Texas Instruments CC2540/41 *Bluetooth*® low energy Software Developer's Guide v1.4.1

Document Number: SWRU271G

References

The following references are included with Texas Instruments *Bluetooth* Low Energy v1.4.1 Stack Release. Note that all path and file references in this document assume that the BLE development kit software has been installed to the default path: C:\Texas Instruments\BLE-CC254X-1.4.1\. Throughout this document, this path will be referred to as $INSTALL$

[1] TI BLE Vendor Specific HCI Reference Guide

$INSTALL$\Documents\TI_BLE_Vendor_Specific_HCI_Guide.pdf

[2] Texas Instruments 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.pd

[5] Texas Instruments 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 the Texas Instruments web site:

[9] Texas Instruments CC2540DK-MINI *Bluetooth* Low Energy User Guide v1.1

http://www.ti.com/lit/pdf/swru270

[10] Measuring Power Consumption App Note

http://www.ti.com/litv/pdf/swra347a

[11] CC2541/43/44/45 Peripherals Software Examples

http://www.ti.com/lit/zip/swrc257

[12] CC254x Chip User's Guide

http://www.ti.com/lit/pdf/swru191

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

[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

Useful Links

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

[16] Latest stack download: www.ti.com/ble-stack

[17] Support forum: www.ti.com/ble-forum

1 Overview

The purpose of this document is to give an overview of the Texas Instruments CC2540/41 *Bluetooth*® low energy (BLE) software development kit. This document also serves as an introduction to the BLE standard; however it should not be used as a substitute for the complete specification. For more details, see [13].

The release history of the BLE software development kit, including detailed information on changes, enhancements, bug fixes, and known issues, can be found in section 9.1.

Note that The TI BLE Stack™ v1.4.1 only supports Bluetooth 4.0. For Bluetooth 4.1 support please see the TI BLE Stack for the SimpleLink™ Bluetooth low energy CC2640 wireless MCU [16].

1.1 Introduction

Version 4.0 of the *Bluetooth*® standard allows for two systems of wireless technology: Basic Rate (BR; often referred to as "BR/EDR" for "Basic Rate / Enhanced Data Rate") and *Bluetooth* low energy (BLE). The BLE protocol was created for the purpose of transmitting very small packets of data at a time, while consuming significantly less power than BR/EDR devices.

Devices that can support BR and BLE are referred to as dual-mode devices and go under the branding *Bluetooth* Smart Ready. Typically in a *Bluetooth* 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 go under the branding *Bluetooth* Smart. These single-mode devices are generally used for application in which low power consumption is a primary concern, such as those that run on coin cell batteries.

Figure 1 *Bluetooth* Smart and Smart Ready Branding Marks

1.2 BLE Protocol Stack Basics

The BLE protocol stack architecture is illustrated here:

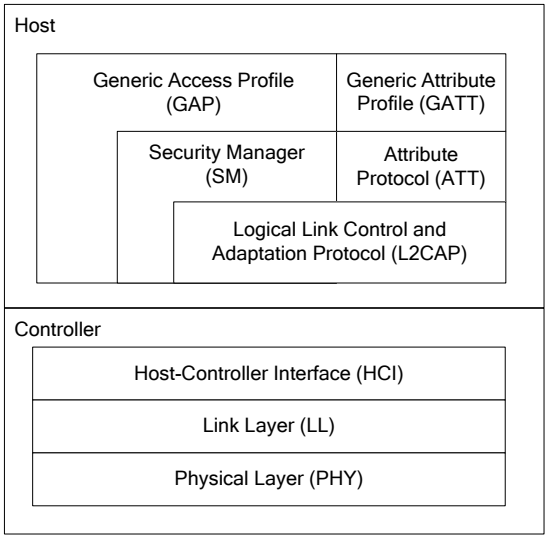


Figure 2: BLE Protocol Stack

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

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

The *link layer* (LL) controls the RF state of the device, with the device being in one of five possible states: standby, advertising, scanning, initiating, or connected. Advertisers transmit data without being in a connection, while scanners listen for advertisers. An Initiator is a device that is responding to an Advertiser with a connection request. If the Advertiser accepts, both the advertiser and initiator will enter a connected state. When a device is in a connection, it will be connected in one of two roles: master or slave. The device that initiated the connection becomes the master, and the device that accepted the request becomes the slave. This layer is implemented in the library code in TI 1.4.1 BLE stack.

The *host control interface* (HCI) layer provides a means of communication between the host and controller via a standardized interface. This layer can be implemented either through a software API, or by a hardware interface such as UART, SPI, or USB. Standard HCI commands and events are specified in the Bluetooth Core Spec [13]. Texas Instruments' proprietary commands and events are specified in the Vendor Specific Guide [1].

The *link logical control and adaption protocol* (L2CAP) layer provides data encapsulation services to the upper layers, allowing for logical end-to-end communication of data. See section 5.5 for more information on Texas Instruments' implementation of the L2CAP layer.

The *security manager* (SM) layer defines the methods for pairing and key distribution, and provides functions for the other layers of the protocol stack to securely connect and exchange data with another device. See section 5.3 for more information on Texas Instruments' implementation of the SM layer.

The *generic access protocol* (GAP) layer directly interfaces with the application and/or profiles, to handle device discovery and connection-related services for the device. In addition, GAP handles the initiation of security features. See section 5.1 for more information on Texas Instruments' implementation of the GAP layer.

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

The *generic attribute protocol* (GATT) layer is a service framework that defines the sub-procedures for using ATT. All data communications that occur between two devices in a BLE connection are handled through GATT sub-procedures. Therefore, the application and/or profiles will directly use GATT. See section 5.4 for more information on Texas Instruments' implementation of the ATT and GATT layers.

2 Texas Instruments BLE software development platform

The Texas Instruments royalty-free BLE software development kit is a complete software platform for developing single-mode BLE applications. It is based on the CC2540/41 complete System-on-Chip (SoC) solution. The CC2540/41 combines a 2.4GHz 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 / application configurations:

1. **Single-Device:** The controller, host, profiles, and application are all implemented on the CC2540/41 as a true single chip solution. This is the simplest and most common configuration when using the CC2540/41. This is the configuration that most of our sample projects use. It is most cost effective and provides the lowest-power performance. The SimpleBLEPeripheral and SimpleBLECentral projects are examples of applications built using the single-device configuration. More information on these projects can be found in section 4.

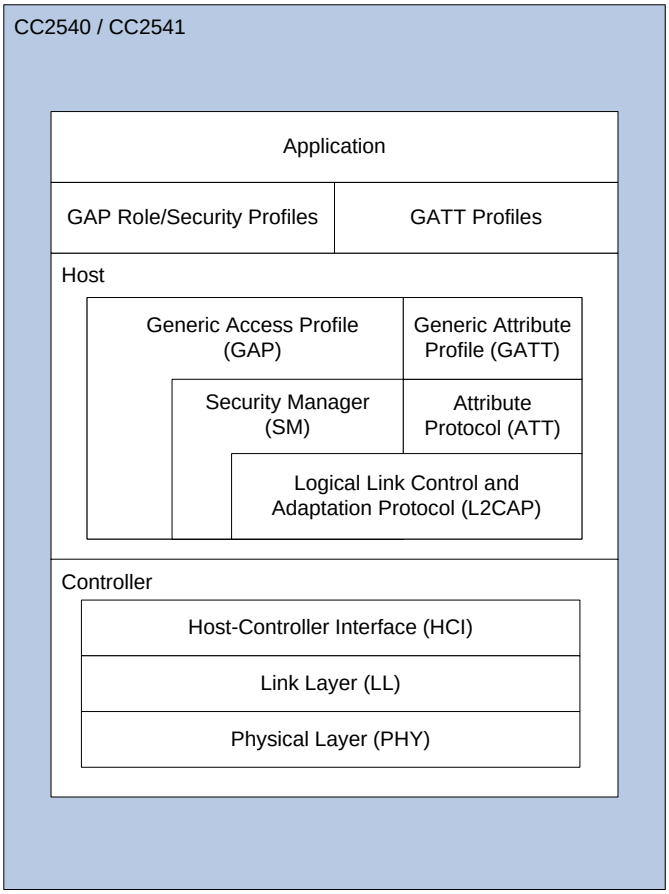


Figure 3: Single-Device Configuration

2. **Network Processor:** The controller and host are implemented together on the CC2540/41, while the profiles and application are implemented separately. The application and profiles communicate with the CC2540/41 by means of vendor-specific HCI commands using a SPI or UART interface, or using a virtual UART interface over USB. This configuration is useful for applications which execute on either another device (such as an external microcontroller) or a PC. In these cases, the application can be developed externally while still running the BLE stack on the CC2540/41. To use the network processor, the HostTestRelease project must be used.

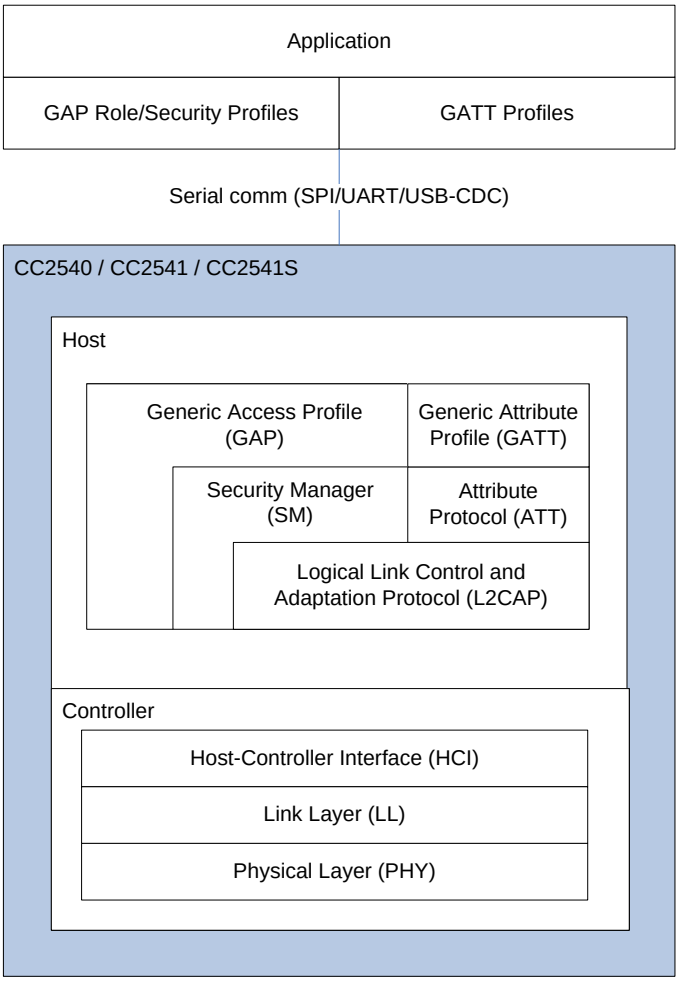


Figure 4: Network Processor Configuration

2.2 Projects

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

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

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

Several other sample projects are included in the BLE development kit, implementing various profiles and demo applications. More information on these other projects can be found in [5]

2.3 Software Overview

Software developed using the BLE software development kit consists of five major sections: the OSAL, HAL, the BLE Protocol Stack, profiles, and the application. The BLE protocol stack is provided as object code, while the OSAL and HAL code is provided as full source. In addition, three GAP profiles (peripheral role, central role, and peripheral bond manager) are provided, as well as several sample GATT profiles and applications. These modules will be described throughout this document.

All path and file references in this document assume that the BLE development kit software has been installed to the default path: C:\Texas Instruments\BLE-CC254X-1.4.1\.

Note that the SimpleBLEPeripheral project will be used as a reference throughout this guide. However all of the BLE projects included in the development kit will follow a similar structure.

3 Operating System Abstraction Layer (OSAL)

The BLE protocol stack, the profiles, and all applications are built around the Operating System Abstraction Layer (OSAL). The OSAL is not an actual operating system in the traditional sense but rather a control loop that allows software to setup the execution of events. If it were an actual operating system, it would be considered non-pre-emptive single-threaded. Each layer of software functions as a task and requires a task identifier (ID), a task initialization routine, and an event processing routine. A message processing routine may optionally be defined as well. These layers must adhere to a strict priority scheme with the LL being the highest priority (since it has very strict timing requirements). For example, here 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
};
```

In addition to task management, the OSAL provides additional services such as message passing, heap management, and timers. All OSAL code is provided as full source.

For more information on the OSAL functions, see the OSAL API Guide [7].

3.1 Task Initialization

In order to use the OSAL, at the end of the `main()` function there should be a call to `osal_start_system()`. This is the OSAL routine that starts the system, and which will call the `osalInitTasks()` function that is defined by the application. In the SimpleBLEPeripheral project, this function can be found 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 that is using the OSAL must have an initialization routine that is called from the function `osalInitTasks()`. Within this function, the initialization routine for every layer of software is called. As each task initialization routine is called, an 8-bit "task ID" value is assigned to the task. Note that when creating an application, it is very important that it be added to the end of the list, such that it has a higher task ID than the others. This is because the priority of tasks is determined by the task ID, with a lower value meaning higher priority. It is important that the protocol stack tasks have the highest priority in order to function properly. Such is the case with the SimpleBLEPeripheral application: its initialization function, `SimpleBLEPeripheral_Init()`, has the highest task ID and therefore the lowest priority.

3.2 Task Events and Event Processing

After the OSAL completes initialization, it runs in an infinite loop checking for task events. This loop can be found in the function `osal_start_system()` in the file *OSAL.c*. Task events are stored as unique bits in a 16-bit variable where each bit corresponds to a unique event. The definition and use of these event flags is completely up to the application. Here is a flow diagram of the OSAL processing scheme:

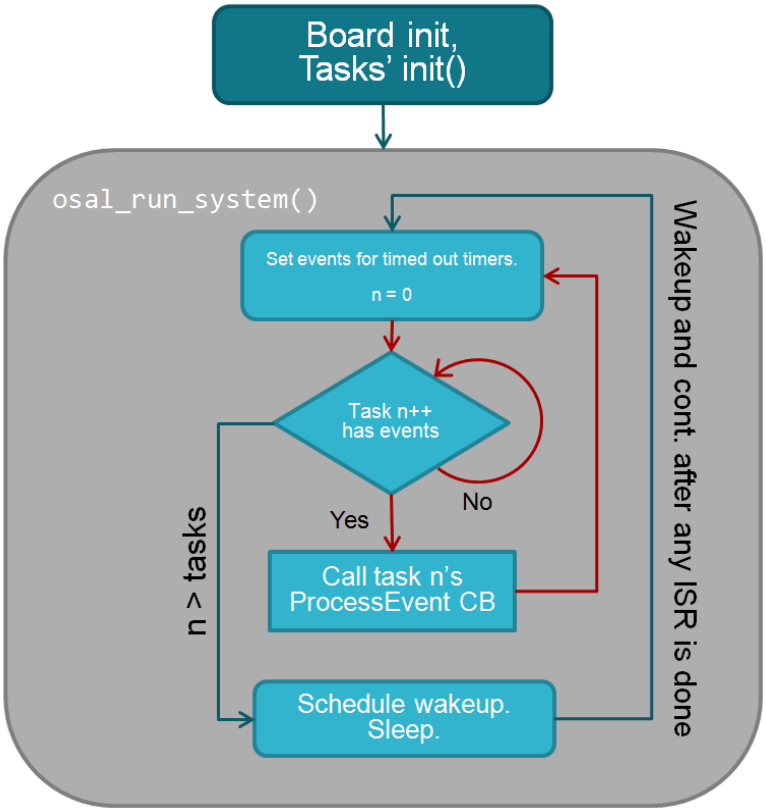


Figure 5: OSAL task loop

For example, the SimpleBLEPeripheral application defines a flag in *simpleBLEPeripheral.h*: `SBP_START_DEVICE_EVT` (0x0001), which indicates that the initial device start has completed, and the application processing should begin. The only flag value which is reserved and cannot be defined by the application is 0x8000, which corresponds to the event `SYS_EVENT_MSG` (this event is used for messaging between tasks, covered in section 3.4).

When the OSAL detects an event set for a task, it will call that task's event processing routine to process the event. The task layer must have added its own event processing routine to the table formed by the array of function pointers called `tasksArr` (located in *OSAL_SimpleBLEPeripheral.c* in the example). You will notice that the order of the event processing routines in `tasksArr` is identical to the order of task ID's in the `osalInitTasks()` function. This is required in order for events to be processed by the correct software layer.

In the case of the SimpleBLEPeripheral application, the event processing function is called `SimpleBLEPeripheral_ProcessEvent()`. Note that once the event is handled and if it is not removed from the event flag, the OSAL will continue to repeatedly call the task's process event handler. As can be seen in the SimpleBLEPeripheral application function `SimpleBLEPeripheral_ProcessEvent()`, 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 );
  }
```

It is possible for any layer of the software to set an OSAL event for any other layer, as well as for itself. The simplest way to set up an OSAL event is to use the `osal_set_event()` function (prototype in *OSAL.h*), which immediately schedules a new event. With this function, you specify the task ID (of the task that will be processing the event) and the event flag as parameters.

Another way to set an OSAL event for any layer is to use the `osal_start_timerEx()` function (prototype in *OSAL_Timers.h*). This function operates just like the `osal_set_event()` function. You select the task ID of the task that will be processing the event and the event flag as parameters. There is also a third parameter in `osal_start_timerEx()` which accepts as input a timeout value in milliseconds. This will cause the OSAL to set a timer and the specified event will get set at the expiration of the timer.

3.3 Heap Manager

OSAL also provides basic memory management functions. The `osal_mem_alloc()` function serves as a memory allocation function similar to the standard C malloc function: it takes a single parameter specifying the number of bytes to allocate and returns a void pointer if successful. If no memory is available, a `NULL` pointer will be returned. Similarly, the `osal_mem_free()` function works similar to the standard C free function: it frees memory that was previously allocated using `osal_mem_alloc()`.

The pre-processor define `INT_HEAP_LEN` is used to reserve memory for dynamic allocation.

To see profile the run-time memory usage, you can set the pre-processor define `OSALMEM_METRICS=TRUE` in the project options. After a stress test of the application where you send as many messages, have as many clients as you will in the worst case, remembering to use bonding and encryption during the test if that's applicable, etc., you can look at the value of the variable `memMax` in *OSAL_Memory.c* to see the maximum amount memory was ever allocated at a given time. This figure can be used as a guideline for lowering INT_HEAP_LEN. However, thorough testing is needed as the heap is also used by the BLE stack.

3.4 OSAL Messages

OSAL also provides a scheme for different subsystems of the software to communicate with each other by sending or receiving messages. Messages can contain any type of data and can be any size (assuming there is enough memory). To send an OSAL message, first the memory used to store the message must be allocated with `osal_msg_allocate()` by passing in parameter specifying the length of the message. A pointer to a buffer containing the allocated space will be returned (you do not need to use `osal_mem_alloc()` when using `osal_msg_allocate()`). If no memory is available, a `NULL` pointer will be returned. You can then copy the data into the buffer. To send the message, `osal_msg_send()` should be called with the destination task for the message indicated as a parameter. Here is 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 will then signal the receiving task that a message is arriving by setting the `SYS_EVENT_MSG` flag for that task. This causes the receiving task's event handler function to be called. The receiving task can then retrieve the data by calling `osal_msg_receive()` and can process accordingly based on the data received. It is recommended that every OSAL task have a local message processing function (the SimpleBLEPeripheral application's message processing function is `simpleBLEPeripheral_ProcessOSALMsg()`) that decides what action to take based on the type of message received. Once the receiving task has completed processing 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 various layers' event processing functions.

4 The Application and Profiles

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

4.1 Project Overview

On the left side of the IAR window, the "Workspace" section will list all of the files used by the project:

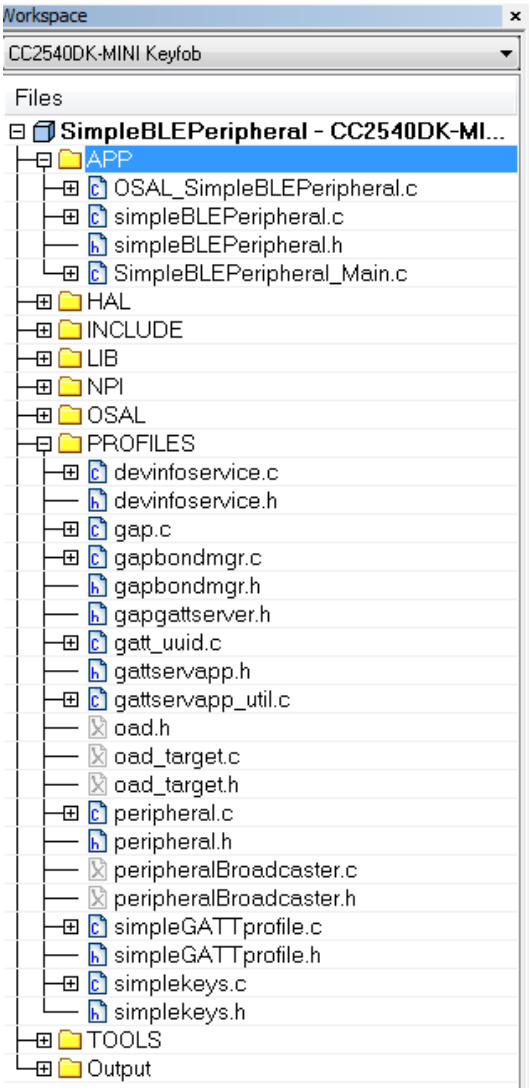


Figure 6: Project Files

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. More information on the HAL can be found in section 6.
- **INCLUDE** – This group includes all of the necessary header files for the BLE protocol stack API (detailed in the Appendix as well as the API Guide [2]).
- **LIB** – This group contains the protocol stack library file **CC2540_BLE_peri.lib**. More information on the protocol stack libraries can be found in 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). If not used, the CC254X_BLE_HCI_TL_None.lib should be used (see SimpleBLEPeripheral) when developing a single-chip application.
- **OSAL** – This group contains the OSAL source code and header files. More information on the OSAL can be found in section 3 and the OSAL API Guide [7].
- **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. More information on these modules can be found in section 5.

- **TOOLS** – This group contains the configuration files *buildComponents.cfg* and *buildConfig.cfg*. These files are described in section 5.7. It also contains the files *OnBoard.c* and *OnBoard.h*, which handle user 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 inside of *SimpleBLEPeripheral_Main.c* is the starting point at runtime. This is where the board is brought up and drivers, OSAL, and SNV are initialized. Also in this function, power management is initialized and the tasks are created / constructed. In the final step, the processing loop (OSAL) is started by calling `osal_start_system(),` which 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. First, the `SimpleBLEPeripheral_Init()` function is called by the OSAL. This function sets up the GAP role profile parameters, GAP characteristics, the GAP bond manager parameters, and simpleGATTprofile parameters. It also sets an OSAL `SBP_START_DEVICE_EVT` event.

This event triggers the second phase of the initialization, which can be found within the `SimpleBLEPeripheral_ProcessEvent()` function. During this phase, the `GAPRole_StartDevice()` function is called, which sets up the GAP functions of the application. The device then is made to be discoverable with connectable undirected advertisements (for CC2540/41DK-MINI keyfob builds, the device does not become discoverable until the right button is pressed). A central device can discover the peripheral device by scanning. If a central device sends a connection request to the peripheral device, the peripheral device will accept the request and go into the connected state as a slave. If no connection request is received, the device will only remain discoverable for 30.72 seconds, before going to the 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 the SimpleGATTProfile characteristic values. It can also enable notifications of one of the characteristics.

4.4 Event Processing

After initialization, the application task will process events in `SimpleBLEPeripheral_ProcessEvent` when a bit is set in its `events` variable. Possible sources of events are described here.

4.4.1 Periodic Event

The application contains an OSAL event defined as `SBP_PERIODIC_EVT`, which is set to occur periodically by means of an OSAL timer. The timer gets set after the `SBP_START_DEVICE_EVT is processed`, with a timeout value of `PERIODIC_EVT_PERIOD` (default value is 5000 milliseconds). Therefore, every five 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 simply gets the value of the third characteristic in the SimpleGATTProfile, and copies that value into the fourth characteristic. This is implemented for demonstration purposes: any application task can be performed within this function. Before calling the function, a new OSAL timer is started, setting up the next periodic task.

4.4.2 OSAL Messages

OSAL Messages can come from various layers of the BLE stack. As an example, consider key presses sent from HAL. The application has some additional code that is specific to the keyfob contained with the CC2540/41DK-MINI development kit. This code is only surrounded by the pre-processor directive "#if defined(CC2540_MINIDK)" and only 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 from the user through the simple keys profile.

Each time one of the keys on the keyfob gets pressed or released, HAL sends an OSAL message to the application. As described in section 3.4, this causes a `SYS_EVENT_MSG` event to occur which is handled in the application by the function `simpleBLEperipheral_ProcessOSALMsg()`. In the current SimpleBLEPeripheral application, the only OSAL message that is recognized (additional types can be defined) is the `KEY_CHANGE` message. This causes the function `simpleBLEPeripheral_HandleKeys()` to be called, which checks the state of the keys.

4.5 Callbacks

Besides processing events, application code can also be run using the various callbacks defined by the application such as `simpleProfileChangeCB()` and `peripheralStateNotificationCB()`. However, it is important to note that these callbacks are processing in the context of the task which called them, not the application. Therefore, processing should be limited in these callbacks. If any intensive processing needs to be done, an event should be sent from the callback to the application so that processing can occur as described in section 4.4.

4.6 Complete Attribute Table

During initialization, various profiles are added to the application as described in section 3.1. The table below shows the SimpleBLEPeripheral complete attribute table, and can be used as a reference when communicating over-the-air with the device. Services are shown in red, characteristic descriptors are shown in yellow, and general attributes are shown in white. These are described in more detail in section 5.4. When working with the SimpleBLEPeripheral application, it might be useful to print out the table 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 7: SimpleBLEPeripheral Complete Attribute Table

4.7 Additional Sample Projects

The BLE development kit includes several sample projects implementing various profiles, such as a heart rate monitor, health thermometer, and proximity keyfob. More information on these projects can be found in [5].

5 The BLE Protocol Stack

The entire BLE protocol stack is provided as object code in library files. Texas Instruments does not provide the protocol stack source code as a matter of policy. The functionality of these layers should be understood as they interact directly with the application and profiles.

5.1 Generic Access Profile (GAP)

5.1.1 Gap Overview

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

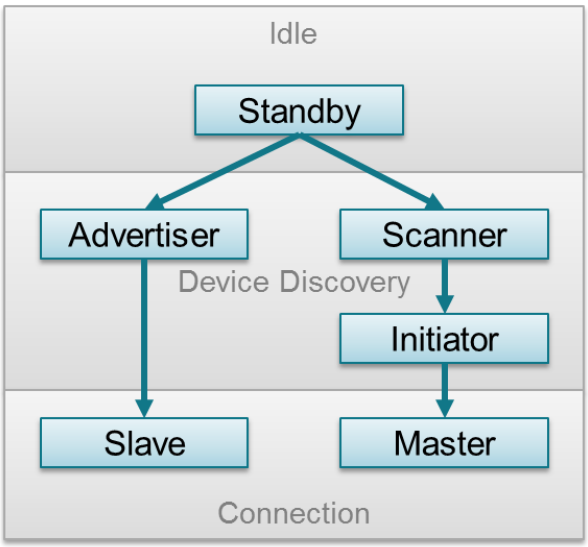


Figure 8: GAP State Diagram

Based on which role the device is configured for, the various states that a device can be in are depicted in Figure 8 and described below:

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

Connection Parameters

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

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

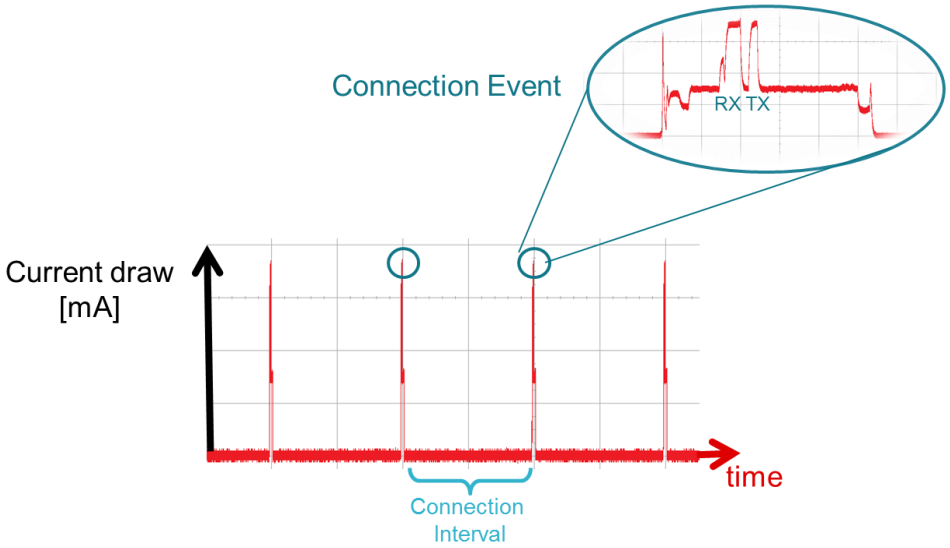


Figure 9: Connection Event and Interval

Different applications may require different connection intervals. As below, this will affect the device's power consumption. For more detailed information on power consumption, please see the power consumption application note [10].

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

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

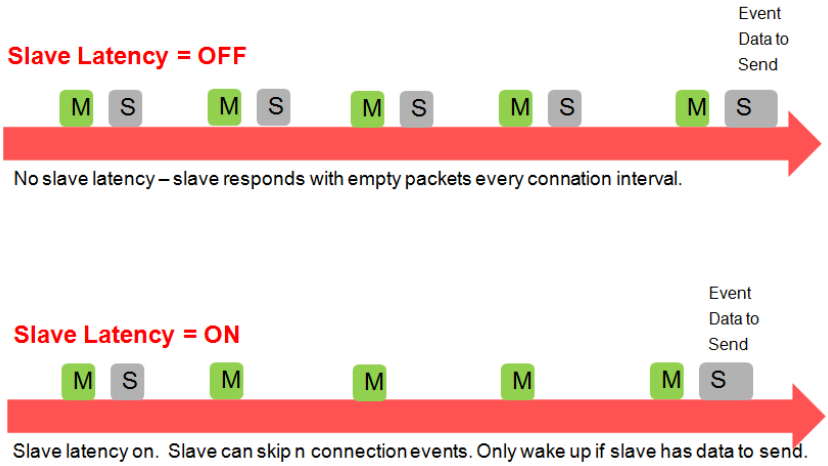


Figure 10: Slave Latency

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

Effective Connection Interval

The "effective connection interval" is equal to the amount of time between two connection events, assuming that the slave skips the maximum number of possible events if slave latency is allowed (the effective connection interval is equal to the actual connection interval if slave latency is set to zero). It can be calculated using the formula:

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

Consider the following example:

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

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

Connection Parameter Considerations

In many applications, the slave will skip the maximum number of connection events. Therefore it is useful to consider the effective connection interval when selecting or requesting connection parameters. Selecting the correct group of connection parameters plays an important role in power optimization of the BLE device. The following list gives a general summary of the trade-offs in connection parameter settings:

Reducing the connection interval will:

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

Increasing the connection interval will:

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

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

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

Increasing the slave latency will:

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

Connection Parameter Update

In some cases, the central device will request a connection with a peripheral device containing connection parameters that are unfavorable to the peripheral device. In other cases, a peripheral device might have the desire to change parameters in the middle of a connection, based on the peripheral application. The peripheral device can request the central device to change the connection settings by sending a "Connection Parameter Update Request." For BT4.0 devices, the L2CAP layer of the protocol stack handles the request.

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

Connection Termination

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

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

5.1.2 GAP abstraction

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

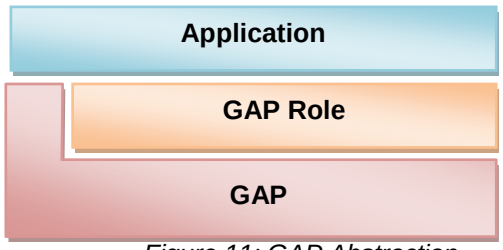


Figure 11: GAP Abstraction

Therefore, it is preferable and easier to configure the GAPRole module and use its API's to interface with the GAP layer. For this reason, the following GAP Layer section will only describe the functions and parameters that are not handled or configured through the GAPRole task and therefore must be modified directly through the GAP layer.

5.1.3 Configuring the GAP layer

The GAP layer functionality is mostly defined in library code. The function headers can be found in *gap.h*. As stated above, most of these functions are used by the GAPRole and will not need to be called directly. For reference, the GAP API is defined in Appendix I. There are several parameters which may be desirable to modify before starting the GAPRole. These can be set / get via the `GAP_SetParamValue()` and `GAP_GetParamValue()` functions and include advertising / scanning intervals, windows, etc (see the API for more information). As an example, here is the configuration of the GAP layer done in `SimpleBLEPeripheral_init()`:

```
// Set advertising interval
{
  uint16 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 seen in section 3.1, the GAPRole task is a separate task (GAPRole_ProcessEvent) which simplifies the application by handling most of the GAP layer functionality. It is enabled and configured by the application upon initialization. Based on this configuration, many BLE protocol stack events are handled directly by the GAPRole task and never passed to the application. However, there are callbacks which the application can register with the GAPRole task so that it can be notified of certain events and proceed accordingly.

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

- **Broadcaster** – an advertiser that is non-connectable
- **Observer** – scans for advertisements, but cannot initiate connections
- **Peripheral** – an advertiser that is connectable, and operates as a slave in a single link-layer connection.

- **Central** – scans for advertisements and initiates connections; operates as a master in a single or multiple link-layer connections. Currently, the BLE central protocol stack supports up to three simultaneous connections.

Furthermore, the BLE specification allows for certain combinations of multiple-roles, all of which are supported by the BLE protocol stack. See the wiki page [15] for sample projects of combo roles. Note that the CC254x does not support simultaneous peripheral / central functionality. This is supported by the CC2640. The peripheral and central roles will be described here.

5.2.1 Peripheral Role

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

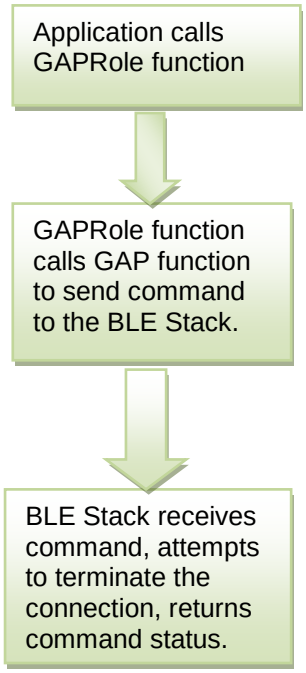
1. Initialize the GAPRole Parameters (see Appendix II.2). This should be done in the application initialization function, (i.e. `SimpleBLEPeripheral_init()`):

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

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

3. Send GAPRole commands as desired from the application. Here is an example of the application using `GAPRole_TerminateConnection()`.



simpleBLEPeripheral.c:

```
GAPRole_TerminateConnection();
```

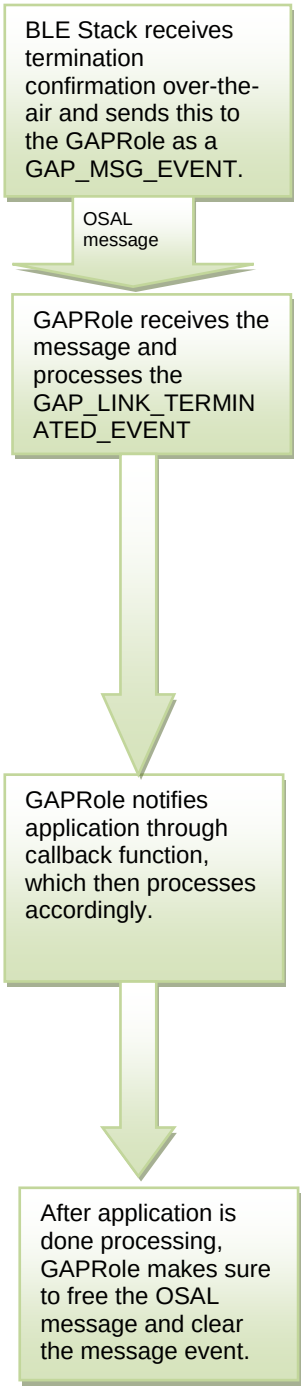
peripheral.c

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

Library Code

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

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



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_States_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*. The full API including commands, configurable parameters, events, and callbacks is described in Appendix III.

The general steps to use this module are:

1. Initialize the GAPRole Parameters if desired. These parameters are defined in Appendix III.2. This should be done in the application initialization function, (i.e. `SimpleBLECentral_init()`):

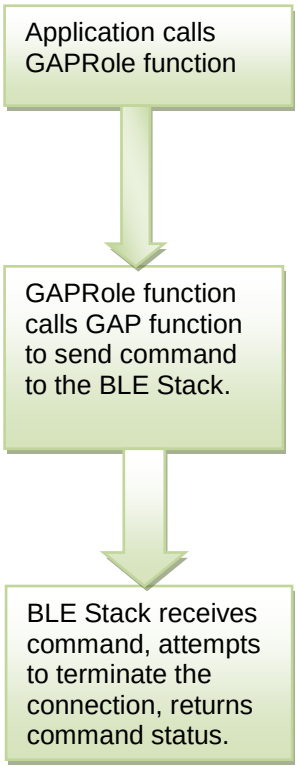
```
  // Setup GAP
uint8 scanRes = DEFAULT_MAX_SCAN_RES;
```

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

2. Initialize the GAPRole. This should be done when processing `START_DEVICE_EVT`. This involves passing function pointers to application callback functions. These callbacks are defined in Appendix III.3 .

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

3. Send GAPRole commands as desired from the application. Here is an example of the application using `GAPCentralRole_StartDiscovery()`.



simpleBLEcentral.c

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

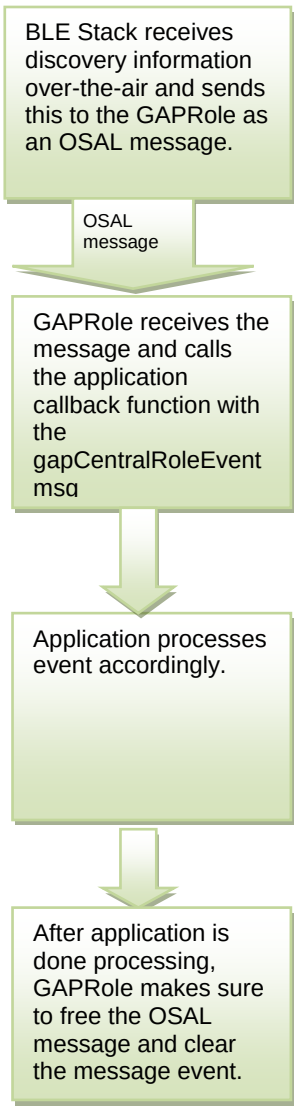
central.c

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

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

  return GAP_DeviceDiscoveryRequest(&params);
}
```

Library Code

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

4. The GAPRole task will process some of the GAP-related events passed to it from the BLE protocol stack. However, there are also some events which it will forward to the application. Here 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. Certain data may be readable or writeable only in an authenticated connection. It is important to define the terminology used in BLE security:

Term	Description
Pairing	The process of exchanging keys.
Encryption	Data is encrypted after pairing, or re-encryption (a subsequent connection where keys are looked up from non-volatile memory)
Authentication	The pairing process completed with MITM (Man in the Middle) protection (passcode, NFC, etc).
Bonding	Storing the encryption keys in non-volatile memory to use for the next encryption sequence.
Authorization	An additional application level key exchange in addition to authentication.
OOB	Out of Band. Keys are not exchanged over the air, but rather over some other source such as serial port or NFC. This also provides MITM protection.
MITM	Man in the Middle Protection. This prevents an attacker from listening to the

	keys transferred over the air to break the encryption.
Just Works	Pairing method where keys are transferred over the air without MITM

The general process to establish security is:

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

Note that it is not necessary to perform all of these steps. For example, it is possible to skip bonding and just re-pair upon each reconnection. Also note that the GAPBondMgr makes use of the SNV flash area for storing bond information. For more information on SNV, see Section 3.10.4.

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

Also note that the GAPBondMgr makes use of the SNV flash area for storing bond information. For more information on SNV, see Section 6.9.

5.3.1 Overview of BLE Security

This section will briefly describe BLE security methods. See the Bluetooth Spec [13] or the wiki page [15] for more information.

Once a connection is formed, the connected devices can go through a process called pairing. When pairing is performed, keys are established which encrypt and optionally authenticate the link. That is, either device may optionally require that a passkey is used to create in order to complete the pairing process. This is called authenticated pairing. This passcode could be a fixed value such as "000000" or it could be a randomly generated value that gets provided to the user (such as on a display). After the correct passkey is received, the two devices exchange security keys to encrypt and authenticate the link. The stated I/O capabilities of the two devices in the pairing request much match in a way that authentication is possible.

In many cases, the same central and peripheral devices will be regularly connecting and disconnecting from each other. BLE has a security feature that allows two devices, when pairing, to give each other a long-term set of security keys so that re-pairing is not needed upon re-connection. This feature, called bonding, allows the two devices to quickly re-establish encryption and authentication after re-connecting without going through the full pairing process every time that they connect as long as they store the long-term key information.

5.3.2 Using the GapBondMgr

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

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

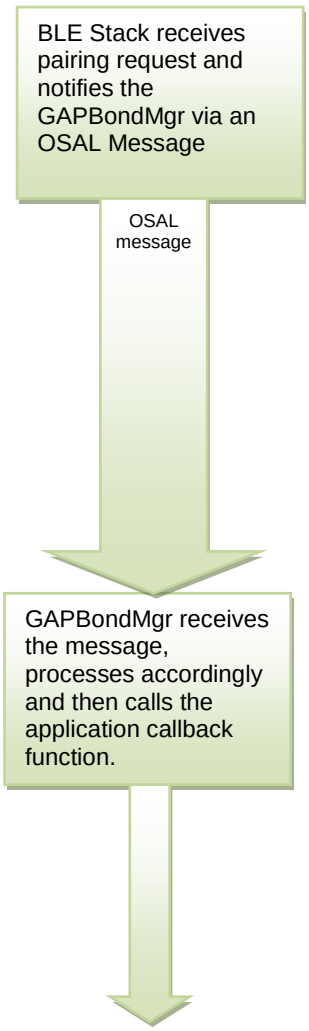
```
// Setup the GAP Bond Manager
{
  uint32 passkey = DEFAULT_PASSCODE;
  uint8 pairMode = GAPBOND_PAIRING_MODE_INITIATE;
  uint8 mitm = DEFAULT_MITM_MODE;
  uint8 ioCap = DEFAULT_IO_CAPABILITIES;
  uint8 bonding = DEFAULT_BONDING_MODE;
```

```
    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. This should be done after the GAPRole has been started 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 );
…
```

3. At this point, the GAPBondMgr is configured and, for the most part, will operate autonomously from the Application's perspective. For example, once a connection is established the GAPBondMgr will initiate pairing and bonding depending on the configuration parameters from step 1. There are a few parameters which can be set asynchronously at this point such as GAPBOND_ERASE_ALLBONDS. However, for the most part, all communication between the GAPBondMgr and the Application will occur through the callbacks which were registered in step 2. The following is a flow diagram example from SimpleBLECentral of the GAPBondMgr notifying the application that pairing has started. These callbacks will be expanded upon in the following sections.



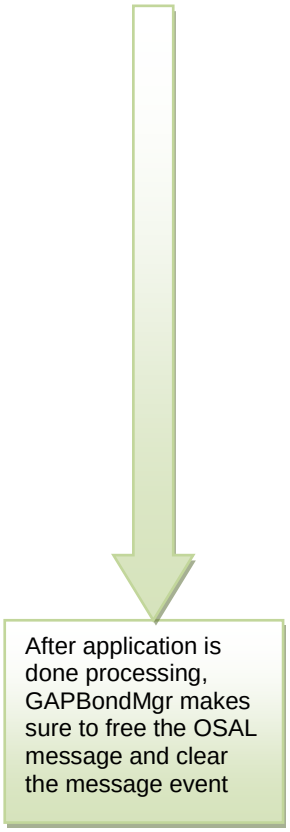
Library Code

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 will provide message diagrams for the various types of security that can be implemented. Note that these all assume acceptable I/O capabilities are present for the given security mode. See the Core Spec for more information [13] on how I/O 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 will automatically reject any attempt at pairing.

5.3.3.2 Just Works Pairing without Bonding

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

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

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

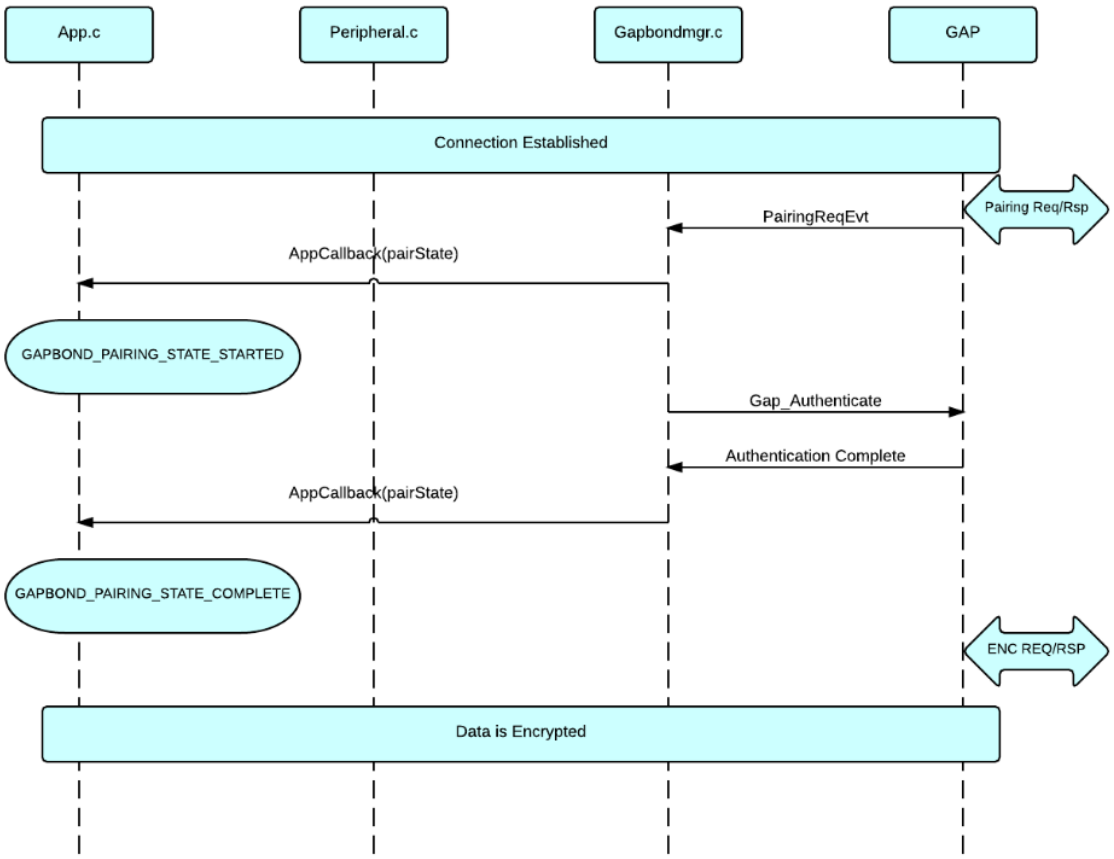


Figure 12: Just Works Pairing

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

5.3.3.3 Just Works Pairing with Bonding Enabled

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

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

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

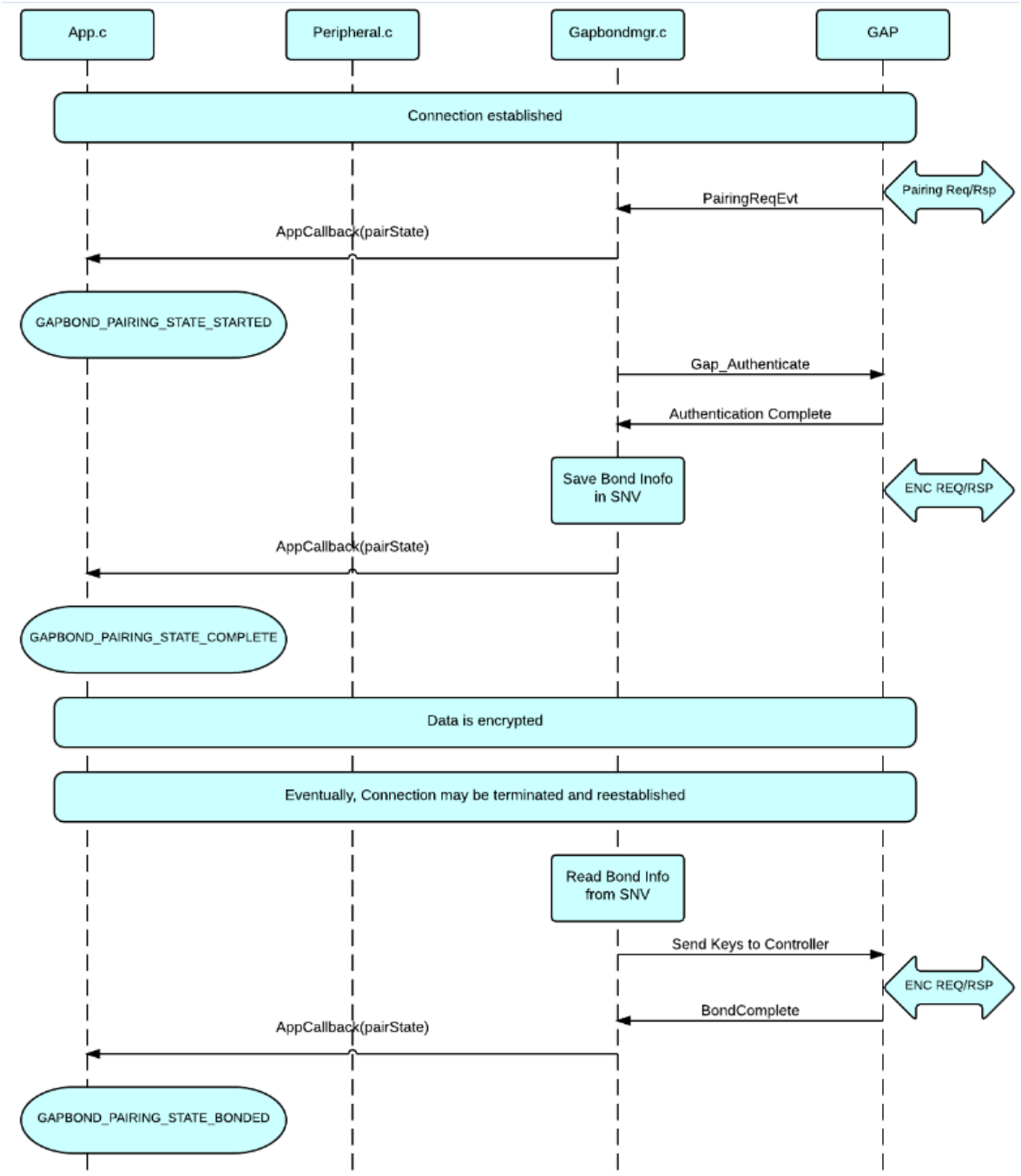


Figure 13: Bonding after Just Works Pairing

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

5.3.3.4 Authenticated Pairing

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

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

```
uint8 mitm = TRUE;
```

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

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

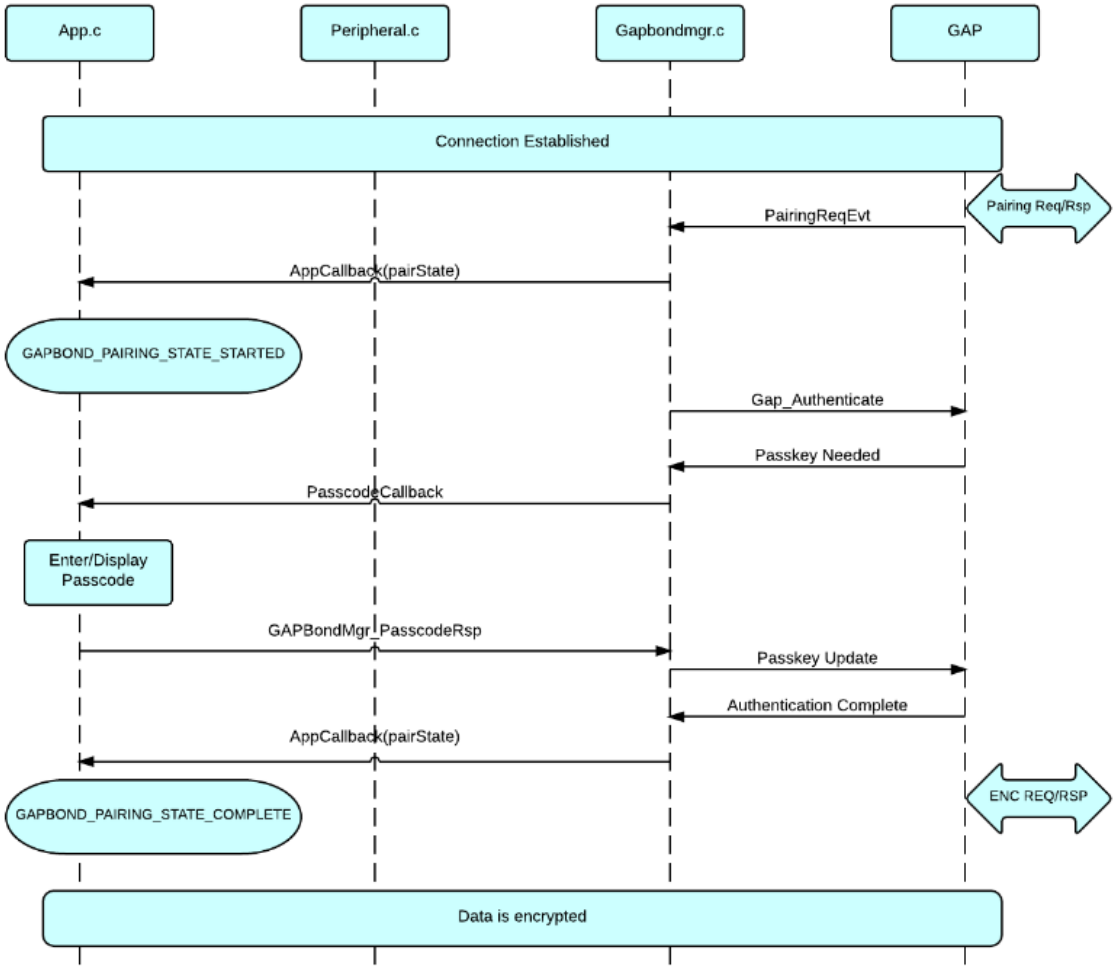


Figure 14: Pairing with MITM Authentication

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

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

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

```
static void 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 this example, a random password is created and displayed the password on an LCD screen. The other connected device must then enter this passcode.

5.3.3.5 Authenticated Pairing with Bonding Enabled

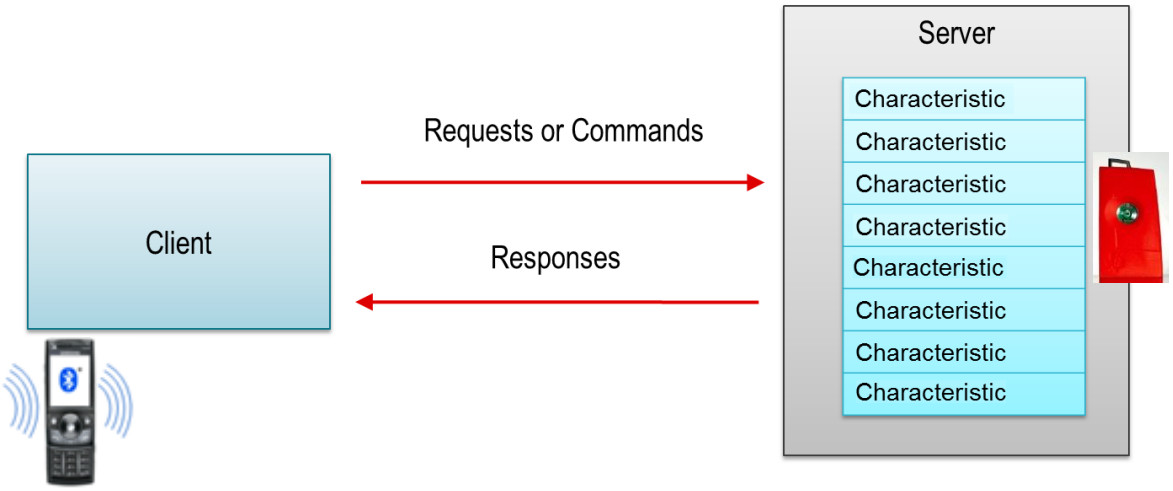
After pairing and encrypting with MITM authentication, bonding occurs in the same manner as in Section 5.3.3.3.

5.4 Generic Attribute Profile (GATT)

Whereas the GAP layer handles most of the connection-related functionality, the GATT layer of the BLE Protocol Stack is designed to be used by the application for data communication between two connected devices. Data is passed and stored in the form of characteristics which are stored in memory on the BLE device. From a GATT standpoint, when two devices are connected they are each in one of two roles:

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

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



is acting as the GATT client.

Figure 15: GATT Client and Server

While this is the typical case, it is important to note that the GATT roles of Client and Server are completely independent from the GAP roles of peripheral and central. A peripheral can be either a GATT Client or a GATT Server, and a central can be either a GATT client or a GATT Server. It is also possible to act as both a GATT Client and a GATT Server.

5.4.1 GATT Characteristics / Attributes

While they are sometimes used interchangeably when referring to BLE, it is helpful to consider "characteristics" as being composed of groups of information called "attributes." Attributes are the base information that is actually transferred between devices. Characteristics organize and use attributes as data values, properties & configuration information. A typical characteristic is composed of the following attributes:

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

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

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

5.4.2 GATT Services / Profile

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

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

- **Mandatory GAP Service** – This service contains device and access information, such as the device name, vendor identification, and product identification, and is a part of the BLE protocol stack. It is required for every BLE device as per the BLE specification. The source code for this service is not provided as it is built into the Stack library.
- **Mandatory GATT Service** – This service contains information about the GATT server and is a part of the BLE protocol stack. It is required for every GATT server device as per the BLE specification. The source code for this service is not provided as it is built into the Stack library.
- **Device Info Service** – This service exposes information about the device such as the hardware, software and firmware version, regulatory & compliance info, and manufacturer name. The Device Info Service is provided as part of the BLE protocol stack and configured by the Application. See [14] for more information.
- **SimpleGATTProfile Service** – This service is a sample profile that is provided for testing and for demonstration. The full source code is provided in the files *simpleGATTProfile.c* and *simpleGATTProfile.h*.

The portion of the attribute table in the SimpleBLEPeripheral project corresponding to the simpleGATTProfile service is shown and described below. This section is meant as an introduction to the attribute table. For information on how this profile is implemented in the code, see Section 5.4.4.2.

Handle	Uuid	Uuid Description	Value	Properties
0x001F	0x2800	GATT Primary Service Declaration	F0:FF	
0x0020	0x2803	GATT Characteristic Declaration	0A:21:00:F1:FF	
0x0021	0xFFF1	Simple Profile Char 1	01	Rd Wr 0x0A
0x0022	0x2901	Characteristic User Description	Characteristic 1	
0x0023	0x2803	GATT Characteristic Declaration	02:24:00:F2:FF	
0x0024	0xFFF2	Simple Profile Char 2	02	Rd 0x02
0x0025	0x2901	Characteristic User Description	Characteristic 2	
0x0026	0x2803	GATT Characteristic Declaration	08:27:00:F3:FF	
0x0027	0xFFF3	Simple Profile Char 3		Wr 0x08
0x0028	0x2901	Characteristic User Description	Characteristic 3	
0x0029	0x2803	GATT Characteristic Declaration	10:2A:00:F4:FF	
0x002A	0xFFF4	Simple Profile Char 4		Nfy 0x10
0x002B	0x2902	Client Characteristic Configuration	00:00	
0x002C	0x2901	Characteristic User Description	Characteristic 4	
0x002D	0x2803	GATT Characteristic Declaration	02:2E:00:F5:FF	
0x002E	0xFFF5	Simple Profile Char 5		Rd 0x02
0x002F	0x2901	Characteristic User Description	Characteristic 5	

Figure 16: Simple GATT Profile Characteristic Table from BTool

The simpleGATTProfile contains five characteristics:

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

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

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

    The value of 0x0A means the characteristic is readable (0x02) and writeable (0x08)
  - **Bytes 1-2:** the byte-reversed handle where the SimpleProfileChar1 value is located (handle 0x0021).

- **Bytes 3-4:** the UUID of the SimpleProfileChar1 value (custom defined 0xFFF1).

- **0x0021:** this is the SimpleProfileChar1 value. It has a UUID of 0xFFF1 (custom-defined). Its value is the actual payload data of the characteristic. As indicated by its characteristic declaration (handle 0x0020), it is readable and writeable.
- **0x0022:** this is the SimpleProfileChar1 user description. It has a UUID of 0x2901 (Bluetooth-defined). Its value is a user-readable string describing the characteristic.
- **0x0023** – **0x002F:** These attributes follow the same structure as the simpleProfileChar1 described above with regard to the remaining four characteristics. The only different attribute, handle 0x002B, is described below
- **0x002B**: this is the SimpleProfileChar4 client characteristic configuration. It has a UUID of 0x2902 (Bluetooth-defined). By writing to this attribute, a GATT server can configure the SimpleProfileChar4 for notifications (writing 0x0001) or indications (writing 0x0002). Writing a 0x0000 to this attribute will disable notifications / indications.

5.4.3 GATT Client abstraction

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

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

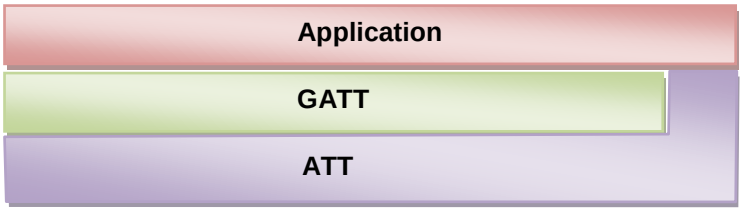


Figure 17: GATT Client Abstraction

5.4.3.1 Using the GATT Layer Directly

This section will describe how to directly use the GATT layer in the Application. The functionality of the GATT layer is implemented in the library code but header functions can be found in *gatt.h*. The complete API for the GATT layer can be found in Appendix IV. More information on the functionality of these commands can be found in the Bluetooth spec [13]. As described above, these functions will be used primarily for GATT client applications. There are a few server-specific functions which are described in the API and not considered here. Note that most of the GATT functions will return ATT events to the application so it also necessary to consider the ATT API in Appendix IV. The general procedure to use the GATT layer when functioning as a GATT Client (i.e. in the SimpleBLECentral project) is as follows:

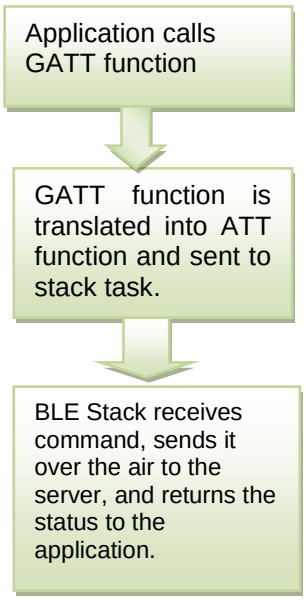
1) Initialize the GATT Client. This should be done in the application initialization function.

```
VOID GATT_InitClient();
```

2) Register to receive incoming ATT Indications / Notifications. This should also be done in the application initialization function.

```
GATT_RegisterForInd(selfEntity);
```

3) Perform a GATT Client procedure. The example here will use `GATT_WriteCharValue()`, which is triggered by a left key press in the SimpleBLECentral application.



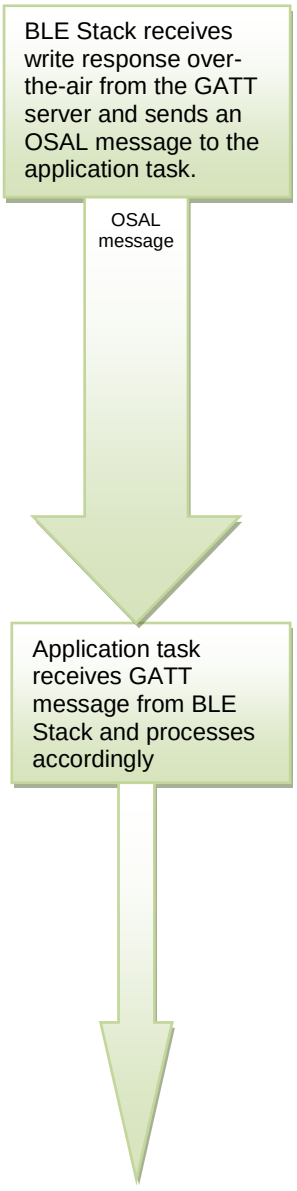
simpleBLECentral.c:

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

Library code

Library Code

4) Receive and handle the response to the GATT Client procedure in the application. In this example, the application will be receiving an `ATT_WRITE_RSP` event. See Appendix IV.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:

After application is done processing, it frees the OSAL message and clears the event.

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

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

Note that even though the event sent to the application is an ATT event, it is still sent as a GATT protocol stack message (`GATT_MSG_EVENT`)

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

5.4.4 GATT Server Abstraction

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

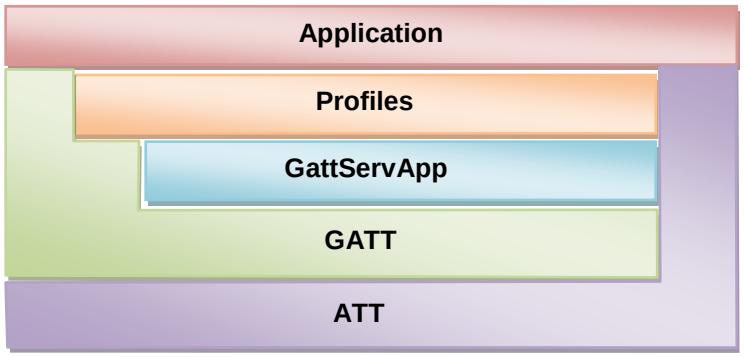


Figure 18: GATT Server Abstraction

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

5.4.4.1 GATTServApp Module

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

5.4.4.1.1 Building up the Attribute Table

Upon power-on / reset, the Application builds the GATT table by using the GATTServApp to add services. Each service consists of a list of attributes with UUIDs, values, permissions, and r/w call-backs. As shown in Figure 19, all of this information is passed through the GATTServApp to GATT and stored in the stack.

This should be done in the application initialization function, i.e. `simpleBLEPeripheral_init()`:

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

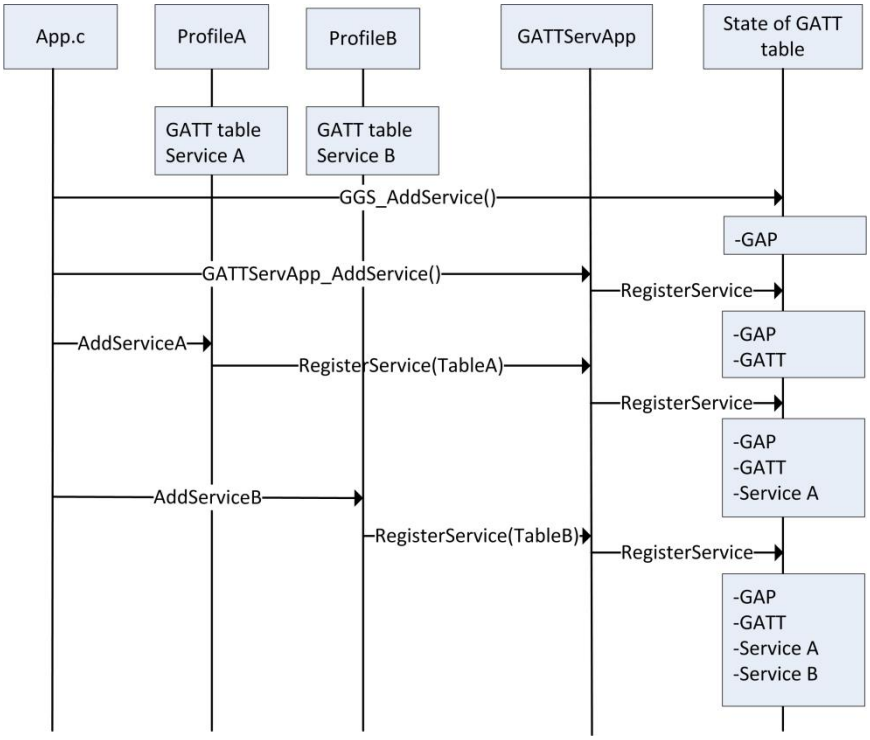


Figure 19: Attribute Table Initialization

5.4.4.2 Profile Architecture

This section will describe the general architecture for all profiles and provide specific functional examples in relation to the simpleGATTProfile in the SimpleBLEPeripheral project. Please refer to Section 5.4.2 for an overview of the simpleGATTProfile.

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

5.4.4.2.1 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:

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

Each attribute in this table is of the type:

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

```
                      //!< value shall be 512 octets.
} gattAttribute_t;
```

The specific elements of this attribute type are detailed here:

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

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

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

- `permissions` – this enforces how/if a GATT client device can access the attribute's value. Possible permissions are defined in *gatt.h* as:
  - `GATT_PERMIT_READ` // Attribute is Readable
  - `GATT_PERMIT_WRITE` // Attribute is Writable
  - `GATT_PERMIT_AUTHEN_READ` // Read requires Authentication
  - `GATT_PERMIT_AUTHEN_WRITE` // Write requires Authentication
  - `GATT_PERMIT_AUTHOR_READ` // Read requires Authorization
  - `GATT_PERMIT_AUTHOR_WRITE` // Write requires Authorization
  - `GATT_PERMIT_ENCRYPT_READ` // Read requires Encryption
  - `GATT_PERMIT_ENCRYPT_WRITE` // Write requires Encryption

  Authentication, authorization, and encryption are further described in Section 5.3.

- `handle` – This is a placeholder in the table where GATTServApp will assign a handle. This is not settable by the user. Handles will be assigned sequentially.

- `pValue` – This is a pointer to the attribute value. Note that the size can't be changed after initialization. Max size is 512 octets.

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

5.4.4.2.1.1 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 */
  },
```

The `type` is set to the Bluetooth SIG-defined "primary service" UUID (0x2800).

A GATT client will need to read this so the `permission` is set to `GATT_PERMIT_READ`.

The `pValue` is a pointer to the service's UUID, custom-defined as 0xFFF0:

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

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

The value of a characteristic declaration was described in Section 5.4.1. For functional purposes here, the only information that needs to be passed to the GATTServApp in `pValue` is a pointer to the characteristic value's properties. The GATTServApp will take care of adding the UUID and the handle of the value. These properties are defined as:

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

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

5.4.4.2.1.3 Characteristic Value

Consider the simpleGATTProfile simpleProfileCharacteristic1 value.

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

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

Because, in the characteristic declaration, it is stated that this characteristic value's properties are readable and writeable, it is necessary to actually set the GATT `permissions` to readable and writeable. ***Note that these two fields must match. The stack does not perform any checking here and errors will occur if they don't match.

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

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

5.4.4.2.1.4 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 will need to read and write to this so the GATT `permissions` are set to readable and writeable.

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

```
static gattCharCfg_t *simpleProfileChar4Config;
```

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

5.4.4.2.2 Add Service Function

As described in Section 5.4.4.1, when an application starts up it will need to add the GATT services it supports. Therefore, each profile needs a global AddService function which can be called from the application. Some of these services are defined in the protocol stack such as GGS_AddService and GATTServApp_AddService. User defined services have to expose their

own AddService function which the application can call for profile initialization. Using `SimpleProfile_AddService()` as an example, these functions should:

- **Allocate space for the client characteristic configuration (CCC) arrays**. As an example, a pointer to one of these arrays was initialized in the profile as described in Section 5.4.4.2.1.4. Here, in the AddService function, is where the number of supported connections is declared and memory is allocated for each array. Note that there is only one CCC defined in the simpleGATTProfile but it is common for there to be multiple CCC's.

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

- **Initialize the CCC arrays**. CCC values are persistent between power downs and between bonded device connections because they are stored in NV. For each CCC in the profile, the `GATTServApp_InitCharCfg()` function must be called. This function will attempt to initialize the CCC's with information from a previously bonded connection and, if not found, set the initial values to default values.

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

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

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

5.4.4.2.3 Register Application Callback Function

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

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

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

Where the callback typedef is defined as:

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

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

The application must then define a callback of this type and pass it to the simpleGATTProfile with the `SimpleProfile_RegisterAppCBs()` function. This is done in *simpleBLEPeripheral.c* via:

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

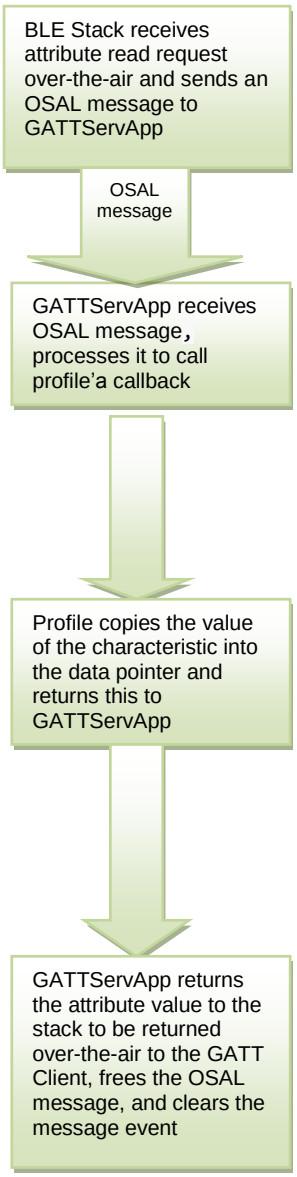
See the next section for the mechanism of how this callback is used.

5.4.4.2.4 Read/Write Callback Functions

The profile must define Read and Write callback functions which the protocol stack will call when one of the profile's attributes are written to / read from. The callbacks must be registered with GATTServApp as mentioned in Section 5.4.4.2.2. These callbacks will perform the characteristic read/write and other processing (possibly calling an application callback) as defined by the specific profile.

Read Request from Client

When a read request from a GATT Client is received for a given attribute, the protocol stack will check the attribute's permissions and, if the attribute is readable, call the profile's read call-back. It is up to the profile to copy in the value, perform any profile-specific processing, and optionally notify the application. This is illustrated in the following flow diagram for a read of simpleprofileChar1 in the simpleGATTProfile.



Library Code

Library Code

simpleGATTProfile.c

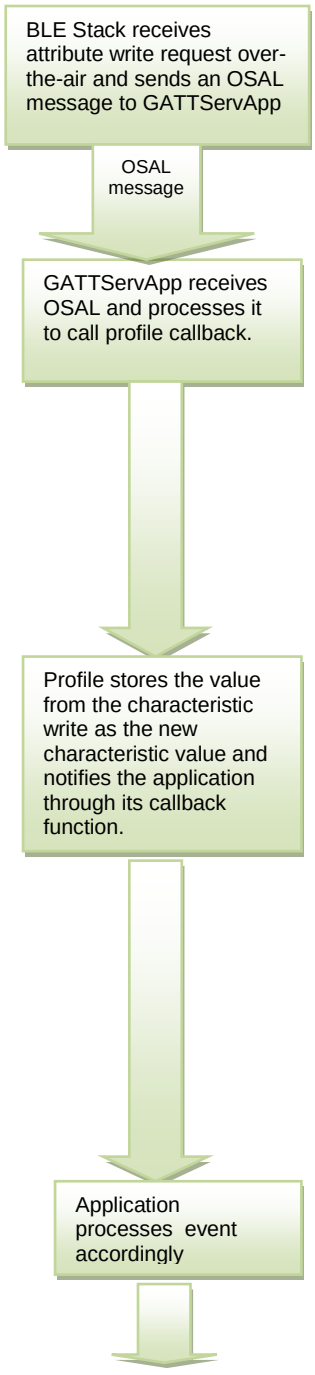
```
static bStatus_t simpleProfile_ReadAttrCB(......)
{
…
  if ( pAttr->type.len == ATT_BT_UUID_SIZE )
  {
    // 16-bit UUID
    uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
      >type.uuid[1]);
    switch ( uuid )
    {
      case SIMPLEPROFILE_CHAR1_UUID:
        *pLen = 1;
        pValue[0]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

Library Code

Note that all of the processing here is in the context of the protocol stack. If there is any intensive profile related processing that needs to be done in the case of an attribute read, this should be split up and done in the context of the Application task. See the following write request for more information.

Write Request from Client

When a write request from a GATT Client is received for a given attribute, the protocol stack will check the attribute's permissions and, if the attribute is write, call the profile's write call-back. It is up to the profile to store the value to be written, perform any profile-specific processing, and optionally notify the application. This is illustrated in the following flow diagram for a write of simpleprofileChar3 in the simpleGATTProfile. Red corresponds to processing in the protocol stack context and green is processing in the application context.



Library Code

Library Code

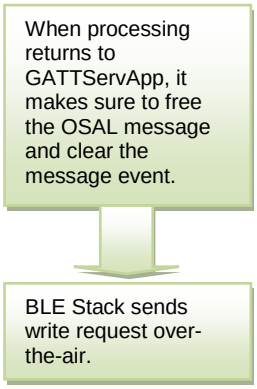
simpleGATTProfile.c

```
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

As stated previously, it is important to minimize the processing done in the stack task. If extensive additional processing beyond storing the attribute write value in the profile is needed, an application should be set so that processing can complete in the application task.

5.4.4.2.5 Get / Set Functions

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

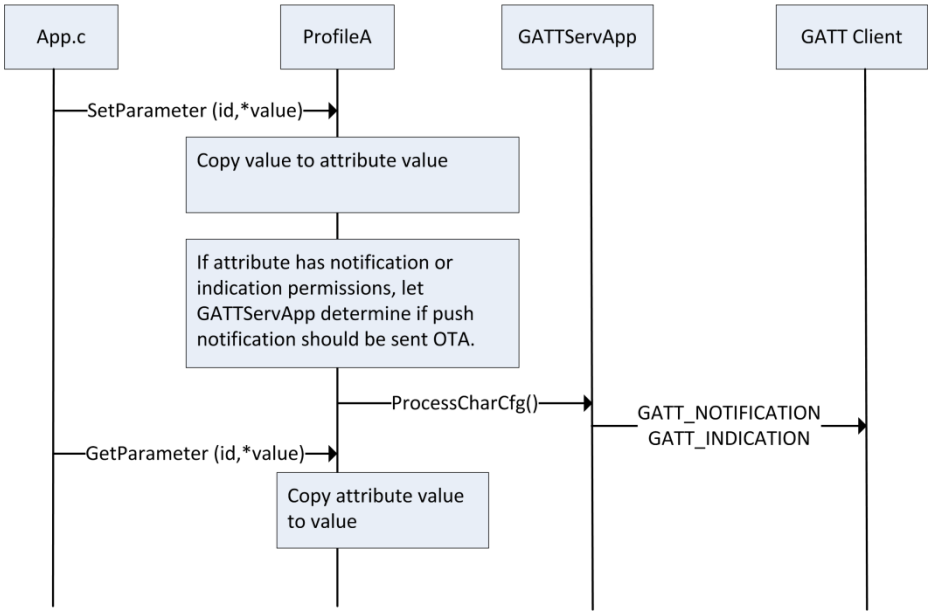


Figure 20: Get / Set Profile Parameter

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

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

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

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

```
      if ( len == sizeof ( uint8 ) )
      {
        simpleProfileChar4 = *((uint8*)value);

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

5.5 L2CAP

The L2CAP layer sits on top of the HCI layer and transfers data between the upper layers of the host (GAP, GATT, Application, etc) and the lower layer protocol stack. This layer is responsible for protocol multiplexing capability and reassembly operation for data exchanged between the host and the protocol stack. L2CAP permits higher level protocols and applications to transmit and receive upper layer data packets (L2CAP Service Data Units, SDU) up to 64 kilobytes in length. The actual size is limited by the amount of memory available on the specific device implementation. CC254x and the 1.4.1 stack only support an effective MTU size of 23, set by `L2CAP_MTU_SIZE` in l2cap.h. This define cannot be modified without breaking the stack.

5.6 HCI

The HCI layer is a thin layer which transports commands and events between the Host and Controller. In a network processor application, the HCI layer is implemented through a transport protocol such as SPI or UART. In embedded SOC projects, as are considered in this document, the HCI layer is implemented through function calls and callbacks. All of the commands and events discussed so far, such as ATT, GAP, etc, pass from the given layer through the HCI layer to the controller and vice versa.

5.6.1 HCI Extension Vendor Specific Commands

There are a number of HCI Extension Vendor Specific Commands which extend some of the functionality of the controller for use by the application / host. Refer to Appendix VII for a description of available HCI Extension Commands and examples for use in an SOC project.

5.6.2 Receiving HCI Extension Events in the Application

Similar to the GAP and ATT layers, HCI Extension Commands will result in HCI Extension Events being passed from the controller to the host. There are some additional steps needed in order to receive these events in the application for processing.

By default, the GAPRole task registers to receive HCI Extension events in `gapRole_init()`:

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

Therefore, it will receive all events related to HCI Extension commands, even if they are called from the application. Because of this, it is necessary to implement a callback function to pass these events from the GAPRole task back to the application if they are needed in the application.

5.7 Library Files

Each project must include two library files:

- BLE Stack Library: this library includes the lower-layer stack functionality and varies based on the GAP role. It is always possible to include the full library but it usually preferable to use a smaller subset in order to conserve code space. Note that this library will be different for a CC2540 and CC2541 project.
- HCI Transport Layer Library: this library includes transport layer functionality in the case of a network processor. This library will be the same for CC2540 and CC2541

The library files are located at $INSTALL$\Projects\ble\Libraries. The table below can be used as a reference to determine the correct library file to use in the project:

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

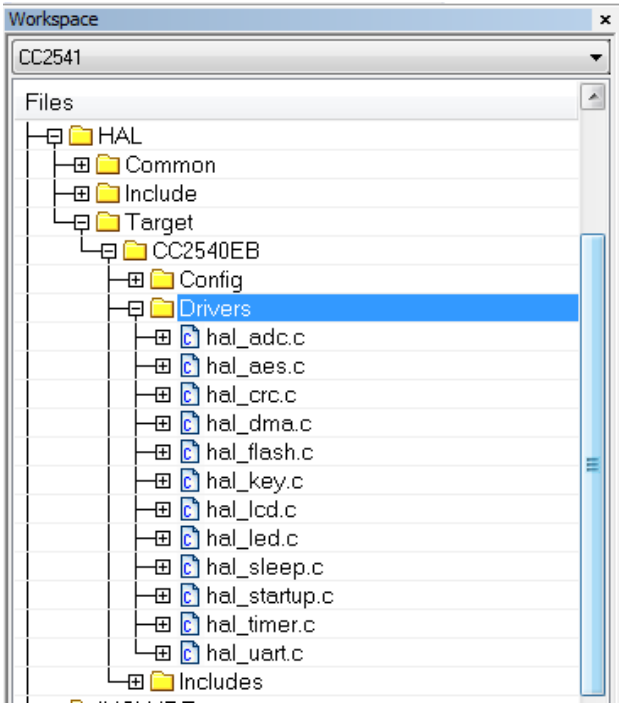
6 Drivers

The Hardware Abstraction Layer (HAL) of the CC254x software provides an interface of abstraction between the physical hardware and the application / protocol stack. This 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 LED's. The HAL drivers that are provided support the following hardware platforms:

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

When developing with a different hardware platform, it might be necessary to modify the HAL source for compatibility.

The HAL files can be found in the sample projects under HAL→Target→CC2540EB→Drivers:



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. There are also several ADC examples in the Peripheral Examples Package [11].

6.2 AES

To include the AES driver, define HAL_AES=TRUE. AES is needed by the stack for encryption so this symbol must always be set to TRUE. The AES driver uses DMA channels 1 and 2 so these cannot be reused. This is set 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
```

The AES API can be found in *hal_ase.h* but it is generally easier for the application to use the HCI encrypt / 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. The driver is designed to function with the LCD on the SmartRF05 boards. Therefore, it will be necessary to modify parts of the driver such as the pin and port definitions in *hal_lcd.c* to use it on custom hardware. See the SimpleBLEPeripheral project, CC2540/1 configuration, for an example of using the LCD driver.

6.4 LED

To include the LED driver, define HAL_LED=TRUE in the preprocessor definitions. The driver is designed to function with the LED's on the Keyfob. Therefore, it will be necessary to modify parts

of the driver such as the LED bit definitions in *hal_led.c* to use it on custom hardware. See the SimpleBLEPeripheral project, CC2540/1DK-miniKeyfob for an example of 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. The driver is designed to function with the buttons on the keyfob or SmartRF05 Board depending on whether CC2540_MINIDK is defined. Therefore, it will be necessary to modify parts of the driver such as the port and pin definitions in *hal_key.c* to use it on custom hardware. See the SimpleBLEPeripheral project for an example of using the KEY driver.

6.6 DMA

To include the DMA driver, define HAL_DMA=TRUE in the preprocessor definitions. Because the DMA driver is used by the AES driver, it must always be included in the project. As stated in section 6.2, channels 1 and 2 are reserved for use by the AES driver. Furthermore, other channels will be used if the UART DMA or SPI driver is being used. The DMA API can be found in *hal_dma.h*. See the SPI driver for an example of using the DMA driver.

6.7 UART / SPI

Describing the UART and SPI drivers is out of the scope of this document. Please see the wiki page [15] for several UART and SPI examples, specifically the "BLE Serial Bridge."

6.8 Other Peripherals

For hardware peripherals that don't have drivers, such as the timers, see the CC2541/43/44/45 Peripherals Software Examples [11] for examples. The hardware peripherals are explained and defined in the chip user's guide [12].

6.9 Simple NV (SNV)

The SNV area of flash is used for storing persistent data in a secure manner, such as encryption keys from Bonding or to store custom parameters. The Protocol Stack reserves two 2 kB flash pages for SNV. By default, these are the last two pages of flash. To minimize the number of erase cycles on the flash, the SNV manager performs "compactions" on the flash sector(s) when the sector has 80% invalidated data. A compaction is the copying of valid data to a temporary area followed by an erase of the sector where the data was previously stored. Note that the SNV driver makes use of the hal_flash driver.

SNV can be read from / written to using the following API's:

uint8 osal_snv_read(osalSnvId_t id, osalSnvLen_t len, void *pBuf)
Description: Read data from NV. Parameters: *id* – valid NV item *len* – Length of data to read. *pBuf* – pointer to buffer to store data read. Returns: SUCCESS: NV item read successfully NV_OPER_FAILED: failure reading NV item

uint8 osal_snv_write(osalSnvId_t id, osalSnvLen_t len, void *pBuf)
Description: Write data to NV. Parameters: *id* – valid NV item *len* – Length of data to write. *pBuf* – pointer to buffer containing data to be written.

Returns:
SUCCESS: NV item read successfully
NV_OPER_FAILED: failure reading NV item

Since SNV is shared with other modules in the BLE SDK such as the GAPBondMgr, it is necessary to carefully manage the NV item ID's. By default, the ID's available to the customer are defined in *bcomdef.h:*

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

7 Creating a Custom BLE Application

By now the BLE system designer should have a firm grasp on the general system architecture, Application & BLE Stack framework required to implement a custom Bluetooth Smart application. This section provides indications on where and how to start writing a custom Application and some considerations.

7.1 Configuring the BLE Stack

First, it's required to decide what role and purpose the custom Application should have. If it's an application that is tied to a specific Service or Profile, it makes sense to start there. One example is the **Heart Rate** sensor project. It's generally a good idea to start with one of the SimpleBLE sample projects:

- SimpleBLECentral
- SimpleBLEPeripheral
- SimpleBLEBroadcaster
- SimpleBLEObserver

After making this decision, the appropriate libraries can be chosen as described in section 5.7.

7.2 Define BLE behavior

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

7.3 Define Application task(s)

The application should contain callbacks from the various stack layers and event handlers to process OSAL messages. If necessary, other tasks can be added ensuring to follow the guidelines from section 3.

7.4 Configure Hardware Peripherals

If desired, add drivers as specified in section 6. If drivers do not exist for a given peripheral, a custom driver must be created.

7.5 Configuring Parameters for Custom Hardware

There are several software parameters which must be adjusted when working with custom hardware.

7.5.1 Board File

The board file (*hal_board_cfg.h*) can be found in the sample projects under HAL→Target→CC2540EB→Config. Depending on the hardware platform (keyfob, EM, etc), the project will contain a different board file. These board files are specific to the given platform and must be adjusted for custom hardware. Some modifications may include:

- modifying the symbols used by drivers for specific pins (i.e. `LED1_SBIT`)

- selecting the 32 kHz oscillator source(`OSC_32KHZ`)
- initializing I/O pins to safe initialization levels to prevent current leakage

7.5.2 Adjusting for 32 MHz Crystal Stabilization Time

Before entering sleep, the sleep timer is set to wake before the next BLE event. The closer the wakeup is to the event, the less power is wasted. However, if the wakeup is too close, it is possible to miss the event. Upon wake, it is necessary to wait for the 32MHz external crystal to stabilize. This stabilization time is not deterministic and is affected by the crystal's inherent stabilization time time, how long the crystal has been off, the temperature, the voltage, etc. Therefore, it is necessary to add a buffer to the wakeup time (i.e. start earlier) in order 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. The larger the value of the definition, the larger the buffer time (and thus more power wasted). This value is generally calculated empirically. If `HAL_SLEEP_ADJ_TICKS` is set too low, there will be spurious advertisement restarts and connection drops. In this case, the definition should be increased until these are no longer seen.

7.5.3 Adjusting For 32 kHz Stabilization Time

There must be enough time before entering sleep in order to allow the external 32 kHz crystal to stabilize. This value can be set using the `HCI_EXT_DelaySleepCmd()` during OSAL task initialization. See Appendix VIII for more information. If this command is never used, the default delay is 400ms on the CC254x. The default value can be modified with the `MIN_TIME_TO_STABLE_32KHZ_XOSC` in *hal_sleep.c*.

7.5.4 Setting the Sleep Clock Accuracy

If necessary to modify the sleep clock accuracy from the default (50 ppm for a master, 40 ppm for a slave), the `HCI_EXT_SetSCACmd()` can be used. See Appendix VIII for more information.

7.6 Software Considerations

7.6.1 Memory Management for GATT Notifications / Indications

The preferred method to send a GATT notification / indication is to use a profile's SetParameter function (i.e. `SimpleProfile_SetParameter()`) and call `GATTServApp_ProcessCharCfg()`. If using `GATT_Notification()` or `GATT_Indication()` directly, there is additional memory management needed:

1. Attempt to allocate memory for the notification / indication using `GATT_bm_alloc()`.
2. If allocation succeeds, send notification / indication using `GATT_Notification()` / `GATT_Indication()`.
3. If the return value of the notification / indication is `SUCCESS` (0x00), this means the memory was freed by the stack. If the return value something other than `SUCCESS` (i.e. `blePending`), free the memory using `GATT_bm_free()`.

There is an example of this in the `gattServApp_SendNotiInd()` function in *gattservapp_util.c*:

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

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

  if ( status != SUCCESS )
```

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

7.6.2 Limit Application Processing During BLE Activity

Because of the time-dependent nature of the BLE protocol, the controller (`LL_ProcessEvent`()) needs to process before each connection event / advertising event. If it does not get to process, advertising will restart or the connection will drop. Since OSAL is not multi-threaded, each task needs to ensure to stop processing in order to allow the controller to process. This will never be an issue with the stack layers. However, it is up to the user to ensure that the application does not process longer than:

(connection / advertising interval) – 2 ms

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

7.6.3 Global Interrupts

During BLE activity, the controller will need to process radio and MAC timer interrupts in a timely manner in order to set up the BLE event post-processing. Therefore, interrupts should never be globally disabled during BLE activity.

8 Development and Debugging

All embedded software for the CC2540/41 is developed using *Embedded Workbench for 8051* 9.10.3 from IAR Software. This section provides information on where to find this software. It also contains some basics on the usage 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 that go beyond the scope of this document. More information and documentation can be found on IAR's website: www.iar.com.

8.1 IAR Overview

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

1. **Download IAR Embedded Workbench 30-day Evaluation Edition** – This version of IAR is completely free of charge and has full functionality; however it is only a 30-day trial. It includes all of the standard features.

   IAR 30-day Evaluation Edition can be downloaded from the following URL:

   http://supp.iar.com/Download/SW/?item=EW8051-EVAL

2. **Purchase the full-featured version of IAR Embedded Workbench** – For complete BLE application development using the CC2540/41, it is recommended that you purchase the complete version of IAR without any restrictions.

   Information on purchasing the complete version of IAR can be found at the following URL:

   http://www.iar.com/en/Products/IAR-Embedded-Workbench/8051/

8.2 Using IAR Embedded Workbench

After installing IAR Embedded Workbench, be sure to download all of the latest patches from IAR, as they will be required in order to build and debug projects with the CC2540/41.

Once all of the patches have been installed, you are ready to develop software for the CC2540/41. This section will describe how to open and build an existing project for a CC2540.

Similar steps apply for a CC2541. The SimpleBLEPeripheral project, which is included with the Texas Instruments BLE software development kit, is used as an example.

8.2.1 Open an Existing Project

First, you must start the IAR Embedded Workbench IDE. When using Windows, this is typically done by clicking **Start > Programs > IAR Systems > IAR Embedded Workbench for 8051 9.10 > IAR Embedded Workbench**.

Once IAR has opened up, click **File > Open > Workspace**. Select the following file:

$INSTALL$\Projects\ble\SimpleBLEPeripheral\CC2540DB\SimpleBLEPeripheral.eww

This is the workspace file for the SimpleBLEPeripheral project. Once it is selected all of the files associated with the workspace should open up as well, with a list of files on the left side.

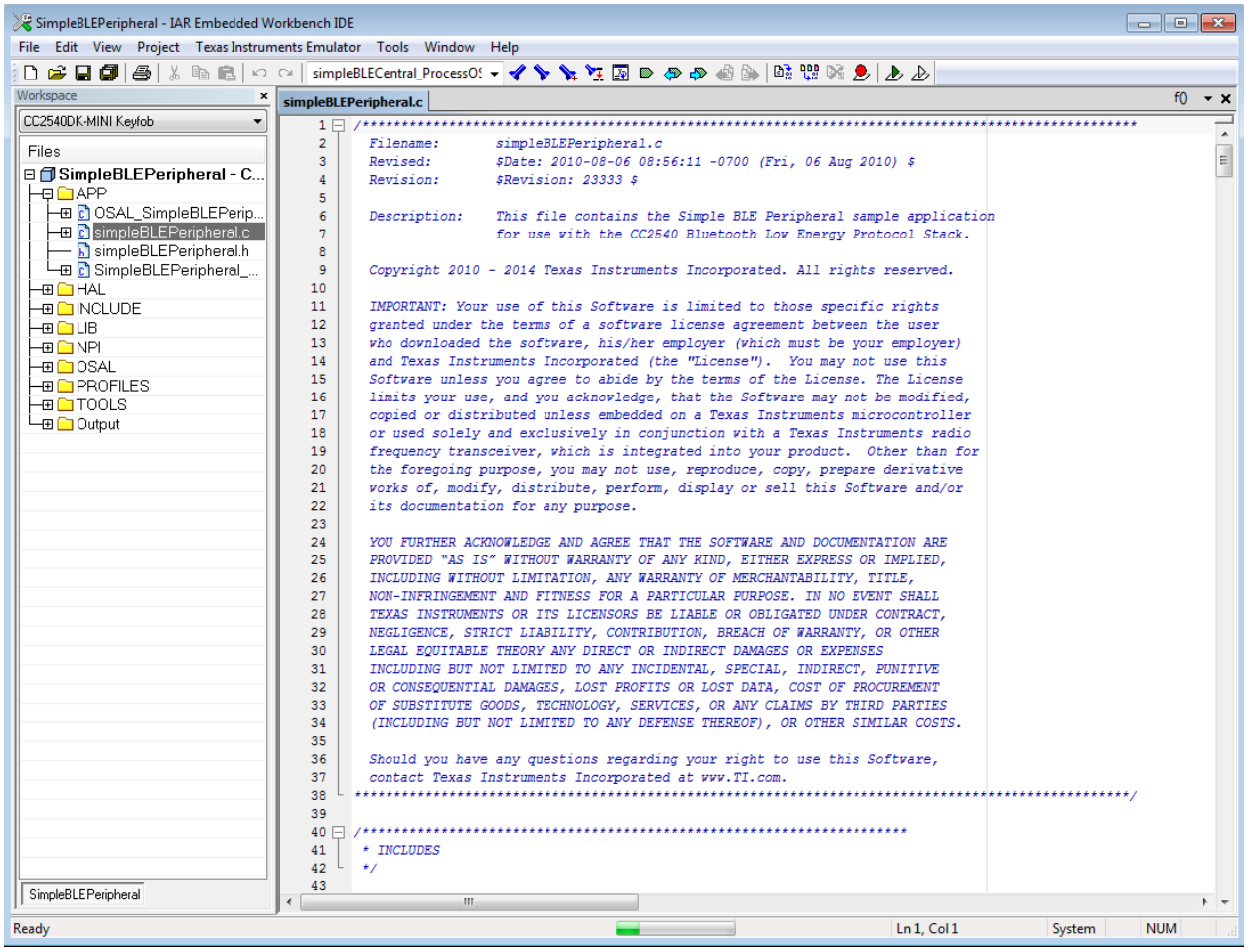


Figure 21: IAR Embedded Workbench

8.2.2 Project Options, Configurations, and Defined Symbols

Every project will have a set of options, which include settings for the compiler, linker, debugger, and more. To view the project options, right click on the project name at the top of the file list and select "Options…" as shown in Figure 22.

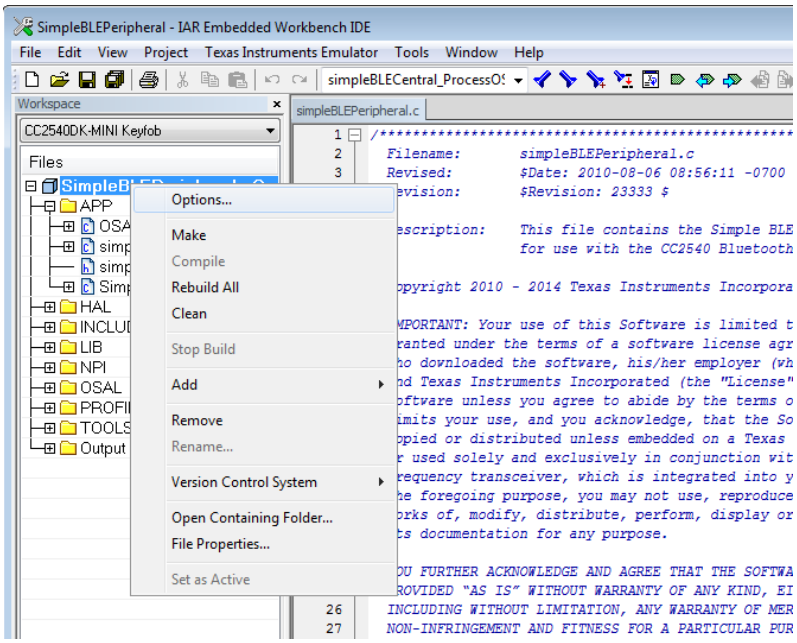
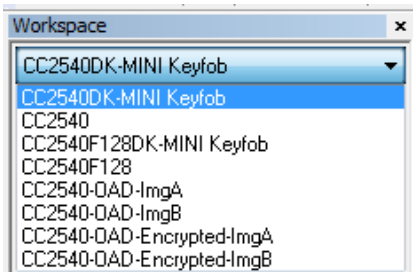


Figure 22: Project Configurations and Options

After clicking "Options…", a new window will pop-up, displaying the project options. Sometimes it is useful to have a few different configurations of options for different setups, such as when multiple hardware platforms are being used. IAR allows for multiple configurations to be created. These configurations are selectable via the drop down menu in the top of the "Workspace" pane. The default configuration in the SimpleBLEPeripheral project is the "CC2540DK-MINI Keyfob" configuration, which is targeted towards the keyfob hardware platform that is included with the CC2540/41DK mini development kit. The other available option, "CC2540", is optimized for the "SmartRF05 + CC2540 EM included with the full development kit. There are other configurations for 128 kB part, OAD, etc:



Project Configurations

One of the important settings when building a project is the compiler preprocessor definitions. symbols. These values can be found (and set) by clicking the "C/C++ Compiler" category on the left, and then clicking the "Preprocessor" tab on the right:

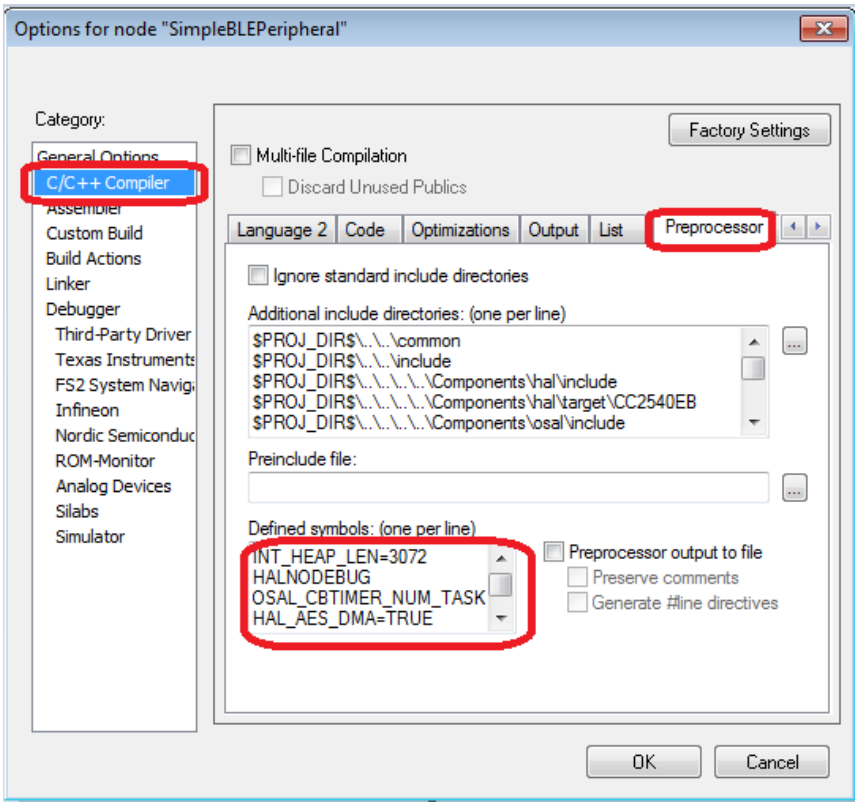
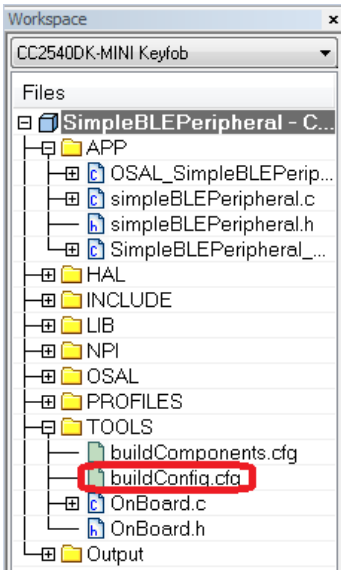


Figure 23: Preprocessor Defined Symbols Settings

Any symbols that are defined here will apply to all files in the project. The symbols can be defined with or without values. For convenience, some symbols may be defined with the character 'x' in front of them. This means that the definitions is not defined, and can be enabled by removing the 'x' and letting the proper name of the symbol be defined.

In addition to the defined symbols list in the compiler settings, symbols can be defined in configuration files, which get included when compiling. The "Extra Options" tab under the compiler settings allows you to set up the configuration files to be included. The file config.cfg must be included with every build, as it defines some required universal constants. The file *buildConfig.h* included with the software development kit will define the appropriate symbols for the project:



This project is fed into the compiler options via the following tab:

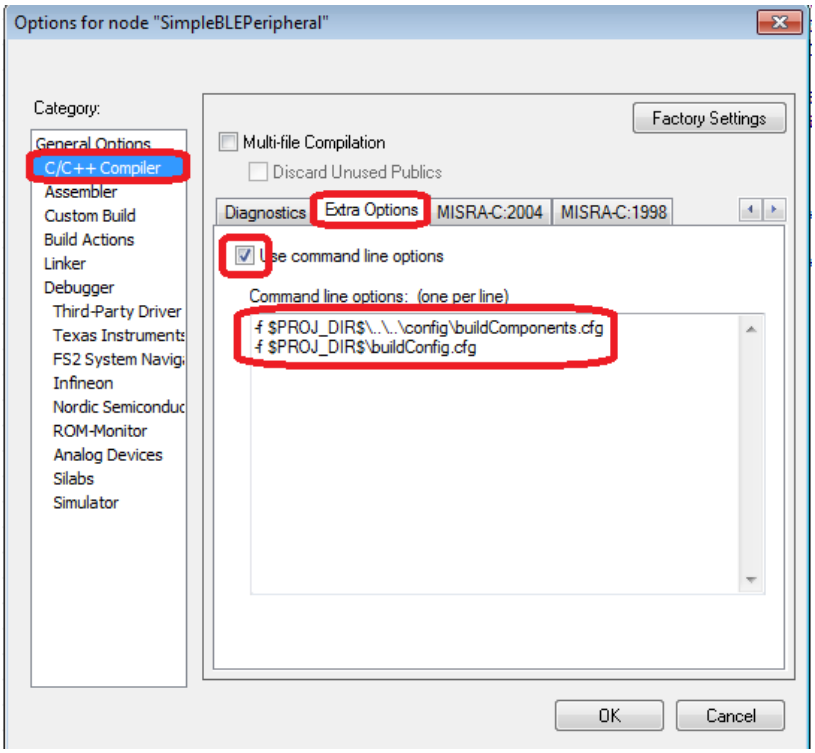


Figure 24: Configuration File Setup

The following symbols are used by the BLE protocol stack and software, and can be found 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. This value can be increased if additional heap memory is required by the application; however if increased too high the RAM limit may be exceeded. If additional memory is needed by the application for local variables, this value may need to be decreased. The memory set aside for the heap will show up in the map file under the module `OSAL_Memory`. For more information on the map file, see section 8.2.4.
- **HALNODEBUG** – This symbol should be defined for all projects in order to disable HAL assertions.
- **OSAL_CBTIMER_NUM_TASKS –** This symbol defines the number of OSAL callback timers that can be used. The BLE protocol stack uses the OSAL callback timer, and therefore this value *must* be defined as either 1 or 2 (there is a maximum of two callback timers allowed). For applications that are not using any callback timers, such as the sample application, this value should be defined as 1.
- **HAL_AES_DMA** – This symbol *must* be defined as **TRUE**, as 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 there aren't any pending tasks.
- **PLUS_BROADCASTER** – This symbol indicates that the device is using the GAP Peripheral / Broadcaster multi-role profile, rather than the single GAP Peripheral role profile. The default option for this in the simpleBLEPeripheral project is for this to be 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** – This symbol should be defined when using the keyfob board contained in the CC2540/41DK-MINI development kit. It configures the hardware based on the board layout.
- **HAL_UART_DMA** – This symbol sets the UART interface to use DMA mode when set to 1. When HAL_UART is defined, one and only one of HAL_UART_DMA and HAL_UART_ISR should be set to 1.
- **HAL_UART_ISR** – This symbol sets the UART interface to use ISR mode when set to 1. When HAL_UART is defined, one and only one of HAL_UART_DMA and HAL_UART_ISR should be set to 1.
- **HAL_UART_SPI** – This symbol indicates whether to include the SPI driver.
- **GAP_BOND_MGR** – This symbol is used by the HostTestRelease network processor project. When this symbol is defined for slave / peripheral configurations, the GAP peripheral bond manager security profile will be used to manage bonds and handle keys. For more information on the peripheral bond manager, see section 5.3.
- **GATT_DB_OFF_CHIP** – This symbol is used by the HostTestRelease network processor project. This symbol sets a GATT client in a network processor configuration to have management of the attributes in the application processor as opposed to being on the CC2540/41.

There are other defines relating to specific use cases that will be defined in the relevant documentation such as the serial bootloader, OAD, etc.

8.2.3 Building and Debugging a Project

To build a project, right click on the workspace name as seen below, and click "Make" or pressing F7.

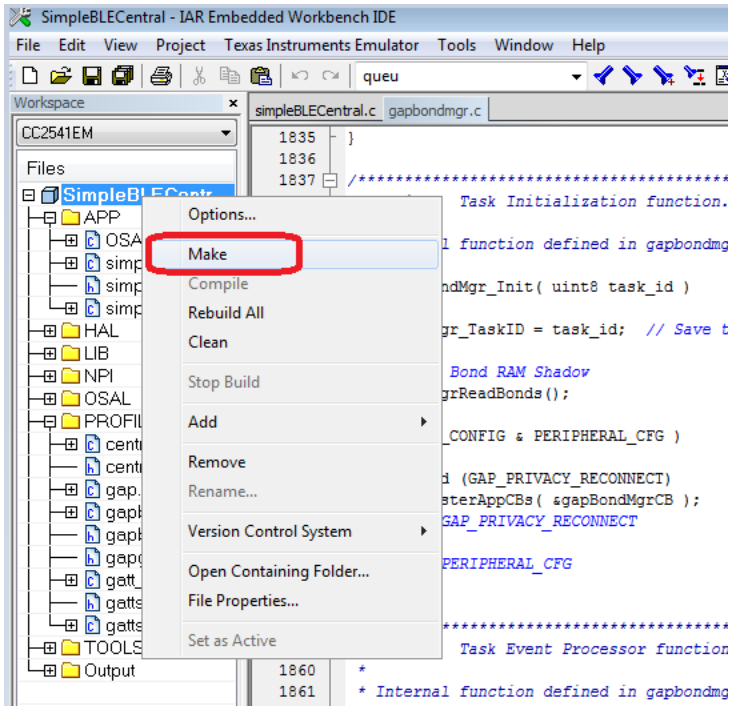


Figure 25: Building a Project

This will compile the source code, link the files, and build the project. Any compiler errors or warnings will appear in the "Build" window.

To download the compiled code onto a CC2540/41 device and debug, connect the keyfob using a hardware debugger (such as the CC Debugger, which is included with the CC2540/41DK-MINI development kit) connected to the PC over USB. Find the "Debug" button in the upper right side of the IAR window:

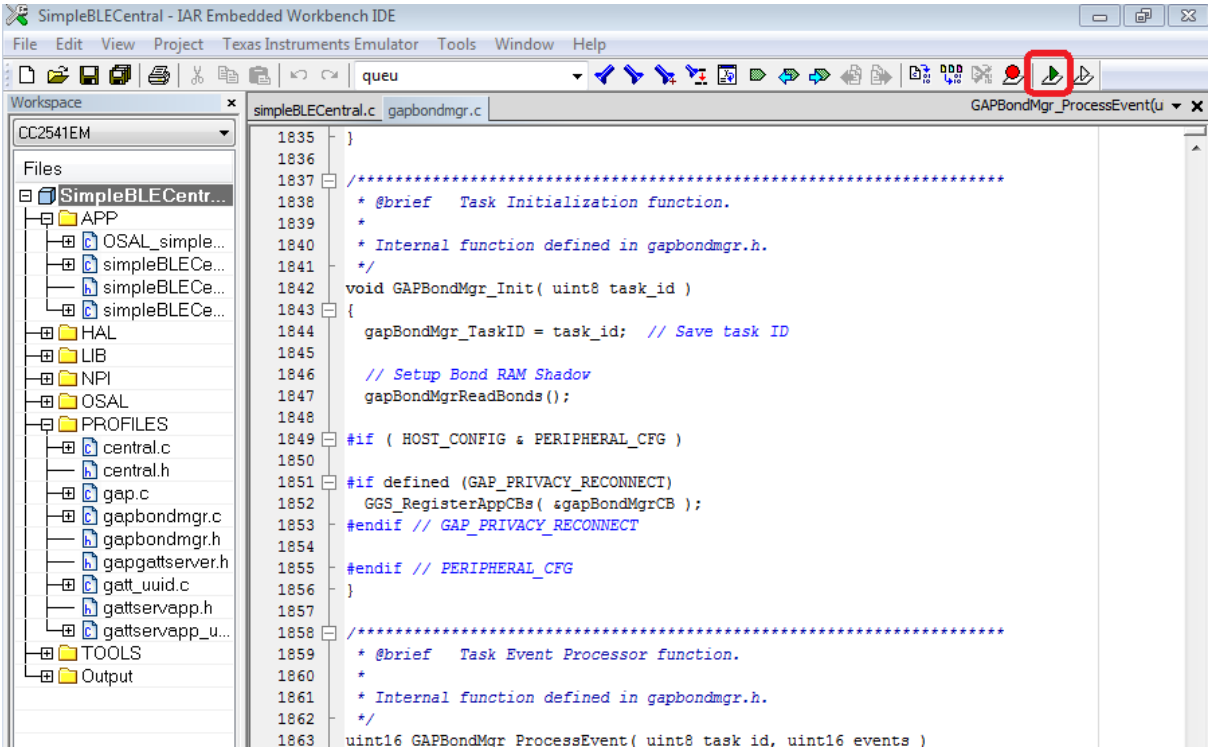


Figure 26: Debug Button in IAR

If there are multiple debug devices connected, the following window will appear to select a device. After selecting a device, the code will be downloaded.

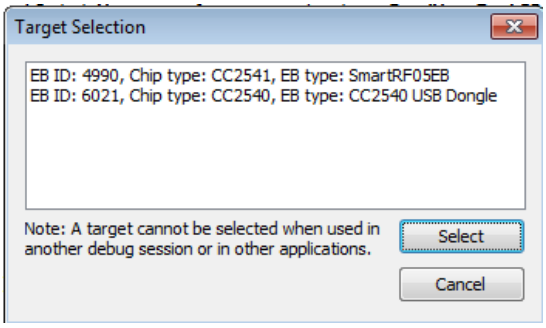


Figure 27

Once the code is downloaded, a toolbar with the debug commands will appear in the upper left corner of the screen. You can start the program's execution by pressing the "Go" button on the toolbar. Once the program is running, you can get out of the debugging mode by pressing the "Stop Debugging" button. Both of these buttons are shown in the image below:

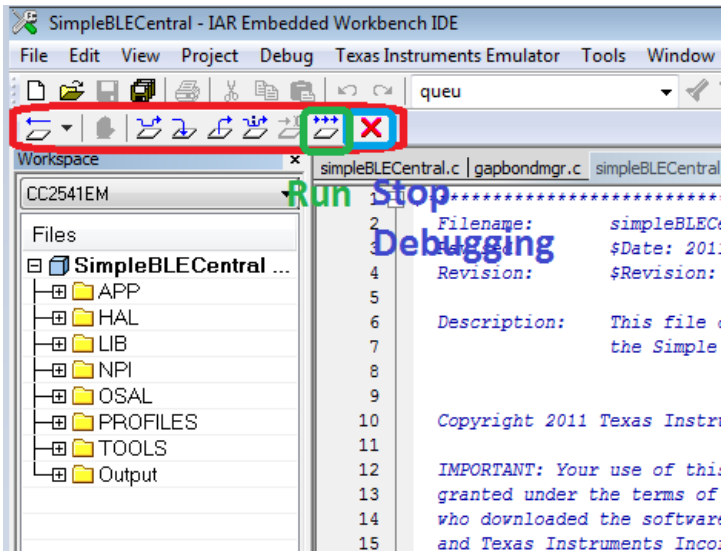


Figure 28: IAR Debug Toolbar

At this point the program should be executing. The hardware debugger can be disconnected from the CC2540/41 and will continue to run as long as the device remains powered-up.

8.2.4 Linker Map File

After building a project, IAR will generate a linker map file, which can be found under the "Output" group in the file list.

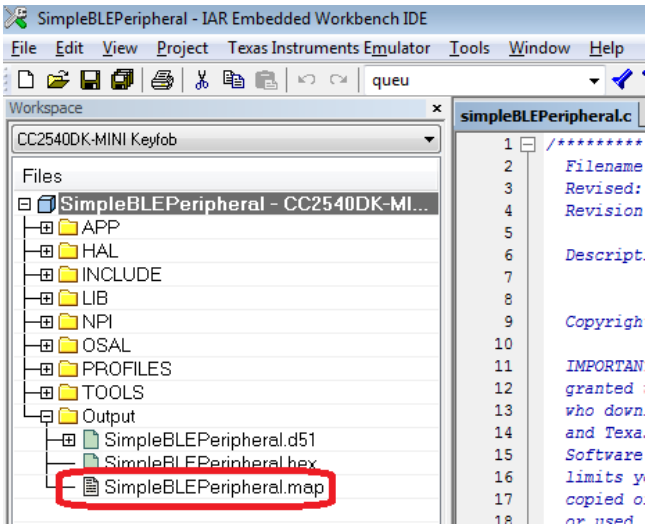


Figure 29: MAP File in File List

The map file contains detailed low-level information about the build. At the very end of the map file, lines of text similar to the following can be found:

```
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 information is useful in that it states the total amount of code space (CODE memory) and RAM (XDATA memory) being used by the project. The sum of the CODE memory plus CONST memory must not exceed the maximum flash size of the device (either 128KB or 256KB, depending on the version of the CC2540/41). The size of the XDATA memory must 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 title "MODULE SUMMARY", which can be found approximately 200-300 lines before the end of the file (the exact location will vary from build-to-build). Within this section, the exact amount of flash and memory being used for every module in the project can be seen.

9 General Information

The current 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. In order to
  build all projects, be sure to 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. 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 please 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. In order to
  build all projects, be sure to 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 please 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 please 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 please 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 please 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 10ms.

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 please 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) it is recommended that 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 below).

- 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 please 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 please 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 please 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: http://www.ti.com/lit/pdf/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 / 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 7.5ms),
          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 "Known Issues" below)


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 please 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

Table 1: Document History

Revision	Date	Description/Changes
1.0	2010-10-07	Initial release
1.1	2011-07-13	Updated for BLEv1.1 software release
1.1b	2011-11-15	Updated for BLEv1.1b software release
1.2	2012-02-13	Updated for BLEv1.2 software release
1.3.1	2012-04-13	Updated for BLEv1.2.1 software release
1.3	2012-12-19	Updated for BLEv1.3 software release
1.3.1	2013-04-05	Updated for BLEv1.3.1 software release
1.3.2	2013-06-12	Updated for BLEv1.3.2 software release
1.4.0	2013-09-12	Updated for BLEv1.4.0 software release
1.4.1	2015-05-15	Updated for BLEv1.4.1 software release

I. *GAP API*

I.1 Commands

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

uint16 GAP_GetParamValue (gapParamIDs_t paramID)
Description: Get a GAP parameter. Parameters: *paramID* – parameter ID (Appendix I.2) Returns: GAP Parameter Value if successful 0xFFFF if *paramID* invalid

bStatus_t GAP_SetParamValue (gapParamIDs_t paramID, uint16 paramValue)
Description: Set a GAP parameter. Parameters: *paramID* – parameter ID (Appendix I.2)) *paramValue* – new param value Returns: SUCCESS INVALIDPARAMETER: *paramID* is invalid

I.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 timeout, 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 timeout (ms) when performing Connection Establishment.
TGAP_CONN_PARAM_TIMEOUT	Timeout (ms) for link layer to wait to receive connection parameter update response.
TGAP_LIM_DISC_ADV_INT_MIN	Minimum advertising interval in limited discovery mode (n * 0.625 ms)
TGAP_LIM_DISC_ADV_INT_MAX	Maximum advertising interval in limited discovery mode (n * 0.625 ms)
TGAP_GEN_DISC_ADV_INT_MIN	Minimum advertising interval in general discovery mode (n * 0.625 ms)
TGAP_GEN_DISC_ADV_INT_MAX	Maximum advertising interval in general discovery mode (n * 0.625 ms)
TGAP_CONN_ADV_INT_MIN	Minimum advertising interval when in connectable mode (n * 0.625 ms)
TGAP_CONN_ADV_INT_MAX	Maximum advertising interval when in connectable mode (n * 0.625 ms)

TGAP_CONN_SCAN_INT	Scan interval used during Link Layer Initiating state, when in Connectable mode (n * 0.625 mSec)
TGAP_CONN_SCAN_WIND	Scan window used during Link Layer Initiating state, when in Connectable mode (n * 0.625 mSec)
TGAP_CONN_HIGH_SCAN_INT	Scan interval used during Link Layer Initiating state, when in Connectable mode, high duty scan cycle scan parameters (n * 0.625 mSec)
TGAP_CONN_HIGH_SCAN_WIND	Scan window used during Link Layer Initiating state, when in Connectable mode, high duty scan cycle scan parameters (n * 0.625 mSec)
TGAP_GEN_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in General Discovery proc (n * 0.625 mSec).
TGAP_GEN_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in General Discovery proc (n * 0.625 mSec)
TGAP_LIM_DISC_SCAN_INT	Scan interval used during Link Layer Scanning state, when in Limited Discovery proc (n * 0.625 mSec)
TGAP_LIM_DISC_SCAN_WIND	Scan window used during Link Layer Scanning state, when in Limited Discovery proc (n * 0.625 mSec)
TGAP_CONN_EST_INT_MIN	Minimum Link Layer connection interval, when using Connection Establishment proc (n * 1.25 mSec)
TGAP_CONN_EST_INT_MAX	Maximum Link Layer connection interval, when using Connection Establishment proc (n * 1.25 mSec)
TGAP_CONN_EST_SCAN_INT	Scan interval used during Link Layer Initiating state, when using Connection Establishment proc (n * 0.625 mSec)
TGAP_CONN_EST_SCAN_WIND	Scan window used during Link Layer Initiating state, when using Connection Establishment proc (n * 0.625 mSec)
TGAP_CONN_EST_SUPERV_TIMEOUT	Link Layer connection supervision timeout, when using Connection Establishment proc (n * 10 mSec)
TGAP_CONN_EST_LATENCY	Link Layer connection slave latency, when using Connection Establishment proc (in number of connection events)
TGAP_CONN_EST_MIN_CE_LEN	Local informational parameter about min len of connection needed, when using Connection Establishment proc (n * 0.625 mSec)
TGAP_CONN_EST_MAX_CE_LEN	Local informational parameter about max len of connection needed, when using Connection Establishment proc (n * 0.625 mSec).
TGAP_PRIVATE_ADDR_INT	Minimum Time Interval between private (resolvable) address changes. In minutes (default 15 minutes)
TGAP_CONN_PAUSE_CENTRAL	Central idle timer. In seconds (default 1 second)
TGAP_CONN_PAUSE_PERIPHERAL	Minimum time upon connection establishment before the peripheral starts a connection update procedure. In seconds (default 5 seconds)
TGAP_SM_TIMEOUT	Time (ms) to wait for security manager response before returning bleTimeout. Default is 30 seconds.
TGAP_SM_MIN_KEY_LEN	SM Minimum Key Length supported. Default 7.
TGAP_SM_MAX_KEY_LEN	SM Maximum Key Length supported. Default 16.
TGAP_FILTER_ADV_REPORTS	TRUE to filter duplicate advertising reports. Default TRUE.
TGAP_SCAN_RSP_RSSI_MIN	Minimum RSSI required for scan responses to be reported to the app. Default -127.
TGAP_REJECT_CONN_PARAMS	Whether or not to reject Connection Parameter Update

	Request received on Central device. Default FALSE.

I.3 Events

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

```
typedef struct
{
  osal_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
  uint8 opcode;                    //!< GAP type of command. Ref: @ref GAP_MSG_EVENT_DEFINES
} gapEventHdr_t;
```

The following is a list of the possible `hdr` and the associated events. See *gap.h* for all other definitions used in these events.

- GAP_DEVICE_INIT_DONE_EVENT: Sent when the Device Initialization is complete.

```
typedef struct
{
  osal_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
  uint8 opcode;                       //!< GAP_DEVICE_INIT_DONE_EVENT
  uint8 devAddr[B_ADDR_LEN];          //!< Device's BD_ADDR
  uint16 dataPktLen;                  //!< HC_LE_Data_Packet_Length
  uint8 numDataPkts;                  //!< HC_Total_Num_LE_Data_Packets
} gapDeviceInitDoneEvent_t;
```

- GAP_DEVICE_DISCOVERY_EVENT: Sent when the Device Discovery Process is complete.

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_DEVICE_DISCOVERY_EVENT
  uint8 numDevs;         //!< Number of devices found during scan
  gapDevRec_t *pDevList; //!< array of device records
} gapDevDiscEvent_t;
```

- GAP_ADV_DATA_UPDATE_DONE_EVENT: Sent when the Advertising Data or SCAN_RSP Data has been updated.

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_ADV_DATA_UPDATE_DONE_EVENT
  uint8 adType;          //!< TRUE if advertising data, FALSE if SCAN_RSP
} gapAdvDataUpdateEvent_t;
```

- GAP_MAKE_DISCOVERABLE_DONE_EVENT: Sent when the Make Discoverable Request is complete.

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_MAKE_DISCOVERABLE_DONE_EVENT
  uint16 interval;       //!< actual advertising interval selected by controller
} gapMakeDiscoverableRspEvent_t;
```

- GAP_END_DISCOVERABLE_DONE_EVENT: Sent when the Advertising has ended.

```
typedef struct
{
  osal_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
  uint8 opcode;          //!< GAP_END_DISCOVERABLE_DONE_EVENT
} gapEndDiscoverableRspEvent_t;
```

- GAP_LINK_ESTABLISHED_EVENT: Sent when the Establish Link Request is complete

```
typedef struct
{
  osal_event_hdr_t  hdr;      //!< GAP_MSG_EVENT and status
  uint8 opcode;               //!< GAP_LINK_ESTABLISHED_EVENT
  uint8 devAddrType;          //!< Device address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 devAddr[B_ADDR_LEN]; //!< Device address of link
  uint16 connectionHandle;    //!< Connection Handle from controller used to ref the device
  uint16 connInterval;        //!< Connection Interval
  uint16 connLatency;         //!< Conenction Latency
  uint16 connTimeout;         //!< Connection Timeout
  uint8 clockAccuracy;        //!< Clock Accuracy
} gapEstLinkReqEvent_t;
```

- GAP_LINK_TERMINATED_EVENT: Sent when a connection was terminated.

```
typedef struct
{
  osal_event_hdr_t  hdr;    //!< GAP_MSG_EVENT and status
  uint8 opcode;             //!< GAP_LINK_TERMINATED_EVENT
  uint16 connectionHandle; //!< connection Handle
  uint8 reason;             //!< termination reason from LL
} gapTerminateLinkEvent_t;
```

- GAP_LINK_PARAM_UPDATE_EVENT: Sent when an Update Parameters Event is received.

```
typedef struct
{
  osal_event_hdr_t hdr;      //!< GAP_MSG_EVENT and status
  uint8 opcode;              //!< GAP_LINK_PARAM_UPDATE_EVENT
  uint8 status;              //!< bStatus_t
  uint16 connectionHandle;  //!< Connection handle of the update
  uint16 connInterval;       //!< Requested connection interval
  uint16 connLatency;        //!< Requested connection latency
  uint16 connTimeout;        //!< Requested connection timeout
} gapLinkUpdateEvent_t;
```

- GAP_RANDOM_ADDR_CHANGED_EVENT: Sent when a random address was changed.

```
typedef struct
{
  osal_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
  uint8 opcode;                       //!< GAP_RANDOM_ADDR_CHANGED_EVENT
  uint8 addrType;                     //!< Address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 newRandomAddr[B_ADDR_LEN];    //!< the new calculated private addr
} gapRandomAddrEvent_t;
```

- GAP_SIGNATURE_UPDATED_EVENT: Sent when the device's signature counter is updated.

```
typedef struct
{
  osal_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
  uint8 opcode;                       //!< GAP_SIGNATURE_UPDATED_EVENT
  uint8 addrType;                     //!< Device's address type for devAddr
  uint8 devAddr[B_ADDR_LEN];          //!< Device's BD_ADDR, could be own address
  uint32 signCounter;                 //!< new Signed Counter
} gapSignUpdateEvent_t;
```

- GAP_AUTHENTICATION_COMPLETE_EVENT: Sent when the Authentication (pairing) process is complete.

```
typedef struct
{
  osal_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
  uint8 opcode;                    //!< GAP_AUTHENTICATION_COMPLETE_EVENT
  uint16 connectionHandle;         //!< Connection Handle from controller used to ref the device
  uint8 authState;                 //!< TRUE if the pairing was authenticated (MITM)
  smSecurityInfo_t *pSecurityInfo; //!< BOUND - security information from this device
  smSigningInfo_t *pSigningInfo;   //!< Signing information
  smSecurityInfo_t *pDevSecInfo;   //!< BOUND - security information from connected device
  smIdentityInfo_t *pIdentityInfo; //!< BOUND - identity information
```

```
} gapAuthCompleteEvent_t;
```

- GAP_PASSKEY_NEEDED_EVENT: Sent when a Passkey is needed. This is part of the pairing process.

```
typedef struct
{
  osal_event_hdr_t  hdr;        //!< GAP_MSG_EVENT and status
  uint8 opcode;                 //!< GAP_PASSKEY_NEEDED_EVENT
  uint8 deviceAddr[B_ADDR_LEN]; //!< address of device to pair with, and could be either public or
random.
  uint16 connectionHandle;      //!< Connection handle
  uint8 uiInputs;               //!< Pairing User Interface Inputs - Ask user to input passcode
  uint8 uiOutputs;              //!< Pairing User Interface Outputs - Display passcode
} gapPasskeyNeededEvent_t;
```

- GAP_SLAVE_REQUESTED_SECURITY_EVENT: Sent when a Slave Security Request is received.

```
typedef struct
{
  osal_event_hdr_t  hdr;        //!< GAP_MSG_EVENT and status
  uint8 opcode;                 //!< GAP_SLAVE_REQUESTED_SECURITY_EVENT
  uint16 connectionHandle;      //!< Connection Handle
  uint8 deviceAddr[B_ADDR_LEN]; //!< address of device requesting security
  uint8 authReq;                  //!< Authentication Requirements: Bit 2: MITM, Bits 0-1: bonding
(0 - no bonding, 1 - bonding)

} gapSlaveSecurityReqEvent_t;
```

- GAP_DEVICE_INFO_EVENT: Sent during the Device Discovery Process when a device is discovered.

```
typedef struct
{
  osal_event_hdr_t  hdr;    //!< GAP_MSG_EVENT and status
  uint8 opcode;             //!< GAP_DEVICE_INFO_EVENT
  uint8 eventType;          //!< Advertisement Type: @ref GAP_ADVERTISEMENT_REPORT_TYPE_DEFINES
  uint8 addrType;           //!< address type: @ref GAP_ADDR_TYPE_DEFINES
  uint8 addr[B_ADDR_LEN];   //!< Address of the advertisement or SCAN_RSP
  int8 rssi;                //!< Advertisement or SCAN_RSP RSSI
  uint8 dataLen;            //!< Length (in bytes) of the data field (evtData)
  uint8 *pEvtData;          //!< Data field of advertisement or SCAN_RSP
} gapDeviceInfoEvent_t;
```

- GAP_BOND_COMPLETE_EVENT: Sent when the bonding process is complete.

```
typedef struct
{
  osal_event_hdr_t  hdr;   //!< GAP_MSG_EVENT and status
  uint8 opcode;            //!< GAP_BOND_COMPLETE_EVENT
  uint16 connectionHandle; //!< connection Handle
} gapBondCompleteEvent_t;
```

- GAP_PAIRING_REQ_EVENT: Sent when an unexpected Pairing Request is received.

```
typedef struct
{
  osal_event_hdr_t hdr;    //!< GAP_MSG_EVENT and status
  uint8 opcode;            //!< GAP_PAIRING_REQ_EVENT
  uint16 connectionHandle; //!< connection Handle
  gapPairingReq_t pairReq; //!< The Pairing Request fields received.
} gapPairingReqEvent_t;
```

II. *GAPRole peripheral Role API*

II.1 Commands

bStatus_t GAPRole_SetParameter(uint16 param, uint8 len, void *pValue)
Description: Set a GAP Role parameter.

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

Returns:
SUCCESS
INVALIDPARAMETER: *param* was not valid
bleInvalidRange: *len* is not valid for the given *param*
blePending*:* previous param update has not been completed
bleIncorrectMode: can not start connectable advertising because non-connectable advertising is enabled

bStatus_t GAPRole_GetParameter(uint16 param, void *pValue)

Description: Set a GAP Role parameter.

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

Returns:
SUCCESS
INVALIDPARAMETER: *param* was not valid

bStatus_t GAPRole_StartDevice(gapRolesCBs_t *pAppCallbacks)

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

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

Returns:
SUCCESS
bleAlreadyInRequestedMode: device was already initialized

bStatus_t GAPRole_TerminateConnection(void)

Description: 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 already in process

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

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

Parameters:
connInterval – the new connection interval
latency – the new slave latency
connTimeout – the new timeout value
handle failure– what to do if the update does not occur. Available actions:

- GAPROLE_NO_ACTION 0 // Take no action upon unsuccessful parameter updates
- GAPROLE_RESEND_PARAM_UPDATE 1 // Continue to resend request until successful update
- GAPROLE_TERMINATE_LINK 2 // Terminate link upon unsuccessful parameter updates

Returns:
SUCCESS: parameter update process has started
bleNotConnected: there is no connection so can not update parameters

II.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPROLE_PROFILEROLE	R	uint8	GAP profile role (peripheral)
GAPROLE_IRK	R/W	uint8[16]	Identity resolving key. Default is all 0, which means the IRK will be randomly generated.
GAPROLE_SRK	R/W	uint8[16]	Signature resolving key. Default is all 0, which means the SRK will be randomly generated.
GAPROLE_SIGNCOUNTER	R/W	uint32	Sign counter.
GAPROLE_BD_ADDR	R	uint8[6]	Device address read from controller. This can be set with the HCI_EXT_SetBDADDRCmd().
GAPROLE_ADVERT_ENABLED	R/W	uint8	Enable / disable advertising. Default is TRUE = enabled.
GAPROLE_ADVERT_OFF_TIME	R/W	uint16	How long to 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 0's.
GAPROLE_ADV_EVENT_TYPE	R/W	uint8	Advertisement type. Default is GAP_ADTYPE_IND (from gap.h)
GAPROLE_ADV_DIRECT_TYPE	R/W	uint8	Direct advertisement type. Default is ADDRTYPE_PUBLIC (from gap.h)
GAPROLE_ADV_DIRECT_ADDR	R/W	uint8[6]	Direct advertisement address. Default is 0.
GAPROLE_ADV_CHANNEL_MAP	R/W	uint8	Which channels to advertise on. Default is GAP_ADVCHAN_ALL (from gap.h)
GAPROLE_ADV_FILTER_POLICY	R/W	uint8	Policy for filtering advertisements. Ignored in direct advertising
GAPROLE_CONNHANDLE	R	uint16	Handle of current connection.
GAPROLE_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 param update.
GAPROLE_MAX_CONN_INTERVAL	R/W	uint16	Maximum connection interval to allow (n * 125 ms). Range: 7.5 ms to 4 sec. Default is 7.5 ms. Also used for param update.
GAPROLE_SLAVE_LATENCY	R/W	uint16	Slave latency to use for a param update. Range: 0 – 499. Default is 0.
GAPROLE_TIMEOUT_MULTIPLIER	R/W	uint16	Supervision timeout to use for a param update (n * 10 ms). Range: 100 ms to 32 sec. Default is 1000 ms.

GAPROLE_CONN_BD_ADDR	R	uint8[6]	Address of connected device.
GAPROLE_CONN_INTERVAL	R	uint16	Current connection interval.
GAPROLE_CONN_LATENCY	R	uint16	Current slave latency.
GAPROLE_CONN_TIMEOUT	R	uint16	Current supervision timeout.
GAPROLE_PARAM_UPDATE_REQ	W	uint8	Set this to true to send a param update request.
GAPROLE_STATE	R	uint8	Gap peripheral role state (enumerated in gaprole_States_t in peripheral.h).

II.3 Callbacks

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

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

See the SimpleBLEPeripheral application for an example.

II.3.1 State Change Callback (pfnStateChange)

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

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

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

- GAPROLE_INIT //!< Waiting to be started
- GAPROLE_STARTED //!< Started but not advertising
- GAPROLE_ADVERTISING //!< Currently Advertising
- GAPROLE_ADVERTISING_NONCONN //!< Currently using non-connectable Advertising
- GAPROLE_WAITING //!< Device is started but not advertising, is in waiting period before advertising again
- GAPROLE_WAITING_AFTER_TIMEOUT //!< Device just timed out from a connection but is not yet advertising, is in waiting period before advertising again
- GAPROLE_CONNECTED //!< In a connection
- GAPROLE_CONNECTED_ADV //!< In a connection + advertising
- GAPROLE_ERROR //!< Error occurred - invalid state

II.3.2 RSSI callback (pfnRssiRead)

This function will, when enabled, report the RSSI back to the application at a rate set by the `GAPROLE_RSSI_READ_RATE` GAPRole parameter. Setting this parameter to 0 will disable the RSSI reporting. This function is defined as:

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

This function will pass a signed one byte value (`newRSSI`) of the last reported RSSI to the application.

III. *GAPRole Central Role API*

III.1 Commands

bStatus_t GAPCentralRole_StartDevice(gapCentralRoleCB_t *pAppCallbacks)

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

Parameters:
pAppCallbacks – pointer to application callbacks

Returns:
SUCCESS
bleAlreadyInRequestedMode: *Device already started.*

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

Description: Set a GAP Role parameter.

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

Returns:
SUCCESS
INVALIDPARAMETER: *param* was not valid
bleInvalidRange: *len* is invalid for the given *param*

bStatus_t GAPCentralRole_GetParameter (uint16 param, void *pValue)

Description: Set a GAP Role parameter.

Parameters:
param – Profile parameter ID (section III.2)
pValue – pointer to buffer to contain the read data

Returns:
SUCCESS
INVALIDPARAMETER: *param* was not valid

bStatus_t GAPCentralRole_TerminateLink (uint16 connHandle);

Description: Terminates an existing connection.

Parameters:
connHandle - connection handle of link to terminate or...
0xFFFE: cancel the current link establishment request or...
0xFFFF: terminate all links

Returns:
SUCCESS: 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)

Description: Establish a link to a peer device.

Parameters:
highDutyCycle - TRUE to high duty cycle scan, FALSE if not
whiteList - determines use of the white list
addrTypePeer - address type of the peer device:

peerAddr - peer device address

Returns:
SUCCESS: link establishment has started
bleIncorrectMode: invalid profile role.
bleNotReady: a scan is in progress.
bleAlreadyInRequestedMode: can't process now.
bleNoResources: too many links.

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

Description: Update the link connection parameters.

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

Returns:
SUCCESS: 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 will not work because it is not a multiple / divisor of other simultaneous connection's intervals, or the connection's interval is not less than the allowed maximum connection interval as determined by the maximum number of connections times the number of slots per connection

bStatus_t GAPCentralRole_StartDiscovery(uint8 mode, uint8 activeScan, uint8 whiteList)

Description: Start a device discovery scan.

Parameters:
mode - discovery mode
activeScan - TRUE to perform active scan
whiteList - TRUE to only scan for devices in the white list

Returns:
SUCCESS: 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)

Description: 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)

Description: 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)

Description: Cancel periodic RSSI reads on a link.

Parameters:
connHandle - connection handle of link

Returns:
SUCCESS
bleIncorrectMode: No active link.

III.2 Configurable Parameters

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

III.3 Callbacks

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

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

See the SimpleBLECentral application for an example.

III.3.1 RSSI callback (rssiCB)

This function will report the RSSI back to the application as a result of the `GAPCentralRole_StartRssi()` command. It is of the type:

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

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

III.3.2 Central event callback (eventCB)

This callback is used to pass GAP state change events to the application. It is of the 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 deallocation will be done by the application. 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 listed here. See Appendix I.3 for more information on these events:

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

IV. *GATT / ATT API*

This section will describe the API of the GATT and ATT layers. The two sections are combined because the general procedure is to send GATT commands and receive ATT events as described in Section 5.4.3.1.

The return values for the commands referenced in this section are described in Appendix IV.3.

The possible return values are similar for all of these commands so they are described in Section IV.3.

IV.1 Server Commands

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

Description: Indicates a characteristic value to a client and expect an acknowledgement. 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
pInd: pointer to indication to be sent
authenticated: whether an authenticated link is required
taskId: task to be notified of acknowledgement

Corresponding Events:
If the return status is SUCCESS, the calling application task will receive a GATT_MSG_EVENT message with type ATT_HANDLE_VALUE_CFM upon an acknowledgement. It is only at this point that this subprocedure is considered complete.

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

Description: Notifies a characteristic value to a client. 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.

IV.2 Client Commands

bStatus_t GATT_InitClient(void)

Description: Initialize the GATT client in the BLE Stack.

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

bStatus_t GATT_RegisterForInd (uint8 taskId)

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

Parameters:
taskId: task to forward indications or notifications to

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

bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId)

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

Parameters:
connHandle: connection to use
taskId: task to be notified of response

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

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

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

Parameters:
connHandle: connection to use
pValue: pointer to value (UUID) to look for
len: length of value
taskId: task to be notified of response

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

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

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

Parameters:
connHandle: connection to use
startHandle: start handle of primary service to search in

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

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

bStatus_t GAPRole_GetParameter(uint16 param, void *pValue)

Description: Get a GAP Role parameter.

Parameters:
param – Profile parameter ID (See Appendix VI.2)
pValue – pointer to a location to get the value. This is dependent on the param ID and will be cast to the appropriate data type.

Returns:
SUCCESS
INVALIDPARAMETER: *param* was not valid

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

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

Parameters:
connHandle: connection to use
startHandle: start handle of service to search in
endHandle: end handle of service to search in
taskId: task to be notified of response

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

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

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

Parameters:
connHandle: connection to use
pReq: pointer to request to be sent, including start and end handles of service and UUID of characteristic value to search for.
taskId: task to be notified of response

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

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

Description: used by a client to find all the characteristic descriptor's Attribute Handles and AttributeTypes within a characteristic definition when only the characteristic handle range is known..

Parameters:
connHandle: connection to use
startHandle: start handle
endHandle: end handle
taskId: task to be notified of response

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

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

Description: Used to read a Characteristic Value from a server when the client knows the Characteristic Value Handle.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

bStatus_t GATT_WriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

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

Parameters:
connHandle: connection to use
pReq: pointer to command to be sent

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

bStatus_t GATT_SignedWriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Description: Used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the ATT Bearer is not encrypted. This sub-procedure shall only be used if the Characteristic Properties authenticated bit is enabled and the client and server device share a bond as defined in the GAP.

Parameters:
connHandle: connection to use
pReq: pointer to command to be sent

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

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

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

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

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

bStatus_t GATT_WriteLongCharValue(uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

Description: Used to write a Characteristic Value to a server when the client knows the Characteristic Value

Handle but the length of the Characteristic Value is longer than can be sent in a single Write Request Attribute Protocol message.

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

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

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

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

Parameters:
connHandle: connection to use
pReq: pointer to requests to be sent (must be allocated)

numReqs - number of requests in pReq

flags - execute write request flags
taskId: task to be notified of response

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

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

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

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

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

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

Description: Used to read a characteristic descriptor from a server when the client knows the characteristic descriptor declaration's Attribute handle and the length of the characteristic descriptor declaration is longer than can be sent in a single Read Response attribute protocol message.

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

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

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

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

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

bStatus_t GATT_WriteLongCharDesc (uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

Description: Used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle but the length of the Characteristic Value is longer than can be sent in a single Write Request Attribute Protocol message.

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

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

IV.3 Return Values

- SUCCESS (0x00): Command was executed as expected. See the individual command API for corresponding events to expect.
- INVALIDPARAMETER (0x02): Invalid connection handle or request field.
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): attribute requires authentication
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): key size used for encrypting is insufficient
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): attribute requires encryption
- MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available. Retry later.
- bleNotConnected (0x14): the device is not currently in a connection.
- blePending (0x17):
    - when returned to a client function: a response is still pending with the server or the GATT sub-procedure is still in progress
    - when returned to server function: confirmation from a client is still pending
- bleTimeout (0x16): the previous transaction timed out. No more ATT/GATT messages can be sent until the connection is re-established.
- bleMemAllocError (0x13): memory allocation error occurred
- bleLinkEncrypted (0x19): link is already encrypted. An Attribute PDU that includes an Authentication Signature should not be sent on an encrypted link

IV.4 Events

These will be received from the BLE stack in the application as a GATT_MSG_EVENT stack message sent as an OSAL message. They will be received as the following structure where the `method` signifies the ATT Event and the `msg` is a union of all the various ATT events:

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

This section will list the various ATT Events by their `method` and display their structure that is used in the `msg` payload. These are listed in the *att.h* file.

- ATT_ERROR_RSP (0x01)

```
typedef struct
{
  uint8 reqOpcode; //!< Request that generated this error response
  uint16 handle;   //!< Attribute handle that generated error response
  uint8 errCode;   //!< Reason why the request has generated error response
} attErrorRsp_t;
attErrorRsp_t
```

- ATT_FIND_INFO_RSP (0x03)

```
typedef struct
{
  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
{
  uint16 handle;               //!< Handle of the attribute that has been read
  uint16 offset;               //!< Offset of the first octet to be written
  uint8 len;                   //!< Length of value
  uint8 value[ATT_MTU_SIZE-5]; //!< Part of the value of the attribute to be written
} attPrepareWriteRsp_t;
```

- ATT_EXECUTE_WRITE_RSP (0x19)
- ATT_HANDLE_VALUE_NOTI (0x1B)

```
typedef struct
{
  uint16 handle;                //!< Handle of the attribute that has been changed (must be first
field)
  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

IV.5 GATT commands and corresponding ATT events

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

ATT Response Events	GATT API calls
ATT_EXCHANGE_MTU_RSP	GATT_ExchangeMTU
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs, GATT_DiscAllCharDescs
ATT_FIND_BY_TYPE_VALUE_RSP	GATT_DiscPrimaryServiceByUUID
ATT_READ_BY_TYPE_RSP	GATT_PrepareWriteReq, GATT_ExecuteWriteReq, GATT_FindIncludedServices, GATT_DiscAllChars, GATT_DiscCharsByUUID, GATT_ReadUsingCharUUID,
ATT_READ_RSP	GATT_ReadCharValue,

	GATT_ReadCharDesc
ATT_READ_BLOB_RSP	GATT_ReadLongCharValue, GATT_ReadLongCharDesc
ATT_READ_MULTI_RSP	GATT_ReadMultiCharValues
ATT_READ_BY_GRP_TYPE_RSP	GATT_DiscAllPrimaryServices
ATT_WRITE_RSP	GATT_WriteCharValue, GATT_WriteCharDesc
ATT_PREPARE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc
ATT_EXECUTE_WRITE_RSP	GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc

IV.6 ATT_ERROR_RSP errCode's

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

- ATT_ERR_INVALID_HANDLE (0x01): Attribute handle value given was not valid on this attribute server
- ATT_ERR_READ_NOT_PERMITTED (0x02): Attribute cannot be read
- ATT_ERR_WRITE_NOT_PERMITTED (0x03): Attribute cannot be written
- ATT_ERR_INVALID_PDU (0x04): The attribute PDU was invalid
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): The attribute requires authentication before it can be read or written
- ATT_ERR_UNSUPPORTED_REQ (0x06): Attribute server doesn't support the request received from the attribute client
- ATT_ERR_INVALID_OFFSET (0x07): Offset specified was past the end of the attribute
- ATT_ERR_INSUFFICIENT_AUTHOR (0x08): The attribute requires an authorization before it can be read or written
- ATT_ERR_PREPARE_QUEUE_FULL (0x09): Too many prepare writes have been queued
- ATT_ERR_ATTR_NOT_FOUND (0x0A): No attribute found within the given attribute handle range
- ATT_ERR_ATTR_NOT_LONG (0x0B): Attribute cannot be read or written using the Read Blob Request or Prepare Write Request
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The Encryption Key Size used for encrypting this link is insufficient
- ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is invalid for the operation
- ATT_ERR_UNLIKELY (0x0E): The attribute request that was requested has encountered an error that was very unlikely, and therefore could not be completed as requested
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): The attribute requires encryption before it can be read or written
- ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10): The attribute type is not a supported grouping attribute as defined by a higher layer specification

- ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient Resources to complete the request

V. GATTServApp API

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

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

V.1 Commands

void GATTServApp_InitCharCfg(uint16 connHandle, gattCharCfg_t *charCfgTbl)

Description: Initialize the client characteristic configuration table for a given connection. This should be used whenever a service is added to the application (Section 5.4.4.2.2)

Parameters:
connHandle – connection handle (0xFFFF for all connections).
charCfgTbl – client characteristic configuration table where this characteristic resides

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

Description: Process Client Characteristic Configuration change.

Parameters:
charCfgTbl– Profile characteristic configuration table
pValue– pointer to attribute value.
authenticated – whether an authenticated link is required
attrTbl– whether attribute table.
numAttrs – number of attributes in attribute table.
tasked – task to be notified of confirmation.
pfnReadAttrCB – read callback function pointer.

Returns:
SUCCESS: 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)

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

Parameters:
pAttrTbl – pointer to attribute table
numAttrs – number of attributes in attribute table
pValue – pointer to attribute value

Returns:
Pointer to attribute record if found.
NULL, if not found.

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

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

Parameters:

connHandle– connection message was received on.
pAttr – pointer to attribute value.
pValue– pointer to data to be written
len – length of data
offset– offset of the first octet to be written
validCfg– valid configuration

Returns:
SUCCESS: 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

VI. *GAPBondMgr API*

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

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

VI.1 Commands

bStatus_t GAPBondMgr_SetParameter(uint16 param, void *pValue)

Description: Set a GAP Bond Manager parameter.

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

Returns:
SUCCESS: 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)

Description: Get a GAP Bond Manager parameter.

Parameters:
param – Profile parameter ID (see Appendix VI.2)
pValue – pointer to a location to get the value. This is dependent on the param ID and will be cast to the appropriate data type.

Returns:
SUCCESS: param was successfully placed in *pValue*
INVALIDPARAMETER: *param* was not valid

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

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

Parameters:
addrType - address type of the peer device:
peerAddr - peer device address
connHandle - connection handle
role - master or slave role

Returns:
SUCCESS: GAPBondMgr was notified of link establishment

void GAPBondMgr_LinkTerm(uint16 connHandle)

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

Parameters:
connHandle - connection handle

void GAPBondMgr_SlaveReqSecurity(uint16 connHandle)

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

Parameters:
connHandle - connection handle

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

Description: Resolve an address from bonding information.

Parameters:
addrType - address type of the peer device:
peerAddr - peer device address
pResolvedAddr - pointer to buffer to put the resolved address

Returns:
Bonding index (0 - (GAP_BONDINGS_MAX-1): if address was found...
GAP_BONDINGS_MAX: if address was not found

bStatus_t GAPBondMgr_ServiceChangeInd(uint16 connectionHandle, uint8 setParam)

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

Parameters:
connHandle - connection handle of the connected device or 0xFFFF for all devices in database.
setParam - TRUE to set the service change indication, FALSE to clear it.

Returns:
SUCCESS - 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)

Description: Update the Characteristic Configuration in a bond record.

Parameters:
connectionHandle - connection handle of the connected device or 0xFFFF for all devices in database.
attrHandle - attribute handle
value - characteristic configuration value

Returns:
SUCCESS - 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)

Description: Register callback functions with the bond manager.

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

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

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

Parameters:
connectionHandle - connection handle of the connected device or 0xFFFF for all devices in database.
status - SUCCESS if passcode is available, otherwise see SMP_PAIRING_FAILED_DEFINES in gapbondmgr.h
passcode - integer value containing the passcode

Returns:
SUCCESS: 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)

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

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

Parameters:
pMsg - GAP event message

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

uint8 GAPBondMgr_CheckNVLen(uint8 id, uint8 len)

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

Parameters:
id - NV ID.
len - lengths in bytes of item.

Returns:
SUCCESS: NV item is the correct length
FAILURE: NV item is an incorrect length

VI.2 Configurable Parameters

ParamID	R/W	Size	Description
GAPBOND_PAIRING_MODE	R/W	uint8	Default is GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_INITIATE_WAIT	R/W	uint16	Pairing Mode Initiate wait timeout. This is the time it will wait for a Pairing Request before sending the Slave Initiate Request. Default is 1000(in milliseconds)
GAPBOND_MITM_PROTECTION	R/W	uint8	Man-In-The-Middle (MITM) basically turns on Passkey protection in the pairing algorithm. Default is 0 (disabled).
GAPBOND_IO_CAPABILITIES	R/W	uint8	Default is GAPBOND_IO_CAP_DISPLAY_ONLY
GAPBOND_OOB_ENABLED	R/W	uint8	OOB data available for pairing algorithm. Default is 0(disabled).
GAPBOND_OOB_DATA	R/W	uint8[16]	OOB Data. Default is all 0's.

GAPBOND_BONDING_ENABLED	R/W	uint8	Request Bonding during the pairing process if enabled. Default is 0(disabled).
GAPBOND_KEY_DIST_LIST		uint8	The key distribution list for bonding. Default is sEncKey, sIdKey, mIdKey, mSign enabled.
GAPBOND_DEFAULT_PASSCODE		uint32	The default passcode for MITM protection. Range is 0 - 999,999. Default is 0.
GAPBOND_ERASE_ALLBONDS	W	None	Erase all of the bonded devices.
GAPBOND_KEYSIZE	R/W	uint8	Key Size used in pairing. Default is 16.
GAPBOND_AUTO_SYNC_WL	R/W	uint8	Clears the White List adds to it each unique address stored by bonds in NV. Default is FALSE.
GAPBOND_BOND_COUNT	R	uint8	Gets the total number of bonds stored in NV. Default is 0 (no bonds).
GAPBOND_BOND_FAIL_ACTION	W	uint8	Possible actions Central may take upon an unsuccessful bonding. Default is 0x02 (Terminate link upon unsuccessful bonding).
GAPBOND_ERASE_SINGLEBOND	W	uint8[9]	Erase a single bonded device. Must provide address type followed by device address.

VI.3 Callbacks

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

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

VI.3.1 Passcode Callback (passcodeCB)

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

```
typedef void (*pfnPasscodeCB_t)
(
uint8  *deviceAddr,                    //!<   address   of   device   to   pair   with,   and
could be either public or random.
  uint16 connectionHandle,             //!< Connection handle
  uint8  uiInputs,                     //!< Pairing User Interface Inputs - Ask user to input
passcode
  uint8  uiOutputs                     //!< Pairing User Interface Outputs - Display passcode
 );
```

Based on the parameters passed to this callback such as the pairing user interface inputs / outputs, the application should act accordingly by displaying the passcode or initiating the entrance of a passcode.

VI.3.2 Pairing State callback (pairStateCB)

This callback will return the current pairing state to the application whenever the state changes as well as the current status of the pairing / bonding process associated with the current state. This function is defined as:

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

The pairing states (`state`) are enumerated as:

- GAPBOND_PAIRING_STATE_STARTED
  - The following `status` are possible for this `state`
    - SUCCESS (0x00): pairing has been initiated.
- GAPBOND_PAIRING_STATE_COMPLETE
  - The following `status` are possible for this `state`
    - SUCCESS (0x00): pairing is complete. Session keys have been exchanged.
    - SMP_PAIRING_FAILED_PASSKEY_ENTRY_FAILED (0x01): user input failed
    - SMP_PAIRING_FAILED_OOB_NOT_AVAIL (0x02): Out-of-band data not available
    - SMP_PAIRING_FAILED_AUTH_REQ (0x03): IO capabilities of devices do not allow for authentication
    - SMP_PAIRING_FAILED_CONFIRM_VALUE (0x04): the confirm value does not match the calculated compare value
    - SMP_PAIRING_FAILED_NOT_SUPPORTED (0x05): pairing is not supported
    - SMP_PAIRING_FAILED_ENC_KEY_SIZE (0x06): encryption key size is insufficient
    - SMP_PAIRING_FAILED_CMD_NOT_SUPPORTED (0x07): The SMP command received is not supported on this device
    - SMP_PAIRING_FAILED_UNSPECIFIED (0x08): encryption failed to start
    - bleTimeout (0x17): pairing failed to complete before timeout
    - bleGAPBondRejected (0x32): keys did not match
- GAPBOND_PAIRING_STATE_BONDED
  - The following `status` are possible for this `state`
    - LL_ENC_KEY_REQ_REJECTED (0x06): encryption key is missing
    - LL_ENC_KEY_REQ_UNSUPPORTED_FEATURE (0x1A): feature is not supported by the remote device
    - LL_CTRL_PKT_TIMEOUT_TERM (0x22): Timeout waiting for response
    - bleGAPBondRejected (0x32): this is received due to one of the above three errors

VII. *HCI Extension API*

This section will describe the vendor specific HCI Extension API. These proprietary commands are specific to the CC254x device. In the case where more detail is needed, an example will be provided.

Note that, unless stated otherwise, the return values for all of these commands will always be SUCCESS. However, this does not indicate successful completion of the command. These commands will result in corresponding events that should be checked by the calling application.

VII.1 Commands

hciStatus_t HCI_EXT_AdvEventNoticeCmd (uint8 taskID, uint16 taskEvent)

Description: This command is used to configure the device to set an event in the user task after each advertisement event completes. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent is taken to be "disable".

Note: This command will not return any events but it does have a meaningful return status.

Parameters:
taskID– User's task ID.
taskEvent- User's task event. This must be a single bit value.

Returns:
SUCCESS: event configured correctly
LL_STATUS_ERROR_BAD_PARAMETER: there is more than one bit set.

Example (code additions to *SimpleBLEPeripheral.c*):

1. Define the event in the application

```
// BLE Stack Events
```

```
#define SBP_ADV_CB_EVT                          0x0001
```

2. Configure the BLE Protocol Stack to return the event (in `simpleBLEPeripheral_init()`)

```
HCI_EXT_AdvEventNoticeCmd(simpleBLEPeripheral_TaskID, SBP_ADV_CB_EVT);
```

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

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

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

hciStatus_t HCI_EXT_BuildRevision(uint8 mode, uint16 userRevNum)

Description: This command is used to a) allow the embedded user code to set their own 16 bit revision number, or b) read the build revision number of the BLE Stack library software. The default value of the user revision number is *zero*.

When the user updates a BLE project by adding their own code, they may use this API to set their own revision number. When called with mode set to HCI_EXT_SET_APP_REVISION, the stack will save this value. No event will be returned from this API when used this way as it is intended to be called from within the target itself. Note however that this does not preclude this command from being received via the HCI. However, no event will be returned.

Parameters:
Mode - HCI_EXT_SET_APP_REVISION, HCI_EXT_READ_BUILD_REVISION
userRevNum – Any 16 bit value

Returns (only when mode == HCI_EXT_SET_USER_REVISION):
SUCCESS: build revision set succesfully
LL_STATUS_ERROR_BAD_PARAMETER: not a valid mode

Corresponding Events (only when mode == HCI_EXT_SET_USER_REVISION):
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ClkDivOnHaltCmd (uint8 control)

Description: This command is used to configure the Link Layer to divide the system clock when the MCU is halted during a radio operation. The default system value for this feature is disabled.

Note: This command is only valid when the MCU is halted during RF operation (please see HCI_EXT_HaltDuringRfCmd).

Parameters:
control – one of…
HCI_EXT_DISABLE_CLK_DIVIDE_ON_HALT
HCI_EXT_ENABLE_CLK_DIVIDE_ON_HALT

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ConnEventNoticeCmd (uint8 taskID, uint16 taskEvent)

Description: This command is used to configure the device to set an event in the user task after each connection event completes. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent is taken to be "disable".

Note: Only a Slave with one connection is supported (this API will only work while device is configured as a slave and connected to one master). This command should be sent AFTER a connection is established.

Note: This command will not return any events but it does have a meaningful return status.

Parameters:
taskID– User's task ID.

taskEvent- User's task event.

Returns:
SUCCESS or FAILURE
LL_STATUS_ERROR_BAD_PARAMETER: there is more than one bit set.

Example (code additions to *SimpleBLEPeripheral.c*):

1. Define the event in the application

```
// BLE Stack Events
#define SBP_CON_CB_EVT                        0x0001
```

2. Configure the BLE Protocol Stack to return the event (in `SimpleBLEPeripheral_processStateChangeEvt()`) AFTER the connection is established.

```
    case GAPROLE_CONNECTED:
      {
         HCI_EXT_ConnEventNoticeCmd (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)

Description: This command is used to inform the Controller whether the Host is using NV memory during BLE operations. The default system value for this feature is NV In Use.

When the NV is not in use during BLE operations, the Controller is able to bypass internal checks that reduce overhead processing, thereby reducing average power consumption.

Note: This command is only allowed when the BLE Controller is idle.
Note: Using NV when declaring it is not in use may result in a hung BLE Connection.

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)*

Description: This command is used to decrypt encrypted data using the AES128 .

Note: This should only be used by the application. Incoming encrypted BLE data is automatically decrypted by the stack and does not require the use of this API.

Parameters:
key – Pointer to 16 byte encryption key.
encText - Pointer to 16 byte encrypted data.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DelaySleepCmd (uint16 delay)

Description: This command is used to set the delay before sleep occurs after Reset or upon waking from sleep to allow the external 32kHz crystal to stabilize. If this command is never used, the default delay is 400ms on the CC254x.

If the customer's hardware requires a different delay or does not require this delay at all, it can be changed by calling this command during their OSAL task initialization. A zero delay value will eliminate the delay after Reset and (unless changed again) all subsequent wakes from sleep; a non-zero delay value will change the delay after Reset and (unless changed again) all subsequent wakes from sleep.

If this command is used any time after system initialization, then the new delay value will be applied the next time the delay is used.

Note: This delay only applies to Reset and Sleep. If a periodic timer is used, or a BLE operation is active, then only Sleep is used, and this delay will only occur after Reset.

Note: There is no distinction made between a hard and soft reset. The delay (if non-zero) will be applied the same way in either case.

Parameters:
delay – 0x0000…0x003E8 in milliseconds

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_DisconnectImmedCmd (uint16 connHandle)

Description: This command is used to disconnect a connection immediately. This command can be useful for when a connection needs to be ended without the latency associated with the normal BLE Controller Terminate control procedure.

Note that the Host issuing the command will still receive the HCI Disconnection Complete event with a Reason status of 0x16 (i.e. Connection Terminated by Local Host), followed by an HCI Vendor Specific Event.

Parameters:
connHandle– The handle of the connection.

Corresponding Events
HCI_Disconnection_Complete
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_EnablePTMCmd (void)

Description: This command is used to enable Production Test Mode (PTM). This mode is used by the customer during assembly of their product to allow limited access to the BLE Controller for testing and configuration. This mode will remain enabled until the device is reset. Please see the related application note for additional details.

Note: This commands will cause a reset of the controller so in order to re-enter the application, the device should be reset.

Note: This command will not return any events.

Return Values:
HCI_SUCCESS: Successfully entered PTM

hciStatus_t HCI_EXT_EndModemTestCmd (void)

Description: This command is used to shut down a modem test. A complete link layer reset will take place.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ExtendRfRangeCmd (void)

Description: This command is used to configure the CC254x to automatically control the TI CC2590 2.4GHz RF Front End device. Using the CC2590 allows a maximum Tx output power of 10dBm (the specified BLE maximum), and increases Rx sensitivity, thus extending the RF range of the CC254x. Once this command is used, the configuration will

not 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 is also taken to control RF gain. These GPIOs are therefore not available when using this feature.

This command can be used in combination with HCI_EXT_SetTxPowerCmd, resulting in a cumulative Tx output power. Therefore, for the CC2540 only, attempting to set Tx output power to 4dBm (i.e. using HCI_EXT_TX_POWER_4_DBM), will instead set the Tx output power to 0dBm.

The command HCI_EXT_SetRxGainCmd should be used to set the Rx gain per usual. That is, the CC254x Rx Standard/High gain setting is mirrored to the CC2590 High Gain Mode (HGM) Low/High setting.

When this command is used, the CC254x Tx output power and Rx gain will retain their previous values, unless the previous Tx output power value was set to 4dBm on the CC2540. In this case, as previously explained, the value will be set to 0dBm..

Corresponding Events
HCI_Disconnection_Complete
HCI_VendorSpecifcCommandCompleteEvent

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

Description: This command is used to get the number of allocated connections, the number of active connections, and for each active connection, 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 only applies to central devices for the CC254x as peripheral devices are limited to one simultaneous connection.

If all parameters are NULL, then the call to this command is considered a network processor call via a transport layer, and the results will be provided via a vendor specific command complete event.

If any parameter is not NULL, then the call to this command is considered a direct function call and the valid pointers will be used to store the result. In this case, it is the user's responsibility to ensure there is sufficient memory allocated! Note that partial results can be obtained by selective use of the pointers. For example, if only the number of active connections is desired, this can be obtained as follows:

```
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: Connection ID, Connection Role, Peer Device Address, and Peer Address Type, which will require (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)

Description: This command is used to enable or disable 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: This command will be disallowed if there are any active BLE connections.

Note: The HCI_EXT_ClkDivOnHaltCmd will be disallowed if the halt during RF is not enabled..

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)

Description: This command is used to configure and map a CC254x I/O Port as a General-Purpose I/O (GPIO) output signal that reflects the Power Management (PM) state of the CC254x device. The GPIO output will be High on Wake, and Low upon entering Sleep. This feature can be disabled by specifying HCI_EXT_PM_IO_PORT_NONE for the ioPort (ioPin is then ignored). The system default value upon hardware reset is disabled.

This command can be used to control an external DC-DC Converter (its actual intent) such has the TI TPS62730 (or any similar converter that works the same way). This command should be used with extreme care as it will override how the Port/Pin was previously configured! This includes the mapping of Port 0 pins to 32kHz clock output, Analog I/O, UART, Timers; Port 1 pins to Observables, Digital Regulator status, UART, Timers; Port 2 pins to an external 32kHz XOSC. The selected Port/Pin will be configured as an output GPIO with interrupts masked. Careless use can result in a reconfiguration that could disrupt the system. For example, if the Port/Pin is being used as part of the serial interface for the device, the Port/Pin will be reconfigured from its original Peripheral function to a GPIO, disrupting the serial port. It is therefore the user's responsibility to ensure the selected Port/Pin does not cause any conflicts in the system.

Note: Only Pins 0, 3 and 4 are valid for Port 2 since Pins 1 and 2 are mapped to debugger signals DD and DC.

Note: Port/Pin signal change will obviously only occur when Power Savings is enabled.

Note: The CC254xEM modules map the TI TPS62730 control signal to P1.2, which happens to map to the SmartRF05EB LCD Chip Select. Thus, the LCD can't be used when setup this way.Parameters:
ioPort – one of:
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:
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)

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

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Note: The device will transmit at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemTestRxCmd(uint8 rxFreq)

Description: This API is used to start a continuous receiver modem test using a modulated carrier wave tone, at the frequency that corresponds to the specific RF channel. Any received data is discarded. Receiver gain may be adjusted using the HCI_EXT_SetRxGain command. RSSI may be read during this test by using the HCI_ReadRssi command. Use HCI_EXT_EndModemTest command to end the test.

Note: The RF channel, not the BLE frequency, is specified! The RF channel can be obtained from the BLE frequency as follows: RF Channel = (BLE Frequency – 2402) / 2.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

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

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ModemTestTxCmd(uint8 cwMode, uint8 txFreq)

Description: This API is used to start a continuous transmitter modem test, using either a modulated or unmodulated carrier wave tone, at the frequency that corresponds to the specified RF channel. Use the HCI_EXT_EndModemTest command to end the test.

Note: The RF channel, not the BLE frequency, is specified by txFreq. The RF channel can be obtained from the BLE frequency as follows: RF Channel = (BLE Frequency – 2402) / 2.

Note: When the HCI_EXT_EndModemTest is issued to stop this test, a Controller reset will take place.

Note: The device will transmit at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

Parameters:
cwMode - HCI_EXT_TX_MODULATED_CARRIER, HCI_EXT_TX_UNMODULATED_CARRIER
txFreq - Transmit RF channel k=0..39, where BLE F=2402+(k*2MHz)

Corresponding Event:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_NumComplPktsLimitCmd (uint8 limit, uint8 flushOnEvt)

Description: This command is used to set the limit on the minimum number of complete packets before a Number of Completed Packets event is returned by the Controller. If the limit is not reached by the end of a connection event, then the Number of Completed Packets event will be returned (if non-zero) based on the **flushOnEvt** flag. The limit can be set from one to the maximum number of HCI buffers (please see the LE Read Buffer Size command in the Bluetooth Core specification). The default limit is *one*; the default **flushOnEvt** flag is *FALSE*.

Note: The purpose of this command is to minimize the overhead of sending multiple Number of Completed Packet events, thus maximizing the processing available to increase over-the-air throughput. This is often used in conjunction with HCI_EXT_OverlappedProcessingCmd.

Parameters:
limit– From 1 to HCI_MAX_NUM_DATA_BUFFERS (returned by HCI_LE_ReadBufSizeCmd).
flushOnEvt-
- HCI_EXT_DISABLE_NUM_COMPL_PKTS_ON_EVENT: only return a Number of Completed Packets event when the number of completed packets is greater than or equal to the *limit*
- HCI_EXT_ENABLE_NUM_COMPL_PKTS_ON_EVENT: return the Number of Completed Packets

event at the end of every connection event.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_OnePacketPerEventCmd (uint8 control)

Description: This command is used to configure the Link Layer to only allow one packet per connection event. The default system value for this feature is *disabled*.

This command can be used to tradeoff throughput and power consumption during a connection. When enabled, power can be conserved during a connection by limiting the number of packets per connection event to one, at the expense of more limited throughput. When disabled, the number of packets transferred during a connection event is not limited, at the expense of higher power consumption per connection event.

Note: A thorough power analysis of the system needs to be performed before it is certain that this command will save power. It may be more power efficient to transfer multiple packets per connection event.

Parameters:
control – HCI_EXT_DISABLE_ONE_PKT_PER_EVT, HCI_EXT_ENABLE_ONE_PKT_PER_EVT

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

hciStatus_t HCI_EXT_OverlappedProcessingCmd (uint8 mode)

Description: This command is used to enable or disable overlapped processing. The default is disabled. See the wiki page for more information.

Parameters:
mode – one of…
HCI_EXT_DISABLE_OVERLAPPED_PROCESSING,
HCI_EXT_ENABLE_OVERLAPPED_PROCESSING

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

hciStatus_t HCI_EXT_PacketErrorRateCmd (uint16 connHandle, uint8 command)

Description: This command is used to Reset or Read the Packet Error Rate counters for a connection. When Reset, the counters are cleared; when Read, the total number of packets received, the number of packets received with a CRC error, the number of events, and the number of missed events are returned.

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

Parameters:
connId– The connection ID on which to perform the command
command- HCI_EXT_PER_RESET, HCI_EXT_PER_READ

Corresponding Event:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_PERbyChanCmd (uint16 connHandle, perByChan_t *perByChan)

Description: This command is used to start or end Packet Error Rate by Channel counter accumulation for a connection, and can be used by an application to make Coexistence assessments. Based on the results, an application can perform an Update Channel Classification command to limit channel interference from other wireless standards. If ***perByChan** is NULL, counter accumulation will be discontinued. If ***perByChan** is not NULL, then it is assumed that there is sufficient memory at this location for the PER data, based on the following type definition **perByChan_t** located in **ll.h**:

```
#define LL_MAX_NUM_DATA_CHAN 37
// Packet Error Rate Information By Channel
typedef struct
{
  uint16 numPkts[ LL_MAX_NUM_DATA_CHAN ];
  uint16 numCrcErr[ LL_MAX_NUM_DATA_CHAN ];
} perByChan_t;
```

Note: **It is the user's responsibility to ensure there is sufficient memory allocated in the perByChan structure!** The user is also responsible for maintaining the counters, clearing them if required before starting accumulation.

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

Note: This command can be used in combination with HCI_EXT_PacketErrorRateCmd.

Parameters:
connHandle – The connection ID on which to accumulate the data.
perByChan- Pointer to PER by Channel data, or NULL.

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_ResetSystemCmd (uint8 mode)

Description: This command is used to issue a hard or soft system reset. A hard reset will be caused by a watchdog timer timeout while a soft reset is caused by resetting the PC to zero.

Note that the reset occurs after a 100 ms delay in order to allow the correspond event to be returned to the application.

Parameters:
mode – HCI_EXT_RESET_SYSTEM_HARD

Corresponding event:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SaveFreqTuneCmd (void)

Description: This PTM-only command is used to save this device's Frequency Tuning setting in non-volatile memory. This setting will be used by the BLE Controller upon reset, and when waking from Sleep.

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

*hciStatus_t HCI_EXT_SetBDADDRCmd(uint8 *bdAddr)*

Description: This command is used to set the device's BLE address (BDADDR). This address will override the device's address determined when the device is reset). To restore the device's initialized address stored in flash, issue this command with an invalid address (0xFFFFFFFFFFFF).

Note: This command is only allowed when the Controller is in the Standby state. This command is intended to only be used during initialization. Changing the device's BDADDR after various BLE operations have already taken place may cause unexpected problems.

Parameters:
bdAddr – A pointer to a buffer to hold this device's address. An invalid address (i.e. all FF's) will restore this device's address to the address set at initialization.

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd (uint8 control)

Description: This command is used to configure the Link Layer fast transmit response time feature. The

default system value for this feature is *enabled*.

Note: This command is only valid for a Slave controller.

When the Host transmits data, the controller (by default) ensures the packet is sent over the LL connection with as little delay as possible, even when the connection is configured to use slave latency. That is, the transmit response time will tend to be no longer than the connection interval (instead of waiting for the next effective connection interval due to slave latency). This results in lower power savings since the LL may need to wake to transmit during connection events that would normally have been skipped due to slave latency. If saving power is more critical than fast transmit response time, then this feature can be disabled using this command. When disabled, the transmit response time will be no longer than the effective connection interval (slave latency + 1 times the connection interval).

Parameters:
control – HCI_EXT_ENABLE_FAST_TX_RESP_TIME, HCI_EXT_DISABLE_FAST_TX_RESP_TIME

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetFreqTuneCmd (uint8 step)

Description: This PTM-only command is used to set this device's Frequency Tuning either up one step or down one step. When the current setting is already at its max value, then stepping up will have no effect. When the current setting is already at its min value, then stepping down will have no effect. This setting will only remain in effect until the device is reset unless HCI_EXT_SaveFreqTuneCmd is used to save it in non-volatile 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)

Description: This command is used to set the Controller's Local Supported Features.

Note: This command can be issued either before or after one or more connections are formed. However, the local features set in this manner are only effective if performed *before* a Feature Exchange Procedure has been initiated by the Master. Once this control procedure has been completed for a particular connection, only the exchanged feature set for that connection will be used. Since the Link Layer may initiate the feature exchange procedure autonomously, it is best to use this command before the connection is formed.

Note that the features are initialized by the controller upon start up. The need for this command is very unlikely. The defines for the feature values are in ll.h.

Parameters:
localFeatures – A pointer to the Feature Set where each bit where each bit corresponds to a feature
0: Feature shall not be used.
1: Feature can be used.

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd (uint8 txPower)

Description: This command is used to override the RF transmitter output power used by the Direct Test Mode (DTM). Normally, the maximum transmitter output power setting used by DTM is the maximum transmitter output power setting for the device (i.e. 5 dBm). This command will change the value used by DTM.

Note: When DTM is ended by a call to HCI_LE_TestEndCmd, or a HCI_Reset is used, the transmitter output power setting is restored to the default value of 0 dBm.
Parameters:
txPower – one of:

HCI_EXT_TX_POWER_MINUS_21_DBM
HCI_EXT_TX_POWER_MINUS_18_DBM
HCI_EXT_TX_POWER_MINUS_15_DBM
HCI_EXT_TX_POWER_MINUS_12_DBM
HCI_EXT_TX_POWER_MINUS_9_DBM
HCI_EXT_TX_POWER_MINUS_6_DBM
HCI_EXT_TX_POWER_MINUS_3_DBM
HCI_EXT_TX_POWER_0_DBM
HCI_EXT_TX_POWER_1_DBM
HCI_EXT_TX_POWER_2_DBM
HCI_EXT_TX_POWER_3_DBM
HCI_EXT_TX_POWER_4_DBM
HCI_EXT_TX_POWER_5_DBM

Corresponding Events
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetRxGainCmd(uint8 rxGain)

Description: This command is used to set the RF receiver gain. The default system value for this feature is "standard receiver gain."

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)

Description: This command is used to set this device's Sleep Clock Accuracy (SCA) value, in parts per million (PPM), from 0 to 500. For a Master device, the value is converted to one of eight ordinal values representing a SCA range per the Bluetooth Spec [13], which will be used when a connection is created. For a Slave device, the value is directly used. The system default value for a Master and Slave device is 50ppm and 40ppm, respectively.

Note: This command is only allowed when the device is *not* in a connection.

Note: The device's SCA value remains unaffected by an HCI Reset.

Parameters:
scaInPPM – This device's SCA in PPM from 0..500.

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd (uint8 mode)

Description: This command is used to enable or disable the Slave Latency Override, allowing the user temporarily suspend Slave Latency even though it is still active for the connection. That is, once enabled, the device will wake up for every connection until Slave Latency Override is disabled again. The default value is *Disable.*

Note: This only applies to devices acting in the Slave role.

This can be helpful when the Slave application knows it will soon receive something that needs to be handled without delay. Note that this does not actually change the slave latency connection parameter: the device will simply wake up for each connection event.

Parameters:
control – HCI_EXT_ENABLE_SL_OVERRIDE, HCI_EXT_DISABLE_SL_OVERRIDE

Corresponding Event
HCI_VendorSpecifcCommandCompleteEvent

hciStatus_t HCI_EXT_SetTxPowerCmd(uint8 txPower)

Description: This command is used to set the RF transmitter output power. The default system value for this feature is *0 dBm*.

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

Corresponding Events:
HCI_VendorSpecifcCommandCompleteEvent:
HCI_EXT_TX_POWER_MINUS_21_DBM
HCI_EXT_TX_POWER_MINUS_18_DBM
HCI_EXT_TX_POWER_MINUS_15_DBM
HCI_EXT_TX_POWER_MINUS_12_DBM
HCI_EXT_TX_POWER_MINUS_9_DBM
HCI_EXT_TX_POWER_MINUS_6_DBM
HCI_EXT_TX_POWER_MINUS_3_DBM
HCI_EXT_TX_POWER_0_DBM
HCI_EXT_TX_POWER_1_DBM
HCI_EXT_TX_POWER_2_DBM
HCI_EXT_TX_POWER_3_DBM
HCI_EXT_TX_POWER_4_DBM
HCI_EXT_TX_POWER_5_DBM

VII.2 Host Error Codes

This section lists the various possible error codes generated by the Host. If an HCI extension command that sent a Command Status with the error code 'SUCCESS' before processing may find an error during execution then the error is reported in the normal completion command for the original command.

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

Value	Parameter Description
0x00	SUCCESS
0x01	FAILURE
0x02	INVALIDPARAMETER
0x03	INVALID_TASK
0x04	MSG_BUFFER_NOT_AVAIL
0x05	INVALID_MSG_POINTER
0x06	INVALID_EVENT_ID
0x07	INVALID_INTERRUPT_ID
0x08	NO_TIMER_AVAIL
0x09	NV_ITEM_UNINIT
0x0A	NV_OPER_FAILED
0x0B	INVALID_MEM_SIZE
0x0C	NV_BAD_ITEM_LEN
0x10	bleNotReady
0x11	bleAlreadyInRequestedMode

0x12	bleIncorrectMode
0x13	bleMemAllocError
0x14	bleNotConnected
0x15	bleNoResources
0x16	blePending
0x17	bleTimeout
0x18	bleInvalidRange
0x19	bleLinkEncrypted
0x1A	bleProcedureComplete
0x30	bleGAPUserCanceled
0x31	bleGAPConnNotAcceptable
0x32	bleGAPBondRejected
0x40	bleInvalidPDU
0x41	bleInsufficientAuthen
0x42	bleInsufficientEncrypt
0x43	bleInsufficientKeySize
0xFF	INVALID_TASK_ID

Table 1: List of Possible Host Error Codes