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 and Writes and Accepts notifications.

4B.1 Central Roles 2

4B.2 WICED Scanning 2

4B.3 Connect 4

4B.4 Attribute Protocol & More GATT Procedures 4

4B.4.1 GATT Client Read 5

4B.4.2 GATT Client Write 7

4B.4.3 GATT Client Notify and Indicate 8

4B.4.4 GATT Client Find Information 8

4B.4.5 GATT Client Find by Type Value 9

4B.4.6 GATT Client Read by Group Type 9

4B.5 GATT Procedure: Service Discovery 9

4B.5.1 Service Discovery Algorithm 9

4B.5.2 Discover Services 10

4B.5.3 Discover Characteristics 10

4B.5.4 Discover Descriptors 10

4B.6 Running a GATT Server 10

4B.7 Exercises 10

Exercise 4B.1 Make an Observer 10

Exercise 4B.2 Add a filter to show only your device 10

Exercise 4B.3 Update to connect to your peripheral device & turn on/off the LED 11

Exercise 4B.4 Add keys to turn on CCCD 12

Exercise 4B.5 Make your project do service discovery 12

Exercise 4B.6 Run the advertising scanner 12

# Central Roles

# WICED 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 it 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. A wiced\_bt\_ble\_scan\_type\_t which tells the controller to either turn on, 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 is configured in the wiced\_bt\_cfg\_settings\_t structure. The next argument is a wiced\_bool\_t that tells the scanner to filter or not. The final argument is a function pointer that has a prototype of

typedef void (wiced\_bt\_ble\_scan\_result\_cback\_t) (wiced\_bt\_ble\_scan\_results\_t \*p\_scan\_result, uint8\_t \*p\_adv\_data);

Each time that your Centrals hears and advertisement, it will call your callback with a pointer to a structure of information about the device it just heard, and a pointer to the raw data. The scan results structure simply has the Bluetooth Device Address, the address type, what type of advertisement packet, and the rssi.

/\*\* 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. There is a function called wiced\_bt\_ble\_check\_advertising data which can help you find information in the packet. This function will search the advertising packet looking for a field and then if it finds that field, will return a pointer to the field and the length (via a pointer). For example:

 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. The enumeration wiced\_bt\_ble\_advert\_type\_e in the file wiced\_bt\_ble.h is list of the legal advertising fields.

So, now what? Well consider the two cases in the screenshot from the Advertising Scanner (ex10 from this chapter). There are two different devices advertising, one named “key\_pair” and one named “key menu”. You can see the raw bytes of the advertising packet and the decode data.



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,&length);

    if(name && strncmp(name,"key\_pair",length))

    {

        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 uint7\_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(serviceUUID,serviceUUID,16) == 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 manufacturers data 0xFA for Cypress Semiconductor (which is what “key menu” is advertising) you might do this:

    uint8\_t \*mfgData = wiced\_bt\_ble\_check\_advertising\_data(p\_adv\_data,BTM\_BLE\_ADVERT\_TYPE\_MANUFACTURER,&length);

    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.

strcmp and strncmp which compare two strings. Both functions take two pointers to strings. The strncmp also takes a int which is the maximum number of characters to compare (this is safer). Both function return a 0 if the strings MATCH (kind of the opposite of a normal function).

memcmp(unit8\_t \*p1,uint8\_t \*p2, int size) allows you to compare two blocks of memory. It returns 0 if the two block of memory are the same.

If you have two wiced\_uuid\_t you can compare them with the wice\_bt\_util\_uuid\_cmp function

int wiced\_bt\_util\_uuid\_cmp(wiced\_bt\_uuid\_t \*p\_uuid1, wiced\_bt\_uuid\_t \*p\_uuid2);

# Connect

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 with the address,

When the connection is 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 now save the conn\_id so that you can perform reads and write to the Peripheral. If you were going to support multiple connections you might make a table of connection id/BD Address tuples.

# 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 operations. Here is a list of all the Attribute operations with the original operation and the applicable response.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Chapter**  **BT Spec**  **Chap Ref** | **Request** | **Operation** | **Chapter**  **BT Spec** | **Response** | **Response Data** |
|  |  |  | 3.4.1.1 | Error Response | Request Op Code in Error  Attribute Handle in Error  Error Code |
| X  3.4.2.1 | Exchange MTU | Client Rx MTU | X  3.4.2.2 | Exchange MTU response | Server Rx MTU |
| 4D  3.4.3.1  4B.4.8 | Find Information | Starting Handle  Ending Handle | 4D  3.4.3.2  4B.4.8 | Find Information Response | Handles  Attribute Type UUIDs |
| 4D  3.4.3.3  4B.4.6 | Find by Type Value | Starting Handle  Ending Handle  Attribute Type  Attribute Value | 4D  3.4.3.4  4B.4.7 | Find by Type Value Response | Start Handle  End of Group Handle |
| 4D  3.4.4.1  4B.4.7 | Read by Type | Starting Handle  Ending Handle  Attribute Type UUID | 4D  3.4.4.2 | Read by Type Response | Handle Value Pairs |
| 4D  3.4.4.3 | Read | Handle | 4A  3.4.4.4 | Read Response | Handle, Value |
| X  3.4.4.5 | Read Blob | Handle, Offset | X  3.4.4.6 | Read Blob Response | Handle, Data |
| X  3.4.4.7 | Read Multiple | Handles | X  3.4.4.8 | Read Multiple Response | One response with all values concatenated |
| 4D  3.4.4.9  4B.4.5 | Read by Group Type | Starting Handle  Ending Handle  Attribute Group Type UUID | 4D  3.4.4.10  4B.4.5 | Read by Group Type Response | For each match:  Handle, Value  Handle of last Attribute in Group |
| 4D  3.4.5.1 | Write | Handle, Value | 4A | X |  |
| 4D  3.4.5.3 | Write Command | Handle, Value | 4A  3.4.5.2 | Write Response | Response code 0x1E |
| X  3.4.5.4 | Signed Write Command | Handle, Value, Signature |  |  | Response code 0x1E  Or  Error Response |
| X  3.4.6.1 | Prepare Write | Handle  Offset  Value | X  3.4.6.2 | Prepare Write Response | Handle  Offset  Value |
| X  3.4.6.3 | Execute Write | Flags (0-cancel, 1-write) | X  3.4.6.4 | Execute Write Response | Response code 0x19  Or  Error Response |
| 4B  3.4.7.1 | Notification | Handle, Value | 4D | X |  |
| 4B  3.4.7.2 | Indication | Handle, Value | 4D  3.4.7.3 | Handle Value Confirmation | Response code 0x1E  Or  Error Response |

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

## 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 including:

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 \*/

plus a bunch more.

The last piece is the response data, wiced\_bt\_gatt\_operation\_complete\_rsp\_t which contains the handle and the att value.

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 \*/

Finally, you get to the actual gatt data:

/\*\*  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;

All of this structure looks 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.

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;

## GATT Client Write

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 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. Then use a pointer to fill in the data.

Just like in the read case above 4B.4.1 ,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.

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.

If your gatt callback function returns WICED\_BT\_GATT\_SUCCESS, the stack sends a legal response of 0x1E. If you want to send an error back you can choose an appropriate one from the wiced\_bt\_gatt\_status\_e enumeration. This enumeration includes (partial list)

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 \*/

Victor must have a sense of humor after all, given error code WICED\_BT\_GATT\_ERR\_UNLIKELY

## GATT Client Notify and Indicate

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 and the value is just like the Read Response. In the case of Indicate you can return an error code from the wiced\_bt\_gatt\_status\_e enumeration which will be sent as part of the error response.

## GATT Group

There is one last GATT Concept that needs to be introduced to understand the next GATT Procedures, 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. For instance, in the above GATT database the Service Group for <<generic access>> starts at Handle 0x14 and ends at 0x18.

## GATT Client Read by Group Type

The GATT Client Ready by Group Type request takes as input starting, ending handle, group type and outputs a list of tuples with handle, end group handle, value. This request can only be used for a “Grouping Type” meaning, <<Service>>, <<Included Service>> and <<Characteristic>>

Consider this GATT Database from exercise 4C.x

|  |  |  |
| --- | --- | --- |
| **Handle** | **Type** | **Value** |
| 0x01 | <<service>> | <<generic attribute>> |
| 0x14 | <<service>> | <<generic access>> |
| 0x15 | <<characteristic>> | 0x16,<<device name>> |
| 0x16 | <<device name>> |  |
| 0x17 | <<characteristic>> | 0x18,<<characteristic appearance>>,0x02 |
| 0x18 | <<characteristic 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 starting handle=0x01, ending handle=0xFFFF, Group Type = <<service>> you would get as output

|  |  |  |
| --- | --- | --- |
| **Start Handle** | **End Group Handle** | **UUID** |
| 0x01 | 0x01 | <<generic attribute>> |
| 0x14 | 0x18 | <<generic access>> |
| 0x28 | 0x2D | \_\_uuid\_wiced101 |

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

## GATT Client Find by Type Value

The GATT Client Find by Type Value request takes as input the starting Handle, ending Handle, Attribute Type and Attribute Value. It then searches the database and returns the starting and ending handles of the group that match that Attribute Type and Attribute Value. The specific purpose of this function is 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 and Type=<<service>> and Value=\_\_uuid\_wiced101 the output would be:

|  |  |
| --- | --- |
| **Start Handle** | **End Group Handle** |
| 0x28 | 0x2D |

## GATT Client Read by Type

The GATT Client Ready by Type request takes as input starting, ending handle, Attribute Type and outputs a list of handle value pairs. In the example above if you the starting handle=0x28, ending handle=0x2D and Type=<<characteristic>> you would get as output

|  |  |  |  |
| --- | --- | --- | --- |
| **Characteristic Handle** | **Value Handle** | **UUID** | **GATT Permission** |
| 0x29 | 0x2A | <<generic attribute>> | 0x0E |
| 0x2B | 0x2C | <<generic access>> | 0x12 |

In words, a list of the Characteristics, the handles, the handles of the values, the permissions and the UUIDs

## GATT Client Find Information

The input GATT Client Find Information Request is simply a starting handle and an ending handle. The GATT Server then responds with a list of every handle in that range, and the Attribute type of the handle.

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 | <<characteristic appearance>> |

In words, all the Descriptors that are associated with the Characteristic handle 0x17

# 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. 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 4B.4.5 through 4B.4.8 , Read by Group Type, Find by Type Value, Read by Type and Find Information.

## Service Discovery Algorithm

The steps in the Service discovery algorithm are

1. Discover all the Services (which gives you the UUID and Start and End Handles of all the Service Groups) using Read by Group Type (to discover all Services). Or possibly discover only one Service by using Find by Type Value.
2. For each Service Group discover all the Characteristics (which gives you the Handles, UUIDs) using Read by Type with the Handle range of the Service Group that you discovered in step (1)
3. Using the Characteristic Handles you can then calculate the start and end ranges of each of the Descriptors for each characteristic, then run a Read by Type with the Type set to <<characteristic>>.
4. After (3) is done you will know the handles of all the Characteristics in a service. You will then need to calculate the ranges of the descriptors for the characteristics.
5. Using the ranges from (4) discover the Characteristics use the GATT Procedure Find Information

## WICED Service Discovery

Inside of WICED we have provided some utilities functions to simplify the service discovery process. To include the library in your project you need to add the library to your makefile.

$(NAME)\_COMPONENTS += gatt\_utils\_lib.a

The library has one 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:

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)

When the discovery is complete you will get one more gatt callback with the event type set to GATT\_DISCOVERY\_CPLT\_EVT.

# Running a GATT Server

# Exercises

* 1. Make an Observer

This project will listen to all the BLE devices that are broadcasting. It should print out the BD Address of each device. To build this project follow these steps:

Run BT designer and make a project with no gatt database

Delete the advertising functions and function calls

Make a function to process the scanned advertising packets which just prints out the BD Address of the remote. This function should have the prototype of:

typedef void (wiced\_bt\_ble\_scan\_result\_cback\_t) (wiced\_bt\_ble\_scan\_results\_t \*p\_scan\_result, uint8\_t \*p\_adv\_data);

Add a call to wiced\_bt\_ble\_scan to the BTM enabled event with a function pointer to your advertiser processor function.

Build and Program

What is the cause of “Unhandled Bluetooth Management Event: 0x16 (22)” and how do you fix it?

* 1. Add a filter to show only your device

Copy Exercise 01. Change your advertising packet call back to only print out devices that match

1. Your Device Name
2. Your Service UUID
3. Your Manufactures Code

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.

You can use strncmp (to compare strings) and memcmp (to compare UUIDs) to see if the fields match what you are looking for.

* 1. Update to connect to your peripheral device & turn on/off the LED

To do this exercise will require two development kits. One to be your peripheral, which should be programmed with the firmware from chapter 04c exercise xx.

Your project will scan and find your peripheral. Then connect to it. Then let you send 0’s and 1’s to led characteristic. To simplify the project, you will hardcode the handle, but in the upcoming exercises you will add service discovery.

The steps are as follows:

Copy the previous project

Build and program it to make sure it still works

Add a keyboard interface that has a ‘?’ To print out help. Remember from chapter 2, you will need to initialize the puart and setup a call back in the \_app\_init function.

Create a RxCallback function to handle key presses. Make a switch with one case per key (used by this program).

Add keys to turn on/off scanning ‘s’ ON and ‘S’ off (call the wiced\_bt\_ble\_scan with the correct arguments)

Remove the start scan from the BTM\_Enabled Event

Program your projects and make sure that the keyboard interface works and that the s/S turns the scanning on and off.

Update your adv packet function to connect to a the Peripheral when it finds one that it recognizes by calling wiced\_bt\_gatt\_le\_connect.

After starting the connection, turn off scanning

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 previous 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);

Register your gatt callback function in the BTM enabled event with function wiced\_bt\_gatt\_register

Add a global variable uint16\_t to save the connection id.

In the gatt callback when you get a GATT\_CONNECTION\_STATUS\_EVT figure out if it is a connect or a disconnect. Then save the connection id to your global variable

Add a ‘d’ to call disconnect

Program your project. Make sure that it finds and connects. Then disconnects. Then start the scanning and make sure that it reconnects

Notice the Unhandled BTM event 0x16 message. Add a case to print out the “right” message

Create a new global variable called ledHandle. HARDCODE its initial value to the handle of your LED Characteristic. You won’t change the variable in this exercise, but in a future one you will find the handle via a service discovery.

Create a new function to write the led 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.

Add cases for 1 and 0 in your command that call your write led function with 0 or 1

Program and test

* 1. Add keys to turn on CCCD

In this project, we will add setting up the CCCD to turn on notifications, and printing out messages when the notifications happens.

Copy the previous project

Build and program it to make sure that it still works.

Add a new global variable called cccdHandle to hold the handle of the CCCD. Setup its initial value to 0x?? (this is a hardcode which we will fix in the next exercise)

Add a new function called writeCCCD. This function is almost the same as writeLED except that its argument should be uint16\_t instead of uint8\_t (because CCCD is a 16-bit number). All the function needs to be updated to reflect the length of 2 instead of 1.

Add cases ‘n’ and ‘N’ to set and unset the CCCD and add those keys to the help printout.

Add a case for GATT\_OPERATION\_CPLT\_EVT into the gatt callback. This case should printout the connection id, operation, status, handle, length and the raw bytes of data

Build Program and Test

* 1. Make your project do service discovery

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’ = Do a service discovery with the UUID of the service
2. ‘w’ = Do a characteristic discovery with the range of handles from (1)
3. ‘e’ = Do a descriptor discovery of the button characteristic to find the CCCD.

Copy your previous project.

Make a new structure to manage the discovered handles.

typedef struct {

    uint16\_t startHandle;

    uint16\_t endHandle;

    uint16\_t valHandle;

    uint16\_t cccdHandle;

} charHandle\_t;

Create

Add a function to launch the service discovery called “startServiceDiscovery”. Instead of finding all the UUIDs you will turn on the filter for just the WICED101 UUID. The function needs a

* 1. Run the advertising scanner