Texas Instruments CC2540 *Bluetooth*® Low Energy Software Developer's Guide v1.1

Document Number: SWRU271A

Table Of Contents

References

Included with Texas Instruments *Bluetooth* Low Energy v1.1 Stack Release (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-CC2540-1.1\):

[1] OSAL API Guide

C:\Texas Instruments\BLE-CC2540-1.1\Documents\osal\OSAL API.pdf

[2] HAL API Guide

C:\Texas Instruments\ BLE-CC2540-1.1\Documents\hal\HAL API.pdf

[3] TI BLE Vendor Specific HCI Reference Guide

C:\Texas Instruments\ BLE-CC2540-1.1\Documents\
TI_BLE_Vendor_Specific_HCI_Guide.pdf

[4] Texas Instruments CC2540 *Bluetooth* Low Energy API Guide

C:\Texas Instruments\ BLE-CC2540-1.1\Documents\BLE_API_Guide_main.htm

[5] Texas Instruments CC2540 *Bluetooth* Low Energy Sample Applications Guide

C:\Texas Instruments\ BLE-CC2540-1.1\Documents\
TI_BLE_Sample_Applications_Guide.pdf

Available for download from the Texas Instruments web site:

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

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

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

[7] *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

[8] *Device Information Service (Bluetooth* Specification*)*, version 1.0 (24-May-2011)

https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=238689

1 Overview

The purpose of this document is to give an overview of the Texas Instruments CC2540 *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 [7].

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 system was created for the purpose of transmitting very small packets of data at a time, while consuming significantly less power than BR/EDR devices.

Devices that can support BR and BLE are referred to as "dual-mode" devices. 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. 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.

1.2 BLE Protocol Stack Basics

The BLE protocol stack architecture is illustrated here:

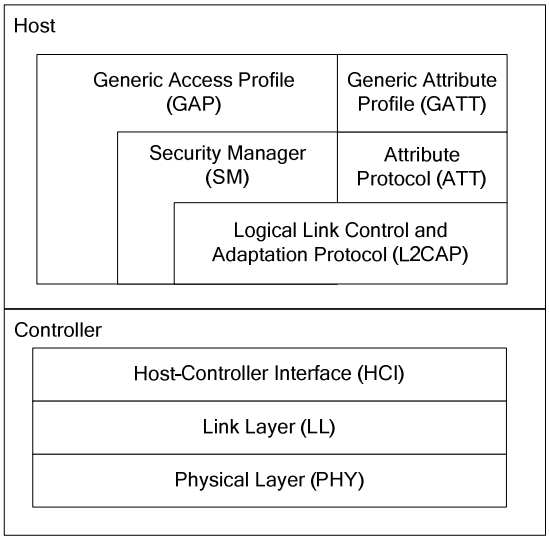

Figure 1: BLE Protocol Stack

The protocol stack consists of two sections: the controller and the host. This separation of control and host goes back to standard *Bluetooth* BR/EDR devices, in which the two sections were often implemented separately. Any profiles and applications that are being used sit on top of the GAP and GATT layers of the stack.

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

The LL essentially controls the RF state of the device, with the device being in one of five possible states: standby, advertising, scanning, initiating, or connected. Advertisers transmit data without being in a connection, while scanners listen for advertisers. An initiator is a device that is responding to an advertiser with a connection request. If the advertiser accepts, both the advertiser and initiator will enter a connected state. When a device is in a connection, it will be connected in one of two roles: master or slave. The device that initiated the connection becomes the master, and the device that accepted the request becomes the slave.

The 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.

The L2CAP layer provides data encapsulation services to the upper layers, allowing for logical end-to-end communication of data.

The SM layer defines the methods for pairing and key distribution, and provides functions for the other layers of the stack to securely connect and exchange data with another device.

The GAP layer directly interfaces with the application and/or profiles, and handles device discovery and connection-related services for the device. In addition, GAP handles the initiation of security features.

The ATT protocol allows a device to expose certain pieces of data, known as "attributes", to another device. In the context of ATT, the device exposing attributes is referred to as the "server", and the peer device is referred to as the "client". The LL state (master or slave) of the device is independent of the ATT role of the device. For example, a master device may either be an ATT server or an ATT client, and a slave device may also be either an ATT server or an ATT client. It is also possible for a device to be both an ATT server and an ATT client simultaneously.

The GATT layer is a service framework that defines the subprocedures for using ATT. GATT specifies the structure of profiles. In BLE, all pieces of data that are being used by a profile or service are called "characteristics". 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.

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, a complete SoC (System-on-Chip) solution. The CC2540 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:

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

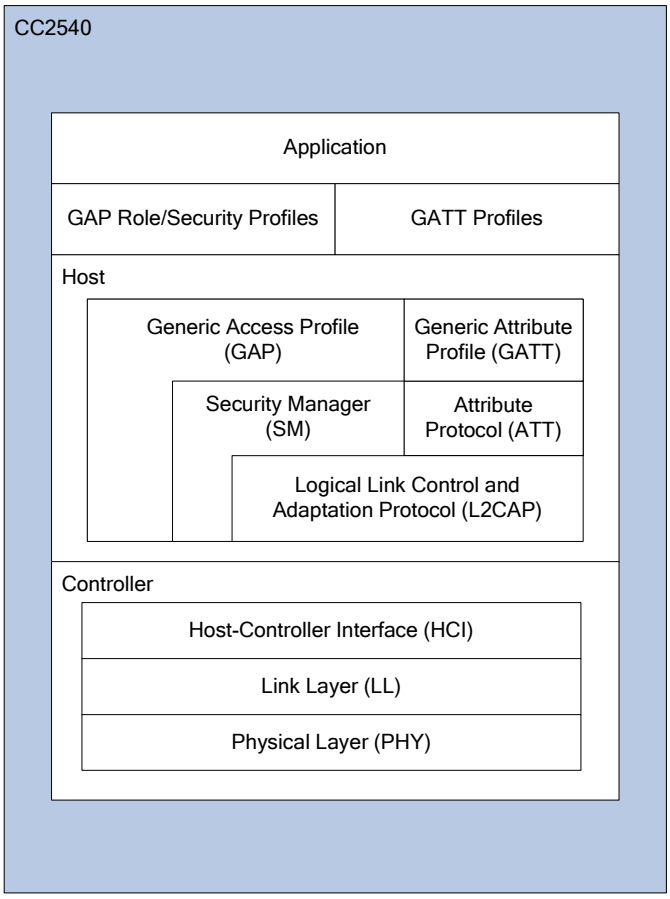

Figure 2: Single-Device Configuration

- **Network Processor:** The controller and host are implemented together on the CC2540, while the profiles and application are implemented separately. The application and profiles communicate with the CC2540 by means of vendor-specific HCI commands using a hardware 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. To use the network processor, the HostTestRelease project must be used. More information on the HostTestRelease project can be found in section 4.5.

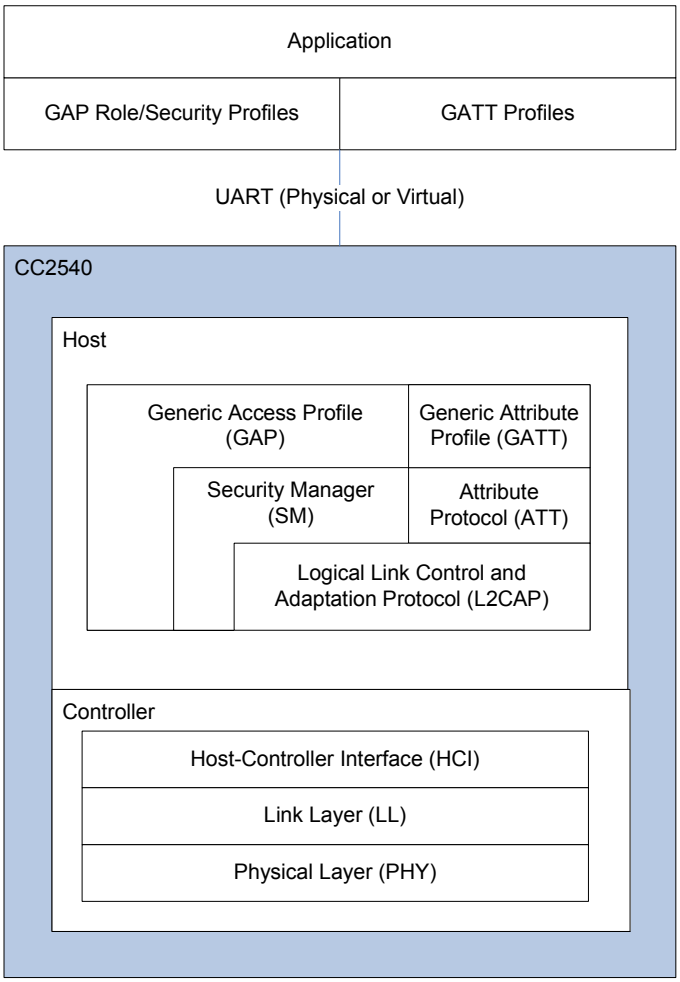

Figure 3: Network Processor Configuration

2.2 Operating System Abstraction Layer (OSAL)

The BLE protocol stack, the profiles, and all applications are all built around the Operating System Abstraction Layer (OSAL), which is similar to an operating system (OS) and allows software to prioritize multiple tasks and events. More information on the OSAL can be found in section 3.1.

2.3 Hardware Abstraction Layer (HAL)

This component provides an abstract interface to the hardware available on chip and on the board. It includes software for the UART communication interface, ADC, soft keys, and LED's. This code is included in source to allow the user to modify it to suit any hardware platform. More information on the HAL can be found in section 3.2.

2.4 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. 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 projects can be found in [5]

3 Software Overview

Software developed using the BLE 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.

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-CC2540-1.1**.

In this section, the SimpleBLEPeripheral project will be used as a reference; however all of the BLE projects included in the development kit will have a similar structure.

3.1 Operating System Abstraction Layer (OSAL)

The BLE protocol stack, the profiles, and all applications are all built around the Operating System Abstraction Layer (OSAL). The OSAL is not an actual operating system (OS) in the traditional sense, but rather a control loop that allows software to setup the execution of events. For each layer of software that requires this type of control, a task identifier (ID) must be created, a task initialization routine must be defined and added to the OSAL initialization, and an event processing routine must be defined. Optionally, a message processing routine may be defined as well. Several layers of the BLE stack, for example, are OSAL tasks, with the LL being the highest priority (since it has very strict timing requirements).

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

Note: The OSAL is capable of providing many more services than are covered in this guide, including message management, timer management, and more; however for many applications this level of depth is not required. This guide should serve as an introduction to the basic framework of the OSAL.

Additional information on the OSAL can be found in [1]:

3.1.1 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 the file **OSAL_SimpleBLEPeripheral.c**.

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 is **SimpleBLEPeripheral_Init**, and it has the highest task ID and therefore the lowest priority.

3.1.2 Task Events and Event Processing

After the OSAL completes initialization, it runs the executive loop checking for task events. This loop can be found in the function **osal_start_system** in the file **OSAL.c**. Task events are implemented as a 16-bit variable (one for each task) where each bit corresponds to a unique event. The definition and use of these event flags is completely up to the application.

For example, the SimpleBLEPeripheral application defines a flag in **simpleBLEPeripheral.h**: **SBP_START_DEVICE_EVT** (0x0001), which indicates that the initial start has completed, and the application 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, which is covered in section 3.1.3).

When the OSAL detects an event for a task, it will call that task's event processing routine. The layer must add its 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 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 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.

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 task ID of the task that will be processing the event and the event flag as parameters; however for a third parameter in **osal_start_timerEx** you input a timeout value in milliseconds. The OSAL will set a timer, and the specified event will not get set until the timer expires.

3.1.3 Heap Manager

OSAL provides basic memory management functions. The **osal_mem_alloc** function serves as a basic memory allocation function similar to the standard C malloc function, taking a single parameter determining the number of bytes to allocate, and returning a void pointer. 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, freeing up memory that was previously allocated using **osal_mem_alloc**.

3.1.4 OSAL Messages

OSAL also provides a system 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. To send an OSAL message, first the memory must be allocated by calling the **osal_msg_allocate** function, passing in the length of the message as the only parameter. 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, the **osal_msg_send** should be called, with the destination task for the message indicated as a parameter.

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**).

3.2 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) of the CC2540 software provides an interface of abstraction between the physical hardware to and the application and 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 drivers that are provided support the SmartRF05EB v1.8.1 + CC2540EM hardware platform, as well as the "keyfob" and USB Dongle boards included with CC2540DK-MINI development kit. When developing with a different hardware platform, it might be necessary to modify the HAL source for compatibility.

More information on the HAL API can be found in [2].

3.3 BLE Protocol Stack

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

3.3.1 Generic Access Profile (GAP)

The GAP layer of the BLE Protocol Stack is responsible for handling the device's access modes and procedures, including device discovery, link establishment, link termination, initiation of security features, and device configuration.

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 stack supports up to three simultaneous connections.

The BLE specification allows for certain combinations of multiple-roles. The default setup of the sample application is to only support the peripheral role, though source code is provided to run a combination peripheral and broadcaster role.

In a typical *Bluetooth* Low Energy system, the peripheral device advertises with specific data letting any central device know that it is a connectable device. This advertisement contains the device address, and can contain some additional data as well, such as the device name. The central device, upon receiving the advertisement, sends a "scan request" to the peripheral. The peripheral responds with a "scan response". This is the process of device discovery, in that the central device is now aware of the peripheral device, and knows that it can form a connection with it. The central device can then send out a request to establish a link with the peripheral device. A connection request contains a few connection parameters:

- **Connection Interval** – In a BLE connection between two devices, a frequency-hopping scheme is used, in that the two devices each send and receive data from one another on a specific channel, then "meet" at a new channel (the link layer of the BLE 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).

Different applications may require different connection intervals. The advantage of having a very long connection interval is that significant power is saved, since the device can sleep most of the time between connection events. The disadvantage is that if a device has data that it needs to send, it must wait until the next connection event.

The advantage of having a very short connection interval is that there is more opportunity for data to be sent or received, as the two devices will connect more frequently. The disadvantage is that more power will be consumed, since the device is frequently waking up for connection events.

- **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 32.0s.

- **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).

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))

Take 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.

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 your connection parameters. Selecting the correct group of connection parameters plays an important role in power optimization of your 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

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". This request is handled by the L2CAP layer of the protocol stack.

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.

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.

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

In many cases, the same central and peripheral devices will be regularly connecting and disconnecting from each other. BLE has a security feature that allows two devices, when pairing, to give each other a long-term set of security keys. This feature, called bonding, allows the two devices to quickly re-establish encryption and authentication after re-connecting without going through the full pairing process every time that they connect, as long as they store the long-term key information.

In the SimpleBLEPeripheral application, the management of the GAP role is handled by the GAP role profile, and the management of bonding information is handled by the GAP security profile. More information on the GAP profiles included with the CC2540 BLE Protocol Stack can be found in section 3.4

3.3.2 Generic Attribute Profile (GATT)

The GATT layer of the BLE Protocol Stack is designed to be used by the application for data communication between two connected devices. From a GATT standpoint, when two devices are connected they are each in one of two roles:

- **GATT Client** – This is the device that is reading/writing data from/to the GATT Server. In the case of the demo application, the USB Dongle is the GATT Client.
- **GATT Server** – This is the device containing the data that is being read/written by the GATT Client. In the case of the demo application, the keyfob is the GATT Server.

It is important to note that the GATT roles of Client and Server are completely independent from the BLE link-layer roles of slave and master, or from the GAP peripheral and central roles. A slave

can be either a GATT Client or a GATT Server, and a master can be either a GATT client or a GATT Server.

A GATT server consists of one or more GATT services, which are collections of data to accomplish a particular function or feature.

In the case of the SimpleBLEPeripheral application, there are three GATT services:

- **Mandatory GAP Service** – This service contains device and access information, such as the device name and vendor 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.
- **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**.

"Characteristics" are values that are used by a service, along with properties and configuration information. GATT defines the sub-procedures for discovering, reading, and writing attributes over a BLE connection.

The characteristic values, along with their properties and their configuration data (known as "descriptors") on the GATT Server are stored in the attribute table. The attribute table is simply a database containing small pieces of data called attributes. In addition to the value itself, each attribute has the following properties associated with it:

- **Handle** – this is essentially the attribute's "address" in the table. Every attribute has a unique handle.
- **Type** – this indicates what the data represents. It is often referred to as a "UUID" (universal unique identifier) assigned by the *Bluetooth* SIG, or a custom type
- **Permissions** – this enforces if and how a GATT client device can access the attribute's value.

GATT defines several sub-procedures for communication between the GATT server and client.

Here are a few of the sub-procedures:

- **Read Characteristic Value** – The client requests to read the characteristic value at a specific handle, and the server responds to the client with the value (assuming that the attribute has read permissions).
- **Read Using Characteristic UUID** – The client requests to read all characteristic values of a certain type, and the server responds to the client with the handles and values (assuming that the attribute has read permissions) of all characteristics with matching type. The client does not need to know the handles of these characteristics.
- **Read Multiple Characteristic Values** – The client requests to read the characteristic values of several handles in a single request, and the server responds to the client with the values (assuming that the attributes all have read permissions). The client must know how to parse the data between the different characteristic values.
- **Read Characteristic Descriptor** – The client requests to read a characteristic descriptor at a specific handle, and the server responds to the client with the descriptor value (assuming that the attribute has read permissions).
- **Discover Characteristic by UUID** – The client requests to discover the handle of a specific characteristic by its type. The server responds with the characteristic declaration, containing the handle of the characteristic value as well as the characteristic's permissions.

- **Write Characteristic Value** – The client requests to write a characteristic value at a specific handle to the server, and the server responds to the client to indicate whether the write was successful or not.
- **Write Characteristic Descriptor** – The client requests to write to a characteristic descriptor at a specific handle to the server, and the server responds to the client to indicate whether the write was successful or not.
- **Characteristic Value Notification** – The server notifies the client of a characteristic value. The client does not need to prompt the server for the data, nor does it need to send any response when a notification is received, but it must first configure the characteristic to enable notifications. A profile defines when the server is supposed to send the data.

Each profile initializes its corresponding service and internally registers the service with the GATT server on the device. The GATT server adds the entire service to the attribute table, and assigns unique handles to each attribute.

There are a few special attribute types that are found in the GATT attribute table, with values defined by the *Bluetooth* SIG:

- **GATT_PRIMARY_SERVICE_UUID** – This indicates the start of a new service, and the type of the service provided
- **GATT_CHARACTER_UUID** – This is known as the "characteristic declaration", and it indicates that the attribute immediately following it is a GATT characteristic value
- **GATT_CLIENT_CHAR_CFG_UUID** – This attribute represents a characteristic descriptor that corresponds to the nearest preceding (by handle) characteristic value in the attribute table. It allows the GATT client to enable notifications of the characteristic value
- **GATT_CHAR_USER_DESC_UUID** – This attribute represents a characteristic descriptor that corresponds to the nearest preceding (by handle) characteristic value in the attribute table. It contains an ASCII string with a description of the corresponding characteristic

These are just few of the special attribute types that are included in the BLE specification. For more details on other attribute types, see [4].

3.3.3 Using the GAP and GATT Stack API

The application and profiles directly call GAP and GATT API functions to perform BLE-related functions, such as advertising, connecting, and reading and writing characteristics. For detailed information on the APIs of the different layers of the BLE protocol stack, the interactive HTML guide can be consulted. For more details on this, see [4].

The general procedure for using the GAP or GATT API is as follows:

1. Call API function with appropriate parameters
2. Stack performs specified action and returns
3. After action is complete, or whenever the stack has information to report back to calling task, the stack sends an OSAL message to the calling task
4. Calling task receives and processes accordingly based on the message
5. Calling task de-allocates the message

An example of this procedure can be seen in the GAP peripheral role profile (**peripheral.c)**:

1. In order to initialize the device, the profile calls the GAP API function **GAP_DeviceInit**.
2. GAP performs the initialization and the function returns a value of **SUCCESS** (0x00).
3. After initialization is complete, the BLE stack sends an OSAL message back to the peripheral role profile with a header event value of **GAP_MSG_EVENT**, and an opcode value of **GAP_DEVICE_INIT_DONE_EVENT**.
4. The profile task receives a **SYS_EVENT_MSG** event, indicating that it has a message waiting. The profile receives the message and looks at the header and opcode values. Based on this, the profile knows to cast the message data to the appropriate type

(**gapDeviceInitDoneEvent_t**) and process accordingly. In this case, the profile stores the generated keys in the NV memory space and locally saves the device address.

5. The profile de-allocates the message and returns.

Here is one more example that would occur if a GATT client device wants to perform a GATT read request on a peer GATT server:

1. The application calls the GATT sub-procedure API function **GATT_ReadCharValue**, passing the connection handle, the characteristic handle (contained within the attReadReq_t data type), and it's own task ID as parameters.
2. GATT processes the request, and returns a value of **SUCCESS** (0x00).
3. The stack sends out the read request at the next connection event. When a read response is received from the remote device, the stack sends an OSAL message containing the data in the read response back to the application. The message contains a header event value of **GATT_MSG_EVENT** and a method value of **ATT_READ_RSP**.
4. The application task receives a **SYS_EVENT_MSG** event, indicating that it has a message waiting. The profile receives the message and looks at at the header and method values. Based on this, the profile knows to cast the message data to the appropriate type (**attReadRsp_t**) and retrieve the data that was received in the read response.
5. The profile de-allocates the message and returns.

3.3.4 GATT Server Application API

The GATT Server Application provides APIs to the higher layer GATT profiles to perform two primary functions:

- Register or deregister service attributes and callback functions from the GATT Server
- Add or delete a GATT Service. This API adds or deletes the GATT Service's attribute list and callback functions to/from the GATT Server Application

A profile may support one or more services. Each of the services may support characteristics or references to other services. Each characteristic contains a value and may contain optional descriptors. The service, characteristic, characteristic value and descriptors are all stored as attributes on the server.

The service attribute list to be registered with the GATT Server Application must start with a service attribute followed by all the attributes associated with that service attribute. Each service is an array of type **gattAttribute_t**, as defined in the file **gatt.h**.

3.3.5 Library Files

Even though a single library file is needed for a BLE application to use the stack, there are eight different files corresponding to eight different configurations (see section 2.1 for more information on stack / application configurations). Table 1 below can be used as a reference to determine the correct library file to use in the project:

Configuration	Library Contains:	LL Role	Power Management	UART HCI	Library File Name
Single-Device	Host + Controller	Slave	On	No	ble_single_chip_slave_pm_on.lib
Single-Device	Host + Controller	Slave	Off	No	ble_single_chip_slave_pm_off.lib
Single-Device	Host + Controller	Master	On	No	ble_single_chip_master_pm_on.lib
Single-Device	Host + Controller	Master	Off	No	ble_single_chip_master_pm_off.lib
Network Processor	Host + Controller	Slave	On	Yes	ble_network_processor_slave_pm_on.lib
Network Processor	Host + Controller	Slave	Off	Yes	ble_network_processor_slave_pm_off.lib

Network Processor	Host + Controller	Master	On	Yes	ble_network_processor_master_pm_on.lib
Network Processor	Host + Controller	Master	Off	Yes	ble_network_processor_master_pm_off.lib

Table 1: BLE Stack Libraries

3.4 Profiles

The BLE software development kit includes three GAP role profiles, one GAP security profile, and several sample GATT service profiles.

3.4.1 GAP Peripheral Role Profile

The peripheral role profile provides the means for the keyfob to advertise, connect with a central device (though the initiation of the connection must come from the central device), and request a specific set of connection parameters from a master device.

The primary API function prototypes for the peripheral role profile can be found in the file **peripheral.h**. The API provides functions to get and set certain GAP profile parameters: **GAPRole_GetParameter** and **GAPRole_SetParameter**, respectively. Here are a few GAP role parameters of interest:

- **GAPROLE_ADVERT_ENABLED** – This parameter enables or disables advertisements. The default value for this parameter is **TRUE**.
- **GAPROLE_ADVERT_DATA** – This is a string containing the data to appear in the advertisement packets. By setting this value to { 0x02, 0x01, 0x05 }, the device will use limited discoverable mode when advertising. More information on advertisement data types and definitions can be found in [7].
- **GAPROLE_SCAN_RSP_DATA** – This is a string containing the name of the device that will appear in scan response data. If an observer or central device is scanning and sends a scan request to the peripheral device, the peripheral will respond back with a scan response containing this string.
- **GAPROLE_ADVERT_OFF_TIME** – This parameter is used when the device is put into limited discoverable mode. It sets how long the device should wait before becoming discoverable again at the end of the limited discovery period. By setting this value to 0, the device will not become discoverable again until the **GAPROLE_ADVERT_ENABLED** is set back to **TRUE**.
- **GAPROLE_PARAM_UPDATE_ENABLE** – This enables automatic connection parameter update requests. The profile default value is **FALSE**.
- **GAPROLE_MIN_CONN_INTERVAL** – This parameter is the minimum desired connection interval value. The default value is 80, corresponding to 100ms.
- **GAPROLE_MAX_CONN_INTERVAL** – This parameter is the maximum desired connection interval value. The default value is 3200, corresponding to 4.0s.
- **GAPROLE_SLAVE_LATENCY** – This parameter is the desired slave latency. The default value is 0.
- **GAPROLE_TIMEOUT_MULTIPLIER** – This parameter is the desired connection supervision timeout. The default value is 1000 (corresponding to 10.0s)

The GAP peripheral role uses callback functions to notify the application of events. These are set up by means of the function **GAPRole_StartDevice**. This function initializes the GAP peripheral role, and should only be called once. Its single parameter is a pointer to a variable of type **gapRolesCBs_t**, which is a structure containing two function pointers:

- **pfnStateChange** – This function gets called every time the GAP changes states, with the new state of the GAP passed as a parameter.

- **pfnRssiRead** – This function gets called every time the RSSI has been read, with the RSSI value passed as a parameter.

It is up to the application to provide the actual callback functions. In the case of the sample application, the state change function is **peripheralStateNotificationCB**, while no function is defined for the RSSI read (a **NULL** pointer is passed as the parameter for **pfnRssiRead**).

The peripheral profile also contains an automatic connection parameter update feature that can be enabled by setting the **GAPROLE_PARAM_UPDATE_ENABLE** parameter to **TRUE**. If the feature is enabled and the peripheral device enters a connection with a connection interval that falls outside of the range of the desired interval, or with a slave latency or timeout setting that is not the desired value, the peripheral profile will automatically send a Connection Parameter Update Request with the desired parameters. As long as the parameter values are legal as per the BLE specification, the central device should accept the request and change the connection parameters. In the sample application, the desired connection parameters are set in the **SimpleBLEPeripheral_Init** function, and can easily be changed to other values.

Additional details on using the GAP Peripheral Role Profile can be found in [4].

3.4.2 GAP Peripheral / Broadcaster Multi-Role Profile

The peripheral / broadcaster multi-role profile operates almost identically to the peripheral role profile; however it provides additional functionality allowing the device to operate in both the peripheral and broadcaster GAP roles simultaneously. In order to use this multi-role functionality, the files **peripheral.c** and **peripheral.h** should be excluded from the build, and the files **peripheralBroadcaster.c** and **peripheralBroadcaster.h** should be included. In addition, the preprocessor value **PLUS_BROADCASTER** should be defined when using the peripheral / broadcaster multi-role profile.

The function names within the peripheral / broadcaster profile are the same as they are in the peripheral profile. This allows the developer to take a single-role application and add multi-role support with minimal changes to the existing source code.

Additional details on using the GAP Peripheral / Broadcaster Multi-Role Profile can be found in [4].

3.4.3 GAP Central Role Profile

The central role profile provides the means for a central device to discover advertising devices, establish a connection with peripheral device, update the connection parameters, and monitor the RSSI.

The primary API function prototypes for the central role profile can be found in the file **central.h**. The API provides functions to get and set certain GAP profile parameters: **GAPCentralRole_GetParameter** and **GAPCentralRole_SetParameter**, respectively.

The GAP Central Role Profile uses callback functions to notify the application of events. These are set up by means of the function **GAPCentralRole_StartDevice**. This function initializes the GAP central role, and should only be called once. Its single parameter is a pointer to a variable of type **gapCentralRolesCBs_t**, which is a structure containing two function pointers:

- **eventCB** – This function gets called every time a GAP event occurs, such as when a device is discovered while scanning, or when a connection is established or terminated.
- **rssiCB** – This function gets called every time the RSSI has been read, with the RSSI value passed as a parameter.

It is up to the application to provide the actual callback functions. In the case of the SimpleBLECentral sample application (see section 4.4), the event callback function is **simpleBLECentralEventCB** and the RSSI callback function is **simpleBLECentralRssiCB**.

The peripheral profile also contains an automatic connection parameter update feature that can be enabled by setting the **GAPROLE_PARAM_UPDATE_ENABLE** parameter to **TRUE**. If the feature is enabled and the peripheral device enters a connection with a connection interval that

falls outside of the range of the desired interval, or with a slave latency or timeout setting that is not the desired value, the peripheral profile will automatically send a Connection Parameter Update Request with the desired parameters. As long as the parameter values are legal as per the BLE specification, the central device should accept the request and change the connection parameters. In the sample application, the desired connection parameters are set in the **SimpleBLEPeripheral_Init** function, and can easily be changed to other values.

Additional details on using the GAP Central Role Profile can be found in [4].

3.4.4 GAP Bond Manager

Note: The GAP Peripheral Bond Manager from the BLEv1.0 software release has been replaced by the GAP Bond Manager in BLEv1.1, which now supports both peripheral and central role configurations. The files **gapperiphbondmgr.c** and **gapperiphbondmgr.h** are still included to support legacy applications; however for it is recommended that future applications use the new bond manager, as it is had additional features and is based on the latest updates.

The GAP Bond Manager allows the device to automatically initiate or respond to pairing requests from a connected device. After pairing, if security keys are exchanged and bonding is enabled, the bond manager saves the security key information in non-volatile memory.

Whenever the bond manager initiates, it loads any previously-stored bonding information into memory. When the device goes into a new connection and if the peer device address matches the address of the information that was loaded, it passes the keys and other necessary data to the GAP layer of the BLE protocol stack. This way, encryption can easily (or automatically) be re-established.

The bond manager is primarily controlled by the GAP role profile; however the application can access a parameter in the bond manager using the **GAPBondMgr_SetParameter** and **GAPBondMgr_GetParameter** functions. The sample application uses the **GAPBondMgr_SetParameter** function during initialization to setup the bond manager. Here are a few bond manager parameters of interest:

- **GAPBOND_PAIRING_MODE** – This parameter tells the bond manager whether pairing is allowed, and if so, whether it should wait for a request from the central device, or initiate pairing on its own. The default setting is to wait for a request from the central device.
- **GAPBOND_MITM_PROTECTION** – This parameter sets whether man-in-the-middle protection is enabled or not. If it is enabled, the pairing request will also authenticate the connection between the slave and master. The profile default value is **FALSE**, though the sample application sets it to **TRUE** during initialization.
- **GAPBOND_IO_CAPABILITIES** – This parameter tells the bond manager the input and output capabilities of the device. This is needed in order to determine whether the device can display and/or enter a passkey. Therefore, the default value is **GAPBOND_IO_CAP_DISPLAY_ONLY**, indicating that the device has a display but no keyboard. Even if the device does not have a physical display, a passkey presented in a user guide (such as this one) can be considered to be a "display". The default passkey value is a six digit string of zeros: "000000".
- **GAPBOND_BONDING_ENABLED** – This parameter enables bonding. The profile default value is **FALSE**, though the SimpleBLEPeripheral application sets it to **TRUE** during initialization.

The bond manager uses callback functions to notify the application of events. These are set up by means of the function **GAPBondMgr_Register**. Its single parameter is a pointer to a variable of type **gapBondCBs_t**, which is a structure containing two function pointers:

- **passcodeCB** – This function gets called during the pairing process if authentication is requested. It allows the application to generate a six-digit passcode.
- **pairStateCB** – This function gets when the pairing state of the device changes, notifying the application if the pairing process has started, has completed, and if the two devices are bonded.

It is up to the application to provide the actual callback functions. In the case of the SimpleBLECentral sample application (see section 4.4), the passcode callback function is **simpleBLECentralPasscodeCB** and the pairing state callback function is **simpleBLECentralPairStateCB**.

Additional details on using the GAP bond manager can be found in [4].

3.4.5 Simple GATT Profile

GATT profiles are used for the storage and handling of data within the GATT server of the device. The simple GATT profile included with the BLE protocol stack provides an example of a generic GATT profile implementation, and is for demonstration purposes. The source code for the simple GATT profile is contained within the files **simpleGATTProfile.c** and **simpleGATTProfile.h**. This source code can be used as a reference in creating additional profiles, either customer-designed or based on *Bluetooth* specifications.

The simple GATT profile contains the following API functions:

- **SimpleProfile_AddService** – Initialization routine that adds the service attributes to the attribute table, and registers read and validate/write callback functions within the profile with the GATT server.
- **SimpleProfile_SetParameter** – Allows the application to set or change parameters in the profile; these parameters correspond to the GATT characteristic values in the profile. If notifications are enabled for a characteristic value, the notifications get sent when this function is called to set a new value of the characteristic.
- **SimpleProfile_GetParameter** – Allows the application to read back parameter values from within the profile.
- **SimpleProfile_RegisterAppCBs** – Allows the application to register a callback function with the simple GATT profile which gets called any time a GATT client device successfully writes a new value to any characteristic in the service with write permissions (**SIMPLEPROFILE_CHAR1** or **SIMPLEPROFILE_CHAR3**). A **simpleProfileCBs_t** value containing a pointer to the callback function must be defined by the application, and passed as a parameter when calling the register function.

The simple GATT profile also contains the following local functions:

- **simpleProfile_ReadAttrCB** – This function must get registered with the GATT server during the AddService routine. Every time a GATT client device wants to read from an attribute in the profile, this function gets called. This function would not be required if the profile didn't contain any attributes with read permissions.
- **simpleProfile_WriteAttrCB** – This function must get registered with the GATT server during the AddService routine. Every time a GATT client device wants to write to an attribute in the profile, this function gets called. Before actually writing the new data to the attribute, it checks whether the data is valid. When a client characteristic configuration value is written to a bonded device, it also notified the bond manager so that it can store the data in non-volatile memory along with the bond data. This function would not be required if the profile didn't contain any attributes with write permissions.
- **simpleProfile_HandleConnStatusCB** – This function must get registered with the link database in order to receive a callback whenever the link status changes. It automatically resets all characteristic configuration values to disable notifications and/or indications if a link is terminated.
- Several additional functions to handle client characteristic configurations. These functions are used to look up the handle of characteristic configurations, read, write, or reset their values, or process a notification of a characteristic value.

The simple GATT profile contains five characteristics, whose values can be set or read back by the application using the **SimpleProfile_SetParameter** and **SimpleProfile_GetParameter** functions, respectively. The parameters are as follows:

- **SIMPLEPROFILE_CHAR1** – This parameter is the value of the first characteristic of the simple GATT profile. It is a one-byte value that can be read or written from a GATT client device.
- **SIMPLEPROFILE_CHAR2** – This parameter is the value of the second characteristic of the simple GATT profile. It is a one-byte value that can be read from a GATT client device, but cannot be written.
- **SIMPLEPROFILE_CHAR3** – This parameter is the value of the third characteristic of the simple GATT profile. It is a one-byte value that can be written from a GATT client device, but cannot be read.
- **SIMPLEPROFILE_CHAR4** – This parameter is the value of the fourth characteristic of the simple GATT profile. It is a one-byte value that cannot be directly read or written from a GATT client device. Its value, however, can be sent as a notification to a GATT client device.
- **SIMPLEPROFILE_CHAR5** - This parameter is the value of the fifth characteristic of the simple GATT profile. It is a five-byte value that can be read (but not written) from a GATT client device only if the link is encrypted (paired).

In order for it to be sent as a notification, the GATT client must turn on notifications by writing a value of 0x0001 (**GATT_CLIENT_CFG_NOTIFY**) to the characteristic configuration attribute. Once notifications are enabled, the data from the characteristic will be sent to the GATT client every time the application sets a new value of the characteristic using **SimpleProfile_SetParameter**.

The SimpleGATTProfile registers the following attributes in the GATT server:

0x2800	GATT_PRIMARY_SERVICE_UUID	0xFFF0 (SIMPLEPROFILE_SERV_UUID)	GATT_PERMIT_READ	Start of Simple GATT Profile Service
0x2803	GATT_CHARACTER_UUID	0A (properties: read/write) 11 00 (handle: 0x0011) F1 FF (UUID: 0xFFF1)	GATT_PERMIT_READ	Characteristic 1 declaration
0xFFF1	SIMPLEPROFILE_CHAR1_UUID	1 (1 byte)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Characteristic 1 value
0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 1" (17 bytes)	GATT_PERMIT_READ	Characteristic 1 user description
0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 14 00 (handle: 0x0014) F2 FF (UUID: 0xFFF2)	GATT_PERMIT_READ	Characteristic 2 declaration
0xFFF2	SIMPLEPROFILE_CHAR2_UUID	2 (1 byte)	GATT_PERMIT_READ	Characteristic 2 value
0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 2" (17 bytes)	GATT_PERMIT_READ	Characteristic 2 user description
0x2803	GATT_CHARACTER_UUID	08 (properties: write only) 17 00 (handle: 0x0017) F3 FF (UUID: 0xFFF3)	GATT_PERMIT_READ	Characteristic 3 declaration
0xFFF3	SIMPLEPROFILE_CHAR3_UUID	3 (1 byte)	GATT_PERMIT_WRITE	Characteristic 3 value
0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 3" (17 bytes)	GATT_PERMIT_READ	Characteristic 3 user description
0x2803	GATT_CHARACTER_UUID	10 (properties: notify only) 1A 00 (handle: 0x001A) F4 FF (UUID: 0xFFF4)	GATT_PERMIT_READ	Characteristic 4 declaration
0xFFF4	SIMPLEPROFILE_CHAR4_UUID	4 (1 byte)	(none)	Characteristic 4 value
0x2902	GATT_CLIENT_CHAR_CFG_UUID	00:00 (2 bytes)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Characteristic 4 configuration
0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 4" (17 bytes)	GATT_PERMIT_READ	Characteristic 4 user description
0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 1E 00 (handle: 0x001E) F5 FF (UUID: 0xFFF5)	GATT_PERMIT_READ	Characteristic 5 declaration
0xFFF5	SIMPLEPROFILE_CHAR5_UUID	01:02:03:04:05 (5 bytes)	GATT_PERMIT_AUTHEN_READ	Characteristic 5 value
0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 5" (17 bytes)	GATT_PERMIT_READ	Characteristic 5 user description

Figure 4: SimpleGATTProfile Attribute List

Note that the types (UUIDs) of the five characteristic values (0xFFF1, 0xFFF2, 0xFFF3, 0xFFF4, and 0xFFF5), as well as the simple profile primary service UUID value (0xFFF0), do not conform to any specifications in the *Bluetooth* SIG. They are simply used as a demonstration.

3.4.6 Simple Keys GATT Profile

The simple keys GATT profile included with the BLE protocol stack provides functionality for sending notifications of key presses from a GATT server device to a GATT client. The profile is designed for use with the keyfob board contained within CC2540DK-MINI development kit. The source code for the simple keys GATT profile is contained within the files **simplekeys.c** and **simplekeys.h**.

It is important to note that the simple keys profile included with the BLE development kit does not conform to any standard profile specification available from the *Bluetooth* SIG. At the time of the release of the software, no official GATT service profile specifications have been approved by the *Bluetooth* SIG. Therefore the profile, including the GATT characteristic definition, the UUID values, and the functional behavior, was developed by Texas Instruments for use with the CC2540DK-MINI development kit.

As the *Bluetooth* SIG begins to approve specifications for different service profiles, Texas Instruments plans to release updates to the BLE software development kit with source code conforming to the specifications.

The simple keys GATT profile contains the following API functions:

- **SK_AddService** – Initialization routine that adds the simple keys service attributes to the attribute table, and registers read and validate/write callback functions within the profile with the GATT server.
- **SK_SetParameter** – Allows the application to set or change the state of the keys in the profile; the parameter corresponds to the GATT characteristic value in the profile. If notifications are enabled for the key press state characteristic value, the notification gets sent when this function is called to set a new value of the characteristic.
- **SK_GetParameter** – Allows the application to read back the parameter value from within the profile.

The simple GATT profile also contains the following local functions:

- **sk_ReadAttrCB** – This function must get registered with the GATT server during the AddService routine. Every time a GATT client device wants to read from an attribute in the profile, this function gets called. This function would not be required if the profile didn't contain any attributes with read permissions.
- **sk_WriteAttrCB** – This function must get registered with the GATT server during the AddService routine. Every time a GATT client device wants to write to an attribute in the profile, this function gets called. Before actually writing the new data to the attribute, it checks whether the data is valid. This function would not be required if the profile didn't contain any attributes with write permissions.

The simple keys GATT profile contains one characteristic. The value of the characteristic can be set or read back by the application using the **SimpleProfile_SetParameter** and **SimpleProfile_GetParameter** functions, respectively. The single parameter in the simple keys profile is the **SK_KEY_ATTR**.

The **SK_KEY_ATTR** parameter represents the current state of the keys. It is a one-byte value, with a range from 0 through 3, with each value representing the following:

- 0 – Neither key on the keyfob is currently pressed
- 1 – The left key on the keyfob is currently pressed
- 2 – The right key on the keyfob is currently pressed
- 3 – Both the left and right keys on the keyfob are currently pressed

Its value can not be directly read or written from a GATT client device; however it can be sent as a notification to a GATT client device. In order for it to be sent as a notification, the GATT client must turn on notifications by writing a value of 0x0001 (**GATT_CLIENT_CFG_NOTIFY**) to the characteristic configuration attribute. Once notifications are enabled, the data from the characteristic will be sent to the GATT client every time the application sets a new value of the characteristic using **SK_SetParameter**.

The simple keys GATT profile registers the following attributes in the GATT server:

Type (hex)	Type (#DEFINE)	Value (default)	Local Parameter Name	Application Permissions	GATT Server Permissions	Description
0x2800	GATT_PRIMARY_SERVICE_UUID	0xFFE0 (SK_SERVICE_UUID)			Read	Start of Service
0x2803	GATT_CHARACTER_UUID	10 (properties: notify only) 1F 00 (handle: 0x001F) E1 FF (UUID: 0xFFE1)			Read	Key Press State Characeristic Declaration
0xFFE1	SK_KEYPRESSED_UUID	0 (1 byte)	SK_KEY_ATTR	Read / Write	Notify	Key Press State Characteristic Value
0x2902	GATT_CLIENT_CHAR_CFG_UUID	00:00 (2 bytes)			Read	Key Press State Characteristic Value
0x2901	GATT_CHAR_USER_DESC_UUID	"Key Press State" (16 bytes)			Read	Key Press State Characteristic Configuration

Figure 5: Simple Keys GATT Profile Attribute List

3.4.7 Device Information Service

The Device Information Service included with the BLE protocol stack is based on an adopted *Bluetooth* specification. In addition to being mandatory as a part of many future BLE profile specifications, it is recommended to include this service in any BLE application requiring interoperability with *Bluetooth*-enabled mobile phones or PCs.

The full specification of the device information can be found in [8].

3.4.8 Additional GATT Profiles

In addition to the generic simple GATT profile, the simple keys profile, and the Device Information Service, the BLE protocol stack contains two additional services that are mandatory as per the BLE specification: the GATT profile service and the GAP profile service. These two services are maintained by the BLE protocol stack and should not affect the application.

Several other GATT services are included in the BLE development kit, and are used by various sample applications. More information on these sample applications and profiles can be found in [5].

4 Working with Projects using IAR Embedded Workbench 7.60

All embedded software for the CC2540 is developed using *Embedded Workbench for 8051* v7.60 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.

4.1 IAR Overview

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

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://www.iar.com/website1/1.0.1.0/1011/1/

2. **Purchase the full-featured version of IAR Embedded Workbench** – For complete BLE application development using the CC2540, 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/website1/1.0.1.0/667/1/

4.2 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.

Once all of the patches have been installed, you are ready to develop software for the CC2540. This section will describe how to open and build an existing project. The SimpleBLEPeripheral project, which is included with the Texas Instruments BLE software development kit, is used as an example.

4.2.1 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 MCS-51 7.60 > IAR Embedded Workbench**.

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

C:\Texas Instruments\BLE-CC2540-1.1\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.

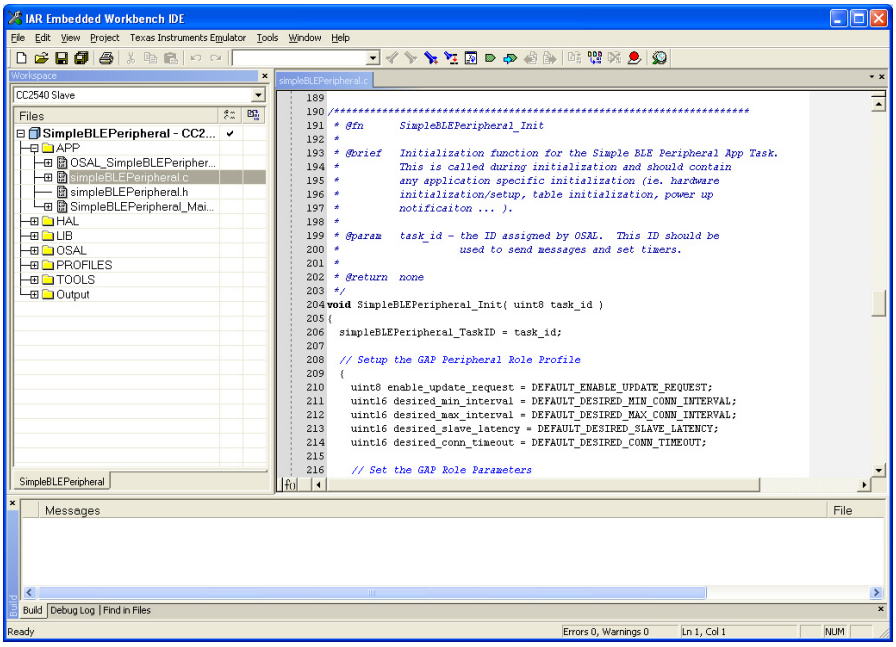

Figure 6: IAR Embedded Workbench

4.2.2 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 7.

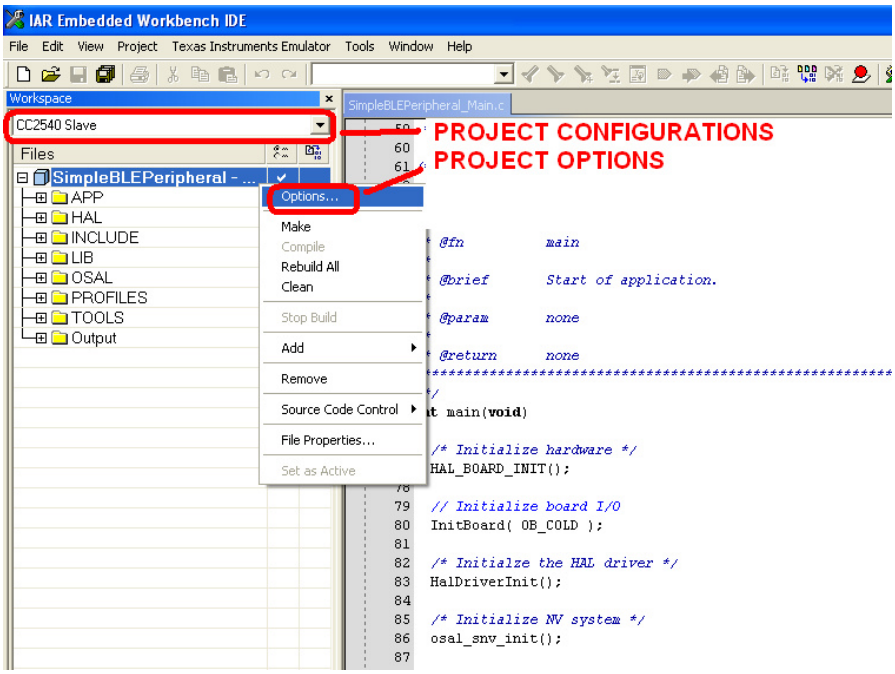

Figure 7: Project Configurations and Options

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. The default configuration in the SimpleBLEPeripheral project is the "CC2540 Slave" configuration, which is targeted towards the SmartRF05 + CC2540EM hardware platform that is included with the CC2540DK development kit. The other available option, "CC2540DK-MINI Keyfob Slave", is optimized for the "keyfob" board in the CC2540DK-MINI development kit.

One of the important settings when building a project is the compiler preprocessor defined 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:

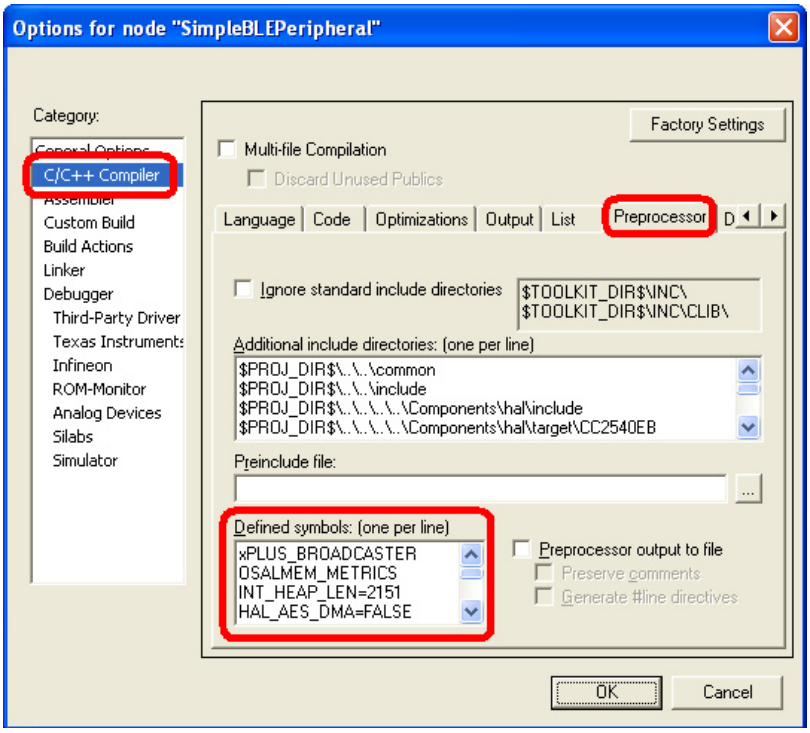

Figure 8: Preprocessor Defined Symbols Settings

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 generally means that a function is being disabled, and can be enabled by removing the 'x' and letting the proper name of the symbol get 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 files **config_master.cfg** and **config_slave.cfg** included with the software development kit will define the appropriate symbols for master and slave builds, respectively.

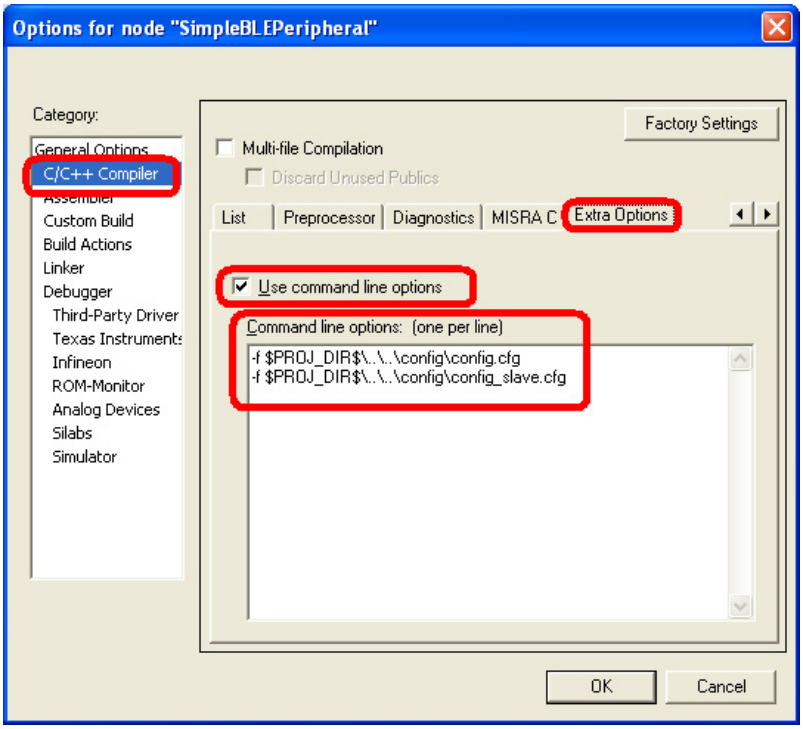

Figure 9: Configuration File Setup

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 3.1.3) 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 4.2.4.
- **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. Note that this has changed from v1.0 of the BLE stack, in which the symbol **HAL_AES_DMA** was required to be defined as **FALSE**
- **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.
- **DEVICE_TYPE –** This value *must* be defined as either **LL_DEV_TYPE_MASTER** or as **LL_DEV_TYPE_SLAVE**, and sets up the BLE device's role as either a master or slave. It is defined in the file **config_master.cfg** or **config_slave.cfg**.

Optional Symbols:

- **POWER_SAVING** – This symbol must be defined if using a library with power management enabled. It 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 (see sections 3.4.1 and 3.4.2). The default option for this in the sample project is for this to be undefined.
- **HAL_LCD** – This symbol indicates whether an LCD display exists, such as with the SmartRF05 board (to be available in the CC2540DK development kit). For the CC2540DK-MINI configuration, this value is defined as **FALSE**.
- **HAL_LED** – This symbol indicates whether the hardware has any LEDs. For the generic configuration of the sample application, this value is defined as **FALSE**. For the CC2540DK-MINI Keyfob Slave configuration it is defined as **TRUE**, since the board contains two LEDs.
- **HAL_UART** – This symbol should be defined (with no value) if the UART interface is being used. For the sample application, this symbol is not defined.
- **CC2540_MINIDK** – This symbol should be defined when using the keyfob board contained in the CC2540DK-MINI development kit. It configures the hardware based on the board layout.
- **HAL_UART_DMA** – This symbol sets the CC2540 UART interface to use DMA mode, and should be either left undefined or set to 0. The CC2540 does not currently support this option.
- **HAL_UART_ISR** – This symbol sets the CC2540 UART interface to use ISR mode, and should be set to **TRUE** for builds that use the actual UART interface for the HCI, but not for builds that use the USB (virtual UART) interface.
- **HAL_UART_ISR_RX_MAX** – This symbol sets the UART buffer size. For builds that use the UART or USB (virtual UART) interface for the HCI (such as the HostTestRelease project), this value should be set to 250 in order to allow for large messages to be sent from the CC2540.
- **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 3.4.4.

4.2.3 Building and Debugging a Project

To build a project, right click on the workspace name "SimpleBLEPeripheral – CC2540 Slave" as seen below, and click "Make":

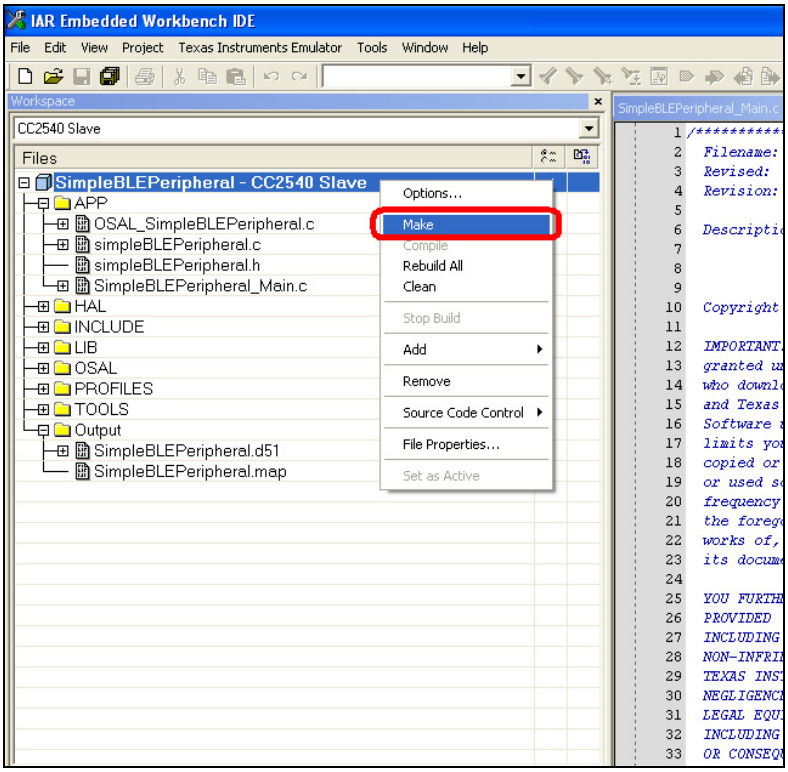

Figure 10: Building a Project

This will compile the source code, link the files, and build the project. Any compiler errors or warnings will appear in the messages window at the bottom of the screen.

To download the compiled code into a CC2540 device and debug, connect the keyfob using a hardware debugger (such as the CC Debugger, which is included with the CC2540DK-MINI development kit) connected to the PC over USB. Find the "Debug" button in the upper right side of the IAR window:

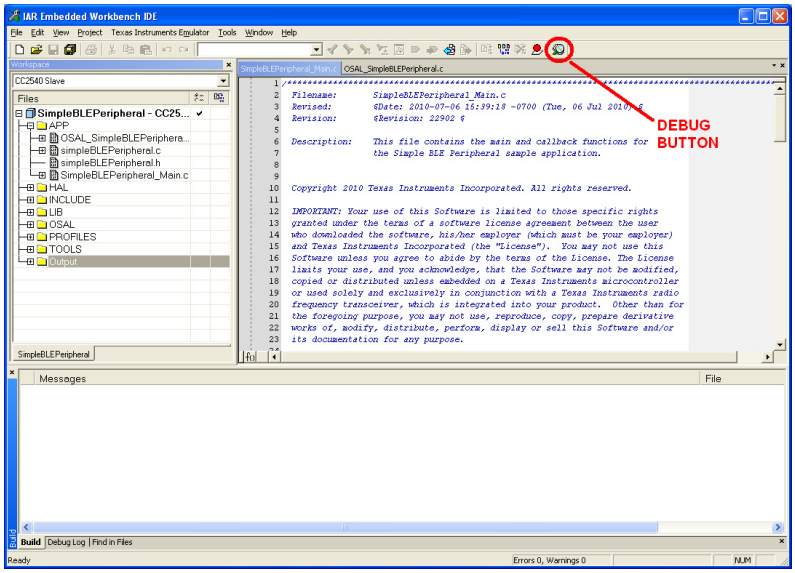

Figure 11: Debug Button in IAR

The following pop-up window should appear on the screen, indicating that the code is being downloaded to the device:

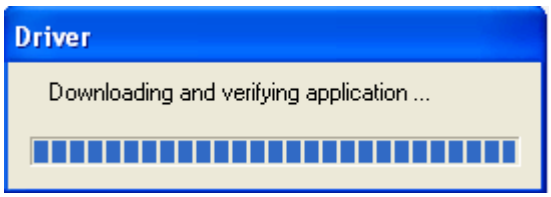

Figure 12

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:

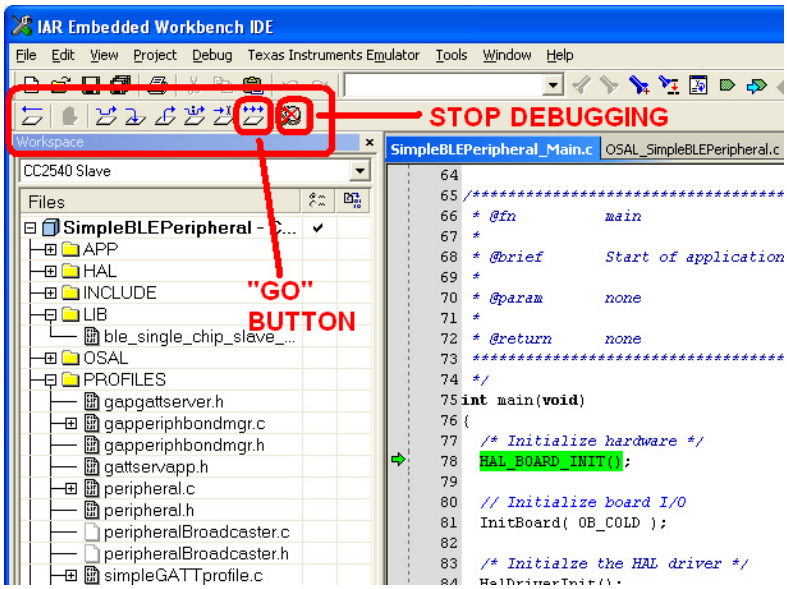

Figure 13: IAR Debug Toolbar

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

4.2.4 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.

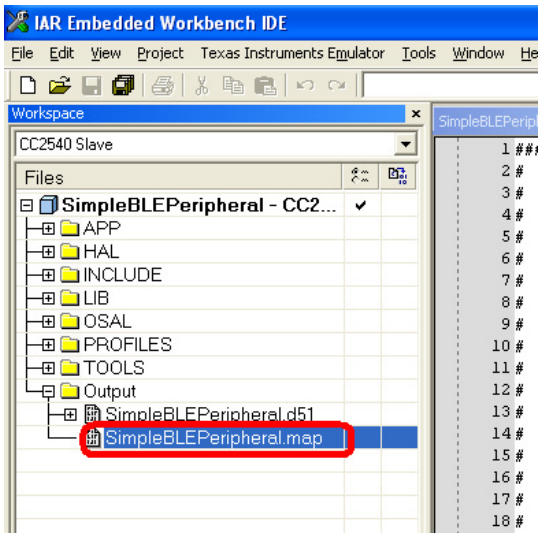

Figure 14: MAP File in File List

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:

```
 101 560 bytes of CODE  memory
      26 bytes of DATA  memory (+ 62 absolute )
   5 699 bytes of XDATA memory
     192 bytes of IDATA memory
       8 bits  of BIT   memory
   3 602 bytes of CONST memory

Errors: none
Warnings: none
```

This information is useful, in that it tells 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). The size of the XDATA memory must not exceed 7936 bytes, as the CC2540 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.

4.3 SimpleBLEPeripheral Sample Project

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

4.3.1 Project Overview

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

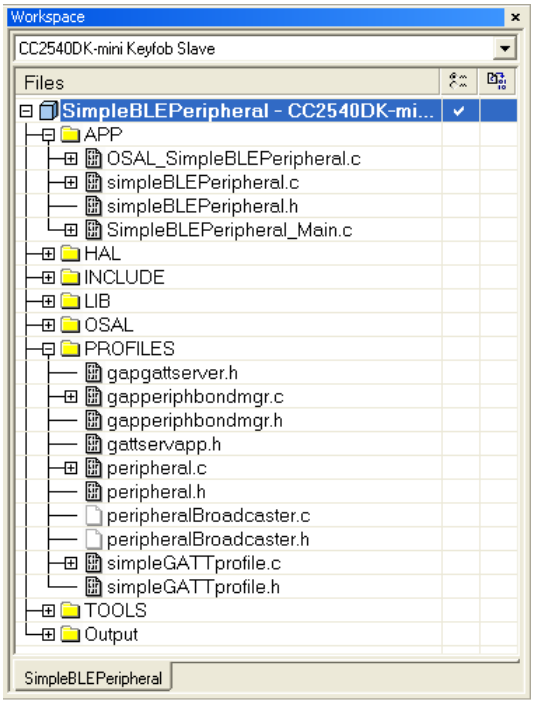

Figure 15: Project Files

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 3.2.
- **INCLUDE** – This group includes all of the necessary header files for the BLE protocol stack API. More information on the API can be found in section 3.3.3
- **LIB** – This group contains the protocol stack library file **ble_single_chip_slave_pm_on.lib**. More information on the protocol stack libraries can be found in section 3.3.5.
- **OSAL** – This group contains the OSAL source code and header files. More information on the OSAL can be found in section 3.1.
- **PROFILES** – This group contains the source code and header files for the GAP role profile, GAP security profile, and the sample GATT profile. More information on these profiles can be found in section 3.4. In addition, this section contains the necessary header files for the GATT server application API. More information on the GATT server application can be found in section 3.3.4.
- **TOOLS** – This group contains the configuration files **config.cfg** and **config_slave.cfg**. 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 4.2.4).

4.3.2 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 paramters, and simpleGATTprofile parameters. It also sets an OSAL **SBP_START_DEVICE_EVT** event.

This 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 CC2540DK-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.

4.3.3 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 device is put in discoverable mode, with a timeout value of **PERIODIC_EVT_PERIOD** (default value is 5000 milliseconds). Every five seconds, the **SBP_PERIODIC_EVT** occurs and the function **performPeriodicTask** is called. The **performPeriodicTask** function simply gets the value of the third characteristic in the SimpleGATTProfile, and copies that value into the fourth characteristic. This is put in 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.

4.3.4 Peripheral State Notification Callback

The application also contains a function called **peripheralStateNotificationCB**. This function gets registered with the peripheral profile when **GAPRole_StartDevice** is called (for example, if the device goes from an advertising state to a connected state). Any time that the peripheral state of the device changes, the callback function gets called and updates the application's local variable **gapProfileState**. This allows the application to perform specific behavior based on the state of the device.

4.3.5 Key Presses (CC2540DK-MINI Keyfob only)

The application has some additional code that is specific to the keyfob contained with the CC2540DK-MINI development kit. This code is only surrounded by the pre-processor directive "#if defined(CC2540_MINIDK)", and therefore only gets compiled when using the "CC2540DK-MINI Keyfob Slave" configuration. This code adds the simple keys service (see section 3.4.6) 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, the HAL sends an OSAL message to the application. 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.

When the device is in an advertising state (before a connection), pressing of the right key will toggle the advertising on and off. This is done via a call to **GAPRole_GetParameter** to read the current advertising state, and a call to **GAPRole_SetParameter** to set a new state.

If the device is in a connection, the appropriate profile parameter value in the attribute table is set using the **SK_SetParameter** function. If notifications of the key press state characteristic value have been enabled, then a notification will be sent to every time the function gets called.

4.3.6 LCD Display (CC2540 Slave only)

The application also contains code which is compiled in when **HAL_LCD** is defined as **TRUE**, such as with the SmartRF05 + CC2540EM hardware platform. The words "BLE Peripheral" along with the device address will be displayed on the LCD screen. In addition, the GAP state of the device will be displayed, such as "Advertising" or "Connected". If a connection is established and a GATT client device writes values to the first or third characteristics in the SimpleGATTProfile (those are the only two characteristics with write permissions), the LCD will display the value that has been written.

4.3.7 Complete Attribute Table

The table below shows the SimpleBLEPeripheral complete attribute table, and can be used as a reference. Services are shown in yellow, characteristics are shown in blue, and characteristic values / descriptors are shown in grey. When working with the SimplyBLEPeripheral application, it might be useful to print out the table as a reference.

SimpleBLEPeripheral Application: Complete Attribute Table

handle (hex)	handle (dec)	Type (hex)	Type (#DEFINE)	Hex / Text Value (default)	GATT Server Permissions	Notes
0x1	1	0x2800	GATT_PRIMARY_SERVICE_UUID	0x1800 (GAP_SERVICE_UUID)	GATT_PERMIT_READ	Start of GAP Service (Mandatory)
0x2	2	0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 03 00 (handle: 0x0003) 00 2A (UUID: 0x2A00)	GATT_PERMIT_READ	Device Name characteristic declaration
0x3	3	0x2A00	GAP_DEVICE_NAME_UUID	"Simple BLE Peripheral"	GATT_PERMIT_READ	Device Name characteristic value
0x4	4	0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 05 00 (handle: 0x0005) 01 2A (UUID: 0x2A01)	GATT_PERMIT_READ	Appearance characteristic declaration
0x5	5	0x2A01	GAP_APPEARANCE_UUID	0x0000	GATT_PERMIT_READ	Appearance characteristic value
0x6	6	0x2803	GATT_CHARACTER_UUID	0A (properties: read/write) 07 00 (handle: 0x0007) 02 2A (UUID: 0x2A02)	GATT_PERMIT_READ	Peripheral Privacy Flag characteristic declaration
0x7	7	0x2A02	GAP_PERI_PRIVACY_FLAG_UUID	0x00 (GAP_PRIVACY_DISABLED)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Peripheral Privacy Flag characteristic value
0x8	8	0x2803	GATT_CHARACTER_UUID	0A (properties: read/write) 09 00 (handle: 0x0009) 03 2A (UUID: 0x2A03)	GATT_PERMIT_READ	Reconnection address characteristic declaration
0x9	9	0x2A03	GAP_RECONNECT_ADDR_UUID	00:00:00:00:00:00	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Reconnection address characteristic value
0xA	10	0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 0B 00 (handle: 0x000B) 04 2A (UUID: 0x2A04)	GATT_PERMIT_READ	Peripheral Preferred Connection Parameters characteristic declaration
0xB	11	0x2A04	GAP_PERI_CONN_PARAM_UUID	50 00 (100ms preferred min connection interval) A0 00 (200ms preferred max connection interval) 00 00 (0 preferred slave latency) E8 03 (10000ms preferred supervision timeout)	GATT_PERMIT_READ	Peripheral Preferred Connection Parameters characteristic declaration
0xC	12	0x2800	GATT_PRIMARY_SERVICE_UUID	0x1801 (GATT_SERVICE_UUID)	GATT_PERMIT_READ	Start of GATT Service (mandatory)
0xD	13	0x2803	GATT_CHARACTER_UUID	20 (properties: indicate only) 0E 00 (handle: 0x000E) 05 2A (UUID: 0x2A05)	GATT_PERMIT_READ	Service Changed characteristic declaration
0xE	14	0x2A05	GATT_SERVICE_CHANGED_UUID	(null value)	(none)	Service Changed characteristic value
0xF	15	0x2800	GATT_PRIMARY_SERVICE_UUID	0x180A (DEVINFO_SERV_UUID)	GATT_PERMIT_READ	Start of Device Information Service
0x10	16	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 11 00 (handle 0x0011) 23 2A (UUID 0x2A23)	GATT_PERMIT_READ	System ID characteristic declaration
0x11	17	0x2A23	DEVINFO_SYSTEM_ID_UUID	xx xx xx 00 00 xx xx xx (xx's are IEEE address)	GATT_PERMIT_READ	System ID
0x12	18	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 13 00 (handle 0x0013) 24 2A (UUID 0x2A24)	GATT_PERMIT_READ	Model Number String characteristic declaration
0x13	19	0x2A24	DEVINFO_MODEL_NUMBER_UUID	"Model Number"	GATT_PERMIT_READ	Model Number String
0x14	20	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 15 00 (handle 0x0015) 25 2A (UUID 0x2A25)	GATT_PERMIT_READ	Serial Number String characteristic declaration
0x15	21	0x2A25	DEVINFO_SERIAL_NUMBER_UUID	"Serial Number"	GATT_PERMIT_READ	Serial Number String
0x16	22	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 17 00 (handle 0x0017) 26 2A (UUID 0x2A26)	GATT_PERMIT_READ	Firmware Revision String characteristic declaration
0x17	23	0x2A26	DEVINFO_FIRMWARE_REV_UUID	"Firmware Revision"	GATT_PERMIT_READ	Firmware Revision String
0x18	24	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 19 00 (handle 0x0019) 27 2A (UUID 0x2A27)	GATT_PERMIT_READ	Hardware Revision String characteristic declaration
0x19	25	0x2A27	DEVINFO_HARDWARE_REV_UUID	"Hardware Revision"	GATT_PERMIT_READ	Hardware Revision String
0x1A	26	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 1B 00 (handle 0x001B) 28 2A (UUID 0x2A28)	GATT_PERMIT_READ	Software Revision String characteristic declaration
0x1B	27	0x2A28	DEVINFO_SOFTWARE_REV_UUID	"Software Revision"	GATT_PERMIT_READ	Software Revision String
0x1C	28	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 1D 00 (handle 0x001D) 29 2A (UUID 0x2A29)	GATT_PERMIT_READ	Manufacturer Name String characteristic declaration
0x1D	29	0x2A29	DEVINFO_MANUFACTURER_NAME_UUID	"Manufacturer Name"	GATT_PERMIT_READ	Manufacturer Name String
0x1E	30	0x2803	GATT_CHARACTER_UUID	02 (read permissions) 1F 00 (handle 0x001F) 2A 2A (UUID 0x2A2A)	GATT_PERMIT_READ	IEEE 11073-20601 Regulatory Certification Data List characteristic declaration
0x1F	31	0x2A2A	DEVINFO_11073_CERT_DATA_UUID	FE 00 65 78 70 65 72 69 6D 65 6E 74 61 6C	GATT_PERMIT_READ	IEEE 11073-20601 Regulatory Certification Data List

Figure 16: SimpleBLEPeripheral Complete GATT Server (part 1)

SimpleBLEPeripheral Application: Complete Attribute Table

	handle (hex)	handle (dec)	Type (hex)	Type (#DEFINE)	Hex / Text Value (default)	GATT Server Permissions	Notes
	0x20	32	0x2800	GATT_PRIMARY_SERVICE_UUID	0xFFF0 (SIMPLEPROFILE_SERV_UUID)	GATT_PERMIT_READ	Start of Simple GATT Profile Service
	0x20	32	0x2803	GATT_CHARACTER_UUID	0A (properties: read/write) 11 00 (handle: 0x0011) F1 FF (UUID: 0xFFF1)	GATT_PERMIT_READ	Characteristic 1 declaration
	0x22	34	0xFFF1	SIMPLEPROFILE_CHAR1_UUID	1 (1 byte)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Characteristic 1 value
	0x23	35	0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 1" (17 bytes)	GATT_PERMIT_READ	Characteristic 1 user description
	0x24	36	0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 14 00 (handle: 0x0014) F2 FF (UUID: 0xFFF2)	GATT_PERMIT_READ	Characteristic 2 declaration
	0x25	37	0xFFF2	SIMPLEPROFILE_CHAR2_UUID	2 (1 byte)	GATT_PERMIT_READ	Characteristic 2 value
	0x26	38	0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 2" (17 bytes)	GATT_PERMIT_READ	Characteristic 2 user description
	0x27	39	0x2803	GATT_CHARACTER_UUID	08 (properties: write only) 17 00 (handle: 0x0017) F3 FF (UUID: 0xFFF3)	GATT_PERMIT_READ	Characteristic 3 declaration
	0x28	40	0xFFF3	SIMPLEPROFILE_CHAR3_UUID	3 (1 byte)	GATT_PERMIT_WRITE	Characteristic 3 value
	0x29	41	0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 3" (17 bytes)	GATT_PERMIT_READ	Characteristic 3 user description
	0x2A	42	0x2803	GATT_CHARACTER_UUID	10 (properties: notify only) 1A 00 (handle: 0x001A) F4 FF (UUID: 0xFFF4)	GATT_PERMIT_READ	Characteristic 4 declaration
	0x2B	43	0xFFF4	SIMPLEPROFILE_CHAR4_UUID	4 (1 byte)	(none)	Characteristic 4 value
	0x2C	44	0x2902	GATT_CLIENT_CHAR_CFG_UUID	00:00 (2 bytes)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Characteristic 4 configuration
	0x2D	45	0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 4" (17 bytes)	GATT_PERMIT_READ	Characteristic 4 user description
	0x2E	46	0x2803	GATT_CHARACTER_UUID	02 (properties: read only) 1E 00 (handle: 0x001E) F5 FF (UUID: 0xFFF5)	GATT_PERMIT_READ	Characteristic 5 declaration
	0x2F	47	0xFFF5	SIMPLEPROFILE_CHAR5_UUID	01:02:03:04:05 (5 bytes)	GATT_PERMIT_AUTHEN_READ	Characteristic 5 value
	0x30	48	0x2901	GATT_CHAR_USER_DESC_UUID	"Characteristic 5" (17 bytes)	GATT_PERMIT_READ	Characteristic 5 user description
CC2540DK-MINI keyfob only	0x31	49	0x2800	GATT_PRIMARY_SERVICE_UUID	0xFFE0 (SK_SERVICE_UUID)	GATT_PERMIT_READ	Start of Simple Keys Service
	0x32	50	0x2803	GATT_CHARACTER_UUID	10 (properties: notify only) 1F 00 (handle: 0x001F) E1 FF (UUID: 0xFFE1)	GATT_PERMIT_READ	Key Press State characteristic declaration
	0x33	51	0xFFE1	SK_KEYPRESSED_UUID	0 (1 byte)	(none)	Key Press State characteristic value
	0x34	52	0x2902	GATT_CLIENT_CHAR_CFG_UUID	00:00 (2 bytes)	GATT_PERMIT_READ | GATT_PERMIT_WRITE	Key Press State characteristic configuration
	0x35	53	0x2901	GATT_CHAR_USER_DESC_UUID	"Key Press State" (16 bytes)	GATT_PERMIT_READ	Key Press State characteristic user description

Figure 17: SimpleBLEPeripheral Complete GATT Server (part 2)

4.4 SimpleBLECentral Sample Project

The BLE software development kit contains a sample project that implements a very simple BLE central device. This project is built using the single-device stack configuration, with the stack, profiles, and application all running on the CC2540. It is designed to be run on a SmartRF05 + CC2540EM hardware platform; however it could be ported to other hardware platforms as well. The application is designed to connect to the SimpleBLEPeripheral sample application to demonstrate the operation of basic GAP and GATT procedures in the stack.

4.4.1 Project Overview

The SimpleBLECentral project structure is very similar to that of the SimpleBLEPeripheral project. The APP directory contains the application source code and header files.

The project contains one configuration, **CC2540EM Master**, using the SmartRF05EB + CC2540EM hardware platform.

4.4.2 User Interface

The SmartRF05EB joystick and display provide a user interface for the application. The joystick and buttons are used as follows:

- Joystick Up: Start or stop device discovery. If connected to a SimpleBLEPeripheral, read or write to the first characteristic in the SimpleGATTProfile service. Each time a write occurs, the value will increment by one for the next write.
- Joystick Left: Scroll through device discovery results.
- Joystick Center: Connect or disconnect to/from the currently selected device.
- Joystick Right: If connected, perform a connection update.
- Joystick Down: If connected, start or stop periodic RSSI readings.

The LCD display is used to display the following information:

- Device BD address
- Device discovery results
- Connection state
- Pairing and bonding status
- Passcode display
- Connection parameter update
- RSSI value
- GATT characteristic value reads and writes

4.4.3 Basic Operation

When the application powers up it displays "BLE Central" and the BD address of the device. Press Joystick Up to start device discovery. Devices that are discovered will be filtered based on the UUIDs of the services that are declared to be included on the peripheral device in the advertisement or scan response data. In particular, only devices containing the SimpleGATTProfile service (UUID of 0xFFF0; see section 3.4.5) will be considered. Any devices not containing this information in the advertisement or scan response data will be ignored. When discovery completes the number of devices found will be displayed. Press Joystick Left to scroll through the devices.

To connect to the selected device press Joystick Center. The connection status will be displayed. Once connected, the application will attempt to discover the Simple BLE profile on the peer device.

If configured to do so, the application or the peer device may also initiate security. If a passcode is required the application will generate and display a random passcode. Enter this passcode on the peer device to proceed with pairing.

Once connected, other operations such as RSSI readings or characteristic read/write can be performed described in the previous section.

To disconnect press Joystick Center again. To reconnect to the same device again press Joystick Center again.

4.4.4 Initialization

The initialization of the application occurs in two phases: first, the **SimpleBLECentral_Init** function is called by the OSAL. This function configures parameters in the central profile, GAP, and GAP bond manager and also initializes GATT for client operation. It also sets up standard GATT and GAP services in the attribute server. Then it sets an OSAL **START_DEVICE_EVT** event. This triggers the second phase of the initialization, which can be found within the **SimpleBLECentral_ProcessEvent** function. During this phase, the **GAPCentralRole_StartDevice** function is called to set up the GAP functions of the application. Then **GAPBondMgr_Register** is called to register with the bond manager.

4.4.5 Event Processing

The application has two main event processing functions, **SimpleBLECentral_ProcessEvent** and **simpleBLECentral_ProcessOSALMsg**.

Function **SimpleBLECentral_ProcessEvent** handles events as follows:

- **SYS_EVENT_MSG**: Service the OSAL queue and process OSAL messages.
- **START_DEVICE_EVT**: Start the device, as described in the previous section.
- **START_DISCOVERY_EVT**: Start service discovery for the SimpleGATTprofile.
- Function **simpleBLECentral_ProcessOSALMsg** handles OSAL messages as follows:
- **KEY_CHANGE** messages: Call function **simpleBLECentral_HandleKeys** to handle key presses.
- **GATT_MSG_EVENT** messages: Call function **simpleBLECentralProcessGATTMsg** to handle messages from GATT.

4.4.6 Callbacks

The application callback functions are as follows:

- **simpleBLECentralRssiCB**: This is the GAP RSSI callback. It displays the latest RSSI value.
- **simpleBLECentralEventCB**: This is the GAP event callback. It processes GAP events for initialization, device discovery, and link connect/disconnect.
- **simpleBLECentralPairStateCB**: This is the GAP bond manager state callback. It displays the status of pairing and bonding operations.
- **simpleBLECentralPasscodeCB**: This is the GAP bond manager passcode callback. It generates and displays a passcode.

4.4.7 Service Discovery

The simpleBLECentral application performs service discovery for the simpleGATTprofile. Discovery is initiated when a connection is established by setting OSAL event **START_DISCOVERY_EVT**. This will result in execution of function **simpleBLECentralStartDiscovery**, which performs primary service discovery for the UUID used by the simpleGATTprofile. When GATT events are received during service discovery function **simpleBLEGATTDiscoveryEvent** is called. This function processes the results of the previous GATT procedure and initiates the next step in the discovery process.

4.5 HostTestRelease Network Processor Project

The BLE software development kit also contains the HostTestRelease project, which implements the network processor configuration on the CC2540 (see section 2.1). The HostTestRelease software simply reads and writes HCI commands through a UART (or virtual UART) interface.

An overview of the HCI command interface can be found in [3].

In order to for the device to perform any actions when running the HostTestRelease project, it must receive the command from an external source. Whenever any messages are received or if an action is required, it simply passes the message on to the external source.

4.5.1 Project Overview

The HostTestRelease project structure is similar to that of the SimpleBLEPeripheral project: OSAL and HAL are used, and the BLE stack is provided as a library. Even though the HostTestRelease project files contain a group called "APP", these files are not truly an application. They are simply a thin layer of source code that translates external messages received from the PC into calls to the BLE stack API. Any messages received from the stack are translated into messages to be sent to the PC. The source code for all of these translations is within the file **hci_ext_app.c**. Looking at this source code can be useful in that it provides working examples of BLE stack API function calls.

The HostTestRelease workspace contains four project configurations:

- **CC2540USB Master** (this is the default configuration for the USB Dongle in the CC2540DK-MINI development kit)
- **CC2540EM Master** (this is the default configuration for the SmartRF05 + CC2540EM hardware in the CC2540DK development kit)
- **CC2540USB Slave**
- **CC2540EM Slave**

The USB builds can be used for devices such as the USB Dongle included with the CC2540DK-MINI development kit. When the dongle is inserted into a PC, it will enumerate as a CDC (communications device class) device, creating a virtual serial port for the HCI. The following Windows INF file can be used to associate the USB Dongle with the appropriate USB-to-serial port driver:

C:\Texas Instruments\BLE-CC2540-1.1\Accessories\usb_cdc_driver_cc2540.inf

More detailed instructions can be found in [6].

The CC2540EM builds are based on the SmartRF05EB v1.8.1 + CC2540EM hardware platform, and use the USART0 port on the CC2540 for the HCI.

4.5.2 External Device Control of BLE Stack

With the HostTestRelease project, a PC can control the CC2540 by means of sending commands over the serial port interface. This allows for the creation of an application on an external device, such as a microcontroller or a PC, to perform any kind of function using the BLE stack. One such example of a PC application is BTool, which is included with the CC2540 software development kit.

The BTool Windows PC application currently only supports the "Master" configurations of the HostTestRelease project. It communicates with the CC2540 through a Windows COM port, which can be either a virtual serial port over USB (such as with the USB Dongle), or a hardware serial port.

BTool allows a user to perform basic BLE central-device functions, such as discovering peripheral and/or broadcaster devices, creating connections with peripheral devices, and performing GATT reads and writes. For this reason, using BTool on a PC along with the HostTestRelease project (with a "Master" configuration) on one CC2540 device can be useful in testing peripheral applications such as the SimpleBLEPeripheral project.

For more information on BTool, please refer to [6].

4.6 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 General Information

5.1 Document History

Table 2: Document History

Revision	Date	Description/Changes
1.0	2010-10-07	Initial release
1.1	2011-07-13	Updated for BLEv1.1 software release

6 Address Information

Texas Instruments Norway AS
Gaustadalléen 21
N-0349 Oslo
NORWAY
Tel: +47 22 95 85 44
Fax: +47 22 95 85 46
Web site: http://www.ti.com/lpw

7 TI Worldwide Technical Support

Internet

TI Semiconductor Product Information Center Home Page: support.ti.com
TI Semiconductor KnowledgeBase Home Page: support.ti.com/sc/knowledgebase

TI LPRF forum E2E community http://www.ti.com/lprf-forum

Product Information Centers

Americas

Phone: +1(972) 644-5580
Fax: +1(972) 927-6377
Internet/Email: support.ti.com/sc/pic/americas.htm

Europe, Middle East and Africa

Phone:
Belgium (English) +32 (0) 27 45 54 32
Finland (English) +358 (0) 9 25173948
France +33 (0) 1 30 70 11 64
Germany +49 (0) 8161 80 33 11
Israel (English) 180 949 0107
Italy 800 79 11 37
Netherlands (English) +31 (0) 546 87 95 45
Russia +7 (0) 95 363 4824
Spain +34 902 35 40 28
Sweden (English) +46 (0) 8587 555 22
United Kingdom +44 (0) 1604 66 33 99
Fax: +49 (0) 8161 80 2045
Internet: support.ti.com/sc/pic/euro.htm

Japan

Fax	International	+81-3-3344-5317
	Domestic	0120-81-0036
Internet/Email	International	support.ti.com/sc/pic/japan.htm
	Domestic	www.tij.co.jp/pic

Asia

Phone	International	+886-2-23786800
	Domestic	Toll-Free Number
	Australia	1-800-999-084
	China	800-820-8682
	Hong Kon	800-96-5941
	India	+91-80-51381665 (Toll)
	Indonesia	001-803-8861-1006
	Korea	080-551-2804
	Malaysia	1-800-80-3973
	New Zealand	0800-446-934
	Philippines	1-800-765-7404
	Singapore	800-886-1028
	Taiwan	0800-006800
	Thailand	001-800-886-0010
Fax		+886-2-2378-6808
Email		tiasia@ti.com or ti-china@ti.com
Internet		support.ti.com/sc/pic/asia.htm

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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