Application Note

BCM920732SW

How to Write WICED Smart™ Applications

BROADCOM
MASS MARKET
PLATFORM

MMP920732SW-AN103-R

5300 California Avenue • Irvine, CA 92617 • Phone: 949-926-5000 • Fax: 949-926-5203

July 17, 2014

Revision History

Revision	***Date***	***Change Description***
MMP920732SW-AN103-R		**Note:** Information on over-the-air updates has been deleted from this document and is now included in *"Secure Over The Air Firmware Upgrade,"* (WICED-Smart-AN100-R). **Deleted:** • "Over the Air Firmware Upgrade" on page 18 • "Create the Private and Public Keys" on page 18 • "Modify the Project to Include the Public Key" on page 19 • "Change the Source Code to Support OTAFU" on page 19 • "Application ID and Version Info" on page 19 • "Changes to the GATT Database" on page 20 • "Write the Callback Handler" on page 21 • "Build the OTAFU Image" on page 21 • "Sign the OTAFU Image" on page 21 • "Upgrading the Firmware" on page 21
MMP920732SW-AN102-R	02/14/14	**Updated:** • "References" on page 5 • "Technical Support" on page 5 • "Using NVRAM" on page 13
MMP920732SW-AN101-R	01/15/14	**Updated:** Converted to Broadcom template
MMP920732SW-AN100-R	01/02/14	Initial release

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Table of Contents

About This Document

Purpose and Scope

Broadcom® Wireless Internet Connectivity for Embedded Devices (WICED; pronounced "wicked") Smart™ SDK (SDK) provides a simple way to write applications for BCM20732 devices. A full API description is included in the help system shipped in the SDK. This document provides design considerations and an overview of application development.

Although some BLE concepts are presented in this document, it is recommended that readers become familiar with BLE technology before attempting to develop an application.

Audience

This document is for software developers who are using the SDK to create applications for Broadcom Bluetooth Smart devices.

Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined on first use. For a comprehensive list of acronyms and other terms used in Broadcom documents, go to:
http://www.broadcom.com/press/glossary.php.

Document Conventions

The following conventions may be used in this document:

Convention	***Description***
Bold	Names of the files and API functions.

`Monospace`	`Command lines and application outputs:`
	`     RAM.hello_sensor-BCM920732TAG_Q32 download`

`< >`	`Placeholders for required elements: <WICED-Smart-SDK>`

References

The references in this section may be used with this document.

Note: Broadcom provides customer access to technical documentation and software through the WICED website (http://community.broadcom.com/welcome). Additional restricted material may be provided through the Customer Support Portal (CSP) and Downloads.

For Broadcom documents, replace the "xx" in the document number with the largest number available to ensure you have the most current version of this document.

Document (or Item) Name	***Number***	***Source***
Broadcom Items		
[1] Quick Start Guide	WICED-Smart-QSG100-R	WICED Smart website
[2] Hardware Interfaces Application Note	920732HW-AN100-R	WICED Smart website
[3] Bluetooth Core Specification v4.0		

Technical Support

Broadcom provides customer access to a wide range of information, including technical documentation, schematic diagrams, bills of material, PCB layout information, and software updates through its customer support portal. For a CSP account, contact your Broadcom Sales or Engineering support representative.

General WICED support is available to registered users within the Broadcom Support Community forums online:

http://community.broadcom.com/welcome.

Introduction

After installation of the WICED Smart SDK, the Apps directory contains BCM20732 sample applications, including:

- Samples for most of the standard Bluetooth Low Energy (BLE) profiles approved by the Bluetooth SIG
- Samples for vendor-specific profiles.

Before starting development of an application it is recommended that the reader examine all samples and identify the one best matches the task required by the application.

BLE Device Types

There are two types of devices:

- Peripheral, for example a sensor
- Central, for example a smart phone

The BCM20732 can function as both a central and a peripheral device. Most of the samples in the SDK are for peripheral devices and this document focuses on developing applications for peripheral devices.

To make itself visible or to establish a connection, a BLE peripheral device sends advertisement packets at configurable intervals. Frequent advertisements increase the chance that the central device will see the advertisement sooner, which results in a smaller latency of the connection establishment. However, the more frequent the advertising rates result in higher power usage.

Typically, the peripheral device sends advertisements at a high rate when a connection must be established, for example when the user pushes a button to connect, or when a new measurement must be delivered. If the central device does not establish a connection after some time, the peripheral typically turns off advertisements or switches to a lower advertisement rate if it still needs to be connectable.

The advertisement data in the advertisement packets can be seen by other devices, even if a connection has not been established. Some commonly used fields include the local name, appearance, transmit power, and UUID of the primary service of the peripheral device.

To see advertisement packets the central device must be configured to perform the BLE scan (listen) procedure. The central device uses the advertisement data to identify the device to which it should connect. The more time a central device stays in scan mode the greater the chance is that it will see the advertisement packet sooner, which results in shorter average latency of the connection establishment. Similar to the peripheral side, the scan consumes power. Typically a central device performs a lazy scan and only when it knows that a connection should be established (for example, based on the user input) does it increase the time it spends scanning.

When a connection is established the central device performs as a master and periodically polls the peripheral, which executes in slave role. The connection interval is negotiated between the master and the slave and results in how often the master polls the slave, which determines data exchange latency and how much power is used during the connection.

BLE Data Exchange

The BLE specification requires that all the data exchanged between a client and a sensor be organized in attributes. Each attribute has unique UUID so that peer can identify it. Each attribute also has a handle so that peer can easily address it. The attribute also has a value that is used to send information between two peers. Each attribute has properties (for example, read and/or write) and permissions (for example, the peer can read and write only over an encrypted link).

Data that can be exchanged between the client and the server is described in characteristics. For example current time, temperature, blood pressure, noise level, or the amount of beer left in a can.

Characteristics are organized in services. A service may have one or several characteristics. For example a thermometer may have an intermediate measurement characteristic and a final measurement characteristic.

A characteristic may have descriptors, for example, characteristic description descriptor, characteristic client configuration descriptor, or a valid range descriptor.

The BLE specification defines valid operations or the ways how a client and a server can exchange data. For example, a client can read or write characteristics values. A server can send notifications, or indications.

Each Bluetooth SIG approved profile defines a service or services that a BLE device should support. Each service defines mandatory and optional characteristics. Characteristic definitions define mandatory and optional characteristic descriptors.

For example, the heart rate profile defines a heart measurement service, which has a mandatory heart rate characteristic, an optional body sensor location characteristic, and an optional heart rate control point characteristic. The heart rate measurement characteristic has a mandatory notify property and a mandatory heart rate measurement characteristic client configuration descriptor with read and write properties.

BLE technology is typically used to send small amounts of data between peripheral and central devices. The size of an attribute is limited to 512 bytes, but in most cases is kept small (below 23 bytes), which is the best from the BLE power-consumption point-of-view.

Developing an Application for the BCM20732

The ROM image of the BCM20732 contains the full LE Stack. The application provides configuration for the stack, for example how often to send advertisements, or what power level to use while transmitting data. Other than that the application needs to focus on application-specific functionality while the stack deals with the low-level details. For example the application does not need to do anything when a client device reads data or performs a full GATT discovery.

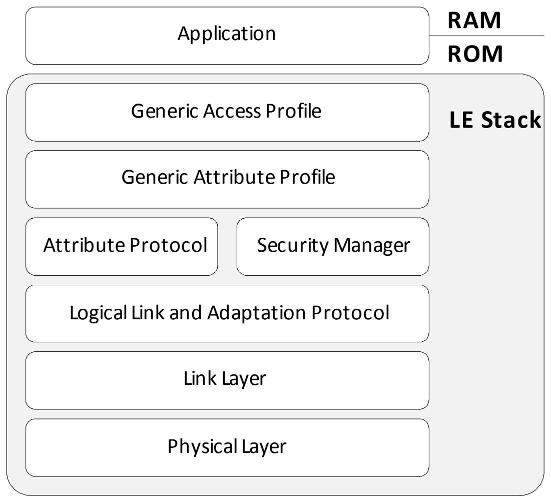

Figure 1: Protocol Stack

The main steps required to develop an application for BCM20732 are:

- Define the data (characteristics) to be exchanged between the client and server, and prepare a GATT database.
- Determine whether additional devices, for example an MCU connected over UART or an SPI connected peripheral sensor will be included in the solution. Refer to [2] for details on how to select GPIOs and how to access peripheral drivers. The UART and GPIO configurations of the application depend on the connected peripheral devices.
- Adjust the application configuration to provide the parameters required by the application. Example adjustments include transmit power, advertisement parameters, device name, etc.
- Define and code the functions for the LE Stack callbacks required by the application. The application will typically require notification from the stack when certain Bluetooth (over the air) events occur. Examples include connection establishment, disconnection, and bonding.

Application Structure

When the BCM20732 starts up it initializes the LE Stack and executes the application initialization function (APPLICATION_INIT). The function should call bleapp_set_cfg, which provides the LE Stack with pointers to application data structures, including the GATT database, application configuration, UART configuration, GPIO configuration, and a pointer to a Create function that is called when application starts.

The following data structures are present in every BCM20732 application:

- The GATT database identifies data objects to the LE Stack that will be exchanged between the application and the client application.
- Application configuration, which specifies parameters shared between the application and the LE Stack.
- UART and GPIO configurations. In some cases the application requires a connection to a peripheral device (for example, a measurement sensor). Although drivers for most peripheral buses are included in the BCM20732 ROM, some code has to be written to support the hardware. For example, some applications may require processing of data received over the UART or SPI interfaces.

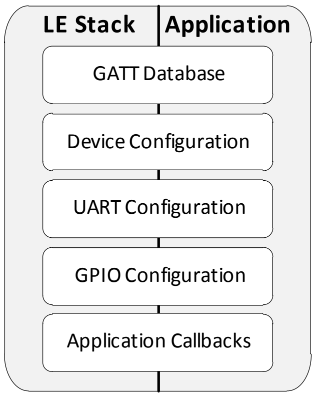

Figure 2: Application Structure

ROM and RAM Applications

In addition to a complete LE Stack ROM image of the BCM20732 contains a few applications. The full source code of the ROM applications is included in the SDK in the **Wiced-Smart/bleapp/app** directory. The new application is flexible in what portions of the ROM code to use. If the ROM code completely matches the requirements, a single **APPLICATION_INIT** is required pointing to data structures and the create function in the ROM. See the Apps\ROM\proximity application for example.

If the GATT database has to be changed a new GATT database must be created in RAM and **APPLICATION_INIT** should point to that RAM area instead of ROM. Similarly, the UART and/or GPIO configurations may be modified. The application can point to its own create function, overwriting the one in the ROM. It is still possible and recommended to use portions of the ROM code to keep the RAM application as small as possible. See the **Apps\ROM\proximity_plus** example in the SDK.

In some cases there is no corresponding ROM application and the developer will need to write all portions of the application using ROM functions only to address the stack and utility functions. Several examples of that are in **Apps\RAM** directory of the SDK.

GATT Database

The GATT database is a central data structure of a BCM20732 application. The peer device can perform GATT discovery to find services, characteristics, and descriptors that are exposed by running the application. After discovery the peer can read and write to the GATT database, subject to characteristic properties and permissions that are setup by the application.

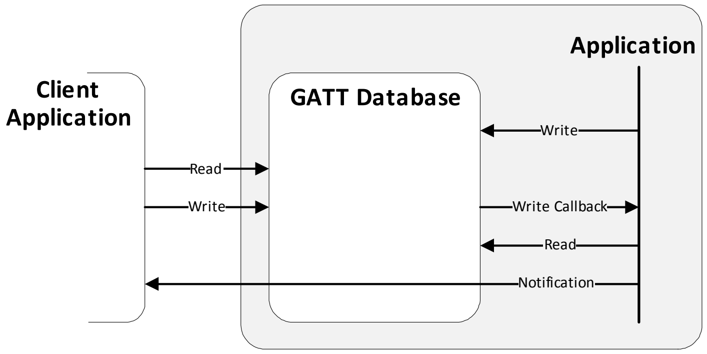

Figure 3: GATT Database Access

A client application can read values of the characteristics and descriptors any time while the connection is up. When a value of some characteristic changes (for example when the application receives new measurements from the sensor), the application may write the value to the GATT database using **bleprofile_WriteHandle**. Depending on the application logic or specification requirements, the application might also send indications (**bleprofile_sendIndication**) or notifications (**bleprofile_sendNotification**) to the client application.

The application can register a callback (**legattdb_regWriteHandleCb**) to receive notification when a peer writes something to the GATT database. When a callback is executed the application receives a handle of the characteristic that has been updated. The application can read the new value from the database using **bleprofile_ReadHandle**.

Create a GATT Database in the BLE Application

The GATT database is a collection of the services, characteristics, and characteristic descriptors that represents the data structure of the BLE application. The SDK provides helper macros to assist in creating the GATT database. See Apps\RAM\hello_sensor.c for an example of a database that has both standard and vendor-specific services.

Each attribute in the GATT database has a handle and a UUID. The UUID can be 16- or128-bits. A UUID uniquely identifies the attribute. For example, 0x1809 is the UUID for the health thermometer service and 0x2A1C is the UUID for the temperature measurement characteristic.

If the Bluetooth SIG does not have a definition for the characteristic your application needs (for example amount of beer left in a can) a vendor-specific unique 128-bit UUID should be used.

A characteristic has two handles, one to identify the characteristic and a second for the value of the characteristic. During GATT discovery the client application can read the UUIDs and their associated handles. For example, during GATT discovery a client can find the UUID of the Temperature Measurements and then read the temperature by sending a read request with the handle of that characteristic value.

Note: According to the Bluetooth specification, the handles in GATT definitions do not need to be consecutive, but should appear in the database in the increasing order.

- **PRIMARY_SERVICE_UUID16** (handle, uuid)

  Creates a definition in the database of a service with a 16-bit UUID. Each BLE device should have the **UUID_SERVICE_GATT and UUID_SERVICE_GAP** services. 16-bit UUIDs are defined by the Bluetooth SIG for standard BLE services, for example the **Find Me** and **Automation IO** services. If the application provides vendor-specific service a 128-bit UUID should be used.
- **PRIMARY_SERVICE_UUID128** (handle, uuid)

  Creates a definition in the database of a service with a 128-bit UUID. 128-bit UUIDs can be used by the application to define a vendor-specific service.
- **CHARACTERISTIC_UUID16** (handle, handle_value, uuid, properties, permission, value_len)

  Adds a characteristic with a 16-bit UUID to a previously defined service. The handle_value is the parameter that the application and the LE Stack use during the data exchange. For example, when the LE Stack calls the Write Callback, it includes a handle_value parameter to indicate which characteristic has been changed.

Note: Do not use this declaration for a characteristic that has Write or Write with No Response property. Use _WRITABLE version of this definition instead.

The properties parameter is a bit field that determines how the characteristic value can be used. The peer can then read the properties of the characteristic while discovering the GATT database of this device.

Table 1 lists the most commonly used definitions.

Table 1: Commonly Used Property Definitions

Property	***Description***
LEGATTDB_CHAR_PROP_BROADCAST	Permits the broadcast of characteristic value.
LEGATTDB_CHAR_PROP_READ	Permits the read of the characteristic value.
LEGATTDB_CHAR_PROP_WRITE_NO_RESPONSE	Permits the write of the characteristic value without response.
LEGATTDB_CHAR_PROP_WRITE	Permits the write of the characteristic value with response.
LEGATTDB_CHAR_PROP_NOTIFY	Permits the notification of the characteristic value without acknowledgement.
LEGATTDB_CHAR_PROP_INDICATE	Permits the indication of the characteristic value with acknowledgement.
LEGATTDB_CHAR_PROP_AUTHD_WRITES	Permits signed writes to the characteristic value.
LEGATTDB_CHAR_PROP_EXTENDED	Additional characteristic properties are defined.

The permission parameter specifies whether the characteristic value can be read and/or written by the client and identifies the security level required for the read, write, notify, and indicate procedures. These are permissions set up by this application (Table 2 on page 12 lists the defined permission bits).

Table 2: Defined Permission Bits

Property	***Definition***
LEGATTDB_PERM_NONE	Characteristic value is not readable or writable (can support indication or notification)
LEGATTDB_PERM_VARIABLE_LENGTH	Characteristic value can be a variable-length array
LEGATTDB_PERM_READABLE	Permits the read of the characteristic value
LEGATTDB_PERM_WRITE_CMD	Permits the write of the characteristic value without response
LEGATTDB_PERM_WRITE_REQ	Permits the write of the characteristic value with response
LEGATTDB_PERM_AUTH_READABLE	Permits the authenticated read operation
LEGATTDB_PERM_RELIABLE_WRITE	Permits reliable write operation
LEGATTDB_PERM_AUTH_WRITABLE	Permits authenticated write operation

- CHARACTERISTIC_UUID128 (handle, handle_value, uuid, properties, permission, value_len)
  Adds a characteristic with a 128-bit UUID to the service. (See Table 1 and Table 2 above for the property and permission descriptions.)

Note: Do not use this declaration for a characteristic that has Write or Write with No Response property. Use _WRITABLE version of this definition (see below) instead.

- CHARACTERISTIC_UUID16_WRITABLE (handle, handle_value, uuid, properties, permission, value_len)
  Adds a 16-bit writable characteristic to a previously defined service. (See Table 1 and Table 2 above for the property and permission descriptions.)
- CHARACTERISTIC_UUID128_WRITABLE (handle, handle_value, uuid, properties, permission, value_len)
  Adds a writable characteristic with a 128-bit UUID to a previously defined service. (See Table 1 and Table 2 for the property and permission descriptions.)
- CHAR_DESCRIPTOR_UUID16 (handle, uuid, permission, value_len)
  One or more descriptors can exist in each characteristic definition. Possible descriptors are defined in the Bluetooth specification. (See Table 1 and Table 2 above for the property and permission descriptions.)

Note: Do not use this declaration for a descriptor that has WRITE_CMD or WRITE_REQ permission. Use _WRITABLE version of this definition (see below) instead.

- CHAR_DESCRIPTOR_UUID16_WRITABLE (handle, uuid, permission, value_len)
  Adds a 16-bit UUID writable characteristic descriptor to a previously defined characteristic (See Table 1 and Table 2 above for the property and permission descriptions.)

Using NVRAM

The LE Stack allows the application to use BCM20732 NVRAM to store application parameters. Use the bleprofile_WriteNVRAM function to write a block of data in NVRAM and bleprofile_ReadNVRAM to retrieve the data.

In some cases certain GATT database attributes should persist when the device is power-cycled. A typical example is the characteristic client configuration descriptor, which allows the client to register to receive indications or notifications when the characteristic value changes. Immediately after the bonding client will typically set the characteristic client configuration descriptor and this will be done once for the duration of the bond. The application should remember the settings even after a power cycle.

The application is responsible for storing persistent GATT database attributes in NVRAM and restore them after a device power-cycle.

Below is an example from the hello_sensor sample of saving the characteristic client configuration descriptor in NVRAM in the write callback.

```
// By writing into Characteristic Client Configuration descriptor
// peer can enable or disable notification or indication
if ((len == 2) &&
     (handle == HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR))
{
     hello_sensor_hostinfo.characteristic_client_configuration =
        attrPtr[0] + (attrPtr[1] << 8);

    // Save update to NVRAM. Client does not need to set it on every connection.
    bleprofile_WriteNVRAM(VS_BLE_HOST_LIST, sizeof(hello_sensor_hostinfo),
         (UINT8 *)&hello_sensor_hostinfo);
}
```

The first parameter in the bleprofile_WriteNVRAM function call is the ID. The application can specify any ID between 0x10 and 0x6F. The stack allocates memory and copies the contents of the payload. The same payload can be retrieved later by using the same ID when calling bleprofile_ReadNVRAM.

The maximum NVRAM read or write block size is 255 bytes.

BLE Advertisements and Connection Establishment

By default the LE Stack adds the following fields to the advertisement packets: local name, appearance, transmission power, and the 16-bit UUID of the primary service of the peripheral device.

The application can overwrite the default and configure application-specific advertisement data using the bleprofile_GenerateADVData and bleprofile_GenerateScanRspData function calls. The application can also set the transmission power during advertisements. The example code below shows how to set advertisement data to include a 128-bit UUID.

```
BLE_ADV_FIELD adv[3];

// flags
adv[0].len = 1 + 1;
adv[0].val = ADV_FLAGS;
adv[0].data[0] = LE_LIMITED_DISCOVERABLE | BR_EDR_NOT_SUPPORTED;

// 128 bit uuid
adv[1].len = 16 + 1;
adv[1].val = ADV_SERVICE_UUID128_COMP;
memcpy(adv[1].data, uuid128, 16);

// name
adv[2].len = strlen(bleprofile_p_cfg->local_name) + 1;
adv[2].val = ADV_LOCAL_NAME_COMP;
memcpy(adv[2].data, bleprofile_p_cfg->local_name, adv[2].len - 1);

bleprofile_GenerateADVData(adv, 3);

// Set transmission power during advertisement
blecm_setTxPowerInADV(0);
bleprofile_Discoverable(HIGH_UNDIRECTED_DISCOVERABLE, NULL);
```

A peripheral device can stay in the advertisement state for a long period of time. If the device uses a coin cell battery the interval between advertisement packets may significantly affect the battery life. When the peripheral device must connect to a central device, for example on a startup, user action, or data transfer, it starts with a high advertisement rate (a short interval between packets). After a certain time the peripheral will typically enter a low advertisement state (a longer interval between packets). If device does not need to be connectable at all times the application should stop advertisements.

Function **bleprofile_Discoverable** allows the application to enter advertisement state. If the first parameter is set to **HIGH_UNDIRECTED_DISCOVERABLE**, the stack executes the advertisement state machine. It starts high, and then goes to low advertisements. When the low advertisement state times out, the LE Stack indicates to the application that advertisements are being stopped through the special callback (see **bleprofile_regAppEvtHandler(BLECM_APP_EVT_ADV_TIMEOUT**…) for details).

The parameters in the application configuration shown below define the interval and duration of each of the advertisement states.

```
/*.high_undirect_adv_interval =*/ 32,// slots
/*.low_undirect_adv_interval  =*/ 1024,// slots
/*.high_undirect_adv_duration =*/ 30,// seconds
/*.low_undirect_adv_duration  =*/ 300,// seconds
```

The application should stop advertising after a connection with the central device is established.

Application Callbacks

Most of the logic required by the Bluetooth specifications is executed by the LE Stack of the BCM20732. Application callbacks are the entry points from the LE Stack to the application. They are used to notify the application of the occurrence of certain events.

Below is an example from the hello_sensor Create function that shows registration for various events that the application wants to process. The functions are described in detail after the example code.

```
// register connection up and connection down handler.
bleprofile_regAppEvtHandler(BLECM_APP_EVT_LINK_UP, hello_sensor_connection_up);
bleprofile_regAppEvtHandler(BLECM_APP_EVT_LINK_DOWN, hello_sensor_connection_down);
bleprofile_regAppEvtHandler(BLECM_APP_EVT_ADV_TIMEOUT,
    hello_sensor_advertisement_stopped);

// handler for Encryption changed.
blecm_regEncryptionChangedHandler(hello_sensor_encryption_changed);

// handler for Bond result
lesmp_regSMPResultCb((LESMP_SINGLE_PARAM_CB) hello_sensor_smp_bond_result);

// register to process client writes
legattdb_regWriteHandleCb((LEGATTDB_WRITE_CB) hello_sensor_write_handler);

// register interrupt handler
bleprofile_regIntCb((BLEPROFILE_SINGLE_PARAM_CB) hello_sensor_interrupt_handler);

// register to be notified on fine and one second timeouts
bleprofile_regTimerCb(hello_sensor_fine_timeout, hello_sensor_timeout);
```

A call to **bleprofile_regAppEvtHandler** registers the application for the one of the following events:

BLECM_APP_EVT_START_UP-register with LE Stack to be called when the stack completes the startup sequence.

BLECM_APP_EVT_LINK_UP- connection established callback. At the time when connection is established the application can query the stack about the Bluetooth device address of the connected device using **emconninfo_getPeerAddr** and the HCI connection handle using **emconinfo_getConnHandle**.

After the connection is established retrieve the data in NVRAM and update the GATT database with the stored data because the peer can start reading from the GATT database as soon as the connection is up.

To conserve power stop advertisements after the connection is established.

BLECM_APP_EVT_LINK_DOWN-disconnected callback. If the device needs to allow connections from the peer, advertisements might need to be restarted.

BLECM_APP_EVT_ADV_TIMEOUT-exiting advertisement state callback. If the application needs the device discoverable at all times advertisements should be restarted.

BLECM_APP_EVT_ENTERING_HIDOFF-entering deep-sleep mode. The LE Stack typically makes the decision to enter deep-sleep (hid-off) based on the time before the next action. For example, deep-sleep is used between advertisements while the device is in the low-duty advertisement state. Before entering deep-sleep the LE Stack asks the application if it is okay to do that.

The application can disable deep-sleep mode or adjust wake-up sources (such as a key scan) using **devlpm_disableWakeFrom**.

BLECM_APP_EVT_ABORTING_HIDOFF-deep-sleep aborted. This callback indicates that the device aborted entering the deep-sleep state because of a pending interrupt.

lesmp_regSMPResultCb-registers the application to be notified when the pairing procedure is complete. If the value of the parameter passed to the registered callback is LESMP_PAIRING_RESULT_BONDED, pairing was successful and a bond was established with the peer device. The application should reinitialize the NVRAM area with the Bluetooth device address and other settings associated with the newly-paired device.

blecm_regEncryptionChangedHandler allows the application to receive notification when the BLE link is encrypted. Most BLE applications require that data exchanged between devices be encrypted. When the application receives data it should verify that the link was encrypted. Similarly the application should verify the link state before sending any indications or notifications because the link can be compromised.

In most cases the application registers a write callback using **legattdb_regWriteHandleCb**. The write callback is the central place for processing of all Write and Write with No Response requests from the client device.

When the peer attempts a write, the LE Stack verifies the permissions setup by the application. If verification passes, the LE Stack stores the value in the GATT database and calls the application. When the function returns the LE Stack, if appropriate, sends a response to the client. The application does not need to do anything other than provide the correct return code.

In the write callback, the application can retrieve the handle of the attribute that has been accessed using **legattdb_getHandle. legattdb_getAttrValueLen** returns the length of the data that was written to the GATT database. The data portion of the modified attribute can be retrieved by the application with **bleprofile_ReadHandle**. Below is an example of how **hello_sensor** handles write requests from the client application.

```
// Process write request or command from peer device
void hello_sensor_write_handler(LEGATTDB_ENTRY_HDR *p)
{
    UINT16 handle = legattdb_getHandle(p);
    int len = legattdb_getAttrValueLen(p);
    UINT8 *attrPtr = legattdb_getAttrValue(p);

    // By writing into Characteristic Client Configuration descriptor
    // peer can enable or disable notification or indication. The handle
    // passed should match the handle in the GATT database
    if ((len == 2) &&
         (handle == HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR))
    {
     hello_sensor_hostinfo.characteristic_client_configuration =
     attrPtr[0] + (attrPtr[1] << 8);
        ble_trace1("hello_sensor_write_handler: client_configuration %04x\n",
        hello_sensor_hostinfo.characteristic_client_configuration);

        // Save update to NVRAM. Client does not need to set it on every connection.
        bleprofile_WriteNVRAM(VS_BLE_HOST_LIST,
            sizeof(hello_sensor_hostinfo), (UINT8 *)&hello_sensor_hostinfo);
    }
    // User can change number of blinks to send when button is pushed
    else if ((len == 1) && (handle == HANDLE_HELLO_SENSOR_CONFIGURATION))
    {
        hello_sensor_hostinfo.number_of_blinks = attrPtr[0];
        if (hello_sensor_hostinfo.number_of_blinks != 0)
        {
            bleprofile_LEDBlink(250, 250, hello_sensor_hostinfo.number_of_blinks);
        }
        // Save update to NVRAM. Client does not need to set it on every connection.
        bleprofile_WriteNVRAM(VS_BLE_HOST_LIST,
            sizeof(hello_sensor_hostinfo), (UINT8 *)&hello_sensor_hostinfo);
        ble_trace1("hello_sensor_write_handler: NVRAM write:%04x\n", writtenbyte);
    }
```

Accessing the BCM20732 Hardware

The LE Stack in the BCM20732 supports a wide variety of hardware interfaces. See Reference [2] on page 5 for details on how to connect and access external sensors that are connected via UART, SPI, I2C, and ADC.

The UART and GPIO configuration of the target device should be provided by the application. The hello_sensor example below shows the configuration for the BCM20732TAG board that is provided with the WICED Smart SDK.

```
// Following structure defines UART configuration
const BLE_PROFILE_PUART_CFG hello_sensor_puart_cfg =
{
    /*.baudrate  =*/ 115200,
    /*.txpin     =*/ PUARTDISABLE | GPIO_PIN_UART_TX,
    /*.rxpin     =*/ PUARTDISABLE | GPIO_PIN_UART_RX,
};

// Following structure defines GPIO configuration used by the application
const BLE_PROFILE_GPIO_CFG hello_sensor_gpio_cfg =
{
    /*.gpio_pin =*/
    {
        GPIO_PIN_WP,// This need to be used to enable/disable NVRAM write protect
        GPIO_PIN_BUTTON,// Button GPIO is configured to trigger either direction of
                            // interrupt
        GPIO_PIN_LED,// LED GPIO, optional to provide visual effects
        GPIO_PIN_BATTERY,// Battery monitoring GPIO. When it is lower than particular
                            // level, it will give notification to the application
        GPIO_PIN_BUZZER,// Buzzer GPIO, optional to provide audio effects
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
// other GPIOs are not used
    },
    /*.gpio_flag =*/
    {
        GPIO_SETTINGS_WP,
        GPIO_SETTINGS_BUTTON,
        GPIO_SETTINGS_LED,
        GPIO_SETTINGS_BATTERY,
        GPIO_SETTINGS_BUZZER,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    }
};
```

If the LED is enabled it can be accessed through the simple **bleprofile_LEDOn, bleprofile_LEDOff**, and **bleprofile_LEDBlink** functions.

The TAG board buzzer can be accessed through **bleprofile_BUZOn, bleprofile_BUZOff, bleprofile_BUZBeep**, and **bleprofile_PWMBUZFreq**. See the function descriptions and parameter usage in the **bleprofile.h** file or in the SDK help.

Timers

The LE Stack supports two free-running timers:

- Normal timer has a resolution of one second.
- Fine timer can be configured in the device configuration to be as low as 12.5 milliseconds. Call **bleprofile_regTimerCb** to register with the LE Stack timer callbacks.

Some BLE profiles require a device to stay connected for a certain time after the last measurement data packet has been processed. If the application starts the connection-idle timer using **bleprofile_StartConnIdleTimer**, the LE Stack monitors the link and disconnects after the specified timeout unless there is data activity on the link. **bleprofile_StopConnIdleTimer** can be used to the stop idle timer.

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Phone: 949-926-5000
Fax: 949-926-5203
E-mail: info@broadcom.com
Web: www.broadcom.com

MMP920732SW-AN102-D1 July 17, 2014