Chapter 4D: BLE Centrals

Time 2 ½ Hours

This chapter introduces you to the Central side of the BLE connection. By the end of this chapter you should be able to create a BLE Central that finds the right BLE Peripheral, connects to it, Reads, Writes and Enables/Handles notifications. It should also be able to perform the GATT Service Discovery Procedure to find the Handles of the Services, Characteristics and Descriptors on the GATT Server.

4D.1 GAP Roles, the Observer and the Central

In the previous three chapters the focus has been on BLE 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
- Peripheral: A device that can advertise and be connected to
- Observer: A device that passively listens to advertisers
- 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 other side of the connection? The answer to that question is the focus of this chapter.

4D.2 Scanning

In the previous chapters I talked about how you 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 WICED BLE device into scanning mode. You do this with a simple call to wiced_ble_bt_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 */
};
```

   The actual parameters of scan fast/slow are configured in the wiced_bt_cfg_settings_t structure typically found in wiced_bt_cfg.c.

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);
```

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;
```

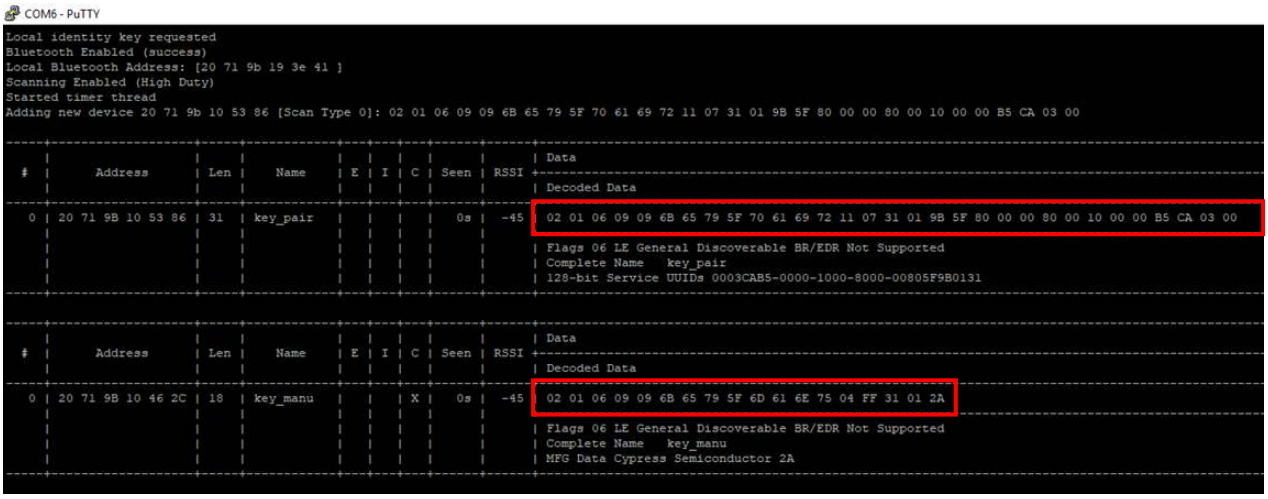
In your advertising callback function you can then parse the advertising data to decide what do next. WICED provides you a function called wiced_bt_ble_check_advertising data which can help you find information in the packet. Recall that every advertising packet is broken up into fields that each have a type. The wiced_bt_ble_check_advertising 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 with 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 wiced_bt_ble.h has a list of the legal advertising field types.

So, now what? Well consider the two cases in the screenshot from the Advertising Scanner (that project is 0 in this chapter). The results shown in the screenshot are from the devices created in Chapter 4B Exercise 03 and Chapter 4C Exercise 01. There are two different devices advertising, one named "key_pair" and one named "key manu". You can see the raw bytes of the advertising packet and the decode of those bytes.



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

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

if(name && (strncmp(name,"key_pair",length) == 0))
{
    WICED_BT_TRACE("Host = %B keypair\r\n ", p_scan_result->remote_bd_addr);
}
```

If you were looking for a device that was advertising the keypair Service UUID you might do this (notice that the UUIDs are stored little endian in the advertising packet):

```
static const uint8_t
serviceUUID[]={0x31,0x01,0x9B,0x5F,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0xB5,0xCA,0x03
,0x00};

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);
}
```

If you were looking for the Peripheral that was advertising the manufacturer's data with the Cypress manufacturer ID of 0x0131 and 0xFA for the data (which is what “key manu” is advertising) you might do this (again, notice the manufacturer ID is little endian in the advertising packet):

```
uint8_t *mfgData = wiced_bt_ble_check_advertising_data(p_adv_data,
                   BTM_BLE_ADVERT_TYPE_MANUFACTURER, &len);

if(mfgData && mfgData[0] == 0x31 && mfgData[1] == 01 && mfgData[2] == 0xFA)
{
    WICED_BT_TRACE("Host = %B length=%d ", p_scan_result->remote_bd_addr, length);
}
```

There are several functions which can be useful in comparing data for when you want to identify a particular device such as:

1. Strcmp(char *srt1, char *str2) and strncmp(char *str1, char *str2, int n) which compare two strings. Both functions take two pointers to strings. The strncmp function also takes an int which is the maximum number of characters to compare (this is safer). Both functions return a 0 if the strings MATCH (the opposite of a normal function).
2. 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.
3. If you have two variables of type wiced_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);
```

4D.3 Connecting, Pairing, and Encrypting

Now that you have found a device that you are interested in what next? 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. That feature is going to be deprecated.

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;  /**< Remmote 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 Client 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 transport can be either BT_TRANSPORT_BR_EDR (for Classic) or BT_TRANSPORT_LE (for BLE). The last two arguments are only for legacy pairing modes, so you should just use 0 for pin_len and NULL for the PIN code.

Note 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. LEGATTDB_PERM_AUTH_READABLE or LEGATTDB_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 BLE). 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:

```
wiced_bt_device_link_keys_t temp_keys;
wiced_bt_ble_sec_action_type_t  encryption_type = BTM_BLE_SEC_ENCRYPT;

for ( i = WICED_NVRAM_VSID_START; i < WICED_NVRAM_VSID_END; i++ ) //Search NVRAM for keys
{
    bytes_read = wiced_hal_read_nvram( 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!

4D.4 Attribute Protocol & More GATT Procedures

In the previous chapters I introduced you to the Peripheral side of several GATT Procedures. Specifically, Read, Write and Notify. Moreover, in those chapters you learned how to create WICED 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.

BT Spec Chap Ref	Request	Request Data	BT 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 4D.4.9	Find Information	Starting Handle Ending Handle	3.4.3.2 4D.4.9	Find Information Response	Handles Attribute Type UUIDs
3.4.3.3 4D.4.7	Find by Type Value	Starting Handle Ending Handle Attribute Type Attribute Value	3.4.3.4 4D.4.64D.4.8	Find by Type Value Response	Start Handle End of Group Handle
3.4.4.1 4D.4.8	Read by Type	Starting Handle Ending Handle Attribute Type UUID	3.4.4.2 4D.4.8	Read by Type Response	Handle Value Pairs
3.4.4.3 4D.4.1	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 4D.4.6	Read by Group Type	Starting Handle Ending Handle Attribute Group Type UUID	3.4.4.10 4D.4.6	Read by Group Type Response	For each match: Handle, Value Handle of last Attribute in Group
3.4.5.1 4D.4.3	Write	Handle, Value	There is no Server response to a Write.		
3.4.5.3 4D.4.3	Write Command	Handle, Value	3.4.5.2 4A	Write Response	Response code 0x1E Or Error Response
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
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

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?

4D.4.1 GATT Library

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 include these functions in your project you need to add the GATT utilities library to your makefile:

```
$(NAME)_COMPONENTS += gatt_utils_lib.a
```

Then you need to include the header file at the top of your C file:

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

4D.4.2 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 different error codes from the enumeration wiced_bt_gatt_status_e which was introduced in Chapter 4A. 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;
```

4D.4.3 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, setup 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 or not you want a response from the server.

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.

4D.4.4 GATT Client Notify and Indicate

To register for Notifications and Indications, the Client just needs to write to the CCCD Descriptor for the Characteristic of interest. There is a utility function that simplifies the process of writing a 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_UNSUPPORT_GRP_TYPE     = 0x10,     /**< Unsupported Group Type */
    WICED_BT_GATT_INSUF_RESOURCE         = 0x11,     /**< Insufficient Resource */
```

4D.4.5 GATT Group

There is one last GATT concept that needs to be introduced to understand the next 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 0x14 and ends at 0x18.

4D.4.6 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>>.

Consider this GATT Database from Chapter 4B Exercise 3 (pairing) and Chapter 4C Exercise 1 (advertise manufacturer's data).

Handle	Type	Value
0x01	<<Service>>	<<Generic Attribute>>
0x14	<<Service>>	<<Generic Access>>
0x15	<<Characteristic>>	0x16, <<Device Name>>
0x16	<<Device Name>>	
0x17	<<Characteristic>>	0x18, <<Appearance>>,0x02
0x18	<< Appearance>>	
0x28	<<Service>>	__UUID_WICED101
0x29	<<Characteristic>>	0x2A, __UUID_WICED101_LED,0x0E
0x2A	__UUID_WICED101_LED	
0x2B	<<Characteristic>>	0x2C, __UUID_WICED101_BUTTONS,0x12
0x2C	__UUID_WICED101_BUTTONS	
0x2D	<<CCCD>>	

In the database above if you input search start Handle=0x01, search end Handle=0xFFFF and Group Type = <<Service>> you would get as output:

Group Start Handle	Group End Handle	UUID
0x01	0x01	<<Generic Attribute>>
0x14	0x18	<<Generic Access>>
0x28	0x2D	__UUID_WICED101

In words, you receive a list of all the Service UUIDs with the start Handle and end Handle of each Group.

4D.4.7 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=0x01, ending handle=0xFFFF, Type=<<Service>> and Value=__UUID_WICED101 the output would be:

Group Start Handle	Group End Handle
0x28	0x2D

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»	Charac-teristic Properties	Character-istic Value Attribute Handle	Character-istic UUID	Read Only, No Authentication, No Authorization

4D.4.8 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=0x28, search ending Handle=0x2D and Type=<<Characteristic>> you would get as output:

Characteristic Handle	Value Handle	UUID	GATT Permission
0x29	0x2A	__UUID_WICED101_LED	0x0E
0x2B	0x2C	__UUID_WICED101_BUTTONS	0x12

In words, you get back a list of the Characteristic Handles, the Handles of the Values, the UUIDs, and the GATT Permissions.

4D.4.9 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 then 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 0x18→0x27 you will get a response of:

Handle	Attribute Type
0x18	<<Appearance>>

In words, the attribute type that is associated with the Characteristic handle 0x18.

4D.5 GATT Procedure: 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 hardcode 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.

4D.5.1 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 you the Characteristic Handles, Characteristic Value Hanels, 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 (4) discover the Descriptors using the GATT Procedure Find Information. This gives you the Attribute type of each Descriptor.

4D.5.2 WICED Service Discovery

The GATT utilities library that we introduced earlier 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 service within a service */
    GATT_DISCOVER_CHARACTERISTICS,              /**< discover characteristics of a service*/
    GATT_DISCOVER_CHARACTERISTIC_DESCRIPTORS,   /**< 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;
```

After you call this function, 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 event parameter can then be decoded using a set of macros provided by WICED.

```
/*****************************************************************************
 *  Macros for parsing results GATT discovery results
 * (while handling GATT_DISCOVERY_RESULT_EVT)
 *****************************************************************************/
/* Discovery type: GATT_DISCOVER_SERVICES_ALL or GATT_DISCOVER_SERVICES_BY_UUID */
#define GATT_DISCOVERY_RESULT_SERVICE_START_HANDLE(p_event_data)
#define GATT_DISCOVERY_RESULT_SERVICE_END_HANDLE(p_event_data)
#define GATT_DISCOVERY_RESULT_SERVICE_UUID_LEN(p_event_data)
#define GATT_DISCOVERY_RESULT_SERVICE_UUID16(p_event_data)
#define GATT_DISCOVERY_RESULT_SERVICE_UUID32(p_event_data)
#define GATT_DISCOVERY_RESULT_SERVICE_UUID128(p_event_data)

/* Discovery type: GATT_DISCOVER_CHARACTERISTIC_DESCRIPTORS */
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_DESCRIPTOR_UUID_LEN(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_DESCRIPTOR_UUID16(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_DESCRIPTOR_UUID32(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_DESCRIPTOR_UUID128(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_DESCRIPTOR_VALUE_HANDLE(p_event_data)

/* Discovery type: GATT_DISCOVER_CHARACTERISTICS */
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_VALUE_HANDLE(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_UUID_LEN(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_UUID16(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_UUID32(p_event_data)
#define GATT_DISCOVERY_RESULT_CHARACTERISTIC_UUID128(p_event_data)
```

There really should be a macro called GATT_DISCOVERY_RESULT_CHARACTERISTIC_HANDLE but for some reason, it isn't in WICED (yet). Therefore, when you want to get a Characteristic's Handle during Characteristic Discovery in your callback, you will have to use something like this:

```
p_data->discovery_result.discovery_data.characteristic_declaration.handle
```

When the discovery is complete you will get one more GATT callback 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.

4D.6 Running a GATT Server

Although somewhat uncommon, there is no reason why a BLE 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 BLE 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.

4D.7 Exercises

To do the exercises you will require two development kits – one to act as a Client and one to act as a Peripheral that you can scan for and connect to. The Peripheral will be programmed with the firmware from Chapter 4B Exercise 3 (pairing) or Chapter 4C Exercise 1 (advertise manufacturer's data).

Exercise - 4D.1 Make an Observer

In this exercise you will build an observer that will listen to all the BLE devices that are broadcasting and will print out the BD Address of each device that it finds. To create this project, follow these steps:

1. Begin by programming your kit with the Peripheral from Chapter 4B Exercise 3.
    a. Hint: If you did not complete that exercise, you can copy it from the solution projects, but make sure you change the device name in ex04_ble_bond.c and wiced_bt_cfg.c to use your initials so that you will be able to find it.
2. Unplug your Peripheral so that you don't accidentally re-program it with the Central project.
3. Get a 2nd kit from an instructor. You will program this kit with the Central project that you create. Note that the Central does NOT need a WBT101 shield – just the base board.
4. Run BT designer and make a project with no GATT database. Use the name ex01_observer.
    a. Hint: Once the project is created, you can move it to a new folder called "ch04d". This will allow you to keep the projects organized.
    b. Hint: Update the Make Target for the new folder location and to add BT_DEVICE_ADDRESS=random.
5. Change the UART to print to WICED_ROUTE_DEBUG_TO_PUART.
6. Delete the advertising functions and function calls.
7. Make a function to process the scanned advertising packets which just prints out the BD Address of the remote. This function declaration should look like this:

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

    Hint: Remember you can use the format code %B in WICED_BT_TRACE to print a BD Address.
8. Add a call to wiced_bt_ble_scan in the BTM_ENABLED event with a function pointer to your advertiser processor function. Enable filtering so that each new device only shows up once – otherwise you will see a LOT of packets from everyone's Peripheral.
9. Build and Program to your kit.
10. Open a terminal window.
11. Plug in the kit programmed with your Peripheral.
12. Open CySmart on your PC or Phone to find the BD Address of your Peripheral.
13. Look for the same BD Address in the list of devices printed by your Central.

Questions:

1. What is the cause of “Unhandled Bluetooth Management Event: 0x16 (22)”? How could you fix it?

Exercise - 4D.2 Read the Device Name to show only your Peripheral device's Info

In this exercise you will modify the previous exercise so that it examines the advertising packet to find the device name. It will only list your device instead of all devices. It will also look for the WICED101 Service UUID and Manufacturer's Data.

1. Unplug your Peripheral so that you don't accidentally re-program it.
2. Copy the previous project to ex02_observer_mydev
    a. Hint: Don't forget to update the name in all locations including the makefile.
3. Change your scanner callback function so that it looks at the advertising packet. Find and print out the BD Address only of devices that match your Peripheral's Device Name.
    a. Hint: You can use the function wiced_bt_ble_check_advertising_data to look at the advertising packet and find fields of type wiced_bt_ble_advert_type_t. If there is a field that matches it will return a pointer to those bytes and a length.
    b. Hint: You can use strncmp (to compare strings) and memcmp (to compare UUIDs) to see if the fields match what you are looking for.
4. For your Peripheral, search for the WICED101 Service UUID and Manufacturer's data in the advertising packet. Print them out if they are found.
    a. Hint: Create a constant uint8_t array with the UUID for the WICED101 Service so that you can compare it to the UUID found in the advertising packet.
5. Program your project to your Central.
6. Plug in your Peripheral and check to see if it is found. It should print:
    a. Device Name of your Peripheral
    b. BD Address
    c. UUID for the WICED101 Service
7. Use CySmart on your PC or your phone to verify that the BD Address matches.
8. Unplug the Central and then program the Peripheral with Chapter 4C Exercise 1.
9. Plug the Central back in, reconnect to the UART terminal window and reset the Central. The terminal window for the Client should now print:
    a. Device Name of your Peripheral
    b. BD Address
    c. Manufacturer's Data

Exercise - 4D.3 Update to connect to your Peripheral device & turn on/off the LED

In this exercise, you will modify the previous exercise to connect to your Peripheral once it finds it. We will decide what to connect to 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 but during class you will need to use the Device Name since there will be a lot of devices with the same Service and Manufacturer Data.

Once connected, you will be able to send 1's and 0's to the LED Characteristic to turn the LED off/on.

To simplify the project, you will hardcode the handle, but in the upcoming exercises you will add Service discovery.

The steps are as follows:

1. Copy the previous project to ex03_connect.
    a. Hint: Don't forget to update the name in all locations including the makefile.
2. Add a keyboard interface that reads in single key commands. Remember from the UART receive exercise in Chapter 2 that you will need to initialize the PUART and setup a call back in the <appname>_app_init function.
    a. Hint: you will need to include wiced_hal_puart.h
3. Create the UART Rx callback function to handle key presses. Make a switch with one case for each key that we will use for this project:
    a. ?: print out help
    b. s: turn on scanning
    c. S: turn off scanning
4. Add the appropriate wiced_bt_ble_scan commands to start/stop scanning and add WICED_BT_TRACE messages so that you can see what is going on from the terminal window.
5. Remove the call to wiced_bt_ble_scan from the BTM_Enabled event.
6. Test 1: Unplug your Peripheral, program your Central, and plug your Peripheral back in.
7. Verify that the keyboard interface works and that s/S turns scanning on and off.
8. In the wiced_bt_cfg.c file, find the GATT configuration entry for client_max_links. Change the value from 0 to 1. If you don't change this setting, the connection will not be allowed.
    a. Hint: This is usually set to 1 by WICED BT Designer, but since we initially created the project without a GATT database it set this value to 0.
9. Update your scan callback function to connect to the Peripheral when it finds one that it recognizes by calling wiced_bt_gatt_le_connect.
    a. Note: As mentioned previously, you will need to use the Device Name of the Peripheral here so that it will connect to your device. If you used the UUID or Manufacturer Data, it would connect to the first matching device it found which would almost certainly belong to another student.
    b. Hint: Check for both possible names (<inits>_pair and <inits>_manu) so that it will work with either Peripheral project.
10. After starting the connection, turn off scanning.

11. Create a new function to serve as the GATT callback function. The easiest way to do this is to copy the callback from one of the Peripheral projects and then remove all the cases except the default case. Regardless, the function must match this prototype:

```
typedef wiced_bt_gatt_status_t wiced_bt_gatt_cback_t (wiced_bt_gatt_evt_t event, wiced_bt_gatt_event_data_t *p_event_data);
```

    a. Hint: By default, WICED BT Designer calls the GATT callback function <appnane>_event_handler.
12. Register your GATT callback function in the BTM enabled event with function wiced_bt_gatt_register.
    a. Hint: You can also copy this from one of the Peripheral projects.
13. Add a global variable uint16_t to save the connection id.
14. In the GATT callback when you get a GATT_CONNECTION_STATUS_EVT figure out if it is a connect or a disconnect. Save the connection id to your global variable on a connect and set it to 0 on a disconnect.
15. Add a 'd' command to the UART interface to call the disconnect function.
16. Test 2: Unplug your Peripheral, program your Central, and plug your Peripheral back in.
17. Test the following:
    a. Start Scanning (s)
    b. Verify that it finds and then connects to your Peripheral
    c. Disconnect from your Peripheral (d)
18. Notice the Unhandled BTM event 0x16 message on the Client. Add a case to print out the "right" message.
19. Also notice the messages about Key Retrieval Failure. These occur because by default WICED Bluetooth Designer does not implement bonding and that it is the message it prints when keys are not available in NVRAM.
20. Create a new global variable called ledHandle. HARDCODE its initial value to the handle of your LED Characteristic's Value. You won't change the variable in this exercise, but in a future one you will find the handle via a Service Discovery.
    a. Hint: Make sure you use the Handle for the Characteristic's Value, not the Characteristic itself.
21. Create a new function to write the LED Characteristic Value with a uint8_t argument that represents either 1 or 0. If there is no connection or the ledHandle is 0 then you should return. Then call the GATT write function.
    a. Hint: Don't forget to include wiced_memory.h to get access to the get and free buffer functions.
22. Add commands for '1' and '0' in the UART interface that call your write LED function with 1 or 0.
23. Test 3: Unplug your Peripheral, program your Central, and plug your Peripheral back in.
24. Test the following:
    a. Start Scanning (s)
    b. Verify that it finds and then connects to your Peripheral
    c. Turn the LED on/off (1 and 0)
    d. Disconnect from your Peripheral (d)

Exercise - 4D.4 (Advanced) Add keys to turn the CCCD on/off

In this project, you will set up the CCCD to turn on/off Notifications for the button Characteristic and print out messages when Notifications are received.

1. Copy the previous project to ex04_connect_notify.
    a. Hint: Don't forget to update the name in all locations including the makefile.
2. After the connection is made using wiced_bt_gatt_le_connect, initiate pairing by calling wiced_bt_dev_sec_bond. 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 (i.e. Authenticated).
3. Add a new global variable called cccdHandle to hold the handle of the CCCD. Setup its initial value to 0x2D (this is a hardcoded which we will fix in the next exercise).
    a. Hint: Depending on how you setup the GATT database your handle might be different.
4. Create a new function to write the CDDD for the Buttons Characteristic with a uint8_t argument to turn Notifications off or on. If there is no connection or the cccdHanle is 0 then you should return.
    a. Hint: You can use the function wiced_bt_util_set_gatt_client_config_descriptor
    b. Hint: Don't forget to include the GATT utility library in the makefile and include the header file at the top of the C file.
5. Add cases 'n' and 'N' to set and unset the CCCD using the function you just created.
6. Add a case for GATT_OPERATION_CPLT_EVT into the GATT callback. This case should print out the connection id, operation, status, handle, length and the raw bytes of data.
    a. Hint: These values are all under the operation_complete structure in the event data. Some of them are 2 levels down under operation_complete.response_data or 3 levels down under operation_complete.response_data.att_value.
7. Unplug your Peripheral, program your Central, and plug your Peripheral back in.
8. Test the following:
    a. Start Scanning (s)
    b. Verify that the connection is made
    c. Touch a CapSense button. Notifications are not enabled so you should not see a response.
    d. Enable Notifications (n)
    e. Touch a CapSense button. You should see a response.
    f. Disable Notifications (N)
    g. Touch a CapSense button. You should not see a response.
    h. Disconnect (d)

Exercise - 4D.5 (Advanced) Make your Central do Service Discovery

In this exercise, instead of hardcoding the Handle for the LED and Button CCCD, we will modify our program to do a Service discovery. Instead of triggering the whole process with a state machine, we will use keyboard commands to launch the three stages.

The three stages are:

1. 'q'= Service discovery with the UUID of the WICED101 Service to get the start and end Handles for the Service Group.
2. 'w'= Characteristic discovery with the range of Handles from step 1 to discover all Characteristic Handles and Characteristic Value Handles.
3. 'e' = Descriptor discovery of the button Characteristic to find the CCCD Handle.

To create the project:

1. Copy the previous project to ex04_discover.
    a. Hint: Don't forget to update the name in all locations including the makefile.
2. Create variables to save the start and end Handle of the WICED101 Service and create a variable to hold the WICED101 Service UUID (if you don't already have one):

```
static const uint8_t serviceUUID[] = { 0x31, 0x01, 0x9b, 0x5f, 0x80, 0x00, 0x00, 0x80,
                                       0x00, 0x10, 0x00, 0x00, 0xb5, 0xca, 0x03, 0x00};
static uint16_t serviceStartHandle = 0x0001;
static uint16_t serviceEndHandle   = 0xFFFF;
```

3. Make a new structure to manage the discovered handles of the Characteristics:

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

4. Create a charHandle_t for the LED and the Button as well as the Characteristic UUIDs to search for:

```
static const uint8_t ledUUID[] = { 0xcf, 0x0a, 0x40, 0x04, 0x41, 0x42, 0x40, 0x70,
                                   0xad, 0x61, 0xe1, 0xce, 0xc0, 0x08, 0x2b, 0x41 };
static charHandle_t  ledChar;

static const uint8_t buttonUUID[] = { 0x31, 0x01, 0x9b, 0x5f, 0x80, 0x00, 0x00, 0x80,
                                      0x00, 0x10, 0x00, 0x00, 0xa3, 0xca, 0x03, 0x00 };
static charHandle_t  buttonChar;
```

    Hint: Your UUID values may be different.

5. Create an array of charHandle_t to temporarily hold the Characteristic Handles. 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.

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

Service Discovery

6. Add 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 WICED101 Service UUID.
    a. Setup the wiced_bt_gatt_discovery_param_t with the starting and ending Handles set to 0x0001 & 0xFFFF.
    b. Setup the UUID to be the UUID of the WICED101 Service.
    c. Use memcpy to copy the Service UUID into the wiced_bt_gatt_discovery_param_t.
    d. Set the discovery type to GATT_DISCOVER_SERVICES_BY_UUID and launch the wiced_bt_gatt_send_discover.
7. Add the case GATT_DISCOVERY_RESULT_EVT to your GATT Event Handler. If the discovery type is GATT_DISCOVER_SERVICES_BY_UUID then update the serviceStart and serviceEnd Handle with the actual start and end Handles (remember GATT_DISCOVERY_RESULT_SERVICE_START_HANDLE and GATT_DISCOVERY_RESULT_SERVICE_END_HANDLE.

Characterisic Discovery

8. Add a function to launch the Characteristic discovery called “startCharacteristicDiscovery” when the user presses ‘w’.
    a. Setup the wiced_bt_gatt_discovery_param_t start and end Handle to be the range you discovered in the previous step.
    b. Call wiced_bt_gatt_send_discover with the discovery type set to GATT_DISCOVER_CHARACTERISTICS.
9. In the GATT_DISCOVERY_RESULT_EVT of your GATT Event Handler add an "if" for the Characteristic result. In the “if” you need to save the startHandle and valueHandle in your charHandles array. Set the endHandle to the end of the Service Group (assume that this Characteristic is the last one in the Service). If this is not the first Characteristic, then set the previous Characteristic end handle:

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

The point is to assume that the 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 is the start of the new one minus 1.

Then you want to see if the Characteristic is the Button or LED Characteristic. If it is one of those, then also save the start, end and value Handles.

Descriptor Discovery

10. Add 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 button Characteristic.
    a. It will need to search for all the Descriptors in the Characteristic Group.
    b. The start will be the Button Value Handle + 1 to the end of the group handle.
    c. Once the parameters are setup, launch wiced_bt_gatt_send_discover with GATT_DISCOVER_CHARACTERISTIC_DESCRIPTORS.
11. In the GATT_DISCOVERY_RESULT_EVT of your GATT Event Handler add an "if" for the Descriptor result. If the Descriptor Attribute is <<CCCD> == 0x2902 then save the Button CCCD Handle.

12. The last change you need to make in the GATT callback is for the GATT_DISCOVERY_CPLT_EVT. You should check to see if it is a Characteristic discovery. If it is, then you will be able figure out the endHandle for each of the LED and Button Characteristic by copying them from the charHandles array. Your code could look something like this:

```
// Once all characteristics are discovered... you need to setup the end handles
if(p_data->discovery_complete.disc_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 == buttonChar.startHandle)
       buttonChar.endHandle = charHandles[i].endHandle;
  }
}
```

13. Add the function calls for ‘q’, ’w’, and ‘e’. Also add those keys to the help print out.
14. Unplug your Peripheral, program your Central, and plug your Peripheral back in.
15. Test the following:
    a. Start Scanning (s)
    b. Discover the WICED101 Service (q)
    c. Discover the Button and LED Characteristics (w)
    d. Discover the Button CCCD (e)
    e. Turn on the LED (1)
    f. Turn off the LED (0)
    g. Turn on notification (n)
    h. Press a CapSense button on the Peripheral and make sure it works
    i. Turn off notification (N)
    j. Press a CapSense button on the Peripheral to make sure the notification is off

Exercise - 4D.6 (Advanced) Run the advertising scanner

In this exercise you will experiment with a full-function advertising scanner project.

1. Copy the project from the class files at templates/ch04d/ex06_AdvScanner to your workspace.
2. Open a UART terminal window.
3. Create a make target and program the project to your kit.
4. Once the project starts running press "?" to get a list of available commands.
5. Use 's' and 'S' to enable/disable scanning
6. Use 't' to print a single line table of all devices that have been found
    a. This table will show raw advertising data
7. Use 'm' to print a multi-line table of all devices
    a. This table will show both raw advertising data and decoded data
    b. Use '?' to get a list of table commands
    c. Use '<' and '>' to change pages
    d. Use 'ESC' to exit the table
8. To filter on a specific device, enter its number from the table
9. Use 'r' to list recent packets from the filtered device
    a. Use '?' to get a list of table commands
    b. Use '<' and '>' to change pages
    c. Use 'ESC' to exit the table