# Attribute Protocol API

Document 2009-0010
Revision 1.5
September 25, 2015

## Table of Contents

# 1 Introduction

This document describes the API of the Attribute Protocol (ATT) subsystem. The attribute protocol is a core component of the Bluetooth LE protocol stack responsible for reading and writing attributes. The ATT API is divided into three parts: The ATT server interface (ATTS), the ATT client interface (ATTC) and the main interface common to both ATTS and ATTC.

Wicentric's ATT subsystem also implements the features of the of the Generic Attribute Profile (GATT) specification.

# 2 Main Interface

This portion of the ATT API is common to both client and server.

## 2.1 Constants and Data Types

### 2.1.1 Status

This parameter indicates the status of an attribute protocol operation.

| Name | Description |
|---|---|
| ATT_SUCCESS | Operation successful. |
| ATT_ERR_HANDLE | Invalid handle. |
| ATT_ERR_READ | Read not permitted. |
| ATT_ERR_WRITE | Write not permitted. |
| ATT_ERR_INVALID_PDU | Invalid pdu. |
| ATT_ERR_AUTH | Insufficient authentication. |
| ATT_ERR_NOT_SUP | Request not supported. |
| ATT_ERR_OFFSET | Invalid offset. |
| ATT_ERR_AUTHOR | Insufficient authorization. |
| ATT_ERR_QUEUE_FULL | Prepare queue full. |
| ATT_ERR_NOT_FOUND | Attribute not found. |
| ATT_ERR_NOT_LONG | Attribute not long. |
| ATT_ERR_KEY_SIZE | Insufficient encryption key size. |
| ATT_ERR_LENGTH | Invalid attribute value length. |
| ATT_ERR_UNLIKELY | Other unlikely error. |
| ATT_ERR_ENC | Insufficient encryption. |
| ATT_ERR_GROUP_TYPE | Unsupported group type. |
| ATT_ERR_RESOURCES | Insufficient resources. |
| ATT_ERR_CCCD | CCCD improperly configured. |
| ATT_ERR_IN_PROGRESS | Procedure already in progress. |
| ATT_ERR_RANGE | Value out of range. |
| ATT_ERR_MEMORY | Out of memory. |
| ATT_ERR_TIMEOUT | Transaction timeout. |
| ATT_ERR_OVERFLOW | Transaction overflow. |
| ATT_ERR_INVALID_RSP | Invalid response PDU. |

| | |
|---|---|
| ATT_ERR_CANCELLED | Request cancelled. |
| ATT_ERR_UNDEFINED | Other undefined error. |
| ATT_ERR_REQ_NOT_FOUND | Required characteristic not found. |
| ATT_CONTINUING | Procedure continuing. |
| ATT_ERR_VALUE_RANGE | Value out of range. |

HCI status values can also be passed through ATT. Since the values of HCI and ATT error codes overlap, the constant ATT_HCI_ERR_BASE is added to HCI error codes before being passed through ATT. See [1] for HCI error code values.

### 2.1.2 Operation

This parameter indicates the over-the-air attribute protocol operation.

| Name | Description |
|---|---|
| ATT_PDU_ERR_RSP | Error response. |
| ATT_PDU_MTU_REQ | Exchange mtu request. |
| ATT_PDU_MTU_RSP | Exchange mtu response. |
| ATT_PDU_FIND_INFO_REQ | Find information request. |
| ATT_PDU_FIND_INFO_RSP | Find information response. |
| ATT_PDU_FIND_TYPE_REQ | Find by type value request. |
| ATT_PDU_FIND_TYPE_RSP | Find by type value response. |
| ATT_PDU_READ_TYPE_REQ | Read by type request. |
| ATT_PDU_READ_TYPE_RSP | Read by type response. |
| ATT_PDU_READ_REQ | Read request. |
| ATT_PDU_READ_RSP | Read response. |
| ATT_PDU_READ_BLOB_REQ | Read blob request. |
| ATT_PDU_READ_BLOB_RSP | Read blob response. |
| ATT_PDU_READ_MULT_REQ | Read multiple request. |
| ATT_PDU_READ_MULT_RSP | Read multiple response. |
| ATT_PDU_READ_GROUP_TYPE_REQ | Read by group type request. |
| ATT_PDU_READ_GROUP_TYPE_RSP | Read by group type response. |
| ATT_PDU_WRITE_REQ | Write request. |
| ATT_PDU_WRITE_RSP | Write response. |
| ATT_PDU_WRITE_CMD | Write command. |
| ATT_PDU_PREP_WRITE_REQ | Prepare write request. |
| ATT_PDU_PREP_WRITE_RSP | Prepare write response. |
| ATT_PDU_EXEC_WRITE_REQ | Execute write request. |
| ATT_PDU_EXEC_WRITE_RSP | Execute write response. |
| ATT_PDU_VALUE_NTF | Handle value notification. |
| ATT_PDU_VALUE_IND | Handle value indication. |
| ATT_PDU_VALUE_CNF | Handle value confirmation. |

### 2.1.3 attCfg_t

This data type contains ATT run-time configurable parameters.

| Type | Name | Description |
|---|---|---|
| wsfTimerTicks_t | discIdleTimeout | ATT server service discovery connection idle timeout in seconds. |
| uint16_t | mtu | Desired ATT MTU. |
| uint8_t | transTimeout | Transaction Timeout in seconds. |
| uint8_t | numPrepWrites | Number of queued prepare writes supported by server. |

## 2.2 Functions

### 2.2.1 void AttRegister(attCback_t cback)

Register a callback with ATT.

- **cback**: Client callback function. See 2.3.1.

### 2.2.2 void AttConnRegister(dmCback_t cback)

Register a connection callback with ATT. The callback is typically used to manage the attribute server database.

- **cback**: DM client callback function. See [3].

### 2.2.3 uint16_t AttGetMtu(dmConnId_t connId)

Get the attribute protocol MTU of a connection.

- **connId**: DM connection ID.

## 2.3 Callback Interface

### 2.3.1 void (*attCback_t)(attEvt_t *pEvt)

This callback function sends ATT events to the client application. A single callback function is used for both ATTS and ATTC.

- **pEvt**: Pointer to ATT event structure.

### 2.3.2 Callback Events

The following callback event values are passed in the ATT event structure.

| Name | Description |
|---|---|
| ATTC_FIND_INFO_RSP | Find information response. |
| ATTC_FIND_BY_TYPE_VALUE_RSP | Find by type value response. |
| ATTC_READ_BY_TYPE_RSP | Read by type value response. |
| ATTC_READ_RSP | Read response. |
| ATTC_READ_LONG_RSP | Read long response. |
| ATTC_READ_MULTIPLE_RSP | Read multiple response. |
| ATTC_READ_BY_GROUP_TYPE_RSP | Read group type response. |
| ATTC_WRITE_RSP | Write response. |

| | |
|---|---|
| ATTC_WRITE_CMD_RSP | Write command response. |
| ATTC_PREPARE_WRITE_RSP | Prepare write response. |
| ATTC_EXECUTE_WRITE_RSP | Execute write response. |
| ATTC_HANDLE_VALUE_NTF | Handle value notification. |
| ATTC_HANDLE_VALUE_IND | Handle value indication. |
| ATTS_HANDLE_VALUE_CNF | Handle value confirmation. |

#### 2.3.3 attEvt_t

This data type is used for all callback events.

| Type | Name | Description |
|---|---|---|
| uint8_t | hdr.event | Callback event. |
| uint16_t | hdr.param | DM connection ID. |
| uint8_t | hdr.status | Event status. |
| uint8_t * | pValue | Pointer to value data, valid if valueLen > 0. |
| uint16_t | valueLen | Length of value data. |
| uint16_t | handle | Attribute handle. |
| bool_t | continuing | TRUE if more response packets expected. |

# 3 Server Interface

This API controls the operation of the attribute protocol server (ATTS).

## 3.1 Attribute Server Operation

An attribute server provides access to an attribute database stored within the server. According to the Bluetooth specification, attributes are collected into groups of characteristics, which are further collected into a service. A service is a collection of characteristics designed to accomplish a particular function, such as an alert service or a sensor service.

Figure 1 shows how services, characteristics, and attributes are organized according to the Bluetooth specification. An attribute database typically contains one or more services. Each service contains a set of characteristics, which is made up of one or more attributes. The type of attribute is uniquely identified by a UUID and an instance of an attribute in a server is uniquely identified by a handle. An attribute typically contains data that can be read or written by the attribute client on a peer device.

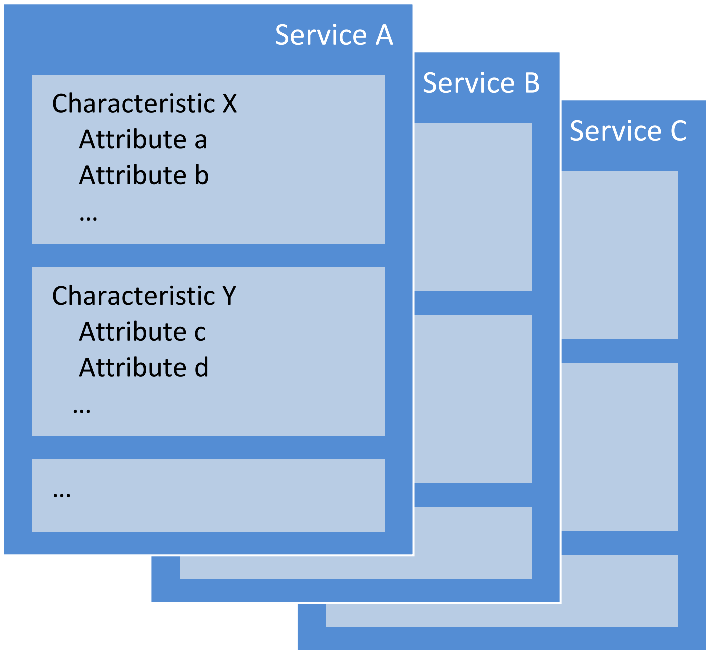

Figure 1. Services, characteristics, and attributes are stored within an attribute server.

In the ATTS implementation, the attribute database consists of a linked list of one or more group structures. Each attribute group structure points to an array of attribute structures. Each attribute structure contains the UUID, data, and other information for the attribute. The data structures in the ATTS database implementation are illustrated in Figure 2.

The group structure contains a pointer to the attribute array, the handle range of the attributes it references, and other data. A database implementation will typically use one group structure per service, although this is not a requirement; a group can contain multiple services, or a service can be implemented with multiple groups.

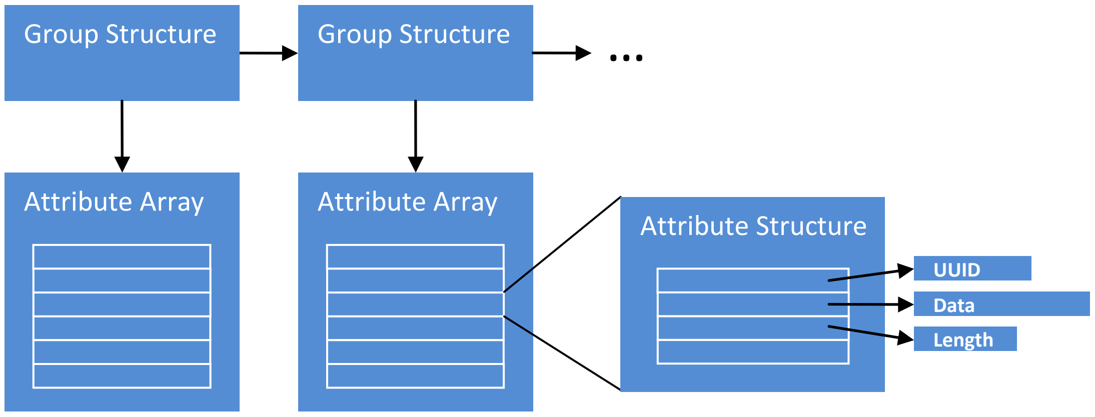

Figure 2. ATTS attribute database data structures.

## 3.2 Constants and Data Types

### 3.2.1 Attribute Settings

This bit mask parameter controls the settings of an attribute.

| Name | Value | Description |
|---|---|---|
| ATTS_SET_UUID_128 | 0x01 | Set if the UUID is 128 bits in length. |
| ATTS_SET_WRITE_CBACK | 0x02 | Set if the group callback is executed when this attribute is written by a client device |
| ATTS_SET_READ_CBACK | 0x04 | Set if the group callback is executed when this attribute is read by a client device. |
| ATTS_SET_VARIABLE_LEN | 0x08 | Set if the attribute has a variable length. |
| ATTS_SET_ALLOW_OFFSET | 0x10 | Set if writes are allowed with an offset. |
| ATTS_SET_CCC | 0x20 | Set if the attribute is a client characteristic configuration descriptor. |
| ATTS_SET_ALLOW_SIGNED | 0x40 | Set if signed writes are allowed. |
| ATTS_SET_REQ_SIGNED | 0x80 | Set if signed writes are required if link is not encrypted. |

### 3.2.2 Attribute Security Settings

This bit mask parameter controls the security settings of an attribute. These values can be set in any combination.

| Name | Value | Description |
|---|---|---|
| ATTS_PERMIT_READ | 0x01 | Set if attribute can be read. |
| ATTS_PERMIT_READ_AUTH | 0x02 | Set if attribute read requires authentication. |
| ATTS_PERMIT_READ_AUTHORIZ | 0x04 | Set if attribute read requires authorization. |
| ATTS_PERMIT_READ_ENC | 0x08 | Set if attribute read requires encryption. |
| ATTS_PERMIT_WRITE | 0x10 | Set if attribute can be written. |
| ATTS_PERMIT_WRITE_AUTH | 0x20 | Set if attribute write requires authentication. |
| ATTS_PERMIT_WRITE_AUTHORIZ | 0x40 | Set if attribute write requires authorization. |
| ATTS_PERMIT_WRITE_ENC | 0x80 | Set if attribute write requires encryption. |

### 3.2.3 Attribute UUID

An attribute UUID is either 16 bits or 128 bits in length. The UUID value is stored as a byte array in little endian format. For example:

```
/* 16 bit UUID value 0x0016 */
uint8 uuid16[] = {0x16, 0x00};

/* 128 bit UUID value 00001234-0000-1000-8000-00805F9B34FB */
uint8 uuid128[] = {0xFB, 0x34, 0x9B, 0x5F, 0x80, 0x00, 0x00, 0x80,
                   0x00, 0x10, 0x00, 0x00, 0x34, 0x12, 0x00, 0x00};
```

### 3.2.4 Attribute Value

The attribute value is stored as a byte array. If the attribute is an integer, the value is stored in little endian format.

### 3.2.5 Attribute Handles

The attribute protocol uses handles to uniquely identify attributes. To save memory, the attribute server does not store a handle for each attribute. Rather, it uses the starting handle value in each group to derive the handle of a particular attribute in the group. The start handle is the handle of the attribute at index zero of the group's attribute array. The handle of each subsequent attribute is simply the start handle plus the attributes index in the array.

### 3.2.6 attsAttr_t

This data type defines the structure used by an attribute in a group.

| Type | Name | Description |
|---|---|---|
| uint8_t * | pUuid | Pointer to the attribute's UUID. |
| uint8_t * | pValue | Pointer to the attribute's value. |
| uint16_t * | pLen | Pointer to the length of the attribute's value. |
| uint16_t | maxLen | Maximum length of attribute's value. |
| uint8_t | settings | Attribute settings. See 3.2.1. |
| uint8_t | permissions | Attribute permissions. See 3.2.2. |

### 3.2.7 attsGroup_t

This data type defines the structure used by a group.

| Type | Name | Description |
|---|---|---|
| attsGroup_t * | pNext | For internal use only. |
| attsAttr_t * | pAttr | Pointer to attribute list for this group. |
| attsReadCback_t | readCback | Read callback function. See 3.4.1. |
| attsWriteCback_t | writeCback | Write callback function. See 3.4.2. |
| uint16_t | startHandle | The handle of the first attribute in this group. |
| uint16_t | endHandle | The handle of the last attribute in this group. |

## 3.3 Functions

### 3.3.1 void AttsInit(void)

This function is called to initialize the attribute server. This function is generally called once during system initialization before any other ATTS API functions are called.

### 3.3.2 void AttsIndInit(void)

This function is called to initialize the attribute server for indications/notifications. This function is generally called once during system initialization before any other ATTS API functions are called.

### 3.3.3 void AttsSignInit(void)

This function is called to initialize the attribute server for data signing. This function is generally called once during system initialization before any other ATTS API functions are called.

### 3.3.4 AttsAuthorRegister(attsAuthorCback_t cback)

This function is called to register an authorization callback with the attribute server. This provides a mechanism to allow user authorization of read or write operations on a particular attribute.

### 3.3.5 void AttsAddGroup(attsGroup_t *pGroup)

Add an attribute group to the attribute server. The memory for the group structure is allocated by the caller and can only be deallocated after AttsRemoveGroup() is called.

- **pGroup**: Pointer to an attribute group structure. See 3.2.7.

### 3.3.6 void AttsRemoveGroup(uint16_t startHandle)

Remove an attribute group from the attribute server.

- **startHandle**: Start handle of attribute group to be removed.

### 3.3.7 void AttsSetAttr(uint16_t handle, uint16_t valueLen, uint8_t *pValue)

Set an attribute value in the attribute server. Before calling this function the group containing the attribute must be added to the server by calling AttsAddGroup().

- **handle**: Attribute handle.
- **valueLen**: Attribute length.
- **pValue**. Attribute value. See 3.2.4.

This function returns ATT_SUCCESS if successful otherwise error.

### 3.3.8 void AttsGetAttr(uint16_t handle, uint16_t *pLen, uint8_t **pValue)

Get an attribute value from the attribute server.

- **handle**: Attribute handle.
- **pLen**: Pointer to the attribute length.
- **pValue**. Attribute value. See 3.2.4.

This function returns ATT_SUCCESS if successful otherwise error.

This function returns the attribute length in pLen and a pointer to the attribute value in pValue. Note that pValue directly accesses memory inside the attribute database.

### 3.3.9 void AttsHandleValueInd(dmConnId_t connId, uint16_t handle, bool_t sendInd)

Send an attribute protocol Handle Value Indication.

- **connId**: DM connection ID.
- **handle**: Attribute handle.
- **len**: Length of value data.
- **pValue**. Pointer to value data.

When the operation is complete the client's callback function is called with an ATTS_HANDLE_VALUE_CNF event.

### 3.3.10 void AttsHandleValueNtf(dmConnId_t connId, uint16_t handle, bool_t sendInd)

Send an attribute protocol Handle Value Notification.

- **connId**: DM connection ID.
- **handle**: Attribute handle.
- **len**: Length of value data.
- **pValue**. Pointer to value data.

When the operation is complete the client's callback function is called with an ATTS_HANDLE_VALUE_CNF event.

### 3.3.11 void AttsSetCsrk(dmConnId_t connId, uint8_t *pCsrk)

Set the peer's data signing key on this connection. This function is typically called from the ATT connection callback when the connection is established. The caller is responsible for maintaining the memory that contains the key.

- **connId**: DM connection ID.
- **pCsrk**: Pointer to data signing key (CSRK).

### 3.3.12 void AttsSetSignCounter(dmConnId_t connId, uint32_t signCounter)

Set the peer's sign counter on this connection. This function is typically called from the ATT connection callback when the connection is established. ATT maintains the value of the sign counter internally and sets the value when a signed packet is successfully received.

- **connId**: DM connection ID.
- **handle**: Attribute handle.

### 3.3.13 uint32_t AttsGetSignCounter(dmConnId_t connId)

Get the current value peer's sign counter on this connection. This function is typically called from the ATT connection callback when the connection is closed so the application can store the sign counter for use on future connections.

- **connId**: DM connection ID.

This function returns the current value of the sign counter.

## 3.4 Callback Interface

### 3.4.1 uint8_t (*attsReadCback_t)( dmConnId_t connId, uint16_t handle, uint8_t operation, uint16_t offset, attsAttr_t *pAttr)

This is the attribute server read callback. It is executed on an attribute read operation if bitmask ATTS_SET_READ_CBACK is set in the settings field of the attribute structure.

- **connId**: DM connection ID.
- **handle**: Attribute handle.
- **operation**: Operation type. See 2.1.2.
- **offset**: Read data offset.
- **pAttr**: Pointer to attribute structure.

This function returns a status value (see 2.1.1). If the operation is successful then ATT_SUCCESS should be returned.

For a read operation, if the operation is successful the function must set pAttr->pValue to the data to be read. In addition, if the attribute is variable length then pAttr->pLen must be set as well.

### 3.4.2 void (*attsWriteCback_t)( dmConnId_t connId, uint16_t handle, uint8_t operation, uint16_t offset, uint16_t len, uint8_t *pValue, attsAttr_t *pAttr)

This is the attribute server write callback. It is executed on an attribute write operation if bitmask ATTS_SET_WRITE_CBACK is set in the settings field of the attribute structure.

- **connId**: DM connection ID.
- **handle**: Attribute handle.
- **operation**: Operation type. See 2.1.2.
- **offset**: Write data offset.
- **len**: Length of data to write.
- **pValue**: Data to write.
- **pAttr**: Pointer to attribute structure.

This function returns a status value (see 2.1.1). If the operation is successful then ATT_SUCCESS should be returned.

### 3.4.3 uint8_t (*attsAuthorCback_t)(dmConnId_t connId, uint8_t permit, uint16_t handle)

This callback function is executed when a read or write operation occurs and the security field of an attribute structure is set to ATTS_PERMIT_READ_AUTHORIZ or ATTS_PERMIT_WRITE_AUTHORIZ respectively.

- **connId**: DM connection ID.
- **permit**: Set to ATTS_PERMIT_WRITE for a write operation or ATTS_PERMIT_READ for a read operation.
- **handle**: Attribute handle.

This function returns a status value (see 2.1.1). If the operation is successful then ATT_SUCCESS should be returned. If the operation fails then ATTS_ERR_AUTHOR is typically returned.

# 4 Server Client Characteristic Configuration Interface

The following ATTS interface functions are a utility service for managing client characteristic configuration descriptors (abbreviated as CCC or CCCD). The client characteristic configuration descriptor is used to enable or disable indications or notifications of the characteristic value associated with the descriptor.

The Bluetooth specification has certain requirements for CCCDs:

1. The server must maintain the value of the CCCD separately for each client.
2. If the server and client are bonded, the value of the CCCD is persistent across connections.
3. If the server and client are not bonded, the value of the CCCD is reset to zero when the client connects.

The functions in this interface simplify and centralize the management of CCCDs. However if a server application does not use notifications or indications, or does not support bonding, then these functions do not need to be used.

An application using this interface is responsible for defining certain data structures, as shown below in Figure 3.

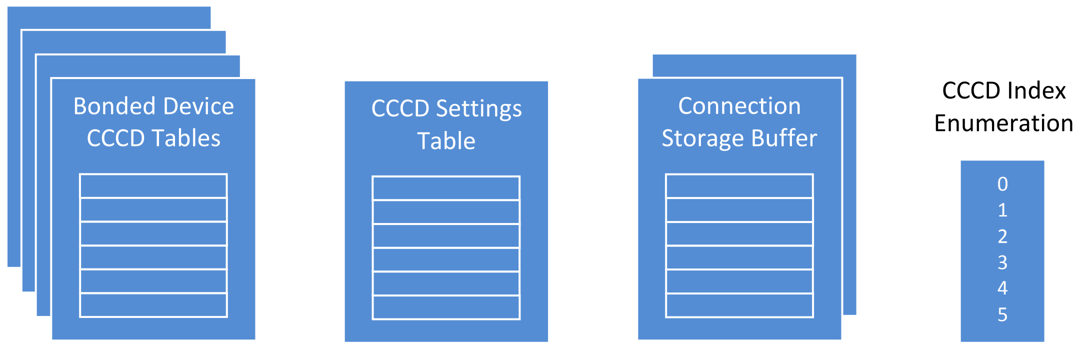

**Figure 3. CCCD data structures defined by the application.**

The data structures consist of bonded device CCCD tables, a CCCD settings table, a connection storage buffer, and a CCCD index enumeration. The Bonded device CCCD tables maintain persistent storage of the CCCD values for each bonded device. The CCCD settings table contains the CCCD attribute handle, security settings, and permitted CCCD values. The connection storage buffer holds separate CCCD values for all simultaneous connections. All tables are indexed by the CCCD index enumeration that defines the position in the table associated with each CCCD.

## 4.1 Constants and Data Types

### 4.1.1 attsCccSet_t

This data type defines the client characteristic configuration settings.

| Type | Name | Description |
|---|---|---|
| uint16_t | handle | Client characteristc configuration descriptor handle. |
| uint16_t | valueRange | Acceptable value range of the descriptor  value. |
| uint8_t | secLevel | Security level of characteristic value associated with the CCCD. |

## 4.2 Functions

### 4.2.1 void AttsCccRegister(uint8_t setLen, attsCccSet_t *pSet, attsCccCback_t cback)

Register the utility service for managing client characteristic configuration descriptors.  This function is typically called once on system initialization.

- **setLen**:  Length of settings array.
- **pSet**:  Array of CCC descriptor settings.
- **cback**:  Client callback function.

### 4.2.2 void AttsCccInitTable(dmConnId_t connId, uint16_t *pCccTbl)

Initialize the client characteristic configuration descriptor value table for a connection.  The table is initialized with the values from pCccTbl.  If pCccTbl is NULL the table will be initialized to zero.  This function is typically called when a connection is established or when a device is bonded.

- **connId**:  DM connection ID.
- **pCccTbl**:  Pointer to the descriptor value array.  The length of the array must equal the value of setLen passed to AttsCccRegister().

### 4.2.3 void AttsCccClearTable(dmConnId_t connId)

Clear and deallocate the client characteristic configuration descriptor value table for a connection.  This function must be called when a connection is closed.

- **connId**:  DM connection ID.

### 4.2.4 uint16_t AttsCccGet(dmConnId_t connId, uint8_t idx)

Get the value of a client characteristic configuration descriptor by its index.  If not found, return zero.

- **connId**:  DM connection ID.
- **idx**:  Index of descriptor in CCC descriptor handle table.

### 4.2.5 void AttsCccSet(dmConnId_t connId, uint8_t idx, uint16_t value)

Set the value of a client characteristic configuration descriptor by its index.

- **connId**:  DM connection ID.
- **idx**:  Index of descriptor in CCC descriptor handle table.
- **value**:  Value of the descriptor.

#### 4.2.6 uint16_t AttsCccEnabled(dmConnId_t connId, uint8_t idx)

Check if a client characteristic configuration descriptor is enabled and if the characteristic's security level has been met.

- **connId**: DM connection ID.
- **idx**: Index of descriptor in CCC descriptor handle table.

## 4.3 Callback Interface

#### 4.3.1 attsCccEvt_t

This data type defines the client characteristic configuration callback structure.

| Type | Name | Description |
|---|---|---|
| uint8_t | hdr.event | Callback event. |
| uint16_t | hdr.param | DM connection ID. |
| uint16_t | handle | CCCD handle. |
| uint16_t | value | CCCD value. |
| uint8_t | idx | CCCD index. |

#### 4.3.2 void (*attsCccCback_t)(attsCccEvt_t *pEvt)

Client characteristic configuration callback. This function is executed when a CCCD value changes. This happens when a peer device writes a new value to the CCCD or when a CCCD table is initialized by calling function AttsCccInitTable().

- **pEvt**: Pointer to callback structure.

# 5 Client Interface

## 5.1 Functions

#### 5.1.1 void AttcInit(void)

This function is called to initialize the attribute client. This function is generally called once during system initialization before any other ATTC API functions are called.

#### 5.1.2 void AttcSignInit(void)

This function is called to initialize the attribute client for data signing. This function is generally called once during system initialization before any other ATTC API functions are called.

#### 5.1.3 void AttcFindInfoReq(dmConnId_t connId, uint16_t startHandle, uint16_t endHandle, bool_t continuing)

Initiate an attribute protocol Find Information Request.

- **connId:** DM connection ID.

- **startHandle:** Attribute start handle.
- **endHandle:** Attribute end handle.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_FIND_INFO_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the start handle appropriately to send the next response.

### 5.1.4 void AttcFindByTypeValueReq(dmConnId_t connId, uint16_t startHandle, uint16_t endHandle, uint16_t uuid16, uint16_t valueLen, uint8_t *pValue, bool_t continuing)

Initiate an attribute protocol Find By Type Value Request.

- **connId:** DM connection ID.
- **startHandle:** Attribute start handle.
- **endHandle:** Attribute end handle.
- **uuid16:** 16-bit UUID to find.
- **valueLen**: Length of value data.
- **pValue:** Pointer to value data.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_FIND_BY_TYPE_VALUE_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the start handle appropriately to send the next response.

### 5.1.5 void AttcReadByTypeReq(dmConnId_t connId, uint16_t startHandle, uint16_t endHandle, uint8_t uuidLen, uint8_t *pUuid, bool_t continuing)

Initiate an attribute protocol Read By Type Request.

- **connId:** DM connection ID.
- **startHandle:** Attribute start handle.
- **endHandle:** Attribute end handle.
- **uuidLen:** Length of UUID (2 or 16).
- **pUuid:** Pointer to UUID data.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_READ_BY_TYPE_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the start handle appropriately to send the next response.

### 5.1.6 void AttcReadReq(dmConnId_t connId, uint16_t handle)

Initiate an attribute protocol Read Request.

- **connId:** DM connection ID.
- **handle:** Attribute handle.

When a response is received the client's callback function is called with an ATTC_READ _RSP.

### 5.1.7 void AttcReadLongReq(dmConnId_t connId, uint16_t handle, uint16_t offset, bool_t continuing)

Initiate an attribute protocol Read Long Request.

- **connId:** DM connection ID.
- **handle:** Attribute handle.
- **offset:** Read attribute data starting at this offset.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_READ_LONG_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the offset appropriately to send the next response.

### 5.1.8 void AttcReadMultipleReq(dmConnId_t connId, uint8_t numHandles, uint16_t *pHandles)

Initiate an attribute protocol Read Multiple Request.

- **connId:** DM connection ID.
- **numHandles:** Number of handles in attribute handle list.
- **pHandles:** List of attribute handles.

When a response is received the client's callback function is called with an ATTC_READ _MULTIPLE_RSP.

### 5.1.9 void AttcReadByGroupTypeReq(dmConnId_t connId, uint16_t startHandle, uint16_t endHandle, uint8_t uuidLen, uint8_t *pUuid, bool_t continuing)

Initiate an attribute protocol Read By GroupType Request.

- **connId:** DM connection ID.
- **startHandle:** Attribute start handle.
- **endHandle:** Attribute end handle.
- **uuidLen:** Length of UUID (2 or 16).
- **pUuid:** Pointer to UUID data.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_READ_BY_GROUP_TYPE_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the start handle appropriately to send the next response.

### 5.1.10 void AttcWriteReq(dmConnId_t connId, uint16_t handle, uint16_t valueLen, uint8_t *pValue)

Initiate an attribute protocol Write Request.

- **connId:** DM connection ID.
- **handle:** Attribute start handle.
- **valueLen**: Length of value data.
- **pValue:** Pointer to value data.

When a response is received the client's callback function is called with an ATTC_WRITE_RSP.

### 5.1.11 void AttcWriteCmd(dmConnId_t connId, uint16_t handle, uint16_t valueLen, uint8_t *pValue)

Initiate an attribute protocol Write Command.

- **connId:** DM connection ID.
- **handle:** Attribute start handle.
- **valueLen**: Length of value data.
- **pValue:** Pointer to value data.

When the packet has been sent the client's callback function is called with an ATTC_WRITE_CMD_RSP.

### 5.1.12 void AttcSignedWriteCmd(dmConnId_t connId, uint16_t handle, uint16_t uint32_t signCounter, valueLen, uint8_t *pValue)

Initiate an attribute protocol Write Command.

- **connId:** DM connection ID.
- **handle:** Attribute start handle.
- **valueLen**: Length of value data.
- **signCounter**: Value of sign counter.
- **pValue:** Pointer to value data.

When the packet has been sent the client's callback function is called with an ATTC_WRITE_CMD_RSP.

Note that the application is responsible for maintaining the value of the sign counter. The sign counter should be incremented each time this function is called.

### 5.1.13 void AttcPrepareWriteReq(dmConnId_t connId, uint16_t handle, uint16_t offset, uint16_t valueLen, uint8_t *pValue, bool_t valueByRef, bool_t continuing)

Initiate an attribute protocol Prepare Write Request.

- **connId:** DM connection ID.
- **handle:** Attribute start handle.
- **offset:** Write attribute data starting at this offset.
- **valueLen**: Length of value data.
- **pValue:** Pointer to value data.
- **valueByRef:** TRUE if pValue data is accessed by reference rather than copied.
- **continuing:** TRUE if ATTC continues sending requests until complete.

When a response is received the client's callback function is called with an ATTC_PREPARE_WRITE_RSP. If parameter continuing is TRUE, ATTC will automatically send the next request until all responses are received or an error is received. If parameter continuing is FALSE, the client application must call this function again and update the offset appropriately to send the next response.

### 5.1.14 void AttcExecuteWriteReq(dmConnId_t connId, bool_t writeAll)

Initiate an attribute protocol Execute Write Request.

- **connId:** DM connection ID.
- **writeAll:** TRUE to write all queued writes, FALSE to cancel all queued writes.

When a response is received the client's callback function is called with an ATTC_EXECUTE_WRITE_RSP.

### 5.1.15 void AttcCancelReq(dmConnId_t connId)

Cancel an attribute protocol request in progress.

- **connId:** DM connection ID.

If the request is cancelled the client's callback function is called with the event corresponding to the request.

# 6 Client Discovery Interface

The ATTC API contains a utility interface that simplifies common GATT client service and characteristic discovery procedures. It also contains interfaces that simplify the configuration of a service, e.g. the reading or writing a set of characteristics or attributes after discovery is complete.

An application using this interface is responsible for defining certain data structures, as shown below in Figure 4.

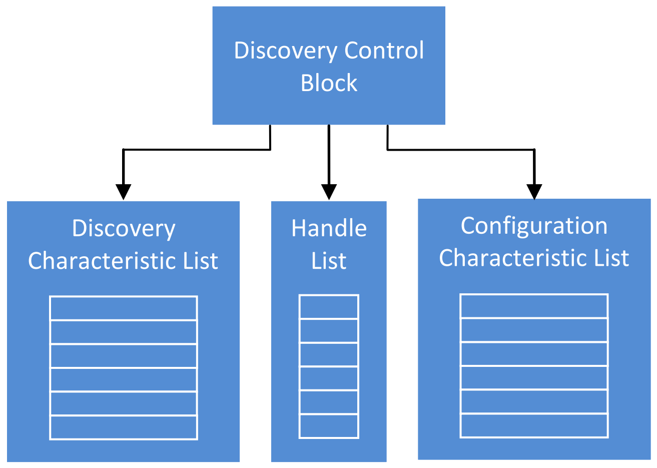
Discovery Control Block
Discovery Characteristic List
Handle List
Configuration Characteristic List

**Figure 4. Client discovery data structures defined by the application.**

The client discovery API uses a discovery control block that contains data used for the discovery and configuration procedure. The control block points to a discovery characteristic list, a configuration characteristic list, and a handle list.

The discovery characteristic list is a list of characteristics and descriptors that are to be discovered. Each item in the list contains the UUID of the characteristic or descriptor and its settings. As characteristics and descriptors are discovered the handle list is populated with their respective handles.

The configuration characteristic list contains a list of characteristics and descriptors to read or write. Each item in the list contains the value (if it is to be written) and the handle index of the characteristic or descriptor in the handle list.

## 6.1 Constants and Data Types

### 6.1.1 Discovery Settings

These settings are used to define the features of a characteristic that is being discovered.

| Name | Description |
|---|---|
| ATTC_SET_UUID_128 | Set if the UUID is 128 bits in length. |
| ATTC_SET_REQUIRED | Set if characteristic must be discovered. |
| ATTC_SET_DESCRIPTOR | Set if this is a characteristic descriptor. |

### 6.1.2 attcDiscChar_t

This data type is the structure for characteristic and descriptor discovery.

| Type | Name | Description |
|---|---|---|
| uint8_t * | pUuid | Pointer to UUID. |
| uint8_t | settings | Characteristic discovery settings. See 6.1.1. |

### 6.1.3 attcDiscCfg_t

This data type is the structure for characteristic and descriptor configuration.

| Type | Name | Description |
|---|---|---|
| uint8_t * | pValue | Pointer to UUID. |
| uint8_t | valueLen | Default value length. |
| uint8_t | hdlIdx | Index of its handle in handle list. |

### 6.1.4 attcDiscCb_t

This data type is the discovery control block.

| Type | Name | Description |
|---|---|---|
| attcDiscChar_t ** | pCharList | Characterisic list for discovery. |
| uint16_t* | pHdlList | Characteristic handle list. |
| attcDiscCfg_t* | pCfgList | Characterisic list for configuration. |
| uint8_t | charListLen | Characteristic and handle list length. |
| uint8_t | cfgListLen | Configuration list length. |

## 6.2 Functions

### 6.2.1 void AttcDiscService(dmConnId_t connId, attcDiscCb_t *pCb, uint8_t uuidLen, uint8_t *pUuid)

This utility function discovers the given service on a peer device. Function AttcFindByTypeValueReq() is called to initiate the discovery procedure.

- **connId:** DM connection ID.
- **pCb**: Pointer to discovery control block.
- **uuidLen**: Length of service UUID (2 or 16).
- **pUuid**: Pointer to service UUID.

### 6.2.2 uint8_t AttcDiscServiceCmpl(attcDiscCb_t *pCb, attEvt_t *pMsg)

This utility function processes a service discovery result. It should be called when an ATTC_FIND_BY_TYPE_VALUE_RSP callback event is received after service discovery is initiated by calling AttcDiscService().

- **pCb**: Pointer to discovery control block.
- **pMsg**: ATT callback event message.

Returns ATT_SUCCESS if successful otherwise error.

### 6.2.3 void AttcDiscCharStart(dmConnId_t connId, attcDiscCb_t *pCb)

This utility function starts characteristic and characteristic descriptor discovery for a service on a peer device. The service must have been previously discovered by calling AttcDiscService() and AttcDiscServiceCmpl().

- **connId:** DM connection ID.
- **pCb**: Pointer to discovery control block.

### 6.2.4 uint8_t AttcDiscCharCmpl(attcDiscCb_t *pCb, attEvt_t *pMsg)

This utility function processes a characteristic discovery result. It should be called when an ATTC_READ_BY_TYPE_RSP or ATTC_FIND_INFO_RSP callback event is received after characteristic discovery is initiated by calling AttcDiscCharStart().

- **pCb**: Pointer to discovery control block.
- **pMsg**: ATT callback event message.

Returns ATT_CONTINUING if successful and the discovery procedure is continuing. Returns ATT_SUCCESS if the discovery procedure completed successfully. Returns ATT_ERR_REQ_NOT_FOUND if discovery failed because a required characteristic was not found. Otherwise the discovery procedure failed.

### 6.2.5 uint8_t AttcDiscConfigStart(dmConnId_t connId, attcDiscCb_t *pCb)

This utility function starts characteristic configuration for characteristics on a peer device. The characteristics must have been previously discovered by calling AttcDiscCharStart() and AttcDiscCharCmpl ().

- **connId:** DM connection ID.
- **pCb**: Pointer to discovery control block.

Returns ATT_CONTINUING if successful and configuration procedure is continuing. Returns ATT_SUCCESS if nothing to configure.

### 6.2.6 uint8_t AttcDiscConfigCmpl(dmConnId_t connId, attcDiscCb_t *pCb)

This utility function initiates the next characteristic configuration procedure. It should be called when an ATTC_READ_RSP or ATTC_WRITE_RSP callback event is received after characteristic configuration is initiated by calling AttcDiscConfigStart().

- **connId**: DM connection ID.
- **pCb**: Pointer to discovery control block.

Returns ATT_CONTINUING if successful and configuration procedure is continuing. Returns ATT_SUCCESS if configuration procedure completed successfully.

### 6.2.7 AttcDiscConfigResume(dmConnId_t connId, attcDiscCb_t *pCb)

This utility function resumes the characteristic configuration procedure. It can be called when an ATTC_READ_RSP or ATTC_WRITE_RSP callback event is received with failure status to attempt the read or write procedure again.

- **connId**: DM connection ID.
- **pCb**: Pointer to discovery control block.
- Returns ATT_CONTINUING if successful and configuration procedure is continuing. Returns ATT_SUCCESS if configuration procedure completed successfully.

# 7 GATT Discovery Procedures

The Generic attribute profile (GATT) of the Bluetooth core specification defines how attribute protocol operations are used to perform GATT procedures. The table below demonstrates how the ATTC API is used to perform GATT discovery procedures.

| GATT Procedure | ATTC API |
|---|---|
| Discover All Primary Services | AttcReadByGroupTypeReq()<br>startHandle = 0x0001<br>EndHandle = 0xFFFF<br>uuidLen = 2<br>pUuid = pointer to ATT_UUID_PRIMARY_SERVICE<br>continuing = TRUE |
| Discover Primary Services by Service UUID | AttcFindByTypeValueReq()<br>startHandle = 0x0001<br>EndHandle = 0xFFFF<br>uuid16 = ATT_UUID_PRIMARY_SERVICE<br>valueLen = 2 or 16<br>pValue = pointer to service UUID<br>continuing = TRUE |
| Find Included Services | AttcReadByTypeReq()<br>startHandle = service start handle<br>EndHandle = service end handle<br>uuidLen = 2<br>pUuid = pointer to ATT_UUID_INCLUDE |
| Discover All Characteristics of a Service | AttcReadByTypeReq()<br>startHandle = service start handle<br>EndHandle = service end handle<br>uuidLen = 2<br>pUuid = pointer to ATT_UUID_CHARACTERISTIC<br>continuing = TRUE |
| Discover Characteristics by UUID | AttcReadByTypeReq()<br>startHandle = service start handle<br>EndHandle = service end handle<br>uuidLen = 2 |

| | |
|---|---|
| | pUuid = pointer to ATT_UUID_CHARACTERISTIC<br>continuing = TRUE |
| Discover All Characteristic Descriptors | AttcFindInfoReq()<br>startHandle = characteristic value handle + 1<br>EndHandle = characteristic end handle<br>continuing = TRUE |

# 8 Scenarios

## 8.1 Server Operations

Figure 5 shows some example server operations. First, a connection is established with an attribute protocol client on a peer device. The peer device sends an attribute protocol read request. In this example, the read request is handled internally by the stack and no interaction is required from the application. Next, the peer device sends a write request. In this example, the attribute being written is configured to execute a write callback function. The callback executes and the application performs whatever operation is necessary for the attribute. Upon return of the callback the stack sends a write response packet. Next, the application sends a handle value notification to the peer device by calling AttsHandleValueInd(). The stack sends a handle value indication packet. When the stack receives a handle value confirmation packet from the peer it executes the application's ATT callback with event ATTS_HANDLE_VALUE_CNF.

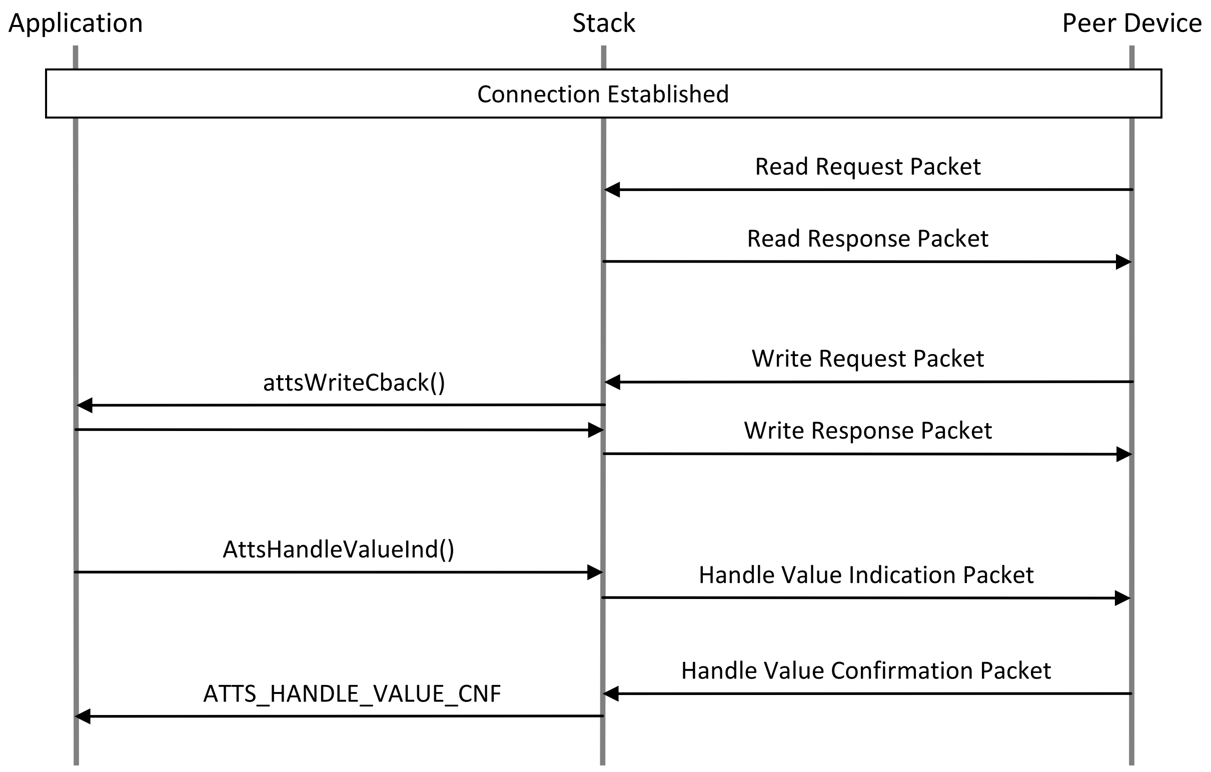

Figure 5. Server operations.

## 8.2 Client Operations

Figure 6 shows some example client operations. First, a connection is established with an attribute protocol server on a peer device. The application initiates a request by calling AttcReadByGroupTypeReq() with the continuing parameter set to TRUE. The client sends an attribute protocol read by group type request, receives a response and executes the ATT callback with event ATTC_READ_BY_GROUP_TYPE_RSP. Since the read by group type procedure is not complete the client automatically sends another read by group type request packet to continue the procedure. When the procedure is complete the ATT callback is executed with event ATTC_READ_BY_GROUP_TYPE_RSP and the continuing parameter set to FALSE.

Next the application sends another request by calling AttcReadByTypeReq(). The stack sends a read by type request packet, receives a response, and executes the ATT callback with event ATTC_READ_BY_TYPE_RSP. In this example the procedure is complete in the first packet transaction and the continuing parameter is set to FALSE.

Finally, the application writes a attribute by calling AttcWriteCmd(). The stack sends a write command packet. This packet does not have a corresponding response packet. When the stack has sent the packet it executes the ATT callback with event ATTC_WRITE_CMD_RSP.

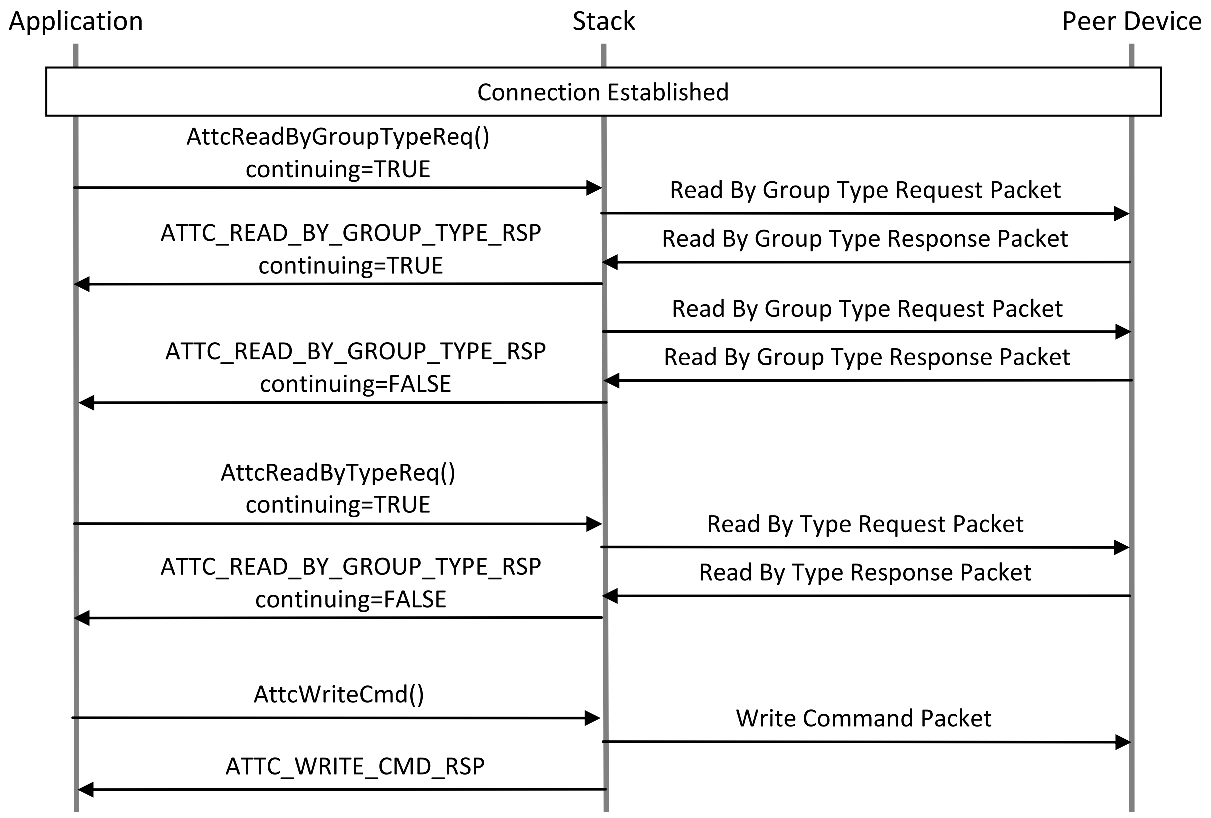
Application
Stack
Peer Device
Connection Established
AttcReadByGroupTypeReq() continuing=TRUE
Read By Group Type Request Packet
ATTC_READ_BY_GROUP_TYPE_RSP continuing=TRUE
Read By Group Type Response Packet
Read By Group Type Request Packet
ATTC_READ_BY_GROUP_TYPE_RSP continuing=FALSE
Read By Group Type Response Packet
AttcReadByTypeReq() continuing=TRUE
Read By Type Request Packet
ATTC_READ_BY_GROUP_TYPE_RSP continuing=FALSE
Read By Type Response Packet
AttcWriteCmd()
Write Command Packet
ATTC_WRITE_CMD_RSP

Figure 6. Client operations.

## 8.3 Client Prepare and Execute Write

Figure 7 shows an example prepare and execute write procedure. The application calls AttcPrepareWriteReq() to write an attribute value. The stack sends prepare write request packets until all the data has been sent to the peer device. The ATT callback is executed with event ATTC_PREPARE_WRITE_RSP each time a response packet is received. When callback event parameter continuing is set to FALSE the procedure is complete.

Next the application calls AttcExecuteWriteReq() to execute the write procedure in the peer device's attribute server. The stack sends and execute write request packet. When it receives a response it executes the ATT callback with event ATTC_EXECUTE_WRITE_RSP.

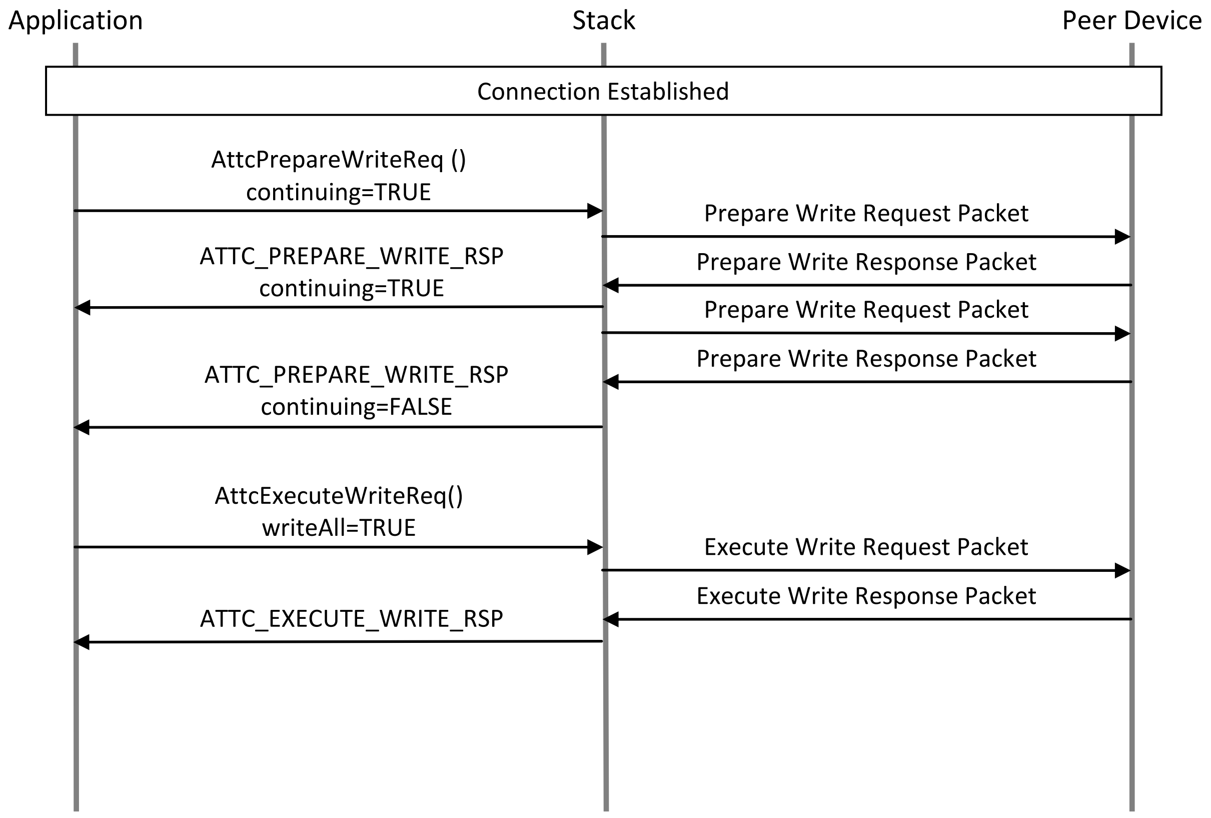

Figure 7. Client prepare and execute write.

## 8.4 Client Discovery and Configuration

Figure 8 shows and example of discovery and configuration using the ATT client discovery API. First, service discovery is initiated by calling AttcDiscService() with the UUID of the service to be discovered.

The ATT callback is executed with event ATTC_FIND_BY_TYPE_VALUE_RSP containing discovery results. The callback message is passed to function AttcDiscServiceCmpl(), which returns ATT_SUCCESS indicating that service discovery completed successfully.

Then the application proceeds with characteristic discovery by calling AttcDiscCharStart(). The ATT callback is executed with event ATTC_READ_BY_TYPE_RSP containing characteristic discovery results. The callback message is passed to function AttcDiscCharCmpl(), which returns ATT_CONTINUING indicating that characteristic discovery is continuing. This procedure repeats until AttcDiscCharCmpl() returns ATT_SUCCESS indicating that characteristic discovery completed successfully.

Then the application proceeds with characteristic configuration by calling AttcDiscConfigStart(). A characteristic read or write is performed according to the contents of the configuration characteristic list, and the ATT callback is executed. The callback message is passed to function AttcDiscConfigCmpl(), which returns ATT_CONTINUING indicating that configuration is not complete. The procedure repeats until AttcDiscConfigCmpl() returns ATT_SUCCESS.

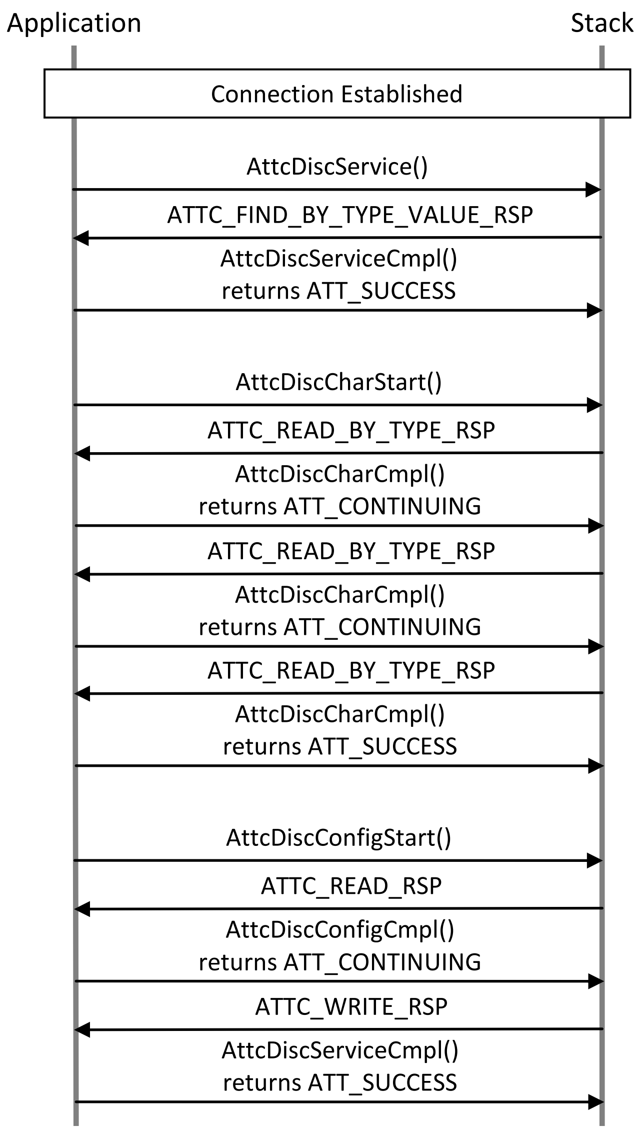
Application
Stack
Connection Established
AttcDiscService()
ATTC_FIND_BY_TYPE_VALUE_RSP
AttcDiscServiceCmpl() returns ATT_SUCCESS
AttcDiscCharStart()
ATTC_READ_BY_TYPE_RSP
AttcDiscCharCmpl() returns ATT_CONTINUING
ATTC_READ_BY_TYPE_RSP
AttcDiscCharCmpl() returns ATT_CONTINUING
ATTC_READ_BY_TYPE_RSP
AttcDiscCharCmpl() returns ATT_SUCCESS
AttcDiscConfigStart()
ATTC_READ_RSP
AttcDiscConfigCmpl() returns ATT_CONTINUING
ATTC_WRITE_RSP
AttcDiscServiceCmpl() returns ATT_SUCCESS

**Figure 8. Client discovery and configuration procedures.**

## 9 References

1. Wicentric, “HCI API”, 2009-0006.
2. Wicentric, “Software Foundation API”, 2009-0003.
3. Wicentric, “Device Manager API”, 2009-0008.

# 10 Definitions

| | |
|---|---|
| ATT | Attribute protocol software subsystem |
| ATTC | Attribute protocol client |
| ATTS | Attribute protocol server |
| CCC or CCCD | Client Characteristic Configuration Descriptor |
| DM | Device Manager software subsystem |
| GATT | Generic Attribute Profile specification |
| HCI | Host Controller Interface |
| LE | (Bluetooth) Low Energy |
| WSF | Wicentric Software Foundation software service and porting layer |