart device such as a smartphone) to push a new firmware image onto a peripheral device and update the firmware. This feature uses the BIM, in which case the downloaded image gets stored in the opposite flash location as the currently running image. For example, if the "A" image is the current image and is used to perform the download, then the downloaded image becomes the "B" image. Upon reset, the "B" image with the updated firmware would be loaded. The OAD feature optionally allows for the firmware image to be signed (using AES). Both the SensorTag and SimpleBLEPeripheral projects include configurations for using the OAD feature. A central "OADManager" application is also included, demonstrating a central implementation for sending a new firmware image to an OAD target device.
- The physical HCI interface used by the network processor (HostTestRelease) has been enhanced to work while power management is enabled on the CC254x device. The UART interface, when using RTS and CTS lines, can be used by an external application processor to wake-up the CC254x network processor. When the network processor has completed all processing, it will go into deep sleep. In addition to UART, an SPI interface has been added as an option for the physical HCI interface. It also supports power management by means of the MRDY and SRDY lines.
- The CC2541 configuration of the KeyFobDemo project has been modified to support the new CC2541 keyfob hardware, contained in the CC2541DK-MINI kit. The accelerometer has been changed, and a TPS62730 DC/DC converter has been added.
- The structure of all projects have been changed to include a Transport Layer ("TL") library and network processor interface "NPI" source code. This new architecture allows for non-network processor applications to have slightly reduced code size by removing unnecessary stack components.
- An API has been provided allowing the device name and appearance characteristics in the GAP service to be modified by the application layer.
- KeyFobDemo project now includes visual feedback from LED to indicate when device has powered up and when device is advertising.
- The HID-over-GATT Profile (HOGP) implementation has been updated to now queue up HID report and send notifications upon reconnection to a HID host.
- A new implementation of the HID service has been included, which supports a combined keyboard, mouse, and consumer class device in its HID report descriptor.
- The API for sending L2CAP Connection Parameter Update Requests from the GAP Peripheral Role Profile has been updated to take both the requested minimum and maximum connection intervals as parameters.
- BTool has been enhanced with a new GATT explorer table, displaying discovered attributes, handles, and values. An XML file is included which allows the user to define descriptions of characteristics based on their UUIDs.
- HCI UART interface baud rate has been changed from 57600 to 115200.

Bug Fixes:

- When power management is used with long connection intervals (>2s), the CC254x remains sleeping properly without unnecessary wake-ups.
- When slave latency is used, peripheral devices now properly wake-up before the next connection event when a data packet is queued
- Various bug fixes on the GlucoseSensor and GlucoseCollector projects to improve compliance with profile and service specifications.
- HID-over-GATT Pofile (HOGP) implementation has been updated to provide better interoperability with HID hosts.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- HCI packet size of 128 bytes or more will be disregarded by the stack, and as such, no HCI event will be returned.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.2.1 Apr 13, 2012

Notices:

- This version of the Texas Instruments BLE stack and software is a minor update to the v1.2 release. It contains some minor enhancements and bug fixes, with no API changes or major functional changes.

Changes and Enhancements:

- When advertising is enabled by calling GAP_MakeDiscoverable, the first advertisement event will now occur within a few milliseconds, rather than waiting for 10 ms.

Bug Fixes:

- The HidEmuKbd project now properly implements the HID Service include of the Battery Service. This bug fix allows for proper interoperability between the CC254x HID Profile and host systems running Windows 8.
- The source code file hal_board_cfg.h has been updated to better support the serial bootloader (SBL) and Universal Bootloader (UBL).
- Scanning in BTool can now be cancelled at any time without hanging or freezing the system.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.2 Feb 13, 2012

Notices:

- This version of the Texas Instruments BLE stack and software includes support for the CC2541, as well as some enhancements and bug fixes. Details of these can be found below. If you have not previously worked with the v1.1b release (which had limited distribution), TI recommends you also read the notes detailing the changes and enhancements from v1.1a to v1.1b in addition to the notes for v1.2.

Changes and Enhancements:

- All projects have been migrated from IAR v7.60 to IAR v8.10.4. In order to build all projects, be sure to upgrade and have IAR v8.10.4. Also, be sure to download and install all of the latest patches from IAR for full CC2540 and CC2541 support.
- Multi-role and combo-role support has been enhanced. The BLE stack can now support simultaneously advertising and/or scanning while in a connection as either a master or a slave. This allows for a central device to perform device discovery while in a connection. All previous rules for multiple simultaneous connections as a central device still apply (see v1.1a release notes).
- New sample projects "SimpleBLEBroadcaster" and "SimpleBLEObserver" have been added, as example projects for pure broadcaster and observer applications with very low code size. The projects make use of new GAP role profiles broadcaster.c and observer.c that are included.
- All projects have a modified architecture from the v1.1, v1.1a, and v1.1b releases. Each project contains a file "buildConfig.cfg" that can be found in the project directory and is included in the IAR project workspace as part of the "TOOLS" group. The settings in this file determine the role of the device in the application. Based on this configuration, different pieces of the BLE stack in object code are linked in, causing the code size to be larger or smaller depending on the roles supported. For example, HostTestRelease by default is now configured to support every single BLE GAP role in a single build, and therefore has a large code size (approx. 165kB). On the other hand, SimpleBLEBroadcaster is configured to only support the GAP broadcaster role, and therefore has a very small code size (approx. 39kB).
- The function GAPRole_SendUpdateParam in peripheral.c has been made public to allow a peripheral application to send an L2CAP connection parameter update request at any time.
- The names and configuration of the BLE stack libraries have changed. Different libraries are used depending on the GAP role (or combination of roles) used by the application. More information can be found in section 3.3.5 of the BLE Software Developer's Guide.
- All library files now support power management. Power management must be enabled by the application by calling osal_pwrmgr_device(PWRMGR_BATTERY);. All sample applications that use power management make this call in the main function.
- All GATT service source code has been cleaned up to make handling of client characteristic configuration descriptors (CCCDs) simpler. All CCCDs are now processing is now handled by GATTServApp and no longer must be handled by the service itself. Examples of this can be found in the included example services such as simpleGATTprofile, Simple Keys service, Accelerometer service, etc...
- The HostTestRelease network processor project now includes HCI Vendor Specific commands for each GATT client sub-procedure, matching the GATT client API. All GATT commands have been added to the "Adv. Commands" tab in BTool. The functions in the BTool GUI "Read / Write" tab now make use of the GATT commands as opposed to ATT commands.
- The old "EmulatedKeyboard" project has been removed and replaced with the new "HIDEmuKbd" project. The new project performs the same functions as the old one, but is now based on the "HID over GATT Profile" v1.0 specification (HOGP_SPEC_V10) that has been adopted by the Bluetooth SIG. The HID profile functionality has been implemented in a OSAL task that runs separate from the application to allow for easy portability to other HID projects. More details on the new application can be found in the BLE Sample Application Guide included as part of the release. The following additional new services / profiles have been included to fully support the HOGP specification:
  - HID Service v1.0 (HIDS_SPEC_V10)
  - Scan Parameters Profile v1.0 (ScPP_SPEC_V10)
  - Scan Parameters Service v1.0 (ScPS_SPEC_V10)

  - Device Information Service v1.1 (DIS_SPEC_V11r00)
  - Battery Service v1.0 (BAS_SPEC_V10)
- The KeyFobDemo project has been updated to use the adopted battery service. The custom battery service that was used in previous released has been removed.
- The TimeApp project has been updated to include support for the Phone Alert Status Profile (PASP_SPEC_V10) in the Client role.
- Support for "Production Test Mode" has been added, allowing a BLE application in a "single-chip" configuration to temporarily expose the HCI over the UART interface when triggered externally to do so (e.g. hold a GPIO pin low during power up). This allows the device to be connected to a Bluetooth tester in order to run direct test mode (DTM) commands on a production line using the final release firmware, while leaving the UART GPIO pins available for the application to use at all other times
- A Universal Boot Loader (UBL) using the USB Mass Storage Device (USB-MSD) class has been added along with a Serial Boot Loader (SBL). The HostTestRelease project includes configurations with examples of both boot loaders. The SBL project is included with the installer. More information on the UBL can be found in the following document: C:\Texas Instruments\BLE-CC254x-1.2\Documents\ Universal Boot Loader for SOC-8051 by USB-MSD Developer's Guide.pdf
- HCI extension command HCI_EXT_MapPmIoPortCmd added to support toggling of a GPIO line as CC254x device goes in and out of sleep. This command can be used to automatically control the bypass line of the TPS62730 DC/DC converter for reducing power consumption in an optimized manner.
- A slave device will now dynamically widen it's Rx window when a previous connection event was missed. This improves connection stability by accounting for additional clock drift that may have occurred since the last successful connection event.
- The application now has the capability to change the permissions of the device name in the GAP service by calling GGS_SetParameter and changing the value of the parameter GGS_W_PERMIT_DEVICE_NAME_ATT. The application can also receive a callback when a client device writes a new name to the device. The application registers the callback by calling GGS_RegisterAppCBs. The prototype for GGS_RegisterAppCBs can be found in gapgattserver.h.

Bug Fixes:

- Duplicate filtering now works with combination states.
- Various minor application / profile bug fixes.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.1b Nov 30, 2011

Notices:

- This version of the Texas Instruments BLE stack and software includes support for the CC2541, as well as some minor enhancements and bug fixes. Details of these can be found below. The general software architecture remains the same as in the v1.1 and v1.1a releases.

Changes and Enhancements:

- BLE stack libraries for the CC2541 are included.
- All BLE libraries are renamed and now indicate whether they are used for CC2540 or CC2541.
- For each project and configuration, new IAR projects are included for use with the CC2541. The only

exception is that any project/configuration that uses the USB interface has not been replicated for the CC2541, as it does not have an on-chip hardware USB interface.

- Link-layer processing has been optimized to provide for reduced power consumption during connection events and advertising events.
- SimpleBLEPeripheral and SimpleBLECentral now use the HCI_EXT_ClkDivOnHaltCmd, which reduces the current level while the CC2540/41 radio is active.
- The bond manager has been updated to allow peripheral devices to properly pair, bond, and resolve the address of central devices that use the private resolvable address type.
- New command HCI_EXT_SetMaxDtmTxPowerCmd included, which allows the maximum Tx power level to be set. This is useful when using Direct Test Mode (DTM), in that the Tx power level will be set to the maximum value set by the HCI_EXT_SetMaxDtmTxPowerCmd command, which may be less than +4dBm for the CC2540 and less than 0dBm for the CC2541. The function prototype can be found in hci.h.

Bug Fixes:

- The command HCI_EXT_SetTxPowerCmd is now properly working.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- Duplicate filtering does not work when scan is used in combination with a connection.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.1a Aug 10, 2011

Changes and Enhancements:

- The thermometer profile sample application has been updated to support stored measurements. The TI_BLE_Sample_Applications_Guide has been updated to match these changes.

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- Duplicate filtering does not work when scan is used in combination with a connection.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum: http://e2e.ti.com/support/low_power_rf/f/538.aspx

---

Version 1.1 July 13, 2011

Notices:

- This version of the Texas Instruments BLE stack and software features several changes, enhancements, and bug fixes from v1.0. Details of these can be found below.

Changes and Enhancements:

- All projects have been migrated from IAR v7.51A to IAR v.7.60. In order to build all projects, be sure to upgrade and have IAR v7.60. Also, be sure to download and install all of the latest patches from IAR for full CC2540 support.
- The stack now supports up to 3 simultaneous connection as a central / master device, with a few constraints:
  - All connection intervals must be a multiple of the minimum connection interval (i.e. the minimum connection interval is the greatest common denominator of all connection intervals).
  - The minimum connection interval allowed is 25ms when using more than one connection.
  - When more than one connection is active, only one data packet per connection event will be allowed in each direction.
  - Scanning is not supported while in a connection. The consequences of this is that device discovery is not possible while in a connection. Therefore, to discover and connect to multiple devices, the device discovery must occur before the first connection is established.
- Several new sample projects are included, with examples of many different BLE applications / profiles. Full details on the sample applications can be found in the BLE Sample Applications Guide, which can be accessed from the Windows Start Menu. These sample applications implement various functions. Some are based on adopted Bluetooth specifications, some are based on draft specifications, and others are custom designed by Texas Instruments. These projects should serve as good examples for various other BLE applications.
- The following updates have been made to BTool (more information on these updates can be found in the CC2540DK-MINI User Guide which can be downloaded here: SWRU270):
  - Improved GUI and robustness.
  - All functions on the GUI been updated to handle multiple simultaneous connections.
  - A new "Pairing / Bonding" tab has been added, allowing link encryption and authentication, passkey entry, and saving / loading of long-term key data (from bonding) to file.
  - Ability to "Cancel" a link establishment while the dongle is initiating.

- The following additional new controller stack features are included in this release:
  - Support for multiple simultaneous connections as a master (details above)
  - HCI Vendor Specific function HCI_EXT_SetSCACmd allows you to specify the exact sleep clock accuracy as any value from 0 to 500 PPM, in order to support any crystal accuracy with optimal power consumption. This feature is only available for slave / peripheral applications.
  - HCI Vendor Specific function HCI_EXT_SetMaxDtmTxPowerCmd allows you to set the maximum transmit output power for Direct Test Mode. This allows you to perform use the LE Transmitter Test command with power levels less than +4dBm.
  - A master device can now advertise while in a connection.
  - New production test mode (PTM) has been added allowing the CC2540 to run Direct Test Mode (DTM) while connected to a tester using a "single-chip" BLE library.
  - The controller now uses DMA to more efficiently encrypt and decrypt packets. All BLE projects must now define HAL_AES_DMA=TRUE in the preprocessor settings when using the v1.1 libraries.
- The following additional new host stack features are included in this release:
  - A new GAP central role profile for single-chip embedded central applications is included, with functions similar to the GAP peripheral role profile. The SimpleBLECentral project serves as an example of an application making use of the central role profile.
  - The GAP peripheral role has been optimized to significantly improve power consumption while advertising with small amounts of data by no longer transmitting non-significant bytes from in the advertisement and scan response data.
- The following additional new application / profile features are included in this release:
  - The GAP peripheral bond manager has been replaced with a general GAP bond manager, capable of managing bond data for both peripheral and central role devices. The gap peripheral bond manager has been included for legacy support; however it is recommend to switch to the general GAP bond manager (gapbondmgr.c/h).
  - The bond manager also now manages the storage of client characteristic configurations for each bond as per the *Bluetooth* 4.0 spec.
  - The simple GATT profile has a new fifth characteristic. This characteristic is 5 bytes long, and has readable permissions only while in an authenticated connection. It should serve as a reference for development of other profiles which require an encrypted link.
  - All GATT profiles have been updated to properly handle client characteristic configurations for both single and multiple connections. Characteristic configurations now get reset to zero (notifications / indications off) after a connection is terminated, and the bond manager now stores client characteristic configurations for bonded devices so that they are remembered for next time when the device reconnects.
  - Added linker configuration file for support of 128kB flash versions of the CC2540. An example is included in the SimpleBLEPeripheral project.
  - The SimpleBLEPeripheral project "CC2540 Slave" configuration has been updated to better support the SmartRF05EB + CC2540EM hardware platform, making use of the LCD display.

Bug Fixes:

- The following bugs have been fixed in the controller stack:
  - Scanning now working for master devices with power savings enabled.
  - RSSI reads no longer require a data packet to update.
  - Improved stability when using very high slave latency setting
  - HCI LE direct test modes now working properly.
  - HCI Read Local Supported Features now returns the proper value.
  - Use of two advertising channels now works.
  - When connecting to a device on the whitelist, the correct peer device address is returned to the host.
- The following bugs have been fixed in the host stack:
  - Pairing no longer fails when either device is using a static, private resolvable, or private non-

resolvable address.

- The following bugs have been fixed in the profiles and applications:
  - Reading of RSSI with peripheral role profile now working.
  - Peripheral role profile now allows all legal whitelist modes.
  - Can now connect with short connection intervals (such as 7.5 ms), since bond manager now reads data from NV memory upon initialization rather than immediately after a connection is established. Pairing still may not be stable when using the bond manager with very short connection intervals (for reason noted in the following Known Issues section)

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.
- Duplicate filtering does not work when scan is used in combination with a connection.

For technical support, visit the Texas Instruments *Bluetooth* low energy E2E Forum:
http://e2e.ti.com/support/low_power_rf/f/538.aspx

Version 1.0 October 7, 2010

Notices:

- The Texas Instruments *Bluetooth* low energy (BLE) software development kit includes all necessary software to get started on the development of single-mode BLE applications using the CC2540 system-on-chip. It includes object code with the BLE protocol stack, a sample project and applications with source code, and BTool, a Windows PC application for testing BLE applications. In addition to the software, the kit contains documentation, including a developer's guide and BLE API guide.
- For complete information on the BLE software development kit, please read the developer's guide:
  - BLE Software Developer's Guide: <Install Directory>\Documents\TI_BLE_Software_Developer's_Guide.pdf, (Also can be accessed through the Windows Start Menu)
- The following additional documentation is included:
  - BLE API Guide:<Install Directory>\Documents\BLE_API_Guide_main.htm
  - Vendor Specific HCI Guide:<Install Directory>\Documents\TI_BLE_Vendor_Specific_HCI_Guide.pdf
  - HAL Drive API Guide:<Install Directory>\Documents\hal\HAL Driver API.pdf
  - OSAL API Guide:<Install Directory>\Documents\osal\OSAL API.pdf
- The following software projects are included, all built using IAR Embedded Workbench v7.51A:
  - SimpleBLEPeripheral:<Install Directory>\Projects\ble\SimpleBLEPeripheral\CC2540DB \SimpleBLEPeripheral.eww
  - HostTestRelease:<Install Directory>\Projects\ble\HostTestApp\CC2540\HostTestRelease.eww
- The following Windows PC application is included:
  - BTool:<Install Directory>\Projects\BTool\BTool.exe (Also can be accessed through the Windows Start Menu)
- Changes:
- Initial Release

Bug Fixes:

- Initial Release

Known Issues:

- Use of the NV memory (to save application data or BLE Host bonding information) during a BLE connection may cause an unexpected disconnect. The likelihood of this happening increases with frequent usage, especially when using short connection intervals. The cause is related to the NV wear algorithm which at some point may cause an NV page erase which can disrupt system real-time processing. It is therefore recommended that the NV memory be used sparingly, or only when a connection is not active.

9.2 Document History

Revision	Date	Description/Changes
1	10/7/2010	Initial release
1.1	7/13/2011	Updated for BLEv1.1 software release
1.1b	11/15/2011	Updated for BLEv1.1b software release
1.2	2/13/2012	Updated for BLEv1.2 software release
1.3.1	4/13/2012	Updated for BLEv1.2.1 software release
1.3	12/19/2012	Updated for BLEv1.3 software release
1.3.1	4/5/2013	Updated for BLEv1.3.1 software release
1.3.2	6/12/2013	Updated for BLEv1.3.2 software release
1.4.0	9/12/2013	Updated for BLEv1.4.0 software release
1.4.1	5/15/2015	Updated for BLEv1.4.1 software release

GAP API

A.1 Commands

This section details the GAP commands from gap.h which the application uses. Other GAP commands are abstracted through the GAPRole or the GAPBondMgr.

uint16 GAP_GetParamValue (gapParamIDs_t paramID)

Get a GAP parameter.

Parameters paramID – parameter ID (Section A.2)

Returns GAP Parameter Value if successful

0xFFFF if paramID invalid

bStatus_t GAP_SetParamValue (gapParamIDs_t paramID, uint16 paramValue)

Set a GAP parameter.

Parameters paramID – parameter ID (Section A.2)

paramValue – new param value

Returns SUCCESS

INVALIDPARAMETER: paramID is invalid

A.2 Configurable Parameters

ParamID	Description
TGAP_GEN_DISC_ADV_MIN	Minimum time (ms) to remain advertising in Discovery mode. Setting this to 0 turns off this time-out, thus advertising infinitely. Default is 0.
TGAP_LIM_ADV_TIMEOUT	Maximum time (sec) to remain advertising in Limited Discovery mode. Default is 180 seconds.
TGAP_GEN_DISC_SCAN	Minimum time (ms) to perform scanning for General Discovery
TGAP_LIM_DISC_SCAN	Minimum time (ms) to perform scanning for Limited Discovery
TGAP_CONN_EST_ADV_TIMEOUT	Advertising time-out (ms) when performing Connection Establishment
TGAP_CONN_PARAM_TIMEOUT	Time-out (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)

ParamID	Description
TGAP_CONN_ADV_INT_MAX	Maximum advertising interval when in connectable mode (n × 0.625 ms)
TGAP_CONN_SCAN_INT	Scan interval used during Link Layer Initiating state, when in Connectable mode (n × 0.625 ms)
TGAP_CONN_SCAN_WIND	Scan window used during Link Layer Initiating state, when in Connectable mode (n × 0.625 ms)
TGAP_CONN_HIGH_SCAN_INT	Scan interval used during Link Layer Initiating state, when in Connectable mode, high duty scan cycle scan parameters (n × 0.625 ms)
TGAP_CONN_HIGH_SCAN_WIND	Scan window used during Link Layer Initiating state, when in Connectable mode, high duty scan cycle scan parameters (n × 0.625 ms)
TGAP_GEN_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in General Discovery procedure (n × 0.625 ms).
TGAP_GEN_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in General Discovery procedure (n × 0.625 ms)
TGAP_LIM_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in Limited Discovery procedure (n × 0.625 ms)
TGAP_LIM_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in Limited Discovery procedure (n × 0.625 ms)
TGAP_CONN_EST_INT_MIN	Minimum Link Layer connection interval, when using Connection Establishment procedure (n × 1.25 ms)
TGAP_CONN_EST_INT_MAX	Maximum Link Layer connection interval, when using Connection Establishment procedure (n × 1.25 ms)
TGAP_CONN_EST_SCAN_INT	Scan interval used during Link Layer Initiating state, when using Connection Establishment procedure (n × 0.625 ms)
TGAP_CONN_EST_SCAN_WIND	Scan window used during Link Layer Initiating state, when using Connection Establishment procedure (n × 0.625 ms)
TGAP_CONN_EST_SUPERV_TIMEOUT	Link Layer connection supervision timeout, when using Connection Establishment procedure (n × 10 ms)
TGAP_CONN_EST_LATENCY	Link Layer connection slave latency, when using Connection Establishment proc (in number of connection events)
TGAP_CONN_EST_MIN_CE_LEN	Local informational parameter about minimum length of connection needed, when using Connection Establishment proc (n × 0.625 mSec)
TGAP_CONN_EST_MAX_CE_LEN	Local informational parameter about maximum length 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 BLE Time-out. 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 application. Default –127.

ParamID	Description
TGAP_REJECT_CONN_PARAMS	Whether or not to reject Connection Parameter Update Request received on Central device. Default FALSE.

A.3 Events

This section details the events relating to the GAP layer that can return to the application from the BLE stack. Some of these events pass directly to the application and some are handled by the GAPRole or GAPBondMgr layers. Whether they are handled by the GAPRole or GAPBondmgr layers, they pass as a GAP_MSG_EVENT with a 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 headers and the associated events. See gap.h for other definitions in these events.

- GAP_DEVICE_INIT_DONE_EVENT: Sent when the Device Initialization completes.

```
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 completes.

```
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 is 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 ends.

```
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 completes.

```
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 terminates.

```
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 changes.

```
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 signature counter of the device updates.

```
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 completes.

```
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 process completes.

```
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;
```

GAPRole Peripheral Role API

B.1 Commands

bStatus_t GAPRole_SetParameter(uint16 param, uint8 len, void *pValue)

Set a GAP Role parameter.

Parameters

param – Profile parameter ID (see Section B.2)

len – length of data to write

pValue – pointer to value to set parameter. This pointer depends on the parameter ID and will be cast to the appropriate data type.

Returns

SUCCESS

INVALIDPARAMETER: param was not valid.

bleInvalidRange: len is not valid for the given param.

blePending: previous param update has not been completed

bleIncorrectMode: cannot start connectable advertising because nonconnectable advertising is enabled

bStatus_t GAPRole_GetParameter(uint16 param, void *pValue)

Set a GAP Role parameter.

Parameters

param – Profile parameter ID (Section B.2)

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

INVALIDPARAMETER: param was not valid

bStatus_t GAPRole_StartDevice(gapRolesCBs_t *pAppCallbacks)

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

Parameters

pAppCallbacks – pointer to application callbacks (Section B.3)

Returns SUCCESS

bleAlreadyInRequestedMode: device was already initialized

bStatus_t GAPRole_TerminateConnection(void)

Terminates an existing connection.

Returns

SUCCESS: connection termination process has started.

bleIncorrectMode: there is no active connection.

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE: disconnect is in progress.

bStatus_t GAPRole_SendUpdateParam(uint16 minConnInterval, uint16 maxConnInterval, uint16 latency, uint16 connTimeout, uint8 handleFailure)

Update the parameters of an existing connection.

Parameters

ConnInterval – the new connection interval

latency – the new slave latency

connTimeout – the new time-out value

handleFailure – 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: parameter update process has started

BleNotConnected: there is no connection so cannot update parameters

B.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPROLE_PROFILEROLE	R	uint8	GAP profile role (peripheral)
GAPROLE_IRK	R/W	uint8[16]	Identity resolving key. Default is all 0s, which means the IRK will be randomly generated.
GAPROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0s, 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 and disable advertising. Default is TRUE = enabled.
GAPROLE_ADVERT_OFF_TIME	R/W	uint16	How long to wait to restart advertising after advertising stops (in ms).
GAPROLE_ADVERT_DATA	R/W	<uint8[32]	Advertisement data. Default is 02:01:01. This third byte sets limited / general advertising.
GAPROLE_SCAN_RSP_DATA	R/W	<uint8[32]	Scan Response data. Default is all 0s.
GAPROLE_ADV_EVENT_TYPE	R/W	uint8	Advertisement type. Default is GAP_ADTYPE_IND (from gap.h).
GAPROLE_ADV_DIRECT_TYPE	R/W	uint8	Direct advertisement type. Default is ADDRTYPE_PUBLIC (from gap.h).
GAPROLE_ADV_DIRECT_ADDR	R/W	uint8[6]	Direct advertisement address. Default is 0.
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_RSSI_READ_RATE	R/W	uint16	How often to read RSSI during a connection. Default is 0 = OFF.
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 parameter 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 parameter update. Range: 0 to 499. Default is 0.

ParamID	R/W	Size	Description
GAPROLE_TIMEOUT_MULTIPLIER	R/W	uint16	Supervision timeout to use for a parameter 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 parameter update request.
GAPROLE_STATE	R	uint8	Gap peripheral role state (enumerated in gaprole_States_t in peripheral.h)

B.3 Callbacks

These callbacks are functions whose pointers are passed from the application to the GAPRole so the GAPRole can return events to the application. They are passed as the following:

```
typedef struct
{
  gapRolesStateNotify_t    pfnStateChange;  //!< Whenever the device changes state
  gapRolesRssiRead_t       pfnRssiRead;     //!< When a valid RSSI is read from controller
} gapRolesCBs_t;
```

See the SimpleBLEPeripheral application for an example.

B.3.1 State Change Callback (pfnStateChange)

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

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

The GAPRole states (newState) are the following:

- GAPROLE_INIT //!< Waiting to be started
- GAPROLE_STARTED //!< Started but not advertising
- GAPROLE_ADVERTISING_NONCONN //!< Currently using nonconnectable Advertising
- GAPROLE_WAITING //!< Device is started but not advertising; it is in waiting period before advertising again
- GAPROLE_WAITING_AFTER_TIMEOUT //!< Device just timed out from a connection but is not yet advertising; it 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

B.3.2 RSSI Callback (pfnRssiRead)

When enabled, this function reports the RSSI to the application at a rate set by the GAPROLE_RSSI_READ_RATE GAPRole parameter. Setting this parameter to 0 disables the RSSI reporting. This function is defined as follows:

```
typedef void (*gapRolesRssiRead_t)(int8 newRSSI);
```

This function passes a signed 1-byte value (newRSSI) of the last reported RSSI to the application.

GAPRole Central Role API

C.1 Commands

bStatus_t GAPCentralRole_StartDevice(gapCentralRoleCB_t *pAppCallbacks)

Start the device in Central role. This function is typically called once during system startup.

Parameters pAppCallbacks – pointer to application callbacks

Returns SUCCESS

bleAlreadyInRequestedMode: Device already started.

bStatus_t GAPCentralRole_SetParameter(uint16 param, uint8 len, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID (see Section C.2)

len – length of data to write

pValue – pointer to value to set parameter. This is dependent on the parameter ID and is cast to the appropriate data type.

Returns SUCCESS

INVALIDPARAMETER: param was not valid

bleInvalidRange: len is invalid for the given param

bStatus_t GAPCentralRole_GetParameter (uint16 param, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID (see Section C.2)

pValue – pointer to buffer to contain the read data

Returns SUCCESS

INVALIDPARAMETER: param was not valid

bStatus_t GAPCentralRole_TerminateLink (uint16 connHandle);

Terminates an existing connection.

Parameters connHandle - connection handle of link to terminate or...

0xFFFE: cancel the current link establishment request or...

0xFFFF: terminate all links

Returns SUCCESS: termination has started

bleIncorrectMode: there is no active connection

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE: terminate procedure already started

bStatus_t GAPCentralRole_EstablishLink(uint8 highDutyCycle, uint8 whiteList, uint8 addrTypePeer, uint8 *peerAddr)

Establishes 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: link establishment has started

bleIncorrectMode: invalid profile role.

bleNotReady: a scan is in progress.

bleAlreadyInRequestedMode: cannot process now.

bleNoResources: too many links.

bStatus_t GAPCentralRole_UpdateLink(uint16 connHandle, uint16 connIntervalMin, uint16 connIntervalMax, uint16 connLatency, uint16 connTimeout)

Update the link connection parameters.

Parameters

connHandle - connection handle

connIntervalMin - minimum connection interval in 1.25 ms units

connIntervalMax - maximum connection interval in 1.25 ms units

connLatency - number of LL latency connection events

connTimeout - connection timeout in 10 ms units

Returns

SUCCESS: parameter update has started

bleNotConnected: No connection to update.

INVALIDPARAMETER: connection parameters are invalid

LL_STATUS_ERROR_ILLEGAL_PARAM_COMBINATION: connection parameters do not meet BLE spec requirements: STO > (1 + Slave Latency) × (Connection Interval × 2)

LL_STATUS_ERROR_INACTIVE_CONNECTION: connHandle is not active

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE: there is already a param update in process

LL_STATUS_ERROR_UNACCEPTABLE_CONN_INTERVAL: connection interval wont work because it is not a multiple or 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 mode, uint8 activeScan, uint8 whiteList)

Start a device discovery scan.

Parameters

mode - discovery mode

activeScan - TRUE to perform active scan

whiteList - TRUE to only scan for devices in the white list

Returns

SUCCESS: device discovery has started

bleAlreadyInRequestedMode: Device discovery already started.

bleMemAllocError: not enough memory to allocate device discovery structure.

LL_STATUS_ERROR_BAD_PARAMETER: bad parameter

LL_STATUS_ERROR_COMMAND_DISALLOWED

bStatus_t GAPCentralRole_CancelDiscovery(void)

Cancel a device discovery scan.

Parameters None

Returns SUCCESS: cancelling of device discovery has started

bleIncorrectMode: Not in discovery mode.

bStatus_t GAPCentralRole_StartRssi(uint16 connHandle, uint16 period)

Start periodic RSSI reads on a link.

Parameters connHandle - connection handle of link

period - RSSI read period in ms

Returns SUCCESS: RSSI calculation has started

bleIncorrectMode: No active link.

bleNoResources: No resources for allocation.

bStatus_t GAPCentralRole_CancelRssi(uint16 connHandle)

Cancel periodic RSSI reads on a link.

Parameters connHandle - connection handle of link

Returns SUCCESS

bleIncorrectMode: No active link.

C.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPCENTRALROLE_IRK	R/W	uint8[16]	Identity resolving key. Default is all 0, which means the IRK are randomly generated.
GAPCENTRALROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0, which means the SRK are randomly generated.
GAPCENTRALROLE_SIGNCOUNTER	R/W	uint32	Sign counter.
GAPCENTRALROLE_BD_ADDR	R	uint8[6]	Device address read from controller. This can be set with the HCI_EXT_SetBDADDRCmd().
GAPCENTRALROLE_MAX_SCAN_RES	R/W	uint8	Maximum number of discover scan results to receive. Default is 8, 0 is unlimited.

C.3 Callbacks

These callbacks are functions whose pointers are passed from the application to the GAPRole so that the GAPRole can return events to the application. They are passed as the following:

```
typedef struct
{
  pfnGapCentralRoleRssiCB_t   rssiCB    //!< RSSI callback.
  pfnGapCentralRoleEventCB_t  eventCB;  //!< Event callback.
} gapCentralRoleCB_t;
```

See the SimpleBLECentral application for an example.

C.3.1 RSSI Callback (rssiCB)

This function reports the RSSI to the application as a result of the GAPCentralRole_StartRssi() command. This function is the following type:

```
typedef void (*pfnGapCentralRoleRssiCB_t)
(
  uint16 connHandle,                  //!< Connection handle of the current RSSI.
  int8  rssi                           //!< New RSSI value.
);
```

This function passes a signed one byte value (newRSSI) of the last reported RSSI to the application for a given connection handle (connHandle).

C.3.2 Central Event Callback (eventCB)

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

```
typedef uint8 (*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 application deallocates. By default, TRUE is always returned. If the event message is to be processed by the application at a later time (not just in the callback context), FALSE should be returned

The possible GAPRole central states are in the following list. See Section A.3 for more information on these events:

- GAP_DEVICE_INIT_DONE_EVENT
- GAP_DEVICE_DISCOVERY_EVENT
- GAP_LINK_ESTABLISHED_EVENT
- GAP_LINK_TERMINATED_EVENT
- GAP_LINK_PARAM_UPDATE_EVENT
- GAP_DEVICE_INFO_EVENT

GATT/ATT API

This section describes the API of the GATT and ATT layers. The sections are combined because the procedure is to send GATT commands and receive ATT events in Section 5.4.3.1. The return values for the commands in this section are in Section D.3. The possible return values are similar for these commands so refer to Section D.3.

D.1 Server Commands

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

Indicates a characteristic value to a client and expect an acknowledgment. Memory must be allocated or freed based on the results of this command. See Section 7.6for more information.

Parameters

connHandle: connection to use

pInd: pointer to indication to be sent

authenticated: whether an authenticated link is required

taskId: task to be notified of acknowledgment

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 acknowledgment. It is only at this point that this subprocedure is considered complete.

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

Notifies a characteristic value to a client. Note that memory must be allocated / freed based on the results of this command. See Section 7.6 for more information.

Parameters

connHandle: connection to use

pNoti: pointer to notification to be sent

authenticated: whether an authenticated link is required

Corresponding Events

If the return status is SUCCESS, the notification has been successfully queued for transmission.

D.2 Client Commands

bStatus_t GATT_InitClient(void)

Initialize the GATT client in the BLE Stack.

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

bStatus_t GATT_RegisterForInd (uint8 taskId)

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

Parameters taskId: task which to forward indications or notifications

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

bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId)

Used by a client to discover all primary services on a server.

Parameters connHandle: connection to use

taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_GRP_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_BY_GRP_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_DiscPrimaryServiceByUUID(uint16 connHandle, uint8 *pValue, uint8 len, uint8 taskId)

Used by a client to discover a specific primary service on a server when only the Service UUID is known.

Parameters connHandle: connection to use

pValue: pointer to value (UUID) 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 receives multiple GATT_MSG_EVENT messages with type ATT_FIND_BY_TYPE_VALUE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_FIND_BY_TYPE_VALUE_RSP (with bleProcedureComplete or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_FindIncludedServices(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find included services with a primary service definition on a server.

Parameters

connHandle: connection to use

startHandle: start handle of primary service to search

endHandle: end handle of primary service to search

taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_FIND_BY_TYPE_VALUE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_FIND_BY_TYPE_VALUE_RSP (with bleProcedureComplete or bleTimeout status) or the calling application task recieves the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GAPRole_GetParameter(uint16 param, void *pValue)

Get a GAP Role parameter.

Parameters

param – Profile parameter ID (See Section F.2)

pValue – pointer to a location to get the value. This pointer depends on the param ID and is cast to the appropriate data type.

Returns

SUCCESS

INVALIDPARAMETER: param was not valid

bStatus_t GATT_DiscAllChars(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find all the characteristic declarations within a service when the handle range of the service is known.

Parameters

connHandle: connection to use

startHandle: start handle of service to search

endHandle: end handle of service to search

taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_DiscCharsByUUID(uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

Used by a client to discover service characteristics on a server when the service handle range and characteristic UUID is known.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent, including start and end handles of service and UUID of characteristic value for which to search.

taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) orthe calling application task recieves the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_DiscAllCharDescs (uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

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 receives 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 the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

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

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 receives 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 the calling application receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_ReadUsingCharUUID (uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

Used to read a Characteristic Value from a server when the client only knows the characteristic UUID and does not know the handle of the characteristic.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

Notes

If the return status is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_BY_TYPE_RSP (with SUCCESS or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

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

Used to read a Characteristic Value from a server when the client knows the Characteristic Value Handle and the length of the Characteristic Value is longer than can be sent in a single Read Response Attribute Protocol message.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

Notes

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_BLOB_RSP (with bleProcedureComplete or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_ReadMultiCharValues (uint16 connHandle, attReadMultiReq_t *pReq, uint8 taskId)

Used to read multiple Characteristic Values from a server when the client knows the Characteristic Value Handles.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

Notes

If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_MULTI_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_MULTI_RSP (with SUCCESS or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_WriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client.

Parameters

connHandle: connection to use

pReq: pointer to command to be sent

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, the write fails and the server generates no error.

bStatus_t GATT_SignedWriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the ATT Bearer is not encrypted. This 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 is sent to the calling application task for this sub-procedure. If the authenticated Characteristic Value is the wrong size or has an invalid value as defined by the profile or the signed value fails to authenticate the client, the write fails and no error is generated by the server.

bStatus_t GATT_WriteCharValue (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

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

Parameters connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

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

bStatus_t GATT_ReliableWrites (uint16 connHandle, attPrepareWriteReq_t *pReq, uint8 numReqs, uint8 flags, uint8 taskId)

Used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle, and assurance is required that the correct Characteristic Value is going to be written by transferring the Characteristic Value to be written in both directions before the write is performed.

Parameters connHandle: connection to use

pReq: pointer to requests to be sent (must be allocated)

numReqs - number of requests in pReq

flags - execute write request flags

taskId: task to be notified of response

Notes If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status), or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status) .

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

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration'.

Parameters connHandle: connection to use

pReq: pointer to request to be sent

taskId: task to be notified of response

Notes If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

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

Used to read a characteristic descriptor from a server when the client knows the characteristic descriptor declaration's Attribute handle and the length of the characteristic descriptor declaration is longer than can send 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 from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or the calling application task receives the ATT_ERROR_RSP (with SUCCESS status).

bStatus_t GATT_WriteCharDesc (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

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)

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 receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This sub-procedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status), or the calling application task recieves the ATT_ERROR_RSP (with SUCCESS status).

D.3 Return Values

- SUCCESS (0x00): Command was executed as expected. See the command API for corresponding events to expect.
- INVALIDPARAMETER (0x02): Invalid connection handle or request field
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): The attribute requires authentication
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The key size for encrypting is insufficient
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): The attribute requires encryption
- MSG_BUFFER_NOT_AVAIL (0x04): An HCI buffer is unavailable. Retry later.
- bleNotConnected (0x14): The device is unconnected.
- blePending (0x17):
  - When returned to a client function, a response is pending with the server or the GATT sub-procedure is in progress.
  - When returned to server function, confirmation from a client is pending.
- bleTimeout (0x16): The previous transaction timed out. No ATT or GATT messages can send until reconnected.
- bleMemAllocError (0x13): A memory allocation error occurred
- bleLinkEncrypted (0x19): The link is encrypted. Do not send an attribute PDU includes an authentication signature on an encrypted link

D.4 Events

Events are received from the BLE stack in the application as a GATT_MSG_EVENT stack message sent as an OSAL message. The events are received as the following structure, where the method signifies the ATT event and the message is a union of the ATT events:

```
typedef struct
{
  osal_event_hdr_t hdr; //!< GATT_MSG_EVENT and status
  uint16 connHandle;    //!< Connection message was received on
  uint8 method;         //!< Type of message
  gattMsg_t msg;        //!< Attribute protocol/profile message
} gattMsgEvent_t;
```

This section lists the ATT events by their method and display their structure that is used in the message payload. These events are 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
{
  uint8 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
{
  uint8 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
{
  uint8 numPairs;                //!< Number of attribute handle-UUID pairs found
  uint8 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
{
  uint8 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
{
  uint8 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
{
  uint8 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
{
  uint8 numGrps;                 //!< Number of attribute handle, end group handle and value sets found
  uint8 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)
- ATT_PREPARE_WRITE_RSP (0x17)

```
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;
```

- 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)
  uint8 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)
  uint8 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

D.5 GATT Commands and Corresponding ATT Events

This table lists the possible commands that may cause an event.

ATT Response Events	GATT API Calls
ATT_EXCHANGE_MTU_RSP	GATT_ExchangeMTU
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs, GATT_DiscAllCharDescs
ATT_FIND_BY_TYPE_VALUE_RSP	GATT_DiscPrimaryServiceByUUID
ATT_READ_BY_TYPE_RSP	GATT_PrepareWriteReq, GATT_ExecuteWriteReq, GATT_FindIncludedServices, GATT_DiscAllChars, GATT_DiscCharsByUUID, GATT_ReadUsingCharUUID,
ATT_READ_RSP	GATT_ReadCharValue, GATT_ReadCharDesc
ATT_READ_BLOB_RSP	GATT_ReadLongCharValue, GATT_ReadLongCharDesc
ATT_READ_MULTI_RSP	GATT_ReadMultiCharValues
ATT_READ_BY_GRP_TYPE_RSP	GATT_DiscAllPrimaryServices
ATT_WRITE_RSP	GATT_WriteCharValue, GATT_WriteCharDesc
ATT_PREPARE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc
ATT_EXECUTE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc

D.6 ATT_ERROR_RSP Error Codes

This section lists the error codes that can exist 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 for encrypting this link is insufficient.
- ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is invalid for the operation.
- ATT_ERR_UNLIKELY (0x0E): The attribute request requested has encountered an error that was unlikely and failed to complete 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 exist to complete the request.

GATTServApp API

This section details the API of the GATTServApp in gattservapp_util.c.

NOTE: These API are only the public commands that should be called by the profile and/or application.

E.1 Commands

void GATTServApp_InitCharCfg(uint16 connHandle, gattCharCfg_t *charCfgTbl)

Initialize the client characteristic configuration table for a given connection. Use this whenever a service is added to the application (see Section 5.4.4.2.2)

Parameters

connHandle – connection handle (0xFFFF for all connections).

charCfgTbl – client characteristic configuration table where this characteristic is

bStatus_t GATTServApp_ProcessCharCfg(gattCharCfg_t *charCfgTbl, uint8 *pValue,uint8 authenticated, gattAttribute_t *attrTbl, uint16 numAttrs, uint8 taskId, pfnGATTReadAttrCB_t pfnReadAttrCB)

Process Client Characteristic Configuration change.

Parameters

charCfgTbl – profile characteristic configuration table

pValue – pointer to attribute value.

authenticated – whether an authenticated link is required

attrTbl – whether attribute table.

numAttrs – number of attributes in attribute table.

tasked – task to be notified of confirmation.

pfnReadAttrCB – read callback function pointer.

Returns

SUCCESS: parameter was set

INVALIDPARAMETER: one of the parameters was a null pointer

ATT_ERR_INSUFFICIENT_AUTHOR: permissions require authorization

bleTimeout: ATT timeout occurred

blePending: another ATT request is pending

LINKDB_ERR_INSUFFICIENT_AUTHEN: authentication is required but link is not authenticated

bleMemAllocError: memory allocation failure occurred when allocating buffer

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

Find the attribute record within a service attribute table for a given attribute value pointer.

Parameters

pAttrTbl – pointer to attribute table

numAttrs – number of attributes in attribute table

pValue – pointer to attribute value

Returns

Pointer to attribute record if found.

NULL, if not found.

bStatus_t GATTServApp_ProcessCCCWriteReq(uint16 connHandle, gattAttribute_t *pAttr, uint8 *pValue, uint8 len, uint16 offset, uint16 validCfg)

Process the client characteristic configuration write request for a given client.

Parameters

connHandle– connection message was received on.

pAttr – pointer to attribute value.

pValue– pointer to data to be written

len – length of data

offset– offset of the first octet to be written

validCfg– valid configuration

Returns

SUCCESS: CCC was written correctly

ATT_ERR_INVALID_VALUE: not a valid value for a CCC

ATT_ERR_INVALID_VALUE_SIZE: not a valid size for a CCC

ATT_ERR_ATTR_NOT_LONG: offset needs to be 0

ATT_ERR_INSUFFICIENT_RESOURCES: CCC not found

GAPBondMgr API

This section details the API of the GAPBondMgr in gapbondmgr.c.

NOTE: Many commands do not need to be called from the application because they are called by the GAPRole or the BLE Stack.

F.1 Commands

bStatus_t GAPBondMgr_SetParameter(uint16 param, void *pValue)

Set a GAP Bond Manager parameter.

Parameters

param – Profile parameter ID (see Section F.2)

len – length of data to write

pValue – pointer to value to set parameter. This depends on the parameter ID and is cast to the appropriate data type

Returns

SUCCESS: parameter was set

INVALIDPARAMETER: param was not valid

bleInvalidRange: len is not valid for the given param

bStatus_t GAPBondMgr_GetParameter(uint16 param, void *pValue)

Get a GAP Bond Manager parameter.

Parameters

param – Profile parameter ID (see Section F.2

pValue – pointer to a location to get the value. This pointer depends on the param ID and is cast to the appropriate data type.

Returns

SUCCESS: param was successfully placed in pValue

INVALIDPARAMETER: param was not valid

bStatus_t GAPBondMgr_LinkEst(uint8 addrType, uint8 *pDevAddr, uint16 connHandle, uint8 role)

Notify the Bond Manager that a connection has been made.

Parameters

addrType – address type of the peer device:

peerAddr – peer device address

connHandle – connection handle

role – master or slave role

Returns

SUCCESS: GAPBondMgr was notified of link establishment

void GAPBondMgr_LinkTerm(uint16 connHandle)

Notify the Bond Manager that a connection has been terminated.

Parameters connHandle – connection handle

void GAPBondMgr_SlaveReqSecurity(uint16 connHandle)

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

Parameters connHandle – connection handle

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

Resolve an address from bonding information.

Parameters addrType – address type of the peer device:

peerAddr – peer device address

pResolvedAddr – pointer to buffer to put the resolved address

Returns Bonding index (0 – (GAP_BONDINGS_MAX-1): if address was found...

GAP_BONDINGS_MAX: if address was not found

bStatus_t GAPBondMgr_ServiceChangeInd(uint16 connectionHandle, uint8 setParam)

Set/clear the service change indication in a bond record.

Parameters connHandle – connection handle of the connected device or 0xFFFF for devices in database.

setParam – TRUE to set the service change indication, FALSE to clear it.

Returns SUCCESS – bond record found and changed

bleNoResources – no bond records found (for 0xFFFF connHandle)

bleNotConnected – connection with connHandle is invalid

bStatus_t GAPBondMgr_UpdateCharCfg(uint16 connectionHandle, uint16 attrHandle, uint16 value)

Update the Characteristic Configuration in a bond record.

Parameters connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database.

attrHandle – attribute handle

value – characteristic configuration value

Returns SUCCESS – bond record found and changed

bleNoResources – no bond records found (for 0xFFFF connectionHandle)

bleNotConnected – connection with connectionHandle is invalid

void GAPBondMgr_Register(gapBondCBs_t *pCB)

Register callback functions with the bond manager.

Parameters pCB – pointer to callback function structure (see Section F.3)

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

Respond to a passcode request and update the passcode if possible.

Parameters

connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database.

status – SUCCESS if passcode is available, otherwise see SMP_PAIRING_FAILED_DEFINES in gapbondmgr.h

passcode – integer value containing the passcode

Returns

SUCCESS: connection found and passcode was changed

bleIncorrectMode: connectionHandle connection not found or pairing has not started

INVALIDPARAMETER: passcode is out of range

bleMemAllocError: heap is out of memory

uint8 GAPBondMgr_ProcessGAPMsg(gapEventHdr_t *pMsg)

This is a bypass mechanism to let the bond manager process GAP messages.

Note

NOTE: This bypass mechanism is an advanced feature and should not 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: not safe to deallocate an incoming gap message

uint8 GAPBondMgr_CheckNVLen(uint8 id, uint8 len)

This function will check the length of a Bond Manager NV Item.

Parameters

id – NV ID

len – lengths in bytes of item.

Returns

SUCCESS: NV item is the correct length

FAILURE: NV item is an incorrect length

F.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPBOND_PAIRING_MODE	R/W	uint8	Default is GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_INITIATE_WAIT	R/W	uint16	Pairing Mode Initiate wait timeout. 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 0s.
GAPBOND_BONDING_ENABLED	R/W	uint8	Request Bonding during the pairing process if enabled. Default is 0(disabled).
GAPBOND_KEY_DIST_LIST		uint8	The key distribution list for bonding. Default is sEncKey, sIdKey, mIdKey, mSign enabled.
GAPBOND_DEFAULT_PASSCODE		uint32	The default passcode for MITM protection. Range is 0 to SWRU2718752999,999. 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 when a bonding is unsuccessful).
GAPBOND_ERASE_SINGLEBOND	W	uint8[9]	Erase a single bonded device. Must provide address type followed by device address.

F.3 Callbacks

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

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

F.3.1 Passcode Callback (passcodeCB)

This callback returns to the application the peer device information when a passcode is requested by the peer device during the paring process. This function is defined as the following:

```
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 like the pairing interface inputs and outputs, the application displays the passcode or initiates the entrance of a passcode.

F.3.2 Pairing State Callback (pairStateCB)

This callback returns the current pairing state to the application whenever the state changes and the current status of the pairing or bonding process associated with the current state and provides the current status of the pairing. This function is defined as the following:

```
typedef void (*pfnPairStateCB_t)
(
  uint16 connectionHandle,   //!< Connection handle
  uint8  state               //!< Pairing state @ref GAPBOND_PAIRING_STATE_DEFINES
  uint8  status              //!< Pairing status
);
```

The pairing state(s) is enumerated as the following:

- GAPBOND_PAIRING_STATE_STARTED
  - The following status is possible for this state:
    - SUCCESS (0x00): pairing has been initiated
- GAPBOND_PAIRING_STATE_COMPLETE
  - The following statuses are possible for this state:
    - SUCCESS (0x00): pairing is complete. Session keys have been exchanged.
    - SMP_PAIRING_FAILED_PASSKEY_ENTRY_FAILED (0x01): input failed
    - SMP_PAIRING_FAILED_OOB_NOT_AVAIL (0x02): Out-of-band data unavailable
    - SMP_PAIRING_FAILED_AUTH_REQ (0x03): input and output capabilities of devices fails to authenticate
    - SMP_PAIRING_FAILED_CONFIRM_VALUE (0x04): the confirm value fails to 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 unsupported on this device
    - SMP_PAIRING_FAILED_UNSPECIFIED (0x08): encryption failed to start
    - bleTimeout (0x17): pairing failed to complete before timeout
    - bleGAPBondRejected (0x32): keys did not match
- GAPBOND_PAIRING_STATE_BONDED
  - The following statuses are possible for this state:
    - LL_ENC_KEY_REQ_REJECTED (0x06): encryption key is missing
    - LL_ENC_KEY_REQ_UNSUPPORTED_FEATURE (0x1A): feature is unsupported by the remote device
    - LL_CTRL_PKT_TIMEOUT_TERM (0x22): Timeout waiting for response
    - bleGAPBondRejected (0x32): This status is received due to one of the above three errors.

HCI Extension API

This section describes the vendor specific HCI Extension API. These proprietary commands are specific to the CC254x device. Where more detail is required, an example is provided.

NOTE: Unless stated otherwise, the return values for these commands is SUCCESS. This value does not indicate successful completion of the command. These commands result in corresponding events that you should check by the calling application

G.1 Commands

hciStatus_t HCI_EXT_AdvEventNoticeCmd (uint8 taskID, uint16 taskEvent)

This command configures the device to set an event in the user task after each advertisement event completes. A non-zero taskEvent value is enable, while a zero valued taskEvent is disable.

Note

NOTE: This command fails to return any events but has a meaningful return status.

Parameters taskID– task ID of the user

taskEvent – task event of the user (must be a single bit value)

Returns SUCCESS: event configured correctly

LL_STATUS_ERROR_BAD_PARAMETER: more than one bit is 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())

```
    case GAPROLE_CONNECTED:
      {
        HCI_EXT_ConnEventNoticeCmd (simpleBLEPeripheral, SBP_CONN_EVT_EVT );
```

3. Check for and receive these events in the application (SimpleBLEPeripheral_ProcessEvent())

```
  if ( events & SBP_CON_CB_EVT )
  {
    //process accordingly

    return ( events ^ SBP_CON_CB_EVT );
  }
……
```

hciStatus_t HCI_EXT_BuildRevision(uint8 mode, uint16 userRevNum)

This command is used to either let the user set their own 16-bit revision number or 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. TI intended this command to be called from within the target itself. This does not preclude this command from being received through the HCI but 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: an invalid mode

Corresponding Events (only when mode == HCI_EXT_SET_USER_REVISION)
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ConnEventNoticeCmd (uint8 taskID, uint16 taskEvent)

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 enable, while a zero valued taskEvent is disable.

Note

NOTE: A slave with one connection is supported (this API only works when the device is configured as a slave and connected to one master). Send this command after establishing a connection.

This command fails to return any events but has a meaningful return status.

Parameters taskID – task ID of the user

taskEvent – task event of the user

Returns SUCCESS or FAILURE

LL_STATUS_ERROR_BAD_PARAMETER: more than one bit is 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

processStateChangeEbt()) after establishing the connection.

```
    case GAPROLE_CONNECTED:
      {
        HCI_EXT_ConnEventNoticeCmd (simpleBLEPeripheral, SBP_CONN_EVT_EVT );
```

3. Check for and receive these events in the application (SimpleBLEPeripheral_taskFxn()).

```
  if ( events & SBP_CON_CB_EVT )
  {
    //process accordingly

    return ( events ^ SBP_CON_CB_EVT );
  }
……
```

hciStatus_t HCI_EXT_DeclareNvUsageCmd (uint8 mode)

This command informs the controller whether the host uses NV memory during BLE operations. The default system value for this feature is NV In Use. When the NV is unused during BLE operations, the controller can bypass internal checks that reduce overhead processing. This capability reduces average power consumption.

Note

NOTE: This command is allowed only when the BLE controller is idle.

If you use NV when declaring it is not in use, a hung BLE connection may occur.

Parameters mode – one of…

HCI_EXT_NV_NOT_IN_USE

HCI_EXT_NV_IN_USE

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DecryptCmd (uint8 *key, uint8 * encText)

This command decrypts encrypted data using the AES128 .

Note

NOTE: Only the application should use this command. Incoming encrypted BLE data is automatically decrypted by the stack and operates free of this API.

Parameters mode – one of…

HCI_EXT_NV_NOT_IN_USE

HCI_EXT_NV_IN_USE

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DelaySleepCmd (uint16 delay)

This command sets the delay before sleep occurs after reset or when waking from sleep to let the external 32-kHz crystal stabilize. If this command is always unused, the default delay is 400 ms on the CC254x. If the hardware of the customer requires a different delay or operates free of this delay, delay can be changed by calling this command during the OSAL task initialization of the application. A zero delay value eliminates the delay after reset and (unless changed again) all subsequent wakes from sleep. A non-zero delay value changes the delay after reset and (unless changed again) subsequent wakes from sleep. If this command is used after system initialization, the new delay value will be applied the next time the delay is used.

Note

NOTE: This delay applies only to reset and sleep. If a periodic timer is used or an active BLE operation and only sleep is used, this delay occurs only after reset.

No distinction exists between a hard and soft reset. The delay (if non-zero) is applied the same way in both cases.

Parameters delay – 0x0000…0x003E8 in milliseconds

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DisconnectImmedCmd (uint16 connHandle)

This command disconnects a connection immediately. When a connection must be ended without the latency associated with the normal BLE controller terminate control procedure, use this command.

Note

NOTE: The host issuing the command receives the HCI disconnection complete event with a reason status of 0x16 (that is, connection terminated by local host), followed by an HCI vendor specific event.

Parameters connHandle– The handle of the connection.

Corresponding Events: HCI_Disconnection_Complete

HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_EnablePTMCmd (void)

This command enables production test mode (PTM). The customer uses this command when assembling their product to allow limited access to the BLE Controller for testing and configuration. This mode remains enabled until the device is reset.

Note

NOTE: This commands resets the controller. To reenter the application, reset the device.

This command fails to return any events.

Return Values HCI_SUCCESS: Successfully entered PTM

hciStatus_t HCI_EXT_EndModemTestCmd (void)

This command shuts down a modem test. A complete link layer reset occurs.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ExtendRfRangeCmd (void)

This command configures the CC254x to automatically control the TI CC2590 2.4-GHz RF Front End device. Using the CC2590 allows a maximum Tx output power of 10 dBm (the specified BLE maximum) and increases Rx sensitivity. This capability extends the RF range of the CC254x. When using this command, the configuration fails to change unless the CC254x is reset. Automatic control of the CC2590 is achieved using the CC254x Observables, which take control of GPIO P1.2 and P1.3. The GPIO P1.1 controls RF gain. These GPIOs are unavailable when using this feature. You can use this command in combination with HCI_EXT_SetTxPowerCmd, resulting in a cumulative Tx output power. For the CC2540 only, attempting to set Tx output power to 4 dBm (that is, using HCI_EXT_TX_POWER_4_DBM) sets the Tx output power to 0 dBm. Use the command HCI_EXT_SetRxGainCmd to set the Rx gain. The CC254x Rx Standard/High gain setting is mirrored by the BLE controller to the CC2590 High Gain Mode (HGM) Low/High setting. When using this command, the CC254x Tx output power and Rx gain retain their previous values unless the previous Tx output power value was set to 4 dBm on the CC2540. In this case, the value is set to 0dBm.

Corresponding Events HCI_Disconnection_Complete

HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_GetConnInfoCmd (uint8 *numAllocConns, uint8 *numActiveConns, hciConnInfo_t *activeConnInfo)

This command gets the number of allocated connections and the number of active connections. For each active connection, this command gets the connection handle, the connection role, the peer device address, and peer device address type. The number of allocated connections is based on a default build value that can be changed in the project using MAX_NUM_BLE_CONNS. The number of active connections refers to active BLE connections. The information per connection is based on the structure hciConnInfo_t provided in hci.h. This command applies only to central devices for the CC254x as peripheral devices can only have one simultaneous connection. If all parameters are NULL, the call to this command is a network processor call through a transport layer and the results are provided by the host through a vendor specific command complete event. If any parameter is not NULL, the call to this command is a direct function call and the valid pointers store the result. Ensure sufficient memory is allocated. Obtain partial results by selective using pointers. If you want to know the number of active connections, do the following:

```
uint8 numActiveConns;
(void)HCI_EXT_GetConnInfoCmd( NULL, &numActiveConns, NULL );
```

Parameters

numAllocConns – pointer to number of build time connections allowed

numActiveConns - pointer to number of active BLE connections

activeConnInfo - pointer for the connection information for each active connection which consists of the following: Connection ID, Connection Role, Peer Device Address, and Peer Address Type, which requires (the number of active connections × 9 bytes) of memory:

```
typedef struct
{
  uint8 connId;                          // device connection handle
  uint8 role;                            // device connection role
  uint8 addr[LL_DEVICE_ADDR_LEN];        // peer device address
  uint8 addrType;                        // peer device address type
} hciConnInfo_t;
```

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_HaltDuringRfCmd (uint8 mode)

This command enables or disables the halting of the MCU while the radio is operating. When the MCU is not halted, the peak current is higher but the system is more responsive. When the MCU is halted, the peak current consumption is reduced but the system is less responsive. The default value is Enable.

Note

NOTE: If there are any active BLE connections, this command is disallowed.

If the halt during RF is disabled, the HCI_EXT_ClkDivOnHaltCmd is disallowed.

Parameters mode – one of…

HCI_EXT_HALT_DURING_RF_DISABLE

HCI_EXT_HALT_DURING_RF_ENABLE

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_MapPmIoPortCmd (uint8 ioPort, uint8 ioPin)

This command configures and maps a CC254x Input and Output Port as a General-Purpose Input and Output (GPIO) output signal that reflects the power management (PM) state of the CC254x device. The GPIO output is High on Wake and Low when entering Sleep. You can disable this feature by specifying HCI_EXT_PM_IO_PORT_NONE for the ioPort (ioPin is then ignored). The system default value when the hardware reset is disabled. You can use this command to control an external DC-DC converter (its intent) such has the TI TPS62730 (or any similar converter). Use this command with extreme care. This command overrides how the port or pin was previously configured, including the mapping of Port 0 pins to 32-kHz clock output, Analog Input and Output, UART, Timers,Port 1 pins to Observables, Digital Regulator status, UART, Timers, Port 2 pins to an external 32-kHz XOSC. The selected port or pin will be configured as an output GPIO with interrupts masked. Using this command carelessly can result in a reconfiguration that could disrupt the system. If a port or pin is used as part of the serial interface for the device, the pin or port is reconfigured from its original peripheral function to a GPIO, disrupting the serial port. Ensure the pin or port does not cause any conflicts in the system.

Note

NOTE: Only pins 0, 3 ,and 4 are valid for port 2 because pins 1 and 2 are mapped to debugger signals DD and DC.

A port or pin signal change occurs only when power savings is enabled.

Parameters

ioPort – one of the following:

- HCI_EXT_PM_IO_PORT_P0
- HCI_EXT_PM_IO_PORT_P1
- HCI_EXT_PM_IO_PORT_P2
- HCI_EXT_PM_IO_PORT_NONE

ioPin – one of the following:

- HCI_EXT_PM_IO_PORT_PIN0
- HCI_EXT_PM_IO_PORT_PIN1
- HCI_EXT_PM_IO_PORT_PIN2
- HCI_EXT_PM_IO_PORT_PIN3
- HCI_EXT_PM_IO_PORT_PIN4
- HCI_EXT_PM_IO_PORT_PIN5
- HCI_EXT_PM_IO_PORT_PIN6
- HCI_EXT_PM_IO_PORT_PIN7

Corresponding Events: HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

This API starts a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37-byte packet of pseudo-random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0 through 39) every 625 µs. Use the HCI_EXT_EndModemTest command to end the test.

Note

NOTE: When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemTestRxCmd(uint8 rxFreq)

This API starts a continuous receiver modem test using a modulated carrier wave tone, at the frequency that corresponds to the specific RF channel. Any received data is discarded. Receiver gain 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

NOTE: The RF channel not the BLE frequency is specified. You can obtain the RF channel from the BLE frequency as follows: RF Channel = (BLE Frequency – 2402) ÷ 2.

When the HCI_EXT_EndModemTest is issued to stop this test, the controller resets.

Parameters rxFreq- selects which channel [0 to 39] on which to receive

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_NumComplPktsLimitCmd (uint8 limit, uint8 flushOnEvt)

This command sets the limit on the minimum number of complete packets before the controller returns a control number of completed packets event . If the limit is not reached by the end of a connection event, the number of completed packets event is returned based on the flushOnEvt flag if nonzero. You can set the limit rom one to the maximum number of HCI buffers (see the LE Read Buffer Size command in). The default limit is one; the default flushOnEvt flag is FALSE.

Note

NOTE: The purpose of this command is to minimize the overhead of sending multiple number of completed packet events. Minimizing this number of events maximizes the processing available to increase wireless throughput. This command 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)

This command configures the link layer to only allow one packet per connection event. The default system value for this feature is disabled. This command can 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

NOTE: Perform a power analysis of the system before determining whether this command will save power. Transferring multiple packets per connection event, may be more power efficient.

Parameters control – HCI_EXT_DISABLE_ONE_PKT_PER_EVT, HCI_EXT_ENABLE_ONE_PKT_PER_EVT

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent: this event is returned only if the setting is changing from enable to disable or disable to enable

hciStatus_t HCI_EXT_OverlappedProcessingCmd (uint8 mode)

This command enables or disables overlapped processing. The default is disabled. See the for more information.

Parameters mode – one of the following:

- HCI_EXT_DISABLE_OVERLAPPED_PROCESSING,
- HCI_EXT_ENABLE_OVERLAPPED_PROCESSING

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent: this event is returned only if the setting is changing from enable to disable or disable to enable

hciStatus_t HCI_EXT_PacketErrorRateCmd (uint16 connHandle, uint8 command)

This command resets or reads the packet error rate counters for a connection. When resetting, the counters are cleared. When reading the total number of packets received, the number of packets received with a CRC error, the number of events, the number of missed events are returned.

Note

NOTE: The counters are only 16 bits. At the shortest connection interval, this command allows for a little over eight minutes of data.

Parameters connId– The connection ID to perform the command

command- HCI_EXT_PER_RESET, HCI_EXT_PER_READ

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_PERbyChanCmd (uint16 connHandle, perByChan_t *perByChan)

This command starts or ends the packet error rate by accumulating channel counters 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 discontinues. If *perByChan is not NULL, sufficient memory is assumed to exist 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

NOTE: You must ensure there is sufficient memory allocated in the perByChan structure. You must also maintain the counters by clearing them if required before starting accumulation.

The counters are 16 bits. At the shortest connection interval, this provides a bit over 8 minutes of data.

This command can be used combined with HCI_EXT_PacketErrorRateCmd.

Parameters connHandle – The connection ID to accumulate the data

perByChan- Pointer to PER by channel data or NULL

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ResetSystemCmd (uint8 mode)

This command issues a hard or soft system reset. A watchdog timer timeout causes a hard reset. Resetting the PC to zero causes a soft reset.

Note

NOTE: The reset occurs after a 100 ms delay to let the correspond event to be returned to the application.

Parameters mode – HCI_EXT_RESET_SYSTEM_HARD

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SaveFreqTuneCmd (void)

This PTM-only command saves this device's the tuning setting of this device in non-volatile memory. The BLE Controller uses this setting when resetting and waking from sleep.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetBDADDRCmd(uint8 *bdAddr)

This command sets the BLE address (BDADDR) of the device. This address overrides the address of the device determined at reset. To restore the initialized address of the device stored in flash, issue this command with an invalid address (0xFFFFFFFFFFFF).

Note

NOTE: This command is allowed only when the controller is in the standby state. TI intends this command to be used only during initialization. Changing the BDADDR of the device after active BLE operations (i.e. Tx and Rx) have occurred may cause undefined behavior.

Parameters bdAddr – A pointer to a buffer to hold this address of the device. An invalid address (that is, all FFs) restores the address of this device to the address set at initialization.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd (uint8 control)

This command configures the link layer fast transmit response time feature. The default system value for this feature is enabled.

Note

NOTE: This command is valid only for a slave controller.

When the host transmits data, the controller ensures the packet is sent over the LL connection with minimal delay even when the connection is configured to use slave latency by default. The transmit response time is no longer than the connection interval, instead of waiting for the next effective connection interval due to slave latency. The result is lower power savings because the LL may need to wake to transmit during connection events that would have been skipped due to slave latency. If saving power is more critical than fast transmit response time, you can disable this feature using this command. When disabled, the transmit response time will be no longer than the effective connection interval (slave latency + 1 × 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_SetFreqTuneCmd (uint8 step)

This PTM-only command sets the frequency tuning of the device either up one step or down one step. When the current setting is at its maximum value, stepping up has no effect. When the current setting is at its mininimum value, stepping down has no effect. This setting remain only in effect until the device is reset unless you use HCI_EXT_SaveFreqTuneCmd to save it in nonvolatile memory.

Parameters step – HCI_PTM_SET_FREQ_TUNE_UP, HCI_PTM_SET_FREQ_TUNE_DOWN

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetLocalSupportedFeaturesCmd (uint8 * localFeatures)

This command sets the local supported features of the controller.

Note

NOTE: This command can be issued either before or after one or more connections are formed. The local features set are only effective if performed before a feature exchange procedure has been initiated by the master. When this control procedure has been completed for a connection, only the exchanged feature set for that connection will be used. Because the link layer may initiate the feature exchange procedure autonomously, TI recommends using this command before the connection is formed.

The features are initialized by the controller when starting up. You might not need this command. Refer to ll.h for a description of the local features.

Parameters localFeatures – A pointer to the feature set where each bit where each bit corresponds to a feature 0: feature shall not be used

0: Feature shall not be used

1: Feature can be used

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd (uint8 txPower)

This command overrides the RF transmitter output power used by the direct test mode (DTM). Typically, the maximum transmitter output power setting used by DTM is the maximum transmitter output power setting for the device (that is, 5 dBm). This command changes the value used by DTM.

Note

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_SetRxGainCmd(uint8 rxGain)

This command sets the RF receiver gain. The default system value for this feature is standard receiver gain.

Note

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 rxGain– one of:

- HCI_EXT_RX_GAIN_STD
- HCI_EXT_RX_GAIN_HIGH

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetSCACmd (uint16 scaInPPM)

This command sets the sleep clock accuracy (SCA) value of the device in parts per million (PPM), from 0 to 500. For a master device, the value is converted to one of eight ordinal values representing a SCA range per , which is used when a connection is created. For a slave device, the value is used directly. The system default value for a master and slave device is 50 ppm and 40 ppm, respectively.

Note

NOTE: This command is allowed only when the device is disconnected.

The SCA value of the device remains unaffected by an HCI reset.

Parameters scaInPPM – The SCA of the device in PPM from 0 to 500.

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd (uint8 mode)

This command enables or disables the slave latency override letting the user temporarily suspend slave latency even though it is active for the connection. When enabled, the device wakes up for every connection until slave latency override is disabled again. The default value is disable.

Note

NOTE: This command applies only to devices in the slave role.

This command can help when the slave application will soon receive something that must be handled immediately. This command fails to change the slave latency connection parameter: the device wakes up for each connection event.

Parameters control – HCI_EXT_ENABLE_SL_OVERRIDE, HCI_EXT_DISABLE_SL_OVERRIDE

Corresponding Events HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetTxPowerCmd(uint8 txPower)

This command sets the RF transmitter output power. The default system value for this feature is 0 dBm.

Parameters txPower– Device's transmit power, one of the following:

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

G.2 Host Error Codes

This section lists the various possible error codes generated by the host. An HCI extension command may respond with a command status of SUCCESS. If an error is detected during subsequent processing of that command, the relevant error code is reported in the command complete event.

The error code 0x00 means SUCCESS. The possible range of failure error codes is 0x01 to 0xFF. The following table provides an error code description for each failure error code.

Table G-1. Host Error Codes

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

Revision History

Changes from F Revision (July 2013) to G Revision	Page
• Added TI BLE Software Development Platform Chapter.	11
• Added images to OSAL Chapter.	15
• Added The Application and Profiles Chapter.	20
• Added content to The BLE Protocol Stack chapter.	27
• Added Drivers Chapter.	65
• Added Creating a Custom BLE Application chapter.	68
• Added Development and Debugging Chapter.	71
• Added General Information chapter.	81
• Added GAP API appendix.	95
• Added GAPRole Peripheral Role API appendix.	102
• Added GAPRole Central Role API appendix.	107
• Added GATT/ATT API appendix.	113
• Added GATTServApp API appendix.	125
• Added GAPBondMgr API appendix.	127
• Added HCI Extension API appendix.	133

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265