Generic Attribute Profile (GATT)

Just as the GAP layer handles most connection-related functionality, the GATT layer of the Bluetooth low energy protocol stack is used by the application for data communication between two connected devices. Data is passed and stored in the form of characteristics which are stored in memory on the Bluetooth low energy device. From a GATT standpoint, when two devices are connected they are each in one of two roles.

- **The GATT server**
  the device containing the characteristic database that is being read or written by a GATT client.

- **The GATT client**
  the device that is reading or writing data from or to the GATT server.

Figure 45. shows this relationship in a sample Bluetooth low energy connection where the peripheral device (that is, a CC2640R2 Launchpad) is the GATT server and the central device (that is, a smart phone) is the GATT client.

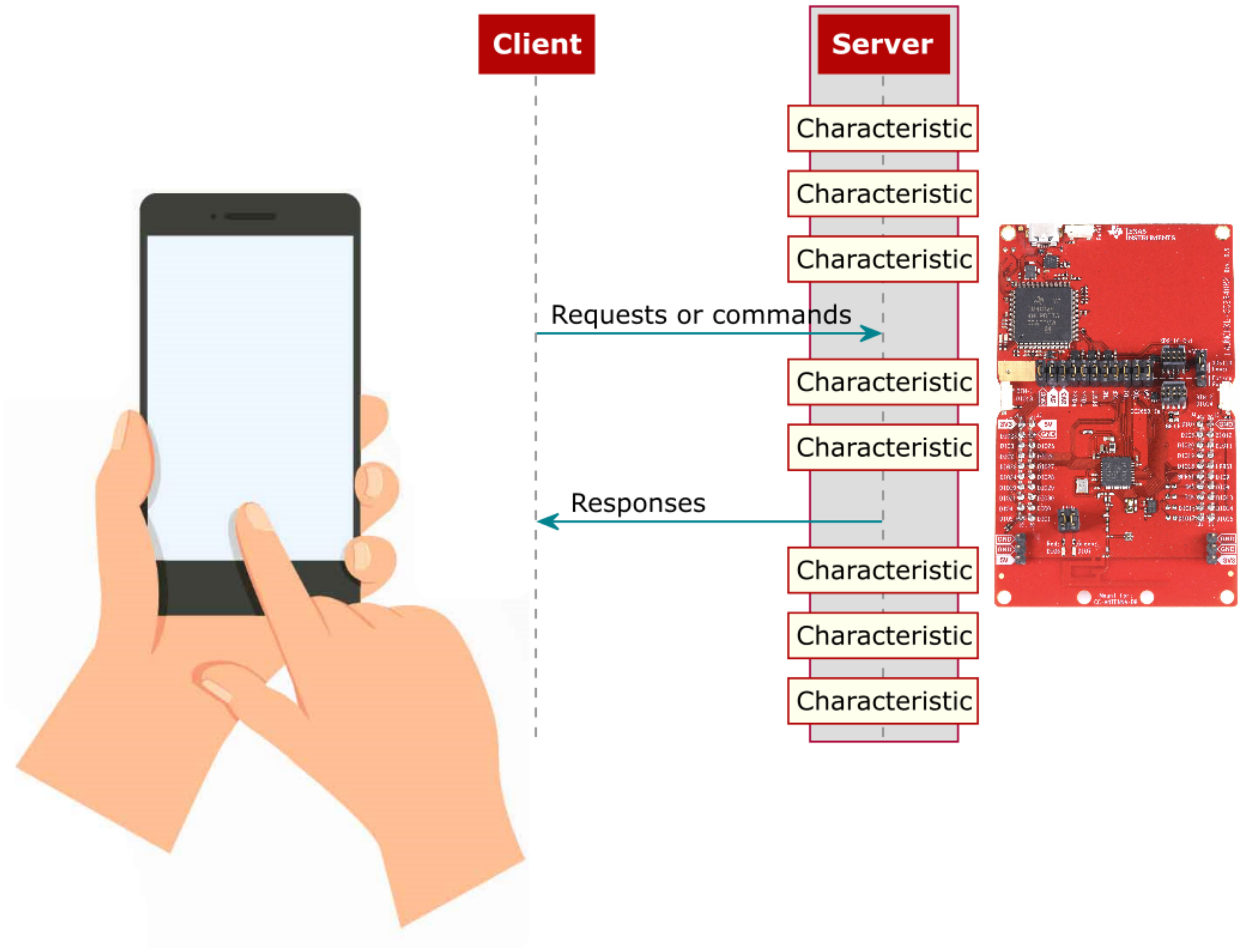


Figure 45. GATT Client and Server Interaction Overview

The GATT roles of client and server are independent from the GAP roles of peripheral and central. A peripheral can be either a GATT client or a GATT server, and a central can be either a GATT client or a GATT server. A peripheral can act as both a GATT client and a GATT server. For a hands-on review of GATT services and characteristics, see SimpleLink Academy.

GATT Characteristics and Attributes

While characteristics and attributes are sometimes used interchangeably when referring to Bluetooth low energy, consider characteristics as groups of information called attributes. Attributes are the information actually transferred between devices. Characteristics organize and use attributes as data values, properties, and configuration information. A typical characteristic is composed of the following attributes.

- **Characteristic Value**
  data value of the characteristic
- **Characteristic Declaration**
  descriptor storing the properties, location, and type of the characteristic value

- **Client Characteristic Configuration**
  a configuration that allows the GATT server to configure the characteristic to be notified (send message asynchronously) or indicated (send message asynchronously with acknowledgment)

- **Characteristic User Description**
  an ASCII string describing the characteristic

These attributes are stored in the GATT server in an attribute table. In addition to the value, the following properties are associated with each attribute.

- **Handle**
  the index of the attribute in the table (Every attribute has a unique handle.)

- **Type**
  indicates what the attribute data represents (referred to as a UUID [universal unique identifier]. Some of these are Bluetooth SIG-defined and some are custom.)

- **Permissions**
  enforces if and how a GATT client device can access the value of an attribute

GATT Client Abstraction

Like the GAP layer, the GATT layer is also abstracted. This abstraction depends on whether the device is acting as a GATT client or a GATT server. As defined by the Bluetooth Core Specification Version 4.2, the GATT layer is an abstraction of the ATT layer.

GATT clients do not have attribute tables or profiles as they are gathering, not serving, information. Most of the interfacing with the GATT layer occurs directly from the application.

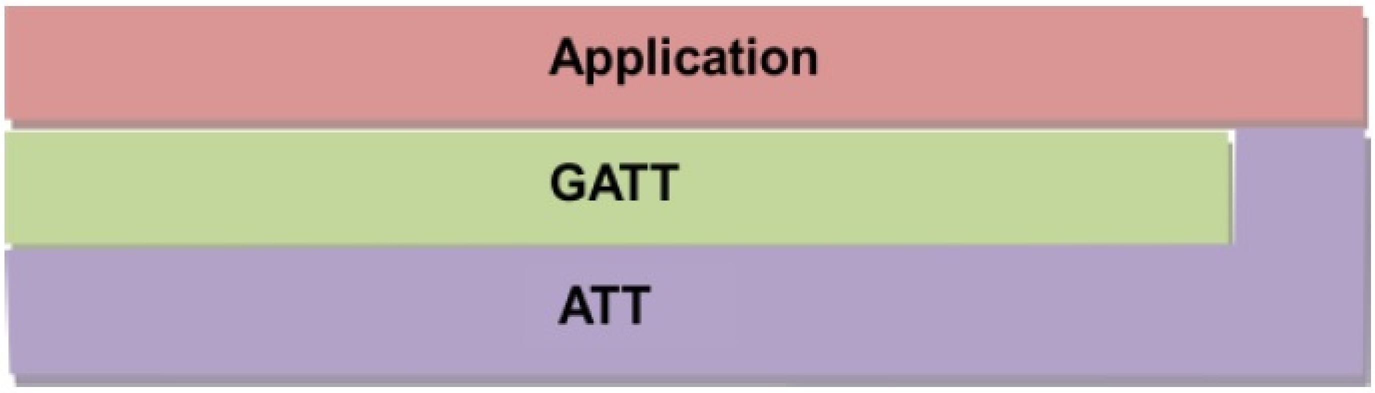


Figure 46. Visualization of GATT Client Abstraction.

GATT Server Abstraction

As a GATT server, most of the GATT functionality is handled by the individual GATT profiles. These profiles use the GATTServApp (see BLE Stack API Reference, GATTServApp Section) (a configurable module that stores and manages the attribute table). Figure 47. shows this abstraction hierarchy.

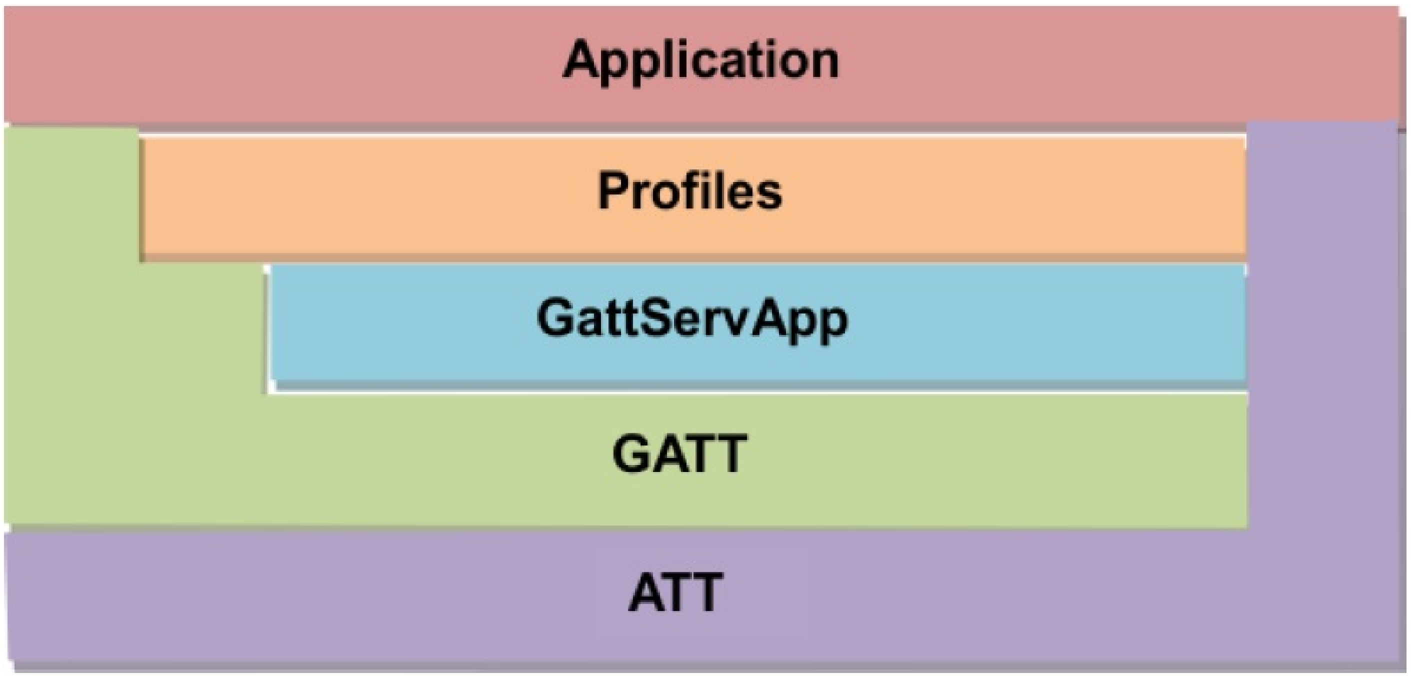


Figure 47. Visualization of GATT Server abstraction.

The design process involves creating GATT profiles that configure the GATTServApp module and use its API to interface with the GATT layer. In this case of a GATT server, direct calls to GATT layer functions are unnecessary. The application then interfaces with the profiles.

GATT Services and Profile

A GATT service is a collection of characteristics. For example, the heart rate service contains a heart rate measurement characteristic and a body location characteristic, among others. Multiple services can be grouped together to form a profile. Many profiles only implement one service so the two terms are sometimes used interchangeably.

> **Note**
>
> TI intends this section as an introduction to the attribute table by using simple_peripheral as an example. For information on how this profile is implemented within the stack, see GATT Server Abstraction.

There are four GATT profiles defined in the simple_peripheral example application project.

- **GAP GATT Service (GGS)**

  This service contains device and access information such as the device name, vendor identification, and product identification.

The following characteristics are defined for this service:

- Device name
- Appearance
- Peripheral preferred connection parameters

Note

See the Gap Service and Characteristics for GATT Server section ([Vol. 3], Part C, Section 12) of the Bluetooth Core Specification Version 4.2 for more information on these characteristics.

- Generic Attribute Service

  This service contains information about the GATT server, is a part of the Bluetooth low energy protocol stack, and is required for every GATT server device as per the Bluetooth Core Specification Version 4.2.

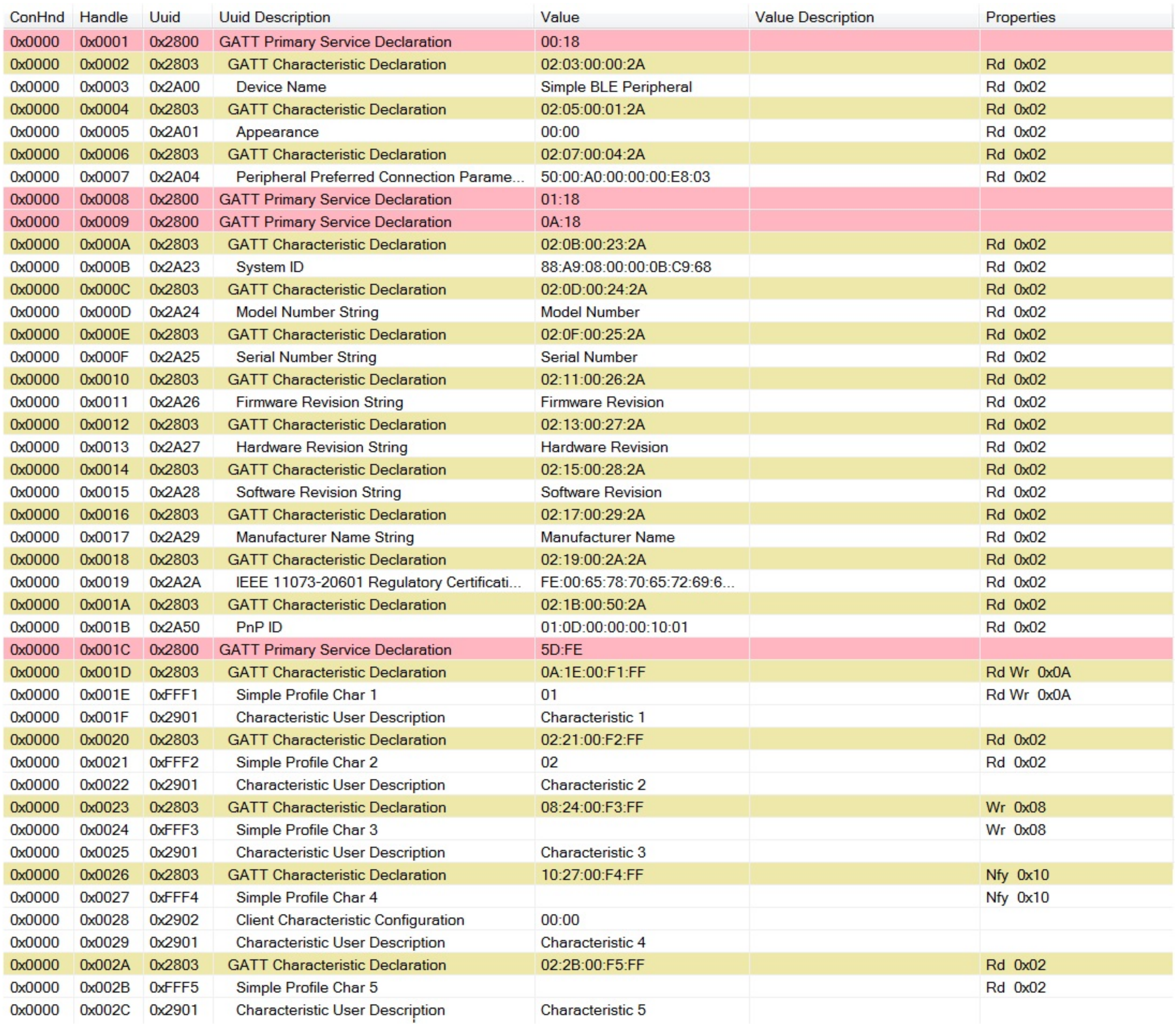
- Device Info Service

  This service exposes information about the device such as the hardware, software version, firmware version, regulatory information, compliance information, and manufacturer name. The Device Info Service is part of the Bluetooth low energy protocol stack and configured by the application.

- simple_gatt_profile Service

  This service is a sample profile for testing and for demonstration. The full source code is provided in the simple_gatt_profile.c and simple_gatt_profile.h files.

Figure 48. shows the attribute table in the simple_peripheral project.



ConHnd	Handle	Uuid	Uuid Description	Value	Value Description	Properties
0x0000	0x0001	0x2800	GATT Primary Service Declaration	00:18		
0x0000	0x0002	0x2803	GATT Characteristic Declaration	02:03:00:00:2A		Rd 0x02
0x0000	0x0003	0x2A00	Device Name	Simple BLE Peripheral		Rd 0x02
0x0000	0x0004	0x2803	GATT Characteristic Declaration	02:05:00:01:2A		Rd 0x02
0x0000	0x0005	0x2A01	Appearance	00:00		Rd 0x02
0x0000	0x0006	0x2803	GATT Characteristic Declaration	02:07:00:04:2A		Rd 0x02
0x0000	0x0007	0x2A04	Peripheral Preferred Connection Parame...	50:00:A0:00:00:00:E8:03		Rd 0x02
0x0000	0x0008	0x2800	GATT Primary Service Declaration	01:18		
0x0000	0x0009	0x2800	GATT Primary Service Declaration	0A:18		
0x0000	0x000A	0x2803	GATT Characteristic Declaration	02:0B:00:23:2A		Rd 0x02
0x0000	0x000B	0x2A23	System ID	88:A9:08:00:00:0B:C9:68		Rd 0x02
0x0000	0x000C	0x2803	GATT Characteristic Declaration	02:0D:00:24:2A		Rd 0x02
0x0000	0x000D	0x2A24	Model Number String	Model Number		Rd 0x02
0x0000	0x000E	0x2803	GATT Characteristic Declaration	02:0F:00:25:2A		Rd 0x02
0x0000	0x000F	0x2A25	Serial Number String	Serial Number		Rd 0x02
0x0000	0x0010	0x2803	GATT Characteristic Declaration	02:11:00:26:2A		Rd 0x02
0x0000	0x0011	0x2A26	Firmware Revision String	Firmware Revision		Rd 0x02
0x0000	0x0012	0x2803	GATT Characteristic Declaration	02:13:00:27:2A		Rd 0x02
0x0000	0x0013	0x2A27	Hardware Revision String	Hardware Revision		Rd 0x02
0x0000	0x0014	0x2803	GATT Characteristic Declaration	02:15:00:28:2A		Rd 0x02
0x0000	0x0015	0x2A28	Software Revision String	Software Revision		Rd 0x02
0x0000	0x0016	0x2803	GATT Characteristic Declaration	02:17:00:29:2A		Rd 0x02
0x0000	0x0017	0x2A29	Manufacturer Name String	Manufacturer Name		Rd 0x02
0x0000	0x0018	0x2803	GATT Characteristic Declaration	02:19:00:2A:2A		Rd 0x02
0x0000	0x0019	0x2A2A	IEEE 11073-20601 Regulatory Certificati...	FE:00:65:78:70:65:72:69:6...		Rd 0x02
0x0000	0x001A	0x2803	GATT Characteristic Declaration	02:1B:00:50:2A		Rd 0x02
0x0000	0x001B	0x2A50	PnP ID	01:0D:00:00:00:10:01		Rd 0x02
0x0000	0x001C	0x2800	GATT Primary Service Declaration	5D:FE		
0x0000	0x001D	0x2803	GATT Characteristic Declaration	0A:1E:00:F1:FF		Rd Wr 0x0A
0x0000	0x001E	0xFFF1	Simple Profile Char 1	01		Rd Wr 0x0A
0x0000	0x001F	0x2901	Characteristic User Description	Characteristic 1		
0x0000	0x0020	0x2803	GATT Characteristic Declaration	02:21:00:F2:FF		Rd 0x02
0x0000	0x0021	0xFFF2	Simple Profile Char 2	02		Rd 0x02
0x0000	0x0022	0x2901	Characteristic User Description	Characteristic 2		
0x0000	0x0023	0x2803	GATT Characteristic Declaration	08:24:00:F3:FF		Wr 0x08
0x0000	0x0024	0xFFF3	Simple Profile Char 3			Wr 0x08
0x0000	0x0025	0x2901	Characteristic User Description	Characteristic 3		
0x0000	0x0026	0x2803	GATT Characteristic Declaration	10:27:00:F4:FF		Nfy 0x10
0x0000	0x0027	0xFFF4	Simple Profile Char 4			Nfy 0x10
0x0000	0x0028	0x2902	Client Characteristic Configuration	00:00		
0x0000	0x0029	0x2901	Characteristic User Description	Characteristic 4		
0x0000	0x002A	0x2803	GATT Characteristic Declaration	02:2B:00:F5:FF		Rd 0x02
0x0000	0x002B	0xFFF5	Simple Profile Char 5			Rd 0x02
0x0000	0x002C	0x2901	Characteristic User Description	Characteristic 5		

Figure 48. Simple GATT Profile Characteristic Table taken with BTool. Red indicates a Profile declaration, Yellow indicates character declaration, and White indicates Attributes related to a particular characteristic declaration.

The simple_gatt_profile contains the following characteristics:

- **SIMPLEPROFILE_CHAR1**
  1-byte value that can be read or written from a GATT client device
- **SIMPLEPROFILE_CHAR2**
  1-byte value that can be read from a GATT client device but cannot be written
- **SIMPLEPROFILE_CHAR3**
  1-byte value that can be written from a GATT client device but cannot be read
- **SIMPLEPROFILE_CHAR4**
  1-byte value that cannot be directly read or written from a GATT client device (This value is notifiable: This value can be configured for notifications to be sent to a GATT client device.)

- **SIMPLEPROFILE_CHAR5**
  5-byte value that can be read (but not written) from a GATT client device

The following is a line-by-line description of the simple profile attribute table, referenced by the following handle.

- **0x001C is the simple_gatt_profile service declaration.**
  This declaration has a UUID of 0x2800 (Bluetooth-defined `GATT_PRIMARY_SERVICE_UUID`). The value of this declaration is the UUID of the simple_gatt_profile (custom-defined).

- **0x001D is the SimpleProfileChar1 characteristic declaration.**
  This declaration can be thought of as a pointer to the SimpleProfileChar1 value. The declaration has a UUID of 0x2803 (Bluetooth-defined `GATT_CHARACTER_UUID`). The value of the declaration characteristic, as well as all other characteristic declarations, is a 5-byte value explained here (from MSB to LSB):

  - Byte 0 is the properties of the SimpleProfileChar1 as defined in the Bluetooth Core Specification Version 4.2 (The following are some of the relevant properties.)
    - 0x02: permits reads of the characteristic value
    - 0x04: permits writes of the characteristic value (without a response)
    - 0x08: permits writes of the characteristic value (with a response)
    - 0x10: permits of notifications of the characteristic value (without acknowledgment)
    - 0x20: permits notifications of the characteristic value (with acknowledgment)

  The value of 0x0A means the characteristic is readable (0x02) and writeable (0x08).

  - Bytes 1-2: the byte-reversed handle where the SimpleProfileChar1's value is (handle 0x001E)
  - Bytes 3-4: the UUID of the SimpleProfileChar1 value (custom-defined 0xFFF1)

- **0x001E is the SimpleProfileChar1 Characteristic Value**
  This value has a UUID of 0xFFF1 (custom-defined). This value is the actual payload data of the characteristic. As indicated by its characteristic declaration (handle 0x01D), this value is readable and writable.

- **0x001F is the SimpleProfileChar1 Characteristic User Description**
  This description has a UUID of 0x2901 (Bluetooth-defined). The value of this description is a user-readable string describing the characteristic.

- **0x0020 - 0x002C**
  are attributes that follow the same structure as the simpleProfileChar1 described previously with regard to the remaining four characteristics. The only different attribute, handle 0x0028, is described as follows.

0x0028 is the SimpleProfileChar4 Client Characteristic Configuration. This configuration has a UUID of 0x2902 (Bluetooth-defined). By writing to this attribute, a GATT server can configure the SimpleProfileChar4 for notifications (writing 0x0001) or indications (writing 0x0002). Writing 0x0000 to this attribute disable notifications and indications.

GATT Security

As described in GATT Server Abstraction, the GATT server may define permissions independently for each characteristic. The server may allow some characteristics to be accessed by any client, while limiting access to other characteristics to only authenticated or authorized clients. These permissions are usually defined as part of a higher level profile specification. For custom profiles, the user may select the permissions as they see fit. For more information about the GATT Security, refer to the Security Considerations section ([Vol 3], Part G, Section 8) of the Bluetooth Core Specification Version 4.2.

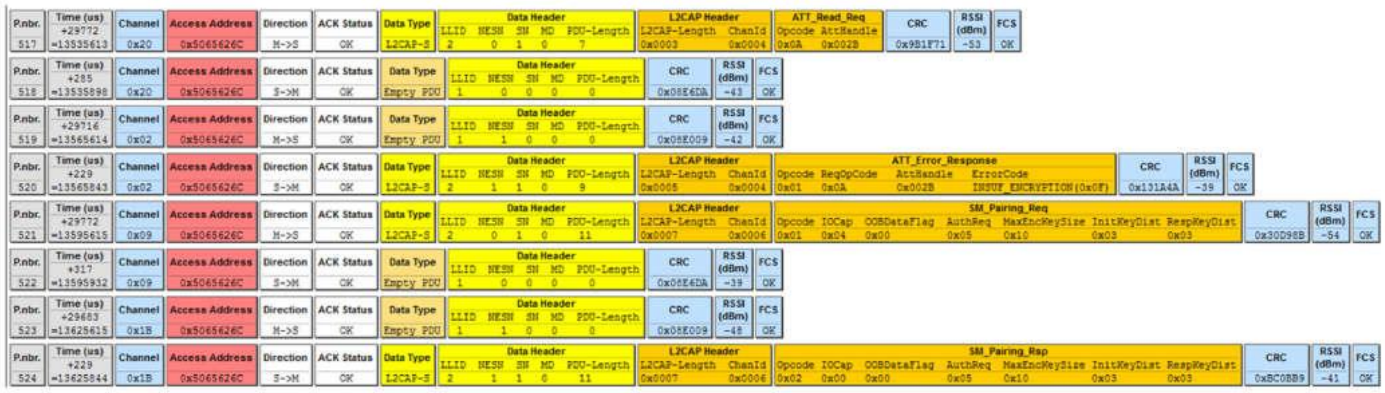
Authentication

Characteristics that require authentication cannot be accessed until the client has gone through an authenticated pairing method. This verification is performed within the stack, with no processing required by the application. The only requirement is for the characteristic to be registered properly with the GATT server.

For example, characteristic 5 of the simple_gatt_profile allows on authenticated reads.

```
// Characteristic Value 5
{
   { ATT_BT_UUID_SIZE, simpleProfilechar5UUID },
   GATT_PERMIT_AUTHEN_READ,
   0,
   simpleProfileChar5
},
```

When an un-authenticated client attempts to read this value, the GATT server automatically rejects it with `ERROR_INSUFFICIENT_AUTHEN (0x41)`, without invoking the simpleProfile_ReadAttrCB(). See an example of this in Sniffer Capture Example.



```
P.nbr. | Time (us) +29772 =13535613 | Channel 0x20 | Access Address 0x5065626C | Direction M->S | ACK Status OK | Data Type L2CAP-S
  Data Header: LLID 2  NESN 0  SN 1  MD 0  PDU-Length 7
  L2CAP Header: L2CAP-Length 0x0003  ChanId 0x0004
  ATT_Read_Req: Opcode 0x0A  AttHandle 0x002B
  CRC 0x9B1F71 | RSSI (dBm) -53 | FCS OK
517

P.nbr. | Time (us) +285 =13535898 | Channel 0x20 | Access Address 0x5065626C | Direction S->M | ACK Status OK | Data Type Empty PDU
  Data Header: LLID 1  NESN 0  SN 0  MD 0  PDU-Length 0
  CRC 0x08E6DA | RSSI (dBm) -43 | FCS OK
518

P.nbr. | Time (us) +29716 =13565614 | Channel 0x02 | Access Address 0x5065626C | Direction M->S | ACK Status OK | Data Type Empty PDU
  Data Header: LLID 1  NESN 1  SN 0  MD 0  PDU-Length 0
  CRC 0x08E009 | RSSI (dBm) -42 | FCS OK
519

P.nbr. | Time (us) +229 =13565843 | Channel 0x02 | Access Address 0x5065626C | Direction S->M | ACK Status OK | Data Type L2CAP-S
  Data Header: LLID 2  NESN 1  SN 1  MD 0  PDU-Length 9
  L2CAP Header: L2CAP-Length 0x0005  ChanId 0x0004
  ATT_Error_Response: Opcode 0x01  ReqOpCode 0x0A  AttHandle 0x002B  ErrorCode INSUF_ENCRYPTION(0x0F)
  CRC 0x131A4A | RSSI (dBm) -39 | FCS OK
520

P.nbr. | Time (us) +29772 =13595615 | Channel 0x09 | Access Address 0x5065626C | Direction M->S | ACK Status OK | Data Type L2CAP-S
  Data Header: LLID 2  NESN 0  SN 1  MD 0  PDU-Length 11
  L2CAP Header: L2CAP-Length 0x0007  ChanId 0x0006
  SM_Pairing_Req: Opcode 0x01  IOCap 0x04  OOBDataFlag 0x00  AuthReq 0x05  MaxEncKeySize 0x10  InitKeyDist 0x03  RespKeyDist 0x03
  CRC 0x30D98B | RSSI (dBm) -54 | FCS OK
521

P.nbr. | Time (us) +317 =13595932 | Channel 0x09 | Access Address 0x5065626C | Direction S->M | ACK Status OK | Data Type Empty PDU
  Data Header: LLID 1  NESN 0  SN 0  MD 0  PDU-Length 0
  CRC 0x08E6DA | RSSI (dBm) -39 | FCS OK
522

P.nbr. | Time (us) +29683 =13625615 | Channel 0x1B | Access Address 0x5065626C | Direction M->S | ACK Status OK | Data Type Empty PDU
  Data Header: LLID 1  NESN 1  SN 0  MD 0  PDU-Length 0
  CRC 0x08E009 | RSSI (dBm) -48 | FCS OK
523

P.nbr. | Time (us) +229 =13625844 | Channel 0x1B | Access Address 0x5065626C | Direction S->M | ACK Status OK | Data Type L2CAP-S
  Data Header: LLID 2  NESN 1  SN 1  MD 0  PDU-Length 11
  L2CAP Header: L2CAP-Length 0x0007  ChanId 0x0006
  SM_Pairing_Rsp: Opcode 0x02  IOCap 0x00  OOBDataFlag 0x00  AuthReq 0x05  MaxEncKeySize 0x10  InitKeyDist 0x03  RespKeyDist 0x03
  CRC 0xBC0BB9 | RSSI (dBm) -41 | FCS OK
524
```

Figure 49. Sniffer Capture Example

After the client has successfully authenticated, read/write requests are forwarded to the profiles read/write callback. See the code below for a simple_gatt_profile example:

```
case SIMPLEPROFILE_CHAR5_UUID:
*pLen = SIMPLEPROFILE_CHAR5_LEN;
VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR5_LEN );
break;
```

Authorization

Authorization is a layer of security provided in addition to what BLE already implements. Because applications are required to define their own requirements for authorization, the stack forwards read/write requests on these characteristics to the application layer of the profile.

For the profile to register for authorization information from the GATT server, it must define an authorization callback with the stack. The simple_gatt_profile does not do this by default, but below is an example of how it could be modified to do this.

1. Register profile level authorization callback.

```
CONST gattServiceCBs_t simpleProfileCBs =
{
   simpleProfile_ReadAttrCB,      // Read callback function pointer
   simpleProfile_WriteAttrCB,     // Write callback function pointer
   simpleProfile_authorizationCB  // Authorization callback function pointer
};
```

2. Implement authorization callback code.

```
static bStatus_t simpleProfile_authorizationCB( uint16 connHandle, gattAttribute_t *pAttr, uint8
opcode )
{
   //This is just an example implementation, normal use cases would require
   //more complex logic to determine that the device is authorized

   if(clientIsAuthorized)
      return SUCCESS;
   else
      return ATT_ERR_INSUFFICIENT_AUTHOR;
}
```

The authorization callback executes in the stack context, thus intensive processing should not be performed in this function. The implementation is left up to the developer; the above callback should be treated as a shell. The return value should be either `SUCCESS` if the client is authorized to access the characteristic, or `ATT_ERR_INSUFFICIENT_AUTHOR` if they have not yet obtained proper authorization. Authorization requires the connection to be authenticated beforehand, or `ATT_ERR_INSUFFICIENT_AUTHEN` will be sent as an error response.

If a characteristic that requires authorization is registered with the GATT server, but no application level authorization callback is defined, the stack will return `ATT_ERR_UNLIKELY`. Because this error can be cryptic, TI recommends using an authorization callback.

Using the GATT Layer Directly

This section describes how to use the GATT layer in an application. The functionality of the GATT layer is implemented in the library but header functions can be found in the gatt.h file. The BLE Stack API Reference within the ATT/GATT section has the complete API for the GATT layer. The Bluetooth Core Specification Version 4.2 provides more information on the functionality of these commands. These functions are used primarily for GATT client applications. A few server-specific functions are described in the API. Most of the GATT functions returns ATT events to the application, review BLE Stack API Reference for details.

The general procedure to use the GATT layer when functioning as a GATT client (in the simple_central project) is as follows:

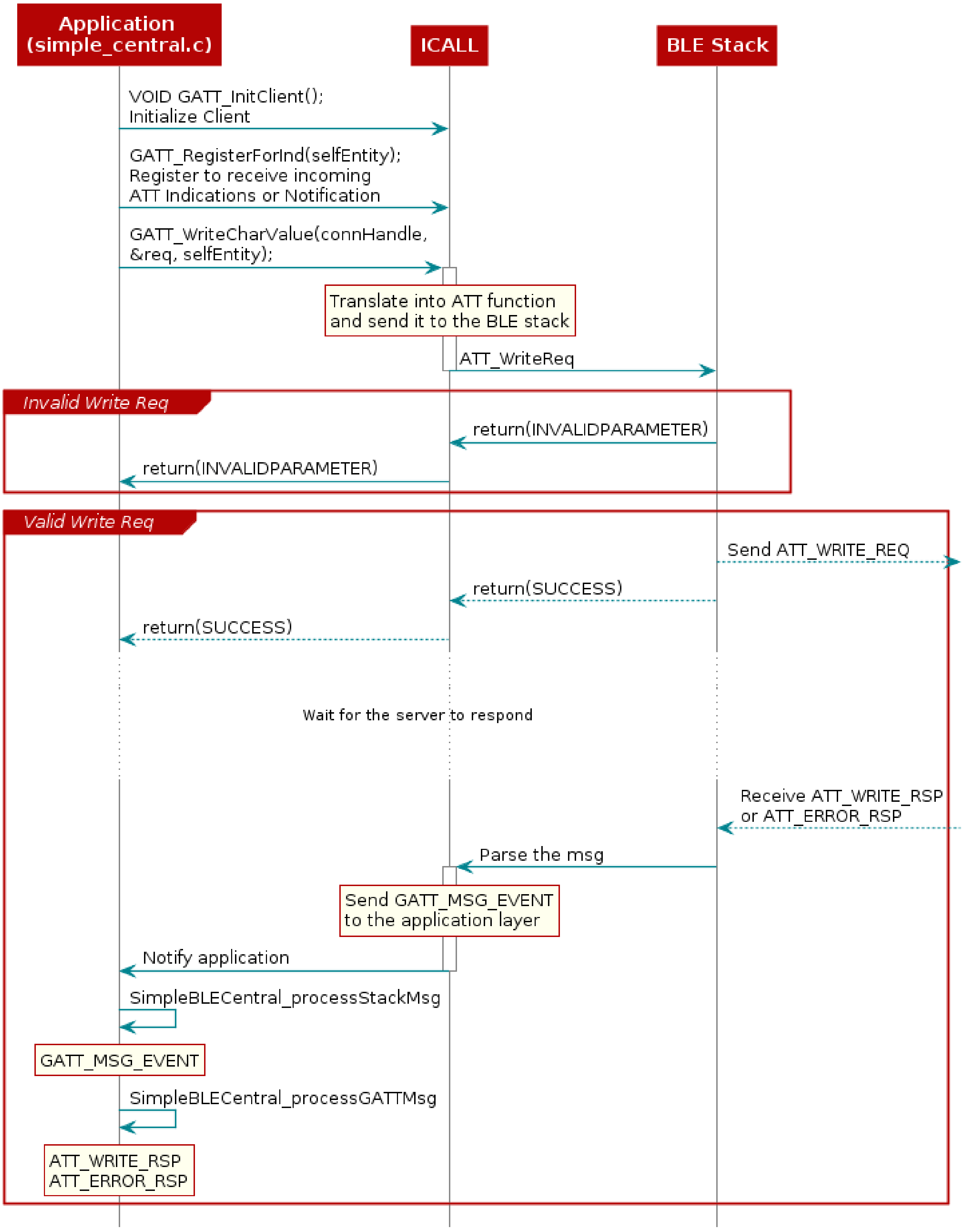

Application (simple_central.c)
ICALL
BLE Stack
VOID GATT_InitClient(); Initialize Client
GATT_RegisterForInd(selfEntity); Register to receive incoming ATT Indications or Notification
GATT_WriteCharValue(connHandle, &req, selfEntity);
Translate into ATT function and send it to the BLE stack
ATT_WriteReq
Invalid Write Req
return(INVALIDPARAMETER)
return(INVALIDPARAMETER)
Valid Write Req
Send ATT_WRITE_REQ
return(SUCCESS)
return(SUCCESS)
Wait for the server to respond
Receive ATT_WRITE_RSP or ATT_ERROR_RSP
Parse the msg
Send GATT_MSG_EVENT to the application layer
Notify application
SimpleBLECentral_processStackMsg
GATT_MSG_EVENT
SimpleBLECentral_processGATTMsg
ATT_WRITE_RSP
ATT_ERROR_RSP

Figure 50. Context Diagram of Application using GATT layer.

Note

Even though the event sent to the application is an ATT event, it is sent as a GATT protocol stack message (`GATT_MSG_EVENT`).

Note

In addition to receiving responses to its own commands, a GATT client may also receive asynchronous data from the GATT server as indications or notifications. Use (`GATT_RegisterForInd(selfEntity)`) to receive these ATT notifications and indications. These notifications and indications are also sent as ATT events (`ATT_HANDLE_VALUE_NOTI` & `ATT_HANDLE_VALUE_IND`) in GATT messages to the application. These events must be handled as described in GATT Services and Profile.

GAP GATT Service (GGS)

The GAP GATT Service (GGS) is required for low-energy devices that implement the **central** or **peripheral** role. Multirole devices that implement either of these roles **must**also contain the GGS. The purpose of the GGS is to aide in the device discovery and connection initiation process.

Note

For more information about the GGS, refer to the GAP Service and and Characteristics for GATT Server section ([Vol 3], Part C, Section 12) of the Bluetooth Core Specification Version 4.2.

Using the GGS

This section describes what the application must do to configure, start, and use the GAP Gatt Service. The GGS is implemented as part of the Bluetooth Low Energy library code, the API can be found in GATTServApp. ATT/GATT describes the full API, including commands, configurable parameters, events, and callbacks.

To review the above mentioned APIs, please see their respective sections in BLE Stack API Reference.

1. Include header

```
#include "gapgattserver.h"
```

2. Initialize GGS parameters.

```
// GAP GATT Attributes
static uint8_t attDeviceName[GAP_DEVICE_NAME_LEN] = "Simple BLE Peripheral";

GGS_SetParameter(GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, attDeviceName);
```

3. Initialize application callbacks with GGS (optional). This notifies the application when any of the characteristics in the GGS have changed.

```
GGS_RegisterAppCBs(&appGGSCBs);
```

4. Add the GGS to the GATT server.

```
bStatus_t GGS_AddService(GATT_ALL_SERVICES);
```

Generic Attribute Profile Service (GATT Service)

The Generic Attribute Profile (GATT) Service provides information about the GATT services registered with a device.

Note

For more information on the GATT Service, refer to the Defined Generic Attribute Profile Service section ([Vol 3], Part G, Section 7) of the Bluetooth Core Specification Version 4.2.

The service changed characteristic is used to inform bonded devices that services have changed on the server upon reconnection. Service changed updates are sent in the form of GATT indications, and the service changed characteristic is not writeable or readable. In the TI Bluetooth low energy stack, the service changed characteristic is implemented as part of the gattservapp, which is part of library code.

Per the Bluetooth Core Specification Version 4.2, it is safe for server devices whose characteristic tables do not change over their lifetime to exclude the service changed characteristic. Support for indications from this characteristic must be supported on GATT client devices.

Using the GATT Service

This section describes what the user must do to enable the GATT service changed feature in the Bluetooth low energy stack. Once the service changed feature is enabled, the GAPBondMgr will handle sending the service changed indication to clients who have enabled it using the CCCD.

1. Use a supported build config for the stack; only stack libraries with v4.1 features and L2CAP connection-oriented channels will support the service changed characteristic.

   Enable this feature with the project’s build_config.opt file, by uncommenting the following line:

   ```
   -DBLE_V41_FEATURES=L2CAP_COC_CFG+V41_CTRL_CFG
   ```

2. From this point, the GAPBondMgr handles sending an indication to connected clients when the service has changed and the CCCD is enabled. If the feature is enabled, the peripheral role invokes the necessary APIs to send the indication through the GAPBondMgr.
3. On the client side, service changed indications can be registered using the same method as registering for other indications.

GATTServApp Module

The GATT Server Application (GATTServApp) stores and manages the application-wide attribute table. Various profiles use this module to add their characteristics to the attribute table. The Bluetooth low energy stack uses this module to respond to discovery requests from a GATT client. For example, a GATT client may send a **Discover all Primary Characteristics** message. The Bluetooth low energy stack on the GATT server side receives this message and uses the GATTServApp to find and send over-the-air all of the primary characteristics stored in the attribute table. This type of functionality is beyond the scope of this document and is implemented in the library code. The GATTServApp functions accessible from the profiles are defined in ‘’gattservapp_util.c’’ and the API described in BLE Stack API Reference (GATTServApp section). These functions include finding specific attributes and reading or modifying client characteristic configurations. See Figure 51. for an example of how GATTServApp functions in an application.

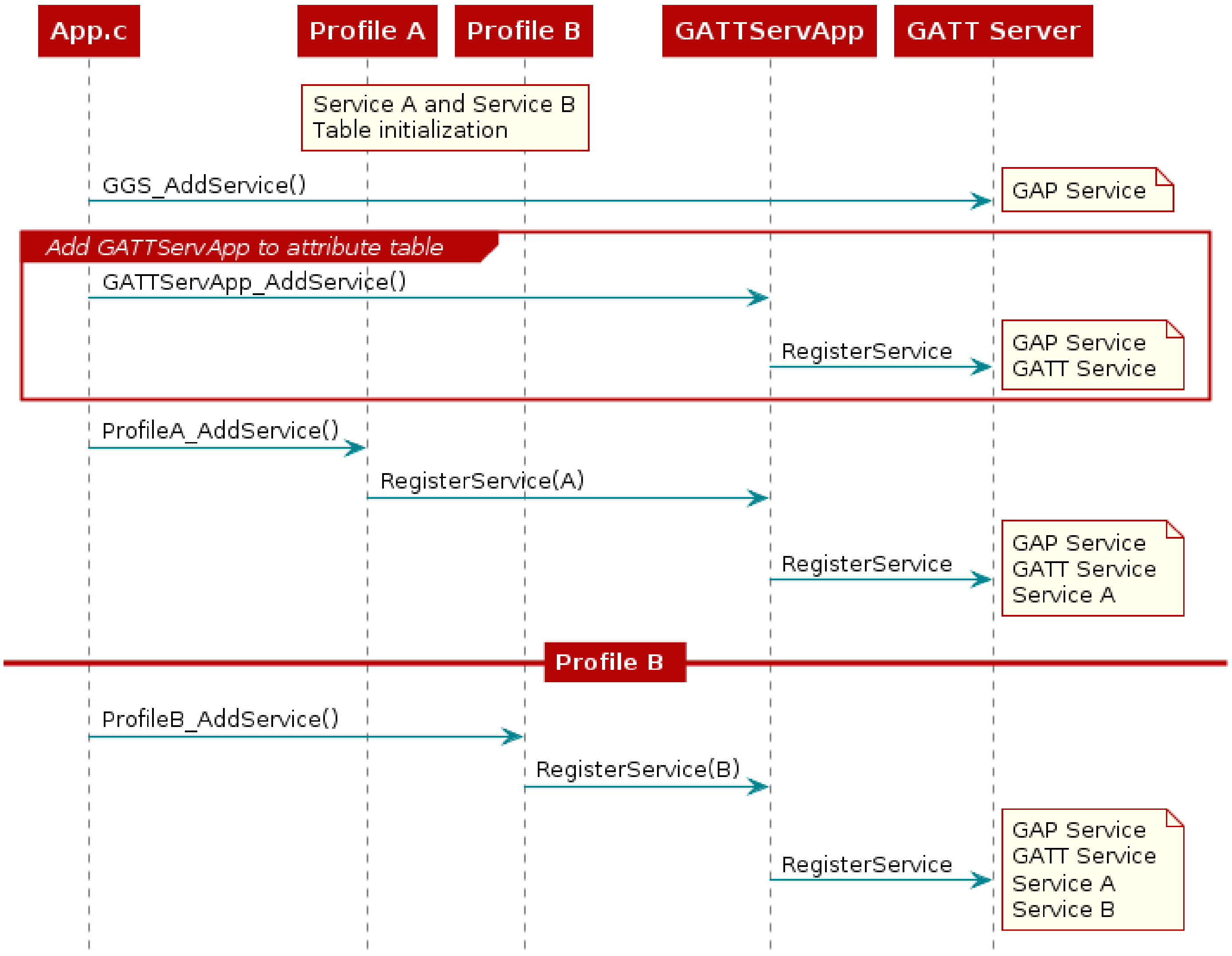


Figure 51. Attribute Table Initialization Diagram.

Building up the Attribute Table

Upon power-on or reset, the application builds the GATT table by using the GATTServApp to add services. Each service consists of a list of attributes with UUIDs, values, permissions, and read and write call-backs. As Figure 51. shows, all of this information is passed through the GATTServApp to GATT and stored in the stack.

Attribute table initialization must occur in the application initialization function, that is, simple_peripheral_init().

```
// Initialize GATT attributes
GGS_AddService(GATT_ALL_SERVICES);         // GAP
GATTServApp_AddService(GATT_ALL_SERVICES); // GATT attributes
```

Implementing Profiles in Attributes Table

This section describes the general architecture for implementing profiles and provides specific functional examples in relation to the simple_gatt_profile in the simple_peripheral project. See GATT Services and Profile for an overview of the simple_gatt_profile.

Attribute Table Definition

Each service or group of GATT attributes must define a fixed size attribute table that gets passed into GATT. This table in simple_gatt_profile.c is defined as the following.

```
static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED]
```

Each attribute in this table is of the following type.

```
typedef struct attAttribute_t
{
   gattAttrType_t type; //!< Attribute type (2 or 16 octet UUIDs)
   uint8 permissions; //!< Attribute permissions

   uint16 handle; //!< Attribute handle - assigned internally by attribute server

   uint8* const pValue; //!< Attribute value - encoding of the octet array is defined in
                        //!< the applicable profile. The maximum length of an attribute
                        //!< value shall be 512 octets.
} gattAttribute_t;
```

When utilizing gattAttribute_t, the various fields have specific meanings.

- **gattAttrType_t type**

  type is the UUID associated with the attribute being placed into the table. gattAttrType_t itself is defined as:

  ```
  typedef struct
  {
     uint8 len;         //!< Length of UUID (2 or 16)
     const uint8 *uuid; //!<Pointer to UUID
  } gattAttrType_t;
  ```

  Where length can be either `ATT_BT_UUID_SIZE` (2 bytes), or `ATT_UUID_SIZE` (16 bytes). The *uuid is a pointer to a number either reserved by Bluetooth SIG (defined in gatt_uuid.c) or a custom UUID defined in the profile.

- **uint8 permissions**

  enforces how and if a GATT client device can access the value of the attribute. Possible permissions are defined in gatt.h as follows:

#define GATT_PERMIT_READ	0x01	//!< Attribute is Readable
#define GATT_PERMIT_WRITE	0x02	//!< Attribute is Writable
#define GATT_PERMIT_AUTHEN_READ	0x04	//!< Read requires Authentication
#define GATT_PERMIT_AUTHEN_WRITE	0x08	//!< Write requires Authentication
#define GATT_PERMIT_AUTHOR_READ	0x10	//!< Read requires Authorization
#define GATT_PERMIT_AUTHOR_WRITE	0x20	//!< Write requires Authorization
#define GATT_PERMIT_ENCRYPT_READ	0x40	//!< Read requires Encryption
#define GATT_PERMIT_ENCRYPT_WRITE	0x80	//!< Write requires Encryption

Allocating Memory for GATT Procedures further describes authentication, authorization, and encryption.

- **uint16 handle**
  handle is a placeholder in the table where GATTServApp assigns a handle. This placeholder is not customizable. Handles are assigned sequentially.

- **uint8* const pValue**
  pValue is a pointer to the attribute value. The size is unable to change after initialization. The maximum size is 512 octets.

Service Declaration

Consider the following simple_gatt_profile service declaration attribute:

```
// Simple Profile Service
{
   { ATT_BT_UUID_SIZE, primaryServiceUUID },    // type
   GATT_PERMIT_READ,                            // permissions
   0,                                           // handle
   (uint8 *)&simpleProfileService               // pValue
}
```

The type is set to the Bluetooth SIG-defined primary service UUID (0x2800). A GATT client is permitted to read this service with the permission is set to `GATT_PERMIT_READ`. The pValue is a pointer to the UUID of the service, custom-defined as 0xFFF0. SimpleProfileService itself is defined to reference the profile's UUID.

```
// Simple Profile Service attribute
static CONST gattAttrType_t simpleProfileService =
   {
      ATT_BT_UUID_SIZE,
      simpleProfileServUUID
   };
```

Characteristic Declaration

Consider the following simple_gatt_profile simpleProfileCharacteristic1 declaration.

```
// Characteristic 1 Declaration
{
   { ATT_BT_UUID_SIZE, characterUUID },
   GATT_PERMIT_READ,
   0,
   &simpleProfileChar1Props
},
```

The type is set to the Bluetooth SIG-defined characteristic UUID (0x2803). This makes simpleProfileCharacteristic1 read only to the GATT client with the permissions set to `GATT_PERMIT_READ`.

For functional purposes, the only information required to be passed to the GATTServApp in pValue is a pointer to the properties of the characteristic value. The GATTServApp adds the UUID and the handle of the value. These properties are defined to allow this particular characteristic in this service to be read and written to.

```
// Simple Profile Characteristic 1 Properties
static uint8 simpleProfileChar1Props = GATT_PROP_READ | GATT_PROP_WRITE;
```

Note

An important distinction exists between these properties and the GATT permissions of the characteristic value. These properties are visible to the GATT client stating the properties of the characteristic value. The GATT permissions of the characteristic value affect its functionality in the protocol stack. These properties must match that of the GATT permissions of the characteristic value.

Characteristic Value

Consider the simple_gatt_profile simpleProfileCharacteristic1 value.

```
// Characteristic Value 1
{
{ ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
  GATT_PERMIT_READ | GATT_PERMIT_WRITE,
  0,
  &simpleProfileChar1
},
```

The type is set to the custom-defined simpleProfilechar1 UUID (0xFFF1). The properties of the characteristic value in the attribute table must match the properties from the characteristic value declaration. The pValue is a pointer to the location of the actual value, statically defined in the profile as follows.

```
// Characteristic 1 Value
static uint8 simpleProfileChar1 = 0;
```

Client Characteristic Configuration

Consider the simple_gatt_profile simpleProfileCharacteristic4 configuration.

```
// Characteristic 4 configuration
{
{ ATT_BT_UUID_SIZE, clientCharCfgUuID },
  GATT_PERMIT_READ | GATT_PERMIT_WRITE,
  0,
  (uint8 *)&simpleProfileChar4Config
 },
```

The type is set to the Bluetooth SIG-defined client characteristic configuration UUID (0x2902) GATT clients must read and write to this attribute so the GATT permissions are set to readable and writable. The pValue is a pointer to the location of the client characteristic configuration array, defined in the profile as follows.

Listing 62. simple_gatt_profile characteristic 4 pValue pointer define.¶

```
static gattCharCfg_t *simpleProfileChar4Config;
```

Note

Client characteristic configuration is represented as an array because this value must be cached for each connection. The catching of the client characteristic configuration is described in more detail in Add Service Function.

Add Service Function

As described in GATTServApp Module, when an application starts up it requires adding the GATT services it supports. Each profile needs a global AddService function that can be called from the application. Some of these services are defined in the protocol stack, such as GAP GATT Service and GATT Service. User-defined services must expose their own AddService function that the application can call for profile initialization. Using SimpleProfile_AddService() as an example, these functions should do as follows.

- Allocate space for the client characteristic configuration (CCC) arrays. As an example, a pointer to one of these arrays was initialized in the profile as described in ref:*client_characteristic_configuration*.

  In the AddService function, the supported connections is declared and memory is allocated for each array. Only one CCC is defined in the imple_gatt_profile but there can be multiple CCCs.

```
// Allocate Client Characteristic Configuration table
simpleProfileChar4Config = (gattCharCfg_t *)ICall_malloc(sizeof(gattCharCfg_t) * linkDBNumConns );

if ( simpleProfileChar4Config == NULL )
{
return ( bleMemAllocError );
}
```

- Initialize the CCC arrays. CCC values are persistent between power downs and between bonded device connections. For each CCC in the profile, the GATTServApp_InitCharCfg() function must be called. This function tries to initialize the CCCs with information from a previously bonded connection and set the initial values to default values if not found.

```
GATTServApp_InitCharCfg( INVALID_CONHANDLE, simpleProfileChar4Config );
```

- Register the profile with the GATTServApp. This function passes the attribute table of the profile to the GATTServApp so that the attributes of the profile are added to the application-wide attribute table managed by the protocol stack and handles are assigned for each attribute. This also passes pointers to the callbacks of the profile to the stack to initiate communication between the GATTServApp and the profile.

```
// Register GATT attribute list and CBs with GATT Server App
status = GATTServApp_RegisterService( simpleProfileAttrTbl,
                                      GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                                      GATT_MAX_ENCRYPT_KEY_SIZE,
                                      &simpleProfileCBs );
```

Register Application Callback Function

Profiles can relay messages to the application using callbacks. In the simple_peripheral project, the simple_gatt_profile calls an application callback whenever the GATT client writes a characteristic value. For these application callbacks to be used, the profile must define a Register Application Callback function that the application uses to set up callbacks during its initialization. The register application callback function for the simple_gatt_profile is the following:

Listing 63. Register application callbacks.¶

```
bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks )
{
   if ( appCallbacks )
   {
      simpleProfile_AppCBs = appCallbacks;
      return ( SUCCESS );
   }

   else
   {
      return ( bleAlreadyInRequestedMode );
   }
}
```

Where the callback typedef is defined as the following.

```
typedef struct
{
   simpleProfileChange_t pfnSimpleProfileChange; // Called when characteristic value changes
} simpleProfileCBs_t;
```

The application must then define a callback of this type and pass it to the simple_gatt_profile with the SimpleProfile_RegisterAppCBs() function. This occurs in simple_peripheral.c as follows.

```
//Simple GATT Profile Callbacks
#ifndef FEATURE_OAD_ONCHIP
static simpleProfileCBs_t SimpleBLEPeripheral_simpleProfileCBs =
{
   SimpleBLEPeripheral_charValueChangeCB // Characteristic value change callback
};
#endif //!FEATURE_OAD_ONCHIP

//...

//Register callback with SimpleGATTprofile
SimpleProfile_RegisterAppCBs(&SimpleBLEPeripheral_simpleProfileCBs);
```

See Read and Write Callback Functions for further information on how this callback is used.

Read and Write Callback Functions

The profile must define Read and Write callback functions which the protocol stack calls when one of the attributes of the profile are written to or read from. The callbacks must be registered with GATTServApp as mentioned in Add Service Function. These callbacks perform the characteristic read or write and other processing (possibly calling an application callback) as defined by the specific profile.

Read Request from Client

When a read request from a GATT Client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is readable, calls the read call-back profile. The profile copies in the value, performs any profile-specific processing, and notifies the application if desired. This procedure is illustrated in Figure 52. for a read of simpleprofileChar1 in the simple_gatt_profile.

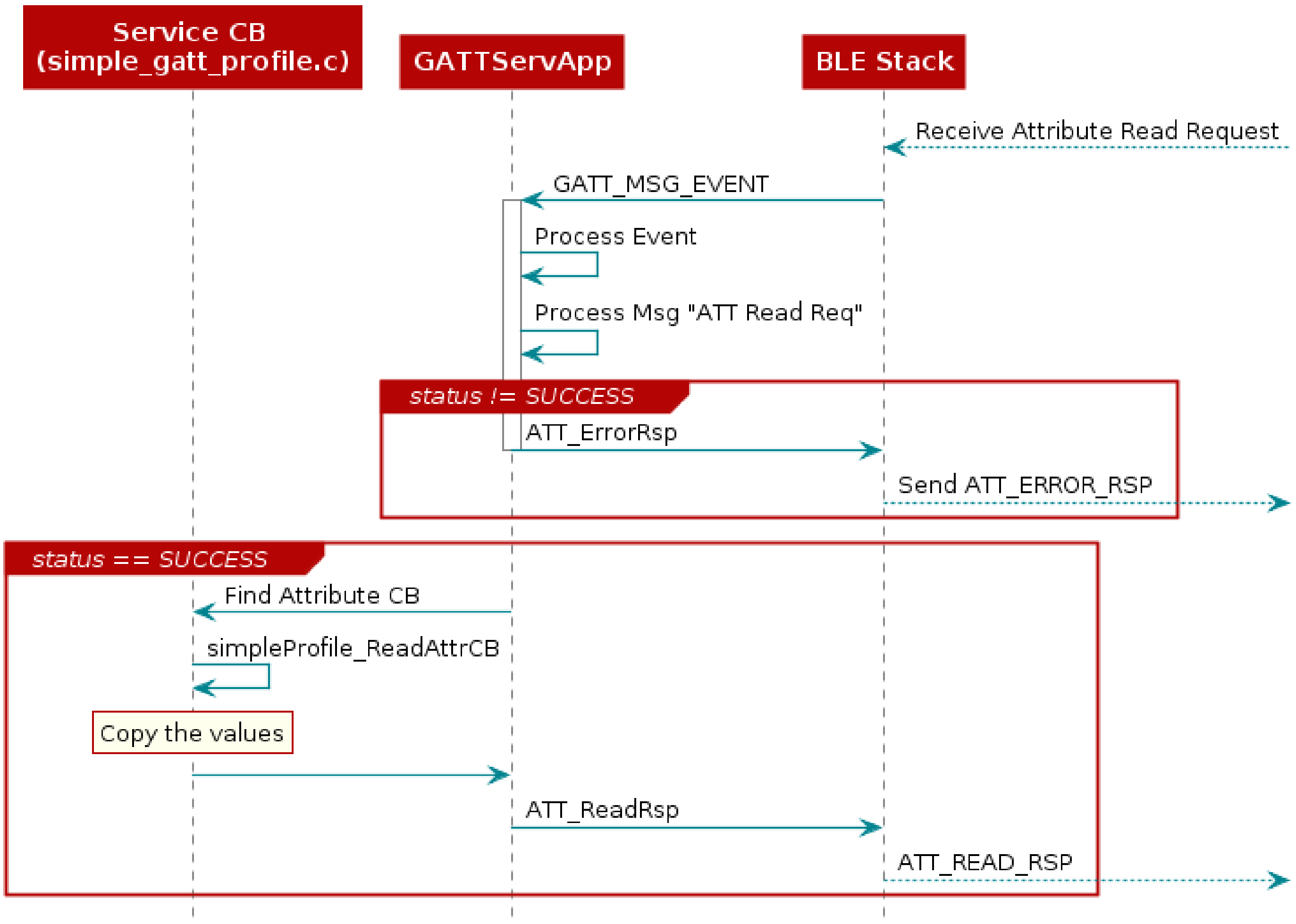


Figure 52. Read Request illustration

The processing is in the context of the protocol stack. If any intensive profile-related processing that must be done in the case of an attribute read, this should be split up and done in the context of the Application task. See the Write Request from Client for more information.

Write Request from Client

When a write request from a GATT client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is write-permitted, calls the write callback of the profile. The profile stores the value to be written, performs any profile-specific processing, and notifies the application if desired. Figure 53. shows this procedure for a write of simpleprofileChar3 in the simple_gatt_profile.

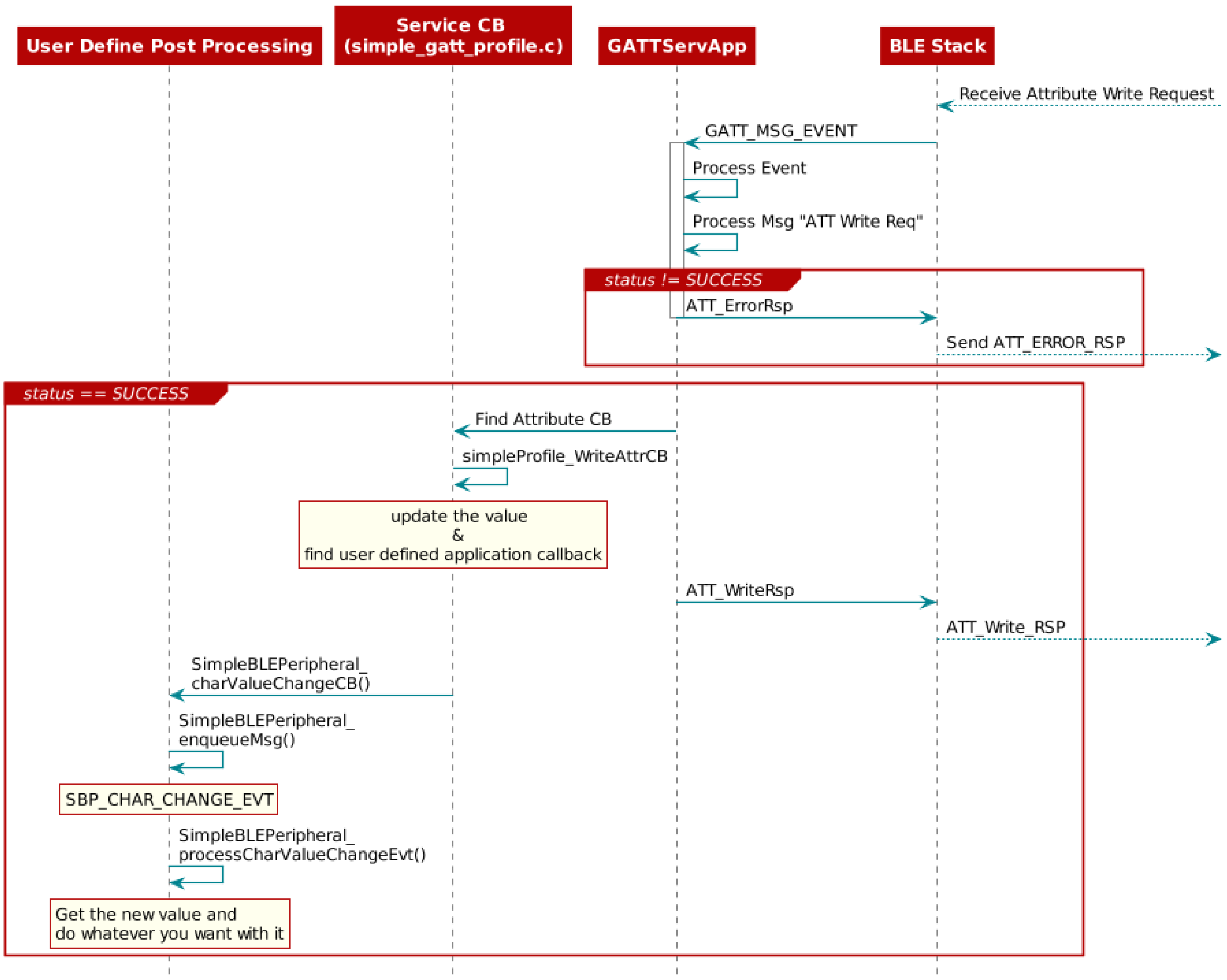

User Define Post Processing
Service CB (simple_gatt_profile.c)
GATTServApp
BLE Stack
Receive Attribute Write Request
GATT_MSG_EVENT
Process Event
Process Msg "ATT Write Req"
status != SUCCESS
ATT_ErrorRsp
Send ATT_ERROR_RSP
status == SUCCESS
Find Attribute CB
simpleProfile_WriteAttrCB
update the value & find user defined application callback
ATT_WriteRsp
ATT_Write_RSP
SimpleBLEPeripheral_charValueChangeCB()
SimpleBLEPeripheral_enqueueMsg()
SBP_CHAR_CHANGE_EVT
SimpleBLEPeripheral_processCharValueChangeEvt()
Get the new value and do whatever you want with it

Figure 53. Write Request illustration

Important

Minimizing the processing in protocol stack context is important. In this example, additional processing beyond storing the attribute write value in the profile occurs in the application context by enqueuing a message in the queue of the application.

Get and Set Functions

The profile containing the characteristics shall provide set and get abstraction functions for the application to read and write a characteristic of the profile. The set parameter function also includes logic to check for and implement notifications and indications if the relevant characteristic has notify or indicate properties. Figure 54. and the following code show this example for setting simpleProfileChacteristic4 in the simple_gatt_profile.

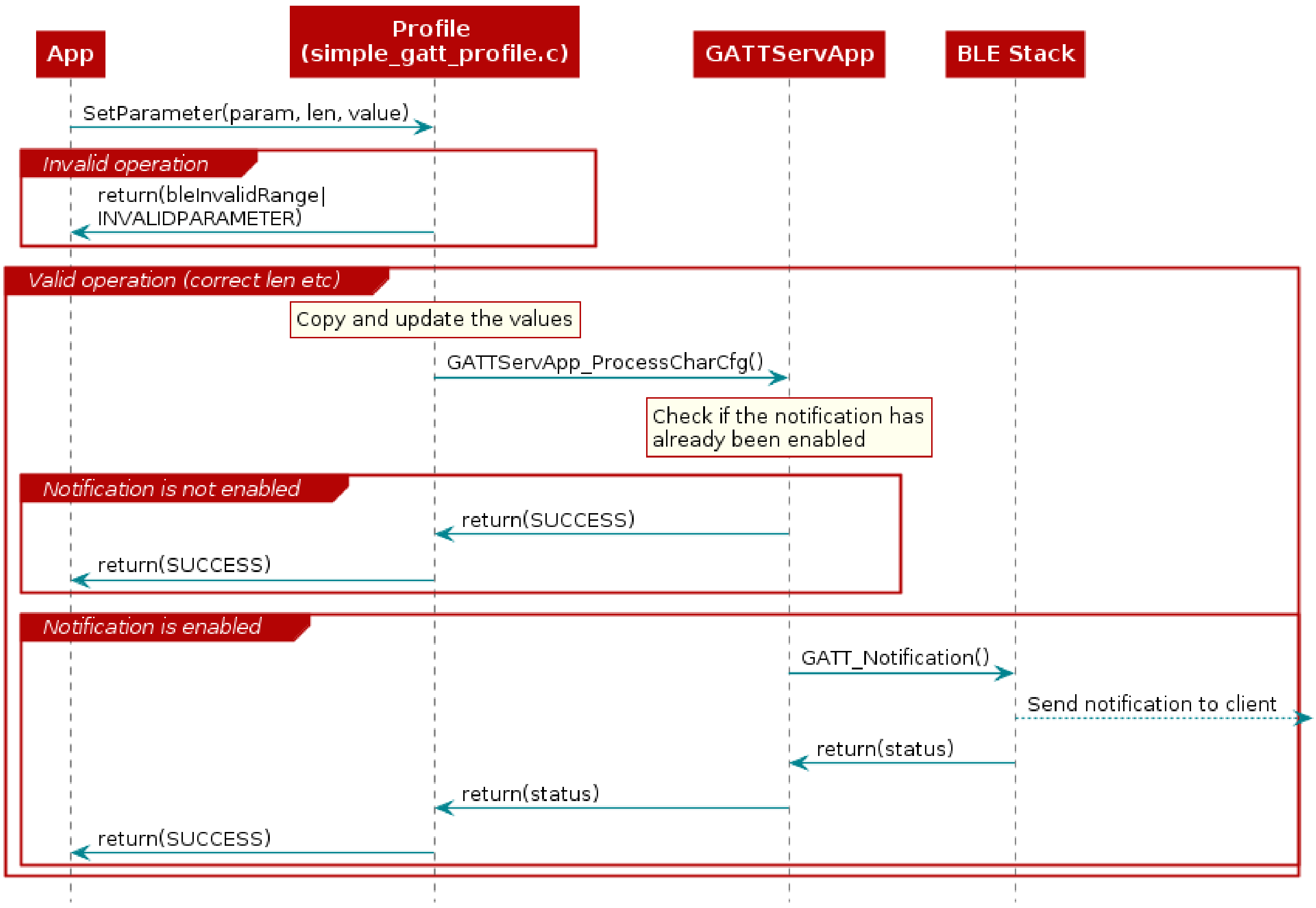

App
Profile (simple_gatt_profile.c)
GATTServApp
BLE Stack
SetParameter(param, len, value)
Invalid operation
return(bleInvalidRange| INVALIDPARAMETER)
Valid operation (correct len etc)
Copy and update the values
GATTServApp_ProcessCharCfg()
Check if the notification has already been enabled
Notification is not enabled
return(SUCCESS)
return(SUCCESS)
Notification is enabled
GATT_Notification()
Send notification to client
return(status)
return(status)
return(SUCCESS)

Figure 54. Set Profile Parameter

For example, the application initializes simpleProfileCharacteristic4 to 0 in simple_peripheral.c through the following.

```
uint8_t charValue4 = 0;
SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8_t), &charValue4);
```

The code for this function is displayed in the following code snippet (from simple_gatt_profile.c). Besides setting the value of the static simpleProfileChar4, this function also calls GATTServApp_ProcessCharCfg() because it has notify properties. This action forces GATTServApp to check if notifications have been enabled by the GATT Client. If so, the GATTServApp sends a notification of this attribute to the GATT Client.

Listing 64. Set characteristic 4 in SimpleProfile_SetParameter().¶

```
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value )
{
bStatus_t ret = SUCCESS switch ( param )
{
case SIMPLEPROFILE_CHAR4:
  if ( len == sizeof ( uint8 ) )
  {
    simpleProfileChar4 = *((uint8*)value);

    // See if Notification has been enabled
    GATTServApp_ProcessCharCfg( simpleProfileChar4Config, &simpleProfileChar4,
FALSE,
                                simpleProfileAttrTbl, GATT_NUM_ATTRS(
simpleProfileAttrTbl ),
                                INVALID_TASK_ID, simpleProfile_ReadAttrCB );
  }
```

Queued Writes

Prepare Write commands allows a GATT server to send more payload data by queuing up multiple write requests. The default queue size is 5. With a default MTU of 23 and payload of 18 bytes, up to 90 bytes of payload can be sent. For more information on queued writes, refer to the Queued Writes section ([Vol 3], Part F, Section 3.4.6) of the Bluetooth Core Specification Version 4.2.

Adjust the Prepare Write queue with GATTServApp_SetParameter() with parameter `GATT_PARAM_NUM_PREPARE_WRITES`. There is no specified limit, but it is bounded by the available HEAPMGR space.

Allocating Memory for GATT Procedures

To support fragmentation, GATT and ATT payload structures must be dynamically allocated for commands sent wirelessly. For example, a buffer must be allocated when sending a GATT_Notification. The stack does this allocation if the preferred method to send a GATT notification or indication is used: calling a SetParameter function of the profile (that is, SimpleProfile_SetParameter()) and callingGATTServApp_ProcessCharCfg() as described in Get and Set Functions.

If using GATT_Notification() or GATT_Indication() directly, memory management must be added as follows.

1. Try to allocate memory for the notification or indication payload using GATT_bm_alloc().
2. Send notification or indication using GATT_Notification() or GATT_Indication() if the allocation succeeds.

Note

If the return value of the notification or indication is `SUCCESS (0x00)`, the stack freed the memory.

3. Use GATT_bm_free() to free the memory if the return value is something other than `SUCCESS` (for example, `blePending`).

   The following is an example of this in the `GATTServApp_SendNotiInd` function in the gattservapp_util.c file.

Listing 65. gattServApp_SendNotiInd().¶

```
noti.pValue = (uint8 *)GATT_bm_alloc( connHandle, ATT_HANDLE_VALUE_NOTI,
GATT_MAX_MTU, &len );

if ( noti.pValue != NULL )
{
   status = (*pfnReadAttrCB)( connHandle, pAttr, noti.pValue, &noti.len, 0, len,
GATT_LOCAL_READ );

   if ( status == SUCCESS )
   {
      noti.handle = pAttr->handle;

      if ( cccValue & GATT_CLIENT_CFG_NOTIFY )
      {
         status = GATT_Notification( connHandle, &noti, authenticated );
      }

      else // GATT_CLIENT_CFG_INDICATE
      {
         status = GATT_Indication( connHandle, (attHandleValueInd_t *)&noti,
authenticated, taskId );
      }
   }

   if ( status != SUCCESS )
   {
      GATT_bm_free( (gattMsg_t *)&noti, ATT_HANDLE_VALUE_NOTI );
   }
}

else
{
   status = bleNoResources;
}
```

For other GATT procedures, take similar steps as noted in BLE Stack API Reference(specifically ATT/GATT section).

Registering to Receive Additional GATT Events in the Application

Using GATT_RegisterForMsgs(), receiving additional GATT messages to handle certain corner cases is possible. This possibility can be seen in simple_peripheral_processGATTMsg(). The following three cases are currently handled.

- GATT server in the stack was unable to send an ATT response (due to lack of available HCI buffers): Attempt to transmit on the next connection interval. Additionally, a status of `bleTimeout` is sent if the ATT transaction is not completed within 30 seconds, as specified in the Bluetooth Core Specification Version 4.2.

Listing 66. See if GATT server was unable to transmit an ATT response.¶

```
// See if GATT server was unable to transmit an ATT response
if (pMsg->hdr.status == blePending)
{
//No HCI buffer was available. Let's try to retransmit the response
//on the next connection event.

if (HCI_EXT_ConnEventNoticeCmd(pMsg->connHandle, selfEntity, SBP_CONN_EVT_END_EVT)
== SUCCESS)
{
//First free any pending response
 SimpleBLEPeripheral_freeAttRsp(FAILURE);

 //Hold on to the response message for retransmission
 pAttRsp = pMsg;

 //Don't free the response message yet
 return (FALSE);
 }
 }
```

- An ATT flow control violation: The application is notified that the connected device has violated the ATT flow control specification, such as sending a Read Request before an Indication Confirm is sent.

  No more ATT requests or indications can be sent wirelessly during the connection. The application may want to terminate the connection due to this violation. As an example in simple_peripheral, the LCD is updated.

Listing 67. ATT flow control violation.¶

```
else if (pMsg->method == ATT_FLOW_CTRL_VIOLATED_EVENT)
{
  //ATT request-response or indication-confirmation flow control is
  //violated. All subsequent ATT requests or indications will be dropped.
  //The app is informed in case it wants to drop the connection.

  //Display the opcode of the message that caused the violation.
  DISPLAY_WRITE_STRING_VALUE("FC Violated: %d", pMsg->msg.flowCtrlEvt.opcode,
LCD_PAGE5);
}
```

- An ATT MTU size is updated: The application is notified in case this affects its processing in any way. See Maximum Transmission Unit (MTU) for more information on the MTU. As an example in simple_peripheral, the LCD is updated.

Listing 68. MTU updated.¶

```
else if (pMsg->method == ATT_MTU_UPDATED_EVENT)
{
    // MTU size updated
    DISPLAY_WRITE_STRING_VALUE("MTU Size: $d", pMsg->msg.mtuEvt.MTU, LCD_PAGE5);
}

```