

Chapter 6: Bluetooth® LE centrals

This chapter introduces you to the central side of the Bluetooth® LE connection. By the end of this chapter you will be able to create a Bluetooth® LE central that finds the right Bluetooth® LE peripheral, connects to it, reads, writes and enables/handles notifications. It will also be able to perform the GATT service discovery procedure to find the handles of the services, characteristics and descriptors on the GATT server.

Table of contents

6.1	GAP Roles, the observer and the central	3
6.2	Stack configuration for central devices.....	3
6.3	Scanning	5
6.4	Connecting, pairing and encrypting.....	8
6.5	Attribute protocol & more GATT procedures	10
6.5.1	GATT utility functions.....	11
6.5.2	GATT client read	11
6.5.3	GATT client write and write command.....	13
6.5.4	GATT client notify and indicate	14
6.5.5	GATT group	15
6.5.6	GATT client read by group type	15
6.5.7	GATT client find by type value	16
6.5.8	GATT client read by type	16
6.5.9	GATT client find information	17
6.6	Service discovery	17
6.6.1	Service discovery algorithm	17
6.6.2	Service discovery implementation	18
6.7	Running a GATT Server.....	19
6.8	Exercises	20
Exercise 1:	Make an observer	20
Exercise 2:	Parse the device name and list only your peripheral.....	22
Exercise 3:	Connect to your peripheral and turn LED ON/OFF.....	23
Exercise 4:	Add commands to enable/disable notifications	26
Exercise 5:	Implement service discovery	28

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR) ; make build
<i>Italics</i>	Displays file names and paths	sourcefile.hex
[bracketed, bold]	Displays keyboard commands in procedures	[Enter] or [Ctrl] [C]
Menu > Selection	Represents menu paths	File > New Project > Clone
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the Debugger icon, and then click Next .

6.1 GAP Roles, the observer and the central

In the previous chapters the focus has been on Bluetooth® LE peripherals. Instead of dividing the world into peripheral and central, it would have been more technically correct to say that Bluetooth® Low Energy has four GAP device roles:

- **Broadcaster:** A device that only advertises (e.g. beacon)
- **Peripheral:** A device that can advertise and be connected to
- **Observer:** A device that passively listens to advertisers (e.g. beacon scanner)
- **Central:** A device that can listen to advertisers and create a connection to a peripheral

So, the previous chapters were really focused on broadcasters and peripherals. But what about the central side of a connection? The answer to that question is the focus of this chapter.

6.2 Stack configuration for central devices

In previous chapters, we looked at the Stack settings for a peripheral but the `wiced_bt_cfg_settings` structure in `app_bt_cfg.c` also allows you to specify settings for a central.

There are two sets of sub-structures that cover most of the settings for centrals – `gatt_cfg` and `ble_scan_config`. Each one is discussed below.

The `gatt_cfg` structure specifies, among other things, how many GATT connections a device allows as a client and as a server. The default allows 0 client links, so if you don't change that value, your central would not be able to connect to a GATT server running on a peripheral. Therefore, you will typically change the max client links to 1 for a central. You can also change the maximum number of allowed server links to 0 since your central will not usually have a GATT server and thus doesn't need to allow server connections to it. The beginning of the structure looks like this (after making changes to the `client_max_links` and `server_max_links` values for a typical central):

```
gatt_cfg = /* GATT configuration */
{
    .appearance = APPEARANCE_GENERIC_TAG, /**< GATT appearance */
    .client_max_links = 1, /**< Max number of servers the local client can connect to */
    .server_max_links = 0, /**< Max number of remote client connections allowed by the
                                local device*/
    ...
}
```

The `ble_scan_config` structure has configuration settings for the scanning that the Central does. It also has settings for the connection configuration. That is, it specifies the connection interval, latency, and timeout that the Central requires.

The structure looks like this:

```
.ble_scan_cfg = /* BLE scan settings */
{
    .scan_mode = BTM_BLE_SCAN_MODE_PASSIVE,

    /* Advertisement scan configuration */
    .high_duty_scan_interval = WICED_BT_CFG_DEFAULT_HIGH_DUTY_SCAN_INTERVAL,
    .high_duty_scan_window = WICED_BT_CFG_DEFAULT_HIGH_DUTY_SCAN_WINDOW,
    .high_duty_scan_duration = 5,

    .low_duty_scan_interval = WICED_BT_CFG_DEFAULT_LOW_DUTY_SCAN_INTERVAL,
    .low_duty_scan_window = WICED_BT_CFG_DEFAULT_LOW_DUTY_SCAN_WINDOW,
}
```

```
.low_duty_scan_duration          = 5,

/* Connection scan configuration */
.high_duty_conn_scan_interval    = WICED_BT_CFG_DEFAULT_HIGH_DUTY_CONN_SCAN_INTERVAL,
.high_duty_conn_scan_window      = WICED_BT_CFG_DEFAULT_HIGH_DUTY_CONN_SCAN_WINDOW,
.high_duty_conn_duration         = 30,

.low_duty_conn_scan_interval     = WICED_BT_CFG_DEFAULT_LOW_DUTY_CONN_SCAN_INTERVAL,
.low_duty_conn_scan_window       = WICED_BT_CFG_DEFAULT_LOW_DUTY_CONN_SCAN_WINDOW,
.low_duty_conn_duration         = 30,

/* Connection configuration */
.conn_min_interval               = WICED_BT_CFG_DEFAULT_CONN_MIN_INTERVAL,
.conn_max_interval               = WICED_BT_CFG_DEFAULT_CONN_MAX_INTERVAL,
.conn_latency                     = WICED_BT_CFG_DEFAULT_CONN_LATENCY,
.conn_supervision_timeout        = WICED_BT_CFG_DEFAULT_CONN_SUPERVISION_TIMEOUT,
},
```

Since the central doesn't usually have a GATT database (it connects to the GATT database on a peripheral device that has a server) and since the central doesn't advertise, there is no need to use the Bluetooth® Configurator. Therefore, the include for the configurator generated file *cycfg_gatt_db.h* is removed from *app.c* and *app_bt_cfg.c* in the template central application. The device name is still required in the *wiced_bt_cfg_settings* structure in *app_bt_cfg.c* so the template just hard-codes in that file:

```
uint8_t deviceName[] = {'c', 'e', 'n', 't', 'r', 'a', 'l', '\0', };

const wiced_bt_cfg_settings_t wiced_bt_cfg_settings =
{
    .device_name          = deviceName,
```

6.3 Scanning

In the previous chapters you learned how to create different peripherals that advertise their existence and some data. How does a central find this information? And how does it use that information to get connected?

First, you must put the central into scanning mode. You do this with a simple call to `wiced_bt_ble_scan`. This function takes three arguments.

1. The first argument is a `wiced_bt_ble_scan_type_t` which tells the controller to either turn off, scan fast (high duty) or scan slowly (low duty).

```
Enum wiced_bt_ble_scan_type_e
{
    BTM_BLE_SCAN_TYPE_NONE,           /**< Stop scanning */
    BTM_BLE_SCAN_TYPE_HIGH_DUTY,      /**< High duty cycle scan */
    BTM_BLE_SCAN_TYPE_LOW_DUTY       /**< Low duty cycle scan */
};
typedef uint8_t wiced_bt_ble_scan_type_t;
```

The actual values of scan high and low duty are set in the Bluetooth® configurator, as you saw earlier.

2. The next argument is a `wiced_bool_t` that tells the scanner to filter or not. If you enable the filter, the scanner will only call you back one time for each unique BD ADDR that it hears even if the advertising packet changes.
3. The final argument is a function pointer to a callback function that looks like this:

```
void myScanCallback(wiced_bt_ble_scan_results_t *p_scan_result, uint8_t *p_adv_data);
```

For example, you may start high duty cycle scanning with filtering like this:

```
wiced_bt_ble_scan( BTM_BLE_SCAN_TYPE_HIGH_DUTY, WICED_TRUE, myScanCallback );
```

Each time that your central hears an advertisement, it will call your callback with a pointer to a scan result structure with information about the device it just heard, and a pointer to the raw advertising data. The scan result structure simply has the Bluetooth® Device Address, the address type, what type of advertisement packet, the RSSI and a flag like this:

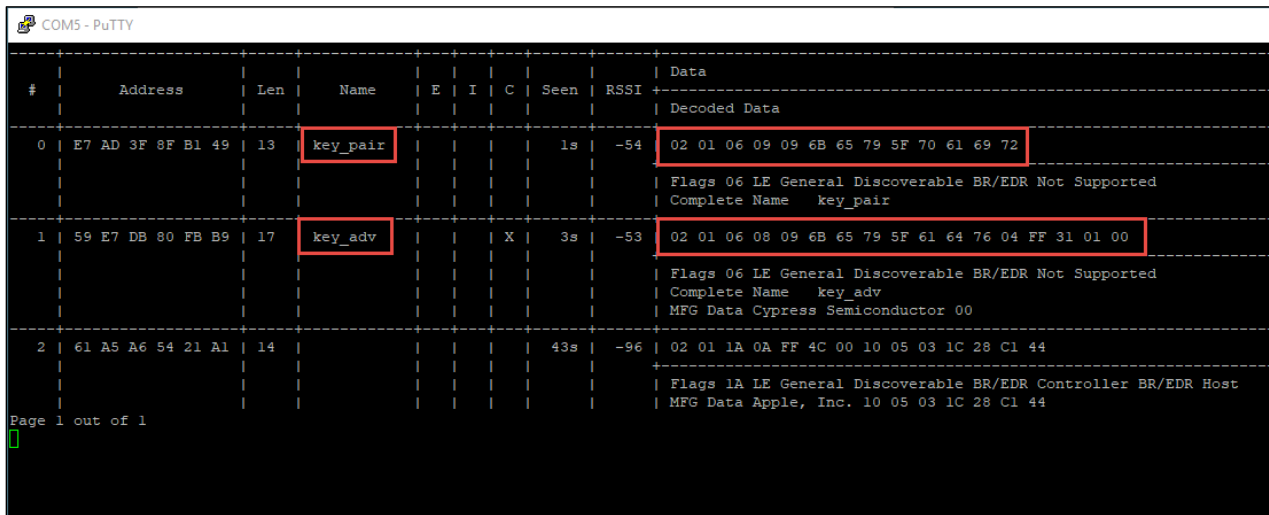
```
/** LE inquiry result type */
typedef struct
{
    wiced_bt_device_address_t    remote_bd_addr;           /**< Device address */
    uint8_t                     ble_addr_type;             /**< LE Address type */
    wiced_bt_dev_ble_evt_type_t ble_evt_type;             /**< Scan result event type */
    int8_t                      rssi;
    uint8_t                     flag;
} wiced_bt_ble_scan_results_t;
```

In your advertising callback function you can then parse the advertising data to decide what to do next. The `wiced_bt_sdk` library provides you a function called `wiced_bt_ble_check_advertising_data` which can help you find information in the packet. Recall that every advertising packet is broken up into fields that each have a type. The `wiced_bt_ble_check_advertising_data` function will search the advertising packet looking for a field that you specify and then, if it finds that field, will return a pointer to the field and the length (via a pointer). For example, you might have this inside the callback function to search for a service UUID:

```
uint8_t len;
uint8_t *findServiceUUID = wiced_bt_ble_check_advertising_data(p_adv_data,
    BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE, &len);
```

After making this call, `findServiceUUID` will either be 0 (it didn't find the field) or will be a pointer to the bytes that make up the service UUID. In addition, `len` will either be 0 or it will be the number of bytes in that field. Remember that the enumeration `wiced_bt_ble_advert_type_e` in the file `mtb_shared/wiced_btsdk/dev-kit/baselib/<device>/<version>/COMPONENT_<device>/include/wiced_bt_ble.h` has a list of the legal advertising field types.

So, now what? Consider the two cases in the following screenshot from an Advertising Scanner (you will use this application in one of the exercises). In this case you can see that there are two different devices advertising, one named "key_pair" and one named "key_adv". You can see the raw bytes of the advertising packet and the decode of those bytes.



#	Address	Len	Name	E	I	C	Seen	RSSI	Data
0	E7 AD 3F 8F B1 49	13	key_pair				1s	-54	02 01 06 09 09 6B 65 79 5F 70 61 69 72 Flags 06 LE General Discoverable BR/EDR Not Supported Complete Name key_pair
1	59 E7 DB 80 FB B9	17	key_adv			X	3s	-53	02 01 06 08 09 6B 65 79 5F 61 64 76 04 FF 31 01 00 Flags 06 LE General Discoverable BR/EDR Not Supported Complete Name key_adv MFG Data Cypress Semiconductor 00
2	61 A5 A6 54 21 A1	14					43s	-96	02 01 1A 0A FF 4C 00 10 05 03 1C 28 C1 44 Flags 1A LE General Discoverable BR/EDR Controller BR/EDR Host MFG Data Apple, Inc. 10 05 03 1C 28 C1 44

Page 1 out of 1

If you were looking for a device named `key_pair` you could do something like this:

```
uint8_t len;
uint8_t * p_name = wiced_bt_ble_check_advertising_data( p_adv_data,
BTM_BLE_ADVERT_TYPE_NAME_COMPLETE, &len );

if( p_name && ( len == strlen("key_pair") ) && ! memcmp( "key_pair", p_name, len ) )
{
    WICED_BT_TRACE("Found Matching Device with BD Address: [%B]",
p_scan_result->remote_bd_addr );
}
```

If you were looking for a device that was advertising the `MySvc` service UUID you might do the following. Notice that the UUIDs are stored little endian in the advertising packet. This UUID is the one that the template peripheral application has – I just copied over the macro from the `GeneratedSource/cycfg_gatt_db.h` file. If you created your own application, it would have a different (random) UUID unless you manually enter this value in the configurator.

```
#define __UUID_SERVICE_MYSVC    0x83u, 0xC3u, 0x54u, 0x15u, 0xC7u, 0x55u, 0x38u, 0x89u, 0x2Bu,
                                0x4Du, 0x46u, 0xA2u, 0xABu, 0x8Cu, 0x2Bu, 0x87u

static const uint8_t serviceUUID[]={ __UUID_SERVICE_MYSVC };

uint8_t *findServiceUUID = wiced_bt_ble_check_advertising_data(p_adv_data,
BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE,&len);

if(findServiceUUID && (memcmp(findServiceUUID,serviceUUID,len) == 0))
{
    WICED_BT_TRACE("Host = %B Found Service UUID\r\n ", p_scan_result->remote_bd_addr);
}
```

Note: The first argument to the "if" function is there to make sure we have a valid (non-null) pointer returned from the advertising packet.

There are several functions which can be useful in comparing data for when you want to identify a specific device such as (note that you must include *string.h* to get access to `memcmp`):

1. `memcmp(unit8_t *p1, uint8_t *p2, int size)` allows you to compare two blocks of memory. It returns 0 if the two blocks of memory are the same.
2. If you have two variables of type `wiced_bt_uuid_t` you can compare them with the `wiced_bt_util_uuid_cmp` function. Like the functions above, it returns 0 if the two UUIDs match.

```
Int wiced_bt_util_uuid_cmp(wiced_bt_uuid_t *p_uuid1, wiced_bt_uuid_t *p_uuid2);
```

3. Remember that `WICED_BT_TRACE_ARRAY` can be used to print the UUID array.

6.4 Connecting, pairing and encrypting

Now that you have found a device that you are interested in, what next? First you need to register a GATT event callback. This looks exactly like it does on the peripheral side. Note that you don't need to initialize the GATT database because the central doesn't contain the database – it's on the peripheral.

```
wiced_bt_gatt_register(app_bt_gatt_callback);
```

To make a connection you just call `wiced_bt_gatt_le_connect`:

```
wiced_bool_t wiced_bt_gatt_le_connect(wiced_bt_device_address_t bd_addr,  
                                       wiced_bt_ble_address_type_t bd_addr_type,  
                                       wiced_bt_ble_conn_mode_t conn_mode,  
                                       wiced_bool_t is_direct);
```

The `bd_addr` and `bd_addr_type` are passed in to the callback as part of the scan result. The `conn_mode` is an enumeration with three possible values which determines how fast the connection is established (and how much power is consumed to establish the connection):

```
enum wiced_bt_ble_conn_mode_e  
{  
    BLE_CONN_MODE_OFF,                /*< Stop initiating */  
    BLE_CONN_MODE_LOW_DUTY,           /*< slow connection scan parameter */  
    BLE_CONN_MODE_HIGH_DUTY,          /*< fast connection scan parameter */  
};
```

The final argument should be set to `WICED_TRUE`.

When the connection has been made, the GATT callback that you registered with `wiced_bt_gatt_register` will be called with the event `GATT_CONNECTION_STATUS_EVT`. The parameter passed to you will be of type `wiced_bt_gatt_connection_status_t` which contains a bunch of information about the connection.

```
typedef struct  
{  
    uint8_t          *bd_addr;    /*< Remote device address */  
    wiced_bt_ble_address_type_t addr_type; /*< Remote device address type */  
    uint16_t         conn_id;     /*< ID of the connection */  
    wiced_bool_t     connected;   /*< TRUE/FALSE connected/disconnected */  
    wiced_bt_gatt_disconn_reason_t reason; /*< Reason code (see @link  
                                       wiced_bt_gatt_disconn_reason_e  
                                       wiced_bt_gatt_disconn_reason_t @endlink) */  
    wiced_bt_transport_t transport; /*< Transport type of the connection */  
    uint8_t          link_role;   /*< Link role on this connection */  
} wiced_bt_gatt_connection_status_t;
```

Typically, you would save the `conn_id` so that you can perform reads and writes to the peripheral. If you were going to support multiple connections, you might make a table of connection ID / Bluetooth® address tuples.

Once connected, the central can initiate pairing (if the devices were not previously bonded). The function is `wiced_bt_dev_sec_bond`:

```
wiced_result_t wiced_bt_dev_sec_bond(wiced_bt_device_address_t bd_addr,  
                                       wiced_bt_ble_address_type_t bd_addr_type,  
                                       wiced_bt_transport_t transport,  
                                       uint8_t pin_len,  
                                       uint8_t *p_pin);
```

The first two arguments are available in the event data passed to the GATT callback function. For example, `p_event_data->connection_status.bd_addr` and `p_event_data->connection_status.addr_type`.

The transport can be either `BT_TRANSPORT_BR_EDR` (for Classic) or `BT_TRANSPORT_LE` (for Bluetooth® LE). The last two arguments are only for legacy pairing modes, so just use 0 for `pin_len` and `NULL` for `p_pin`.

Remember that some characteristic values can only be read or written once the device is paired. This is determined by the GATT characteristic permissions (e.g. `LEGATTTDB_PERM_AUTH_READABLE` or `LEGATTTDB_PERM_AUTH_WRITABLE`).

If you previously saved bonding information on both the peripheral and client, then you don't need to initiate pairing on subsequent connections. In that case instead of `wiced_bt_dev_sec_bond` you just need to enable encryption by calling `wiced_bt_dev_set_encryption`:

```
wiced_result_t wiced_bt_dev_set_encryption (wiced_bt_device_address_t bd_addr,
                                             wiced_bt_transport_t transport,
                                             void *p_ref_data);
```

The transport is again either `BT_TRANSPORT_BR_EDR` (for Classic) or `BT_TRANSPORT_LE` (for Bluetooth® LE). The last argument is a pointer to an enumeration of type `wiced_bt_ble_sec_action_type_t` which returns the encryption status. The enumeration is:

```
/** LE encryption method */
enum
{
    BTM_BLE_SEC_NONE,           /**< No encryption */
    BTM_BLE_SEC_ENCRYPT,        /**< encrypt the link using current key */
    BTM_BLE_SEC_ENCRYPT_NO_MITM, /**< encryption without MITM */
    BTM_BLE_SEC_ENCRYPT_MITM    /**< encryption with MITM*/
};
typedef uint8_t wiced_bt_ble_sec_action_type_t;
```

To summarize, the client would typically do something like this after making a connection:

```
#define VSID_KEYS_START          (WICED_NVRAM_VSID_START)
#define VSID_NUM_KEYS           (4)
#define VSID_KEYS_END           (VSID_KEYS_START+VSID_NUM_KEYS)

wiced_bt_device_link_keys_t temp_keys;
wiced_bt_ble_sec_action_type_t encryption_type = BTM_BLE_SEC_ENCRYPT;

for ( I = VSID_KEYS_START; I < VSID_KEYS_END; i++ ) //Search NVRAM for keys
{
    bytes_read = wiced_hal_read_nvr( I, sizeof( temp_keys ), //Attempt to read NVRAM
                                     (uint8_t *)&temp_keys, &result );
    if ( result == WICED_SUCCESS ) // NVRAM had something at this location
    {
        if ( memcmp( temp_keys.bd_addr, bd_address, BD_ADDR_LEN ) == 0 )
        {
            isBonded = WICED_TRUE; // We found keys for the Peripheral's BD Address
        }
    }
}

if ( isBonded ) /* Device is bonded so just need to enable encryption */
{
    status = wiced_bt_dev_set_encryption( p_peer_info->peer_addr,
                                          p_peer_info->transport,
                                          &encryption_type );
    WICED_BT_TRACE( "wiced_bt_dev_set_encryption %d \n", status );
}
else /* Device not bonded so we need to pair */
{
    status = wiced_bt_dev_sec_bond( p_peer_info->peer_addr,
                                   p_peer_info->addr_type,
                                   p_peer_info->transport, 0, NULL );
    WICED_BT_TRACE( "wiced_bt_dev_sec_bond %d \n", status );
}
```

When you want to disconnect, just call `wiced_bt_gatt_disconnect` with the connection ID as a parameter. Note that the connect function has "le" in the name but the disconnection function does not!

6.5 Attribute protocol & more GATT procedures

In the previous chapters you saw the peripheral side of several GATT Procedures. Specifically, read, write and notify. Moreover, in those chapters you learned how to create firmware to respond to those requests. You will recall that each of those GATT procedures are mapped into one or more attribute requests. Here is a list of all the attribute requests with the original request and the applicable response.

Note that the request can be initiated by either the client or server depending on the operation. For example, a write is initiated by the GATT client while a notification is initiated by the GATT server.

Bluetooth® Spec * Chap Ref	Request	Request Data	Bluetooth® Spec * Chap Ref	Response	Response Data
			3.4.1.1 N/A	Error Response	Request Op Code in Error Attribute Handle in Error Error Code
3.4.2.1 N/A	Exchange MTU	Client Rx MTU	3.4.2.2 N/A	Exchange MTU response	Server Rx MTU
3.4.3.1 6.5.9	Find Information	Starting Handle Ending Handle	3.4.3.2 6.5.9	Find Information Response	Handles Attribute Type UUIDs
3.4.3.3 6.5.7	Find by Type Value	Starting Handle Ending Handle Attribute Type Attribute Value	3.4.3.4 4D.4.60	Find by Type Value Response	Start Handle End of Group Handle
3.4.4.1 0	Read by Type	Starting Handle Ending Handle Attribute Type UUID	3.4.4.2 0	Read by Type Response	Handle Value Pairs
3.4.4.3 6.5.1	Read	Handle	3.4.4.4 4A	Read Response	Handle, Value
3.4.4.5 N/A	Read Blob	Handle, Offset	3.4.4.6 N/A	Read Blob Response	Handle, Data
3.4.4.7 N/A	Read Multiple	Handles	3.4.4.8 N/A	Read Multiple Response	One response with all values concatenated
3.4.4.9 6.5.6	Read by Group Type	Starting Handle Ending Handle Attribute Group Type UUID	3.4.4.10 6.5.6	Read by Group Type Response	For each match: Handle, Value Handle of last Attribute in Group
3.4.5.1 6.5.3	Write	Handle, Value	3.4.5.2 4A	Write Response	Response code 0x1E Or Error Response
3.4.5.3 6.5.3	Write Command	Handle, Value	There is no Server response to a Write.		
3.4.5.4 N/A	Signed Write Command	Handle, Value, Signature	3.4.5.2 4A	Write Response	Response code 0x1E Or Error Response

Bluetooth® Spec * Chap Ref	Request	Request Data	Bluetooth® Spec * Chap Ref	Response	Response Data
3.4.6.1 N/A	Prepare Write	Handle Offset Value	3.4.6.2 N/A	Prepare Write Response	Handle Offset Value
3.4.6.3 N/A	Execute Write	Flags (0-cancel, 1- write)	3.4.6.4 N/A	Execute Write Response	Response code 0x19 Or Error Response
3.4.7.1 4B	Notification	Handle, Value	There is no Client response to a Notification, but you can read about what happens on the Client side in 0		
3.4.7.2 4B	Indication	Handle, Value	3.4.7.3 0	Handle Value Confirmation	Response code 0x1E Or Error Response

* Bluetooth® spec references are from Volume 3, Part F (Attribute Protocol). Additional details on the GATT procedures that use the ATT protocols can be found in Volume 3, Part G (Generic Attribute Profile).

This leads us to some obvious questions: What happens with read, write and notify on the GATT client side of a connection? And what about these other operations?

6.5.1 GATT utility functions

There is a library of functions available to simplify setting up and performing various GATT operations such as reading characteristic values, setting characteristic descriptor values and doing service discovery. To use these functions, you need to include the GATT utilities to your application. The steps are:

1. Open the library manager and include the "Bluetooth® Low Energy" library. Click **Update**. Once the update is finished, click **Close**.
2. In the *makefile*, add a component entry to include the GATT utilities portion of the library:

```
COMPONENTS += gatt_utils_lib
```

3. In the source code, include the header file for the GATT utilities:

```
#include "wiced_bt_gatt_util.h"
```

Note: The library source code is in *mtb_shared/wiced_btsdk/dev-kit/libraries/btsdk-ble/<version>/COMPONENT_gatt_utils_lib*. The header file is in *mtb_shared/wiced_btsdk/dev-kit/baselib/<device>/<version>/COMPONENT_<device>/include*.

6.5.2 GATT client read

To initiate a read of the value of a Characteristic you need to know two things, the Handle of the Characteristic and the connection ID. To execute a read you just call:

```
wiced_bt_util_send_gatt_read_by_handle( conn_id, handle );
```

This function call will cause the Stack to send a read request to the GATT Server. After some time, you will get a callback in your GATT event handler with the event code `GATT_OPERATION_CPLT_EVENT`. The callback

parameter can then be cast into a `wiced_bt_gatt_operation_complete_t`. This structure has everything you need. Specifically:

```
/** Response to read/write/disc/config operations (used by GATT_OPERATION_CPLT_EVT notification) */
typedef struct
{
    uint16_t          conn_id;          /**< ID of the connection */
    wiced_bt_gatt_optype_t op;          /**< Type of operation completed */
    wiced_bt_gatt_status_t status;      /**< Status of operation */
    wiced_bt_gatt_operation_complete_rsp_t response_data; /**< Response data */
} wiced_bt_gatt_operation_complete_t;
```

This same event is used to handle many of the responses from a GATT Server. Exactly which response can be determined by the `wiced_bt_gatt_optype_t` which is just an enumeration of operations. For instance, 0x02 means you are getting the response from a read.

```
enum wiced_bt_gatt_optype_e
{
    GATTC_OPTYPE_NONE           = 0,    /**< None */
    GATTC_OPTYPE_DISCOVERY      = 1,    /**< Discovery */
    GATTC_OPTYPE_READ           = 2,    /**< Read */
    GATTC_OPTYPE_WRITE          = 3,    /**< Write */
    GATTC_OPTYPE_EXE_WRITE      = 4,    /**< Execute Write */
    GATTC_OPTYPE_CONFIG         = 5,    /**< Configure */
    GATTC_OPTYPE_NOTIFICATION   = 6,    /**< Notification */
    GATTC_OPTYPE_INDICATION     = 7,    /**< Indication */
};
```

The `wiced_bt_gatt_status_t` is an enumeration of different error codes from the enumeration `wiced_bt_gatt_status_e` which was introduced previously. As a reminder, the first few values are:

```
enum wiced_bt_gatt_status_e
{
    WICED_BT_GATT_SUCCESS           = 0x00,    /**< Success */
    WICED_BT_GATT_INVALID_HANDLE    = 0x01,    /**< Invalid Handle */
    WICED_BT_GATT_READ_NOT_PERMIT   = 0x02,    /**< Read Not Permitted */
    WICED_BT_GATT_WRITE_NOT_PERMIT  = 0x03,    /**< Write Not permitted */
    WICED_BT_GATT_INVALID_PDU       = 0x04,    /**< Invalid PDU */
};
```

The last piece is the response data, `wiced_bt_gatt_operation_complete_rsp_t` which contains the handle and an attribute value structure which contains the actual GATT data along with information about it.

```
typedef union
{
    wiced_bt_gatt_data_t att_value; /**< Response data for read operations
                                     (initiated using #wiced_bt_gatt_send_read) */
    uint16_t mtu; /**< Response data for configuration
                  operations */
    uint16_t handle; /**< Response data for write operations
                     (initiated using #wiced_bt_gatt_send_write) */
} wiced_bt_gatt_operation_complete_rsp_t; /**< GATT operation complete response type */
```

The attribute value structure containing the GATT data looks like this:

```
/** Response data for read operations */
typedef struct
{
    uint16_t handle; /**< handle */
    uint16_t len; /**< length of response data */
    uint16_t offset; /**< offset */
    uint8_t *p_data; /**< attribute data */
} wiced_bt_gatt_data_t;
```

These structures look a bit overwhelming, but you can easily use this information to find out what happened. Here is an example of how you might deal with this callback to print out the response and all the raw bytes of the data that were sent.

```
case GATT_OPERATION_CPLT_EVT:

    // When you get something back from the peripheral... print it out.. the data
    WICED_BT_TRACE("Event Complete Conn=%d Op=%d status=0x%X Handle=0x%X len=%d Data=",
        p_data->operation_complete.conn_id,
        p_data->operation_complete.op,
        p_data->operation_complete.status,
        p_data->operation_complete.response_data.handle,
        p_data->operation_complete.response_data.att_value.len);

    for(int i=0;i<p_data->operation_complete.response_data.att_value.len;i++)
    {
        WICED_BT_TRACE("%02X ",
            p_data->operation_complete.response_data.att_value.p_data[i]);
    }
    WICED_BT_TRACE("\r\n");
    break;
```

6.5.3 GATT client write and write command

In order to send a GATT write, all you need to do is make a structure of type `wiced_bt_gatt_value_t`, set up the handle, offset, length, authorization and value; then call `wiced_bt_gatt_send_write`. Here is an example:

```
wiced_bt_gatt_value_t *p_write = ( wiced_bt_gatt_value_t* )wiced_bt_get_buffer( sizeof(
wiced_bt_gatt_value_t ) + sizeof(myData)-1);
if ( p_write )
{
    p_write->handle = myHandle;
    p_write->offset = 0;
    p_write->len = sizeof(myData);
    p_write->auth_req = GATT_AUTH_REQ_NONE;
    memcpy(p_write->value, &myData, sizeof(myData));

    wiced_bt_gatt_send_write ( conn_id, GATT_WRITE, p_write );

    wiced_bt_free_buffer( p_write );
}
```

The second argument to `wiced_bt_gatt_send_write` function can either be `GATT_WRITE` or `GATT_WRITE_NO_RSP` depending on whether you want a response from the server or not.

The one trick is that the length of the structure is variable with the length of the data. In the above example, you allocate a block big enough to hold the structure + the length of the data minus 1 using `wiced_bt_get_buffer`. Then use a pointer to fill in the data. The buffer is freed up once the write is done.

Note that the `wiced_bt_get_buffer` and `wiced_bt_free_buffer` functions require that you include `wiced_memory.h` in the C file.

If you asked for a write with response (i.e. `GATT_WRITE`), sometime after you call the `wiced_bt_gatt_send_write` you will get a GATT callback with the event code `GATT_OPERATION_CPLT_EVT`. You can then figure out if the write was successful by checking to see if you got a Write Response or an Error Response (with the error code from the `wiced_bt_gatt_status_e` enumeration).

Don't forget the you can only issue one Read/Write at a time and that you cannot send the next Read/Write until the last one is finished.

6.5.4 GATT client notify and indicate

To register for Notifications and Indications, the Client just needs to write to the CCCD for the Characteristic of interest. There is a utility function that simplifies the process of writing the Descriptor called `wiced_bt_util_set_gatt_client_config_descriptor`. The function takes three arguments:

1. The connection ID
2. The handle of the Descriptor
3. The value to write

For a CCCD, the LSB (bit 0) is set to 1 for Notifications, and bit 1 is set to 1 for Indications. That is:

```
/** characteristic descriptor: client configuration value */
enum wiced_bt_gatt_client_char_config_e
{
    GATT_CLIENT_CONFIG_NONE           = 0x0000,    /**< No notifications or indications */
    GATT_CLIENT_CONFIG_NOTIFICATION = 0x0001,    /**< Send notifications */
    GATT_CLIENT_CONFIG_INDICATION   = 0x0002     /**< Send indications */
};
```

When the GATT Server initiates a Notify or an Indicate you will get a GATT callback with the event code set as `GATT_OPERATION_CPLT_EVT`. You will see that the Operation value is `GATTC_OPTYPE_NOTIFICATION` or `GATTC_OPTYPE_INDICATE` and the value is just like the Read Response. In the case of Indicate you can return `WICED_BT_GATT_SUCCESS` which means the Stack will return a Handle Value Confirmation to the Client, or you can return something other than `WICED_BT_GATT_SUCCESS` in which case the Stack sends an Error Response with the code that you choose from the `wiced_bt_gatt_status_e` enumeration. Remember from Chapter 4A that the (partial) list of available return values is:

```
enum wiced_bt_gatt_status_e
{
    WICED_BT_GATT_SUCCESS           = 0x00,    /**< Success */
    WICED_BT_GATT_INVALID_HANDLE    = 0x01,    /**< Invalid Handle */
    WICED_BT_GATT_READ_NOT_PERMIT    = 0x02,    /**< Read Not Permitted */
    WICED_BT_GATT_WRITE_NOT_PERMIT   = 0x03,    /**< Write Not permitted */
    WICED_BT_GATT_INVALID_PDU       = 0x04,    /**< Invalid PDU */
    WICED_BT_GATT_INSUF_AUTHENTICATION = 0x05,    /**< Insufficient Authentication */
    WICED_BT_GATT_REQ_NOT_SUPPORTED   = 0x06,    /**< Request Not Supported */
    WICED_BT_GATT_INVALID_OFFSET     = 0x07,    /**< Invalid Offset */
    WICED_BT_GATT_INSUF_AUTHORIZATION = 0x08,    /**< Insufficient Authorization */
    WICED_BT_GATT_PREPARE_Q_FULL     = 0x09,    /**< Prepare Queue Full */
    WICED_BT_GATT_NOT_FOUND          = 0x0a,    /**< Not Found */
    WICED_BT_GATT_NOT_LONG           = 0x0b,    /**< Not Long Size */
    WICED_BT_GATT_INSUF_KEY_SIZE     = 0x0c,    /**< Insufficient Key Size */
    WICED_BT_GATT_INVALID_ATTR_LEN   = 0x0d,    /**< Invalid Attribute Length */
    WICED_BT_GATT_ERR_UNLIKELY       = 0x0e,    /**< Error Unlikely */
    WICED_BT_GATT_INSUF_ENCRYPTION   = 0x0f,    /**< Insufficient Encryption */
    WICED_BT_GATT_UNSUPPORTED_GRP_TYPE = 0x10,    /**< Unsupported Group Type */
    WICED_BT_GATT_INSUF_RESOURCE     = 0x11,    /**< Insufficient Resource */
};
```

6.5.5 GATT group

There is one last GATT concept that needs to be introduced to understand the next set of GATT procedures. That is the group. A group is a range of handles starting at a service, service include or a characteristic, and ending at the last handle that is associated with the group. To put it another way, a group is all of the rows in the GATT database that logically belong together. For instance, in the GATT database below, the service group for <<Generic Access>> starts at handle 0x0001 and ends at 0x0005.

6.5.6 GATT client read by group type

The GATT client "Read by Group Type" request takes as input a search starting handle, search ending handle and group type. It outputs a list of tuples with the group start handle, group end handle, and value. This request can only be used for a "Grouping Type" meaning, <<Service>>, <<Included Service>> and <<Characteristic>>.

Note that anything inside double angle brackets << >> indicates a Bluetooth® SIG defined UUID. For example, <<Primary Service>> is defined by the SIG to be 0x2800.

Consider this GATT database for a peripheral with a service called MySvc that contains two characteristics; LED and Counter (which happens to be the application you will control from your central device in this chapter's exercises).

Handle	Type	Value
0x01	<<Primary Service>>	<<Generic Access>>
0x02	<<Characteristic>>	0x02, 0x03, <<Device Name>>
0x03	<<Device Name>>	key_per
0x04	<<Characteristic>>	0x02, 0x05, <<Appearance>>
0x05	<<Appearance>>	0x00
0x06	<<Primary Service>>	<<Generic Attribute>>
0x07	<<Primary Service>>	__UUID_SERVICE_MYSVC
0x08	<<Characteristic>>	0x0A, 0x09, __UUID_CHARACTERISTIC_MYSVC_LED
0x09	__UUID_CHARACTERISTIC_MYSVC_LED	RGB LED value
0x0A	<<Characteristic>>	0x1A, 0x0B, __UUID_CHARACTERISTIC_MYSVC_COUNTER
0x0B	__UUID_CHARACTERISTIC_MYSVC_COUNTER	Button Count Value
0x0C	<<CCCD>>	0x0000 (notify off) or 0x0001 (notify on)

Remember that the type and values for different attributes are:

Attribute	Type	Value
Service	<<Primary Service>> or <<Secondary Service>>	Service UUID
Characteristic	<<Characteristic>>	Properties, Handle to the Value, Characteristic UUID
Characteristic Value	Characteristic UUID	Characteristic Value

In the database above if you input search start handle=0x0001, search end handle=0xFFFF and group type = <<Service>> you would get as output:

Group Start Handle	Group End Handle	UUID
0x0001	0x0005	<<Generic Access>>
0x0006	0x0006	<<Generic Attribute>>
0x0007	0x000C	__UUID_SERVICE_MYSVC

In words, you receive a list of all the service UUIDs with the start handle and end handle of each service group.

6.5.7 GATT client find by type value

The GATT client "Find by Type Value" request takes as input the search starting handle, search ending handle, attribute type and attribute value. It then searches the attribute database and returns the starting and ending handles of the group that match that attribute type and attribute value. This function was put into GATT specifically to find the range of handles for a specific service.

Consider the example above. If your input to the "Find by Type Value" was starting handle 0x0001, ending handle 0xFFFF, type <<Service>> and value __UUID_SERVICE_MYSVC, the output would be:

Group Start Handle	Group End Handle
0x0007	0x000C

This function cannot be used to search for a specific characteristic because the attribute value of a characteristic declaration cannot be known a-priori. This is because the characteristic properties and characteristic value attribute handle (which are part of the attribute value) are not known up front. As a reminder, here is the characteristic declaration:

Attribute Handle	Attribute Types	Attribute Value			Attribute Permissions
0xNNNN	0x2803—UUID for «Characteristic»	Characteristic Properties	Characteristic Value Attribute Handle	Characteristic UUID	Read Only, No Authentication, No Authorization

6.5.8 GATT client read by type

The GATT client "Read by Type" request takes as input the search starting handle, search ending handle and attribute type. It outputs a list of handle value pairs. In the example above if you entered the search starting handle 0x0007, search ending handle 0x000C and type <<Characteristic>>, you would get as output:

Characteristic Handle	Value Handle	UUID	GATT Permission
0x08	0x09	__UUID_CHARACTERISTIC_MYSVC_LED	0x0A
0x0A	0x0B	__UUID_CHARACTERISTIC_MYSVC_COUNTER	0x0A

In words, you get back a list of the characteristic handles, the handles of the values, the UUIDs, and the GATT permissions.

6.5.9 GATT client find information

The input to the GATT client "Find Information" request is simply a search starting handle and a search ending handle. The GATT server responds with a list of every handle in that range, and the attribute type of the handle. Notice that this is the only GATT procedure that returns the attribute type.

If you execute a GATT client "Find Information" with the handle range set to $0x0005 > 0x0005$ you will get a response of:

Handle	Attribute Type
0x0005	<<Appearance>>

In words, the attribute type that is associated with the characteristic handle $0x0005$.

6.6 Service discovery

Given that all transactions between the GATT client and GATT server use the handle instead of the UUID, one huge question left unanswered is how do you find the handles for the different services, characteristics and descriptors on the GATT server?

A very, very bad answer to that question is that you hard-code the handles into the GATT client (although some devices with custom applications do just that).

A much better answer is that you do service discovery. The phrase service discovery includes discovering all the attributes of a device including services, characteristics and descriptors. This is done using the GATT procedures that were introduced in sections just above: "Read by Group Type", "Find by Type Value", "Read by Type" and "Find Information".

6.6.1 Service discovery algorithm

The steps in the service discovery algorithm are:

1. Do one of the following:
 - a. Discover all the services using "Read by Group Type" which gives you the UUID and start and end handles of all the service groups.
 - b. Discover one specific service by using "Find by Type Value" which gives you the start and end handles of the specified service group.
2. For each service group discover all the characteristics using "Read by Type" with the handle range of the service group or groups that you discovered in step (1). This gives you the characteristic handles, characteristic value handles, UUIDs, and permissions of each characteristic.
3. Using the characteristic handles from (2) you can then calculate the start and end handle ranges of each of the descriptors for each characteristic.
 - a. The range for a given characteristic starts at the next handle after the characteristic's value handle and ends either at the end of the service group (if it's the last characteristic in the service group) or just before the next characteristic handle (if it isn't the last characteristic in the service group).
4. Using the ranges from (3) discover the descriptors using the GATT procedure "Find Information". This gives you the attribute type of each descriptor.

6.6.2 Service discovery implementation

The *wiced_bt_sdk* library has a service discovery API that can discover services, characteristics and descriptors:

```
wiced_bt_gatt_status_t wiced_bt_gatt_send_discover (uint16_t conn_id,
                                                    wiced_bt_gatt_discovery_type_t discovery_type,
                                                    wiced_bt_gatt_discovery_param_t *p_discovery_param );
```

The discovery type is an enumeration (note that `GATT_DISCOVER_MAX` is not a legal parameter):

```
enum wiced_bt_gatt_discovery_type_e
{
    GATT_DISCOVER_SERVICES_ALL = 1,           /**< discover all services */
    GATT_DISCOVER_SERVICES_BY_UUID,          /**< discover service by UUID */
    GATT_DISCOVER_INCLUDED_SERVICES,         /**< discover an included svc within a svc */
    GATT_DISCOVER_CHARACTERISTICS,           /**< discover characteristics of a service*/
    GATT_DISCOVER_CHARACTERISTIC_DESCRIPTOR, /**< discover characteristic descriptors */
    GATT_DISCOVER_MAX                        /* maximum discovery types */
};
```

The discovery parameter contains:

```
typedef struct
{
    wiced_bt_uuid_t uuid;           /**< Service or Characteristic UUID */
    uint16_t s_handle;              /**< Start handle for range to search */
    uint16_t e_handle;              /**< End handle for range to search */
} wiced_bt_gatt_discovery_param_t;
```

The UUID entry in the discovery parameter structure is itself a structure that allows you to specify a 2, 4, or 16-byte UUID (i.e. 16, 32, or 128 bits):

```
/** UUID Type */
typedef struct
{
    #define LEN_UUID_16      2    /**< 2 Byte UUID */
    #define LEN_UUID_32      4    /**< 4 Byte UUID */
    #define LEN_UUID_128     16   /**< 16 Byte UUID */

    uint8_t len;                /**< UUID length */

    /** UUID Data */
    union
    {
        uint16_t uuid16; /**< 16-bit UUID */
        uint32_t uuid32; /**< 32-bit UUID */
        uint8_t  uuid128[LEN_UUID_128]; /**< 128-bit UUID */
    } uu;
} wiced_bt_uuid_t;
```

After you call `wiced_bt_gatt_send_discover`, the stack will issue the correct GATT Procedure. Then, each time the GATT server responds with some information you will get a GATT callback with the event type set to `GATT_DISCOVERY_RESULT_EVT`.

The result is provided in the `discovery_result` entry in the event data structure. It contains the connection ID, type of discovery that the result came from, and the data itself. You can work your way through this structure to find out what you need to know about the server. For example, to find the service start and end handles after running `GATT_DISCOVER_SERVICES_BY_UUID` you could do this:

```
if( p_event_data->discovery_result.discovery_type == GATT_DISCOVER_SERVICES_BY_UUID )
{
    serviceStartHandle = p_event_data->discovery_result.discovery_data.group_value.s_handle;
    serviceEndHandle = p_event_data->discovery_result.discovery_data.group_value.e_handle;
}
```

When the discovery is complete you will get one more GATT callbacks with the event type set to `GATT_DISCOVERY_CPLT_EVT`.

Your firmware would typically be a state machine that would sequence through the service, characteristic and descriptor discoveries as the `GATT_DISCOVERY_CPLT_EVT` is completed.

6.7 Running a GATT Server

Although somewhat uncommon, there is no reason why a Bluetooth® LE central cannot run a GATT server. In other words, all combinations of GAP peripheral/central and GATT server/client are legal. An example of this might be a TV that has a Bluetooth® LE remote control. Recall that the device that needs to save power is always the peripheral, in this case the remote control. However, the TV is the thing being controlled, so it would have the GATT database remembering things like channel, volume, etc.

The firmware that you write on a central to run a GATT database is EXACTLY the same as on a peripheral.

6.8 Exercises

The exercises in this chapter require two kits – a peripheral and a central. You can use another CYW920835M2EVB-01 as the peripheral.

The peripheral application is the pairing exercise from the previous chapter with a couple minor modifications: (1) it advertises the device name and the service UUID; and (2) advertising never times out. As you may recall, the LED can be turned on and off by writing to a characteristic that does not require authentication. The Counter characteristic, which is incremented by pressing the user button on the peripheral, requires pairing before the central can read it or enable notifications.

Exercise 1: Make an observer

In this exercise you will first program the peripheral and then you will build an observer that will listen to all the Bluetooth® LE devices that are advertising and will print out the BDA of each device that it finds.

Create/Test peripheral



1. Make sure only the kit that you intend to use as a peripheral is plugged into your computer.



2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

Use the Import functionality in project creator to start from the template found in *Templates/ch06_ex01_peripheral*. Keep the same name.



3. Use the Bluetooth® Configurator to change the device name to `<init>_per`.

Note: Because the advertising packet is limited to 31 bytes, the name you use must be 8 characters or less.



4. Review the GATT properties and permissions for the characteristics and CCCD for this application. This will be important when we read/write the values in later exercises. They are:

LED: read and write authentication are not required, write and write without response are both supported

COUNTER: read authentication is required

CCCD: read and write authentication are required, write is supported, but write without response is not supported



5. Open a UART terminal to the kit if it isn't already open, then build the application and program it to the kit.



6. Write down the device's address. You will need to know it later.



7. Use CySmart to test the application:

- i. Write 1 and 0 to the LED characteristic to turn the LED on and off.

- ii. If using the PC version of CySmart, pair the device.

- iii. Enable notifications and press the button on the kit to receive notifications. Allow pairing when requested.



4. Disconnect from the kit and close the UART window.



5. Unplug the peripheral kit so that you don't accidentally re-program it with the central application.

Note: You should not need to re-program this kit again and so, if possible, plug it into a USB charger instead of your computer.

Create/Test observer



1. Plug the kit that you are going to use for the observer into your computer.



2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

Use the Import functionality in project creator to start from the template found in *Templates/ch06_ex01_observer*. Keep the same name.



3. Open *app_bt_cfg.c*. Find the GATT configuration and change the following to allow the client to make one connection to a server:



```
.client_max_links = 1,  
.server_max_links = 0,
```



4. Review the *.scan_mode* settings to understand how scanning is configured.



5. In *app.c* create a function declaration and a callback function to process the scanned advertising packets. The function should look like this:

```
void scanCallback(wiced_bt_ble_scan_results_t *p_scan_result, uint8_t *p_adv_data)  
{  
    printf("Host = [%B]\n", p_scan_result->remote_bd_addr);  
}
```

Note: The template has TODO comments where changes are required.



6. Add a call to *wiced_bt_ble_scan* in the *BTM_ENABLED* event with a function pointer to the callback function you created in the previous step.

Enable filtering so that each new device only shows up once – otherwise you will see a LOT of packets from all the devices out there.



7. Open a UART terminal, then build the application and program it to the kit.



8. Once the application starts, you should see a list of devices that are advertising. When high duty scanning times out, you will get a similar list for low duty scanning. Scanning will stop after the low duty timeout.

Look for your peripheral's device address in the list to make sure your central sees it.

```
**** App Start ****  
Bluetooth Management Event: 0x15 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT  
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT  
Local Bluetooth Device Address: [20 81 9a 12 1b d8 ]  
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT  
High duty scanning.  
Host = [fa 0d ac e3 ed 79 ]  
Host = [20 81 9a 11 40 b7 ]  
Host = [64 16 66 74 07 48 ]  
Host = [6f a7 f7 23 44 e0 ]  
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT  
Low duty scanning.  
Host = [fa 0d ac e3 ed 79 ]  
Host = [20 81 9a 11 40 b7 ]  
Host = [00 04 20 00 c1 03 ]  
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT  
Scanning stopped.
```

Exercise 2: Parse the device name and list only your peripheral

In this exercise you will extend the previous exercise so that it examines the advertising packet to find the device name. It will only print out the address of your device instead of all devices that are found during the scan.

- ☐ 1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.
- ☐ 2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

Use the Import functionality in project creator to start from the previous completed exercise. If you did not complete that exercise, the solution can be found in *Projects/key_ch06_ex01_observer*.

Name the new application **ch06_ex02_mydev**.

- ☐ 3. Update the scanner callback function so that it looks at the advertising packet. Find and print out the device name and address only for devices that match your peripheral's device Name.

Note: Use the function `wiced_bt_ble_check_advertising_data` to look at the advertising packet and find fields of type `BTM_BLE_ADVERT_TYPE_NAME_COMPLETE`. If there is a field that matches it will return a pointer to those bytes and a length.

Note: Use `memcmp` to see if the field contents match the name you are looking for.

- ☐ 4. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 5. Plug your peripheral back in (if necessary).
- ☐ 6. On the UART terminal for the central, you should see a result just for your device.

```
**** App Start ****
Bluetooth Management Event: 0x15 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Local Bluetooth Device Address: [20 81 9a 12 6b 8b ]
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT
High duty scanning.
Found Device "key_per" with BD Address: [20 81 9a 11 40 b7 ]
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT
Low duty scanning.
Found Device "key_per" with BD Address: [20 81 9a 11 40 b7 ]
Bluetooth Management Event: 0x16 BTM_BLE_SCAN_STATE_CHANGED_EVT
Scanning stopped.
```

Exercise 3: Connect to your peripheral and turn LED ON/OFF

In this exercise, you will modify the previous exercise to connect to your peripheral once it finds it. We will decide the device to which to connect based on your peripheral's device name. In real-world applications, it would be more common to search for a service UUID or manufacturer data, depending on what the peripheral provides.

Once connected, you will be able to send values to the LED characteristic to turn the LED off/on. You will also be able to read the current state of the LED from the peripheral.

To simplify the application, you will hard-code the handle, but in the upcoming exercises you will add service discovery.

Since there are a lot of additions to be made, we will do them in 4 parts and will test each one along the way.

Use UART interface to start/stop scanning

- ☐ 1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.
- ☐ 2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.
- ☐ Use the Import functionality in project creator to start from the previous completed exercise. If you did not complete that exercise, the solution can be found in *Projects/key_ch06_ex02_mydev*. Name the new application **ch06_ex03_connect**.
- ☐ 3. The template that you started with already has a UART with a receive interrupt to handle character input. Change the value for `#define UART_ENABLE` in *app.c* to `WICED_TRUE` to enable it.
- ☐ 4. Review the interrupt service routine (`rx_cback`) to understand how it works.
- ☐ 5. Update the UART task to start scanning (lower case **s**) and stop scanning (upper case **S**).
- ☐ 6. Remove the call to `wiced_bt_ble_scan` from the `BTM_ENABLED_EVT` event.
- ☐ 7. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 8. Plug your peripheral back in (if necessary).
- ☐ 9. Verify that the keyboard interface works by typing **s** in the UART terminal to start scanning and then **S** to stop scanning. Type **?** to see the help message.

Add GATT connect/disconnect functionality

- ☐ 1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.
- ☐ 2. Update your scan callback function to connect to the Peripheral when it finds one that it recognizes by calling `wiced_bt_gatt_le_connect`.

Note: Use the Device Name of your peripheral here so that it will only connect to your device.

- ☐ 3. After making the connection, turn off scanning.
- ☐ 4. Add a **d** command to the UART interface to call the disconnect function.

Note: Remember that the disconnect function does not have "le" in its name.

Note: Don't forget to add the new command to the help message.

- ☐ 5. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 6. Plug your peripheral back in (if necessary).
- ☐ 7. Test the following:
 - a. Start scanning and connect (**s**).
 - b. Verify that the peripheral is found and the central connects to it.
 - c. Disconnect (**d**).

Add LED control functionality

- ☐ 1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.
- ☐ 2. Add an include for the `wiced_memory.h` file.
- ☐ 3. Create a new global `uint16_t` variable called `ledHandle`.

Hard-code its initial value to the handle of the LED characteristic's Value from the peripheral. You won't change the variable in this exercise, but in a future one you will find the handle via a service discovery.

Note: Make sure you use the handle for the characteristic's value, not the characteristic declaration.

Note: The easiest way to find the handle number is to open the file `GeneratedSource/cycfg_gatt_db.h` from the peripheral application. The characteristic handles have the prefix `HDLC`.

- ☐ 4. Create a new function called `writeAttribute` to write a GATT attribute. It will take the connection ID, handle, offset, authorization required, length of the data to write, and a pointer to the value to write.

If there is no connection or the handle is 0 then the function should just return. Otherwise, set up the write parameter structure and then call `wiced_bt_gatt_send_write` to send the value to the peripheral.

Note: For now, the handle will never be 0 since we hard-coded it, but this will be useful once we add service discovery.

Note: The `GATT_OPERATON_CPLT_EVT` already prints out a message when a GATT operation is successful. For simplicity, we won't do any other checking specific to write responses.

- ☐ 5. In the UART interface, call the `writeAttribute` function you created when 0 and 1 are entered.

Note: The case is already provided for you, but don't forget to add a help message.

Note: Since we are not pairing yet, use `GATT_AUTH_REQ_NONE` for the authorization required. Recall that the LED characteristic on the server does not require authentication for read or write so this will work.

- ☐ 6. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 7. Plug your peripheral back in (if necessary).



8. Test the following:

- a. Start scanning and connect (**s**).
- b. Verify that the peripheral is found and the central connects to it.
- c. Turn the LED off and on by pressing **0** and **1**. Look at the peripheral kit to see if the LED turns off and on correctly.
- d. Disconnect (**d**).

Add LED read functionality



1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.



2. Open the *makefile* and add `gatt_utils_lib` to the `COMPONENTS` variable.



3. Add an include for the *wiced_bt_gatt_util.h* file.

Note: The function `wiced_bt_util_send_gatt_read_by_handle` is in the GATT utilities library.



4. Create a new global `uint16_t` variable called `ledStatus`.



5. Add a UART command (**r**) to read the value of the LED from the peripheral and print it to the terminal.

Note: Use `GATT_AUTH_REQ_NONE` as the required authorization so that the value can be read without pairing first.

Note: Don't forget to add the new command to the help message.



6. The GATT event callback function already has a case for `GATT_OPERATON_CPLT_EVT` and it checks if the command was successful or not. Add nested `if` statements to check the operation type and the handle whose value was read and print out the value of the LED if the event was a read of that characteristic.



7. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.



8. Plug your peripheral back in (if necessary).



9. Test the following:

- a. Start scanning and connect (**s**).
- b. Verify that the peripheral is found and the central connects to it.
- c. Turn the LED off and by pressing **0** and **1**. Verify each value by viewing the LED on the peripheral kit and by typing **r** after writing to it.
- d. Disconnect (**d**).

Exercise 4: Add commands to enable/disable notifications

In this exercise, you will enhance the previous exercise to allow the CCCD to be turned on/off so that notifications for the Counter characteristic can be enabled/disabled. You will print out messages when notifications are received.

- ☐ 1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.
- ☐ 2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.
- ☐ 3. Use the Import functionality in project creator to start from the previous completed exercise. If you did not complete that exercise, the solution can be found in *Projects/key_ch06_ex03_connect*. Name the new application **ch06_ex04_connect_notify**.
- ☐ 4. In the GATT connect event handler, initiate pairing by calling `wiced_bt_dev_sec_bond` for the connected case.

Note: This is necessary because the CCCD permissions for your peripheral are set such that it cannot be read or written unless the devices are paired first.

Note: This must be done after the connection is established so that's why it's inside the GATT connect event handler.

- ☐ 5. Add a new `uint16_t` global variable called `counterCccdHandle` to hold the handle of the CCCD for the Counter characteristic.

Hard-code the initial value to the handle from the peripheral. You won't change the variable in this exercise, but in a future one you will find the handle via a service discovery.

Note: The easiest way to find the handle number is to open the file *GeneratedSource/cycfg_gatt_db.h* from the peripheral application. The characteristic descriptor handles have the prefix `HDLD`.

- ☐ 6. Add cases in the UART to set (n) and unset (N) to set and unset the CCCD using the `wiced_bt_util_set_gatt_client_config_descriptor` function.
- ☐ 7. In the `GATT_OPERATION_CPLT_EVT`, if the operation was successful, check to see if the operation was `GATTC_OPTYPE_NOTIFICATION`. If it was, print out the notification value that was received.

Note: The operation type can be found in:
`p_event_data->operation_complete.op`

Note: The data is provided as a `uint8_t` pointer in:
`p_event_data->operation_complete.response_data.att_value.p_data`
The length of the data is provided in:
`p_event_data->operation_complete.response_data.att_value.len`

Note: You can use the `WICED_BT_TRACE_ARRAY` function to easily print the value.

- ☐ 8. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 9. Plug your peripheral back in (if necessary).
- ☐ 10. Test the following:
 - a. Start Scanning and connect (s)

- b. Verify that the connection is made
- c. Press the button on the peripheral. Notifications are not enabled so you should not see a response.
- d. Enable Notifications (**n**)
- e. Press the button on the peripheral. You should see a response in the UART.
- f. Disable Notifications (**N**)
- g. Press the button on the peripheral. You should not see a response.
- h. Disconnect (**d**)

Exercise 5: Implement service discovery

In this exercise, instead of hardcoding the handles for the LED and Counter CCCD, you will modify the previous exercise to do a service discovery. Instead of triggering the whole process with a state machine, you will use keyboard commands to launch each stage.

The three stages are:

- 'q' = Service discovery with the UUID of the service to get the start and end handles for the service group.
- 'w' = Characteristic discovery with the range of handles from step 1 to discover all characteristic handles and characteristic value handles.
- 'e' = Descriptor discovery of the Counter characteristic to find the CCCD handle.



1. If your peripheral is plugged into your computer, unplug it so that you don't accidentally re-program it.



2. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.



3. Use the Import functionality in project creator to start from the previous completed exercise. If you did not complete that exercise, the solution can be found in *Projects/key_ch06_ex04_connect_notify*. Name the new application **ch06_ex05_discover**.



4. Open the file *GeneratedSource/cycfg_gatt_db.h* from the peripheral application and copy over the macros for the following into the *app.c* file for use in this exercise:

```
__UUID_SERVICE_MYSVC
__UUID_CHARACTERISTIC_MYSVC_LED
__UUID_CHARACTERISTIC_MYSVC_COUNTER
__UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION
```

Note: This guarantees that the Central uses the same UUIDs that are used in the Peripheral.

Global Variables



5. Create variables to save the start and end handles of the MySvc service group and create a variable to hold the MySvc service UUID.

```
static const uint8_t serviceUUID[] = { __UUID_SERVICE_MYSVC };
static uint16_t serviceStartHandle = 0x0001;
static uint16_t serviceEndHandle = 0xFFFF;
```



6. Define a new structure type to manage the discovered handles of the characteristics:

```
typedef struct {
    uint16_t startHandle;
    uint16_t endHandle;
    uint16_t valHandle;
    uint16_t cccdHandle;
} charHandle_t;
```



7. Create a `charHandle_t` structure for the LED and the Counter characteristic groups and variables with the characteristic UUIDs to search for.

```
static const uint8_t ledUUID[] = { __UUID_CHARACTERISTIC_MYSVC_LED };
static charHandle_t ledChar;
static const uint8_t counterUUID[] = { __UUID_CHARACTERISTIC_MYSVC_COUNTER };
static charHandle_t counterChar;
```

- ☐ 8. Create an array of `charHandle_t` to temporarily hold the characteristic handles while they are discovered.

```
#define MAX_CHARS_DISCOVERED (10)
static charHandle_t charHandles[MAX_CHARS_DISCOVERED];
static uint32_t charHandleCount;
```

Note: When you discover the characteristics, you won't know what order they will occur in, so you need to save the handles temporarily to calculate the end of group handles.

Service Discovery (q)

- ☐ 9. Create a function to launch the service discovery called `startServiceDiscovery`.

This function will be called when the user presses **q**. Instead of finding all the UUIDs you will turn on the filter for just the MySvc Service UUID.

- Set up a structure of type `wiced_bt_gatt_discovery_param_t` with the starting and ending handles set to `0x0001` and `0xFFFF`. This covers all possible handle values.
- Set the UUID to be the UUID of the MySvc service.

Note: The MySvc service uses a 128-bit UUID so you will need something like this:

```
discovery_param.uuid.len = LEN_UUID_128;
memcpy( &discovery_param.uuid.uu.uuid128, serviceUUID, LEN_UUID_128 );
```

- Use `memcpy` to copy the service UUID into the `wiced_bt_gatt_discovery_param_t` structure.
- Set the discovery type to `GATT_DISCOVER_SERVICES_BY_UUID` and call the `wiced_bt_gatt_send_discover` function.

- ☐ 10. Add the case `GATT_DISCOVERY_RESULT_EVT` to the GATT event handler.

If the discovery type is `GATT_DISCOVER_SERVICES_BY_UUID` then update the `serviceStart` and `serviceEnd` handles with the returned start and end handles (remember you can use `GATT_DISCOVERY_RESULT_SERVICE_START_HANDLE` and `GATT_DISCOVERY_RESULT_SERVICE_END_HANDLE`).

Characteristic Discovery (w)

- ☐ 11. Create a function to launch the Characteristic discovery called `startCharacteristicDiscovery` when the user presses **w**.

- Set the global variable `charHandleCount` to 0. This must be done here so that each time a new characteristic discovery is started the variable that keeps track of how many characteristics have been found is reset.
- Set up a structure of type `wiced_bt_gatt_discovery_param_t` with the start and end handles to the range you discovered in the previous step (i.e. the service group start/end handles).

Note: Use `serviceStartHandle + 1` for the search start handle since we already know the first handle must be the service UUID.

- Call `wiced_bt_gatt_send_discover` with the discovery type set to `GATT_DISCOVER_CHARACTERISTICS`.



12. In the `GATT_DISCOVERY_RESULT_EVT` of the GATT event handler add an `if` statement for the characteristic result.

- a. Inside the `if`, save the `startHandle` and `valueHandle` in the `charHandles` array. Set the `endHandle` to the end of the service group (assume that this new characteristic is the last one in the service). If this is not the first characteristic that you found, then re-set the previous characteristic's end handle now that you know it wasn't the last one. It will be just before the new start handle. For example:

```
if( charHandleCount != 0 )
{
    charHandles[charHandleCount-1].endHandle =
        charHandles[charHandleCount].startHandle-1;
}
charHandleCount += 1;
```

Note: *The point is to assume that the current characteristic ends at the end of the service group. But, if you find another characteristic, then you know that the end of the previous characteristic's end handle is the start of the new characteristic minus 1.*

- b. Check if the characteristic is the LED or Counter characteristic based on the UUID. If it is one of those, then copy the start and value handles in the appropriate structure that you created (`ledChar` or `counterChar`). The end handles will not be done until characteristic discovery is complete because we won't know the true end handles until then. We'll add that next.



13. Add a case in the GATT event callback for the `GATT_DISCOVERY_CPLT_EVT` to get the LED and Counter end handles from the temporary array once characteristic discovery has finished.

You should first check to see if it is a characteristic discovery that just completed. If it is, copy the LED and Counter `endHandle` from the `charHandles` array. Your code could look something like this:

```
// Once all characteristics are discovered... you need to set up the end handles
if( p_event_data->discovery_complete.discovery_type ==
    GATT_DISCOVER_CHARACTERISTICS )
{
    for( int i=0; i<charHandleCount; i++ )
    {
        if( charHandles[i].startHandle == ledChar.startHandle )
            ledChar.endHandle = charHandles[i].endHandle;
        if( charHandles[i].startHandle == counterChar.startHandle )
            counterChar.endHandle = charHandles[i].endHandle;
    }
}
```

Descriptor Discovery (e)



14. Create a function to launch the Descriptor discovery called `startDescriptorDiscovery` when the user presses **e**.

The purpose of this function is to find the CCCD handle for the Counter characteristic.

Note: *Rather than make a generic descriptor discovery function, we will just look at the Counter characteristic since that's the only characteristic that we want to control notifications on.*

- a. It will need to search for all descriptors in the Counter characteristic's group.

Note: Since you know the first 2 attributes in the characteristic group are the characteristic declaration and the characteristic value, you can use the counter value handle + 1 for the start of the search space and the end handle for the end of the search space.

- b. Once the parameters are set up, launch `wiced_bt_gatt_send_discover` with `GATT_DISCOVER_CHARACTERISTIC_DESCRIPTOR`.

- ☐ 15. In the `GATT_DISCOVERY_RESULT_EVT` of the GATT event handler add an `if` statement for the descriptor result.

If the descriptor UUID is `__UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION`, save the CCCD handle.

- ☐ 16. Add the commands for 'q', 'w', and 'e' to call your three new functions to the UART task. Also add those keys to the help message.

Use Discovered Handles

- ☐ 17. Remove the variables that have the hard-coded handle values for the LED value, and Counter CCCD value (`ledHandle` and `counterCccdHandle`) so that we can see that service discovery works.
- ☐ 18. In the places where the old variables are used, change the code to use the `valHandle` or `cccdHandle` element from the appropriate `charHandle_t` structure for the value or CCCD handle (`ledChar.valHandle` or `counterChar.cccdHandle`).

Testing

- ☐ 19. Open a UART terminal to the central if it isn't already open, then build the application and program it to the kit.
- ☐ 20. Plug your peripheral back in (if necessary).
- ☐ 21. Test the following:
- ☐ a. Start scanning and connect (**s**).
 - ☐ b. Try to control the LED (**0, 1**) and notice that nothing happens because the handles have not been discovered yet.
 - ☐ c. Discover the MySvc service (**q**).
 - ☐ d. Discover the LED and counter characteristics (**w**).
 - ☐ e. Discover the Counter CCCD (**e**).
 - ☐ f. Control the LED (**0, 1**).
 - ☐ g. Turn on notifications (**n**).
 - ☐ h. Press the button on the peripheral to observe the notification..
 - ☐ i. Turn off notifications (**N**)
 - ☐ j. Press the button on the peripheral to observe notifications are disabled.
 - ☐ k. Disconnect (**d**).

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2022 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.