Introduction

The main purpose of this document is to provide developers with reference programming guidelines on how to develop a Bluetooth® Low Energy (Bluetooth LE) application using the Bluetooth LE stack v3.x family APIs and related event callbacks.

The document describes the Bluetooth LE stack v3.x Bluetooth Low Energy stack library framework, API interfaces and event callbacks allowing access to the Bluetooth Low Energy functions provided by the STMicroelectronics Bluetooth Low Energy devices system-on-chip.

The following Bluetooth Low Energy device supports the Bluetooth LE stack v3.x family:

- BlueNRG-LP device

The document also focuses on the key changes about APIs and the callback interface, Bluetooth LE stack initialization versus the Bluetooth LE stack v2.x family.

This programming manual also provides some fundamental concepts about the Bluetooth Low Energy technology in order to associate the Bluetooth LE stack v3.x APIs, parameters, and related event callbacks with the Bluetooth LE protocol stack features. The user is expected to have a basic knowledge of Bluetooth LE technology and its main features.

For more information about the supported devices and the Bluetooth Low Energy specifications, refer to Section 5 References at the end of this document.

The manual is structured as follows:

- Fundamentals of the Bluetooth Low Energy technology
- Bluetooth LE stack v3.x library APIs and the event callback overview
- How to design an application using the Bluetooth LE stack v3.x library APIs and event callbacks.

Note: The document content is valid for all the specified Bluetooth Low Energy devices. Any specific difference is highlighted whenever required.
1 Bluetooth Low Energy technology

The Bluetooth Low Energy (Bluetooth LE) wireless technology has been developed by the Bluetooth special interest group (SIG) in order to achieve a very low power standard operating with a coin cell battery for several years.

Classic Bluetooth technology has been developed as a wireless standard allowing cables to be replaced connecting portable and/or fixed electronic devices, but it cannot achieve an extreme level of battery life because of its fast hopping, connection-oriented behavior, and relatively complex connection procedures.

The Bluetooth Low Energy devices consume only a fraction of the power of standard Bluetooth products and enable devices with coin cell batteries to be connected via wireless to standard Bluetooth enabled devices.

![Image](GAMSEC201411251047)

Figure 1. Bluetooth Low Energy technology enabled coin cell battery devices

Bluetooth Low Energy technology is used on a broad range of sensor applications transmitting small amounts of data:

- Automotive
- Sport and fitness
- Healthcare
- Entertainment
- Home automation
- Security and proximity

1.1 Bluetooth LE stack architecture

Bluetooth Low Energy technology has been formally adopted by the Bluetooth core specification version 4.0 (Section 5 References). This version of the Bluetooth standard supports two systems of wireless technology:

- Basic rate
- Bluetooth Low Energy

The Bluetooth Low Energy technology operates in the unlicensed industrial, scientific and medical (ISM) band at 2.4 to 2.485 GHz, which is available and unlicensed in most countries. It uses a spread spectrum, frequency hopping, full-duplex signal. Key features of Bluetooth Low Energy technology are:

- Robustness
- Performance
- Reliability
- Interoperability
- Low data rate
• Low-power.

In particular, Bluetooth Low Energy technology has been created for the purpose of transmitting very small packets of data at a time, while consuming significantly less power than basic rate/enhanced data rate/high speed (BR/EDR/HS) devices.

The Bluetooth Low Energy stack consists of two components:
• Controller
• Host

The controller includes the physical layer and the link layer.

Host includes the logical link control and adaptation protocol (L2CAP), the security manager (SM), the attribute protocol (ATT), generic attribute profile (GATT) and the generic access profile (GAP). The interface between the two components is called host controller interface (HCI).

In addition, Bluetooth specifications v4.1, v4.2, v5.x have been released with new supported features.

For more information about these new features, refer to the related specification document.

Figure 2. Bluetooth Low Energy stack architecture

1.2 Physical layer

The physical layer is a 1 Mbps adaptive frequency-hopping Gaussian frequency shift keying (GFSK) radio. It operates in the license free 2.4 GHz ISM band at 2400-2483.5 MHz. Many other standards use this band: IEEE 802.11, IEEE 802.15.

The Bluetooth LE system uses 40 RF channels (0-39), with 2 MHz spacing. These RF channels have frequencies centered at:

\[ 240 + k \times 2 \text{MHz}, \text{where } k = 0, \ldots, 39 \]  

There are two types of channel:
1. Advertising channels that use three fixed RF channels (37, 38 and 39) for:
   a. Advertising channel packets
   b. Packets used for discoverability/connectability
   c. Used for broadcasting/scanning
2. The data physical channel uses the other 37 RF channels for bidirectional communication between the connected devices.

### Table 1. Bluetooth LE RF channel types and frequencies

<table>
<thead>
<tr>
<th>Channel index</th>
<th>RF center frequency</th>
<th>Channel type</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>2402 MHz</td>
<td>Advertising channel</td>
</tr>
<tr>
<td>0</td>
<td>2404 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>1</td>
<td>2406 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Data channel</td>
</tr>
<tr>
<td>10</td>
<td>2424 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>38</td>
<td>2426 MHz</td>
<td>Advertising channel</td>
</tr>
<tr>
<td>11</td>
<td>2428 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>12</td>
<td>2430 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Data channel</td>
</tr>
<tr>
<td>36</td>
<td>2478 MHz</td>
<td>Data channel</td>
</tr>
<tr>
<td>39</td>
<td>2480 MHz</td>
<td>Advertising channel</td>
</tr>
</tbody>
</table>

Bluetooth LE is an adaptive frequency hopping (AFH) technology that can only use a subset of all the available frequencies in order to avoid all frequencies used by other non-adaptive technologies. This allows moving from a bad channel to a known good channel by using a specific frequency hopping algorithm, which determines the next good channel to be used.

1.2.1 LE 2M and LE Coded physical layers

Bluetooth Low Energy specification v5.x adds two other PHY variants to the PHY specification (LE 1M) provided by Bluetooth Low Energy specifications v4.x:

- LE 2M
- LE Coded

Standard HCI APIs are defined on Bluetooth LE specifications v5.x to set, respectively, the PHY preferences (LE 1M, LE 2M, LE Coded) for the transmitter PHY and receiver PHY for all subsequent connections over the LE transport, or to set the transmitter PHY and receiver PHY for a specific connection.

**Note:** *LE 1M support on Bluetooth LE specification v5.x is still mandatory.*

1.2.2 LE 2M PHY

There are several application use cases demanding a higher throughput:

- Over-the-air FW upgrade procedure
- Sports, fitness, medical applications use cases require to collect big amount of data with a greater accuracy and also to send data more frequently through some medical devices.

LE 2M PHY allows the physical layer to work at 2 Mbps and, as a consequence, PHY can achieve higher data rates than LE 1M. It uses adaptive frequency-hopping Gaussian frequency shift keying (GFSK) radio. LE 2M PHY uses a frequency deviation of at least 370 kHz.
1.2.3 LE Coded PHY

Several application scenarios ask for an increased range. By increasing the range, the signal-to-noise ratio (SNR) starts decreasing and, as a consequence, the probability of decoding errors rises: the bit error rate (BER) increases.

LE Coded PHY uses the forward error correction (FEC) to fix mistakes on received packets. This allows the received packet to be correctly decoded with a lower signal-to-noise ratio (SNR) values and, as a consequence, it increases the transmitter distance without the need to increase the transmitter power level (range can be up to 4 times the one allowed with Bluetooth Low Energy v4.x).

FEC method adds some specific bits to the transmitted packet, which allows FEC to determine the correct values that the wrong bits should have. FEC method adds 2 further steps to the bit stream processing:

1. FEC encoding, which generates 2 further bits for each bit
2. Pattern mapper, which converts each bit from previous step in P symbols depending on 2 coding schemes:
   - S= 2: no change is done. This doubles the range
   - S= 8: each bit is mapped to 4 bits. This leads to a quadruple range

Since the FEC method adds several bits to the overall packet, the number of data to be transmitted is increased: therefore the communication data rate is decreased.

<table>
<thead>
<tr>
<th>Table 2. LE PHY key parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol rate</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Data rate</td>
</tr>
<tr>
<td>Error detection</td>
</tr>
<tr>
<td>Error correction</td>
</tr>
<tr>
<td>Range increase</td>
</tr>
<tr>
<td>Bluetooth LE specification 5.x requirement type</td>
</tr>
</tbody>
</table>

1.3 Link layer (LL)

The link layer (LL) defines how two devices can use a radio to transmit information between each other. The link layer defines a state machine with five states:
• Standby: the device does not transmit or receive packets
• Advertising: the device broadcasts advertisements in advertising channels (it is called an advertiser device)
• Scanning: the device looks for advertiser devices (it is called a scanner device)
• Initiating: the device initiates a connection to the advertiser device
• Connection: the initiator device is in a master role: it communicates with the device in the slave role and it defines timings of transmissions
• Advertiser device is in slave role: it communicates with a single device in master role.

1.3.1 Bluetooth LE packets
A packet is a labeled data transmitted by one device and received by one or more other devices. The Bluetooth LE data packet structure is described below.

The Bluetooth Low Energy specification v4.2 defines the LE data packet length extension feature, which extends the link layer PDU of LE from 27 to 251 bytes of data payload.
Figure 5. Packet structure with LE data packet length extension feature

<table>
<thead>
<tr>
<th>8</th>
<th>32</th>
<th>8</th>
<th>8</th>
<th>0 to (8 * 255)</th>
<th>24</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td>Access Address</td>
<td>Header</td>
<td>Length</td>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The length field has a range from 0 to 255 bytes. When encryption is used, the message integrity code (MIC) at the end of the packet is 4 bytes; so this leads to 251 bytes as current maximum available payload size.

- **Preamble**: RF synchronization sequence
- **Access address**: 32 bits, advertising or data access addresses (it is used to identify the communication packets on physical layer channel)
- **Header**: its content depends on the packet type (advertising or data packet)
- **Advertiser packet header**:
  
  **Table 3. Advertising data header content**

<table>
<thead>
<tr>
<th>Advertising packet type</th>
<th>Reserved</th>
<th>Tx address type</th>
<th>Rx address type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 bits)</td>
<td>(2 bits)</td>
<td>(1 bit)</td>
<td>(1 bit)</td>
</tr>
</tbody>
</table>

- The advertising packet type is defined as follows:

  **Table 4. Advertising packet types**

<table>
<thead>
<tr>
<th>Packet type</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADV_IND</td>
<td>Connectable undirected advertising</td>
<td>Used by an advertiser when it wants another device to connect to it. The device can be scanned by a scanning device, or go into a connection as a slave device on connection request reception.</td>
</tr>
<tr>
<td>ADV_DIRECT_IND</td>
<td>Connectable directed advertising</td>
<td>Used by an advertiser when it wants a particular device to connect to it. The ADV_DIRECT_IND packet contains advertiser’s address and initiator address only.</td>
</tr>
<tr>
<td>ADV_NONCONN_IND</td>
<td>Non-connectable undirected advertising</td>
<td>Used by an advertiser when it wants to provide some information to all the devices, but it does not want other devices to ask it for more information or to connect to it. The device simply sends advertising packets on related channels, but it does not want to be connectable or scanned by any other device.</td>
</tr>
<tr>
<td>ADV_SCAN_IND</td>
<td>Scannable undirected advertising</td>
<td>Used by an advertiser which wants to allow a scanner to require more information from it. The device cannot connect, but it is discoverable for advertising data and scans response data.</td>
</tr>
<tr>
<td>SCAN_REQ</td>
<td>Scan request</td>
<td>Used by a device in scanning state to request additional information from the advertiser.</td>
</tr>
<tr>
<td>SCAN_RSP</td>
<td>Scan response</td>
<td>Used by an advertiser device to provide additional information to a scan device.</td>
</tr>
<tr>
<td>CONNECT_REQ</td>
<td>Connection request</td>
<td>Sent by an initiating device to a device in connectable/discoverable mode.</td>
</tr>
</tbody>
</table>
The advertising event type determines the allowable responses:

**Table 5. Advertising event type and allowable responses**

<table>
<thead>
<tr>
<th>Advertising event type</th>
<th>Allowable response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCAN_REQ</td>
</tr>
<tr>
<td>ADV_IND</td>
<td>YES</td>
</tr>
<tr>
<td>ADV_DIRECT_IND</td>
<td>NO</td>
</tr>
<tr>
<td>ADV_NONCONN_IND</td>
<td>NO</td>
</tr>
<tr>
<td>ADV_SCAN_IND</td>
<td>YES</td>
</tr>
</tbody>
</table>

- Data packet header:

**Table 6. Data packet header content**

<table>
<thead>
<tr>
<th>Link layer identifier</th>
<th>Next sequence number</th>
<th>Sequence number</th>
<th>More data</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 bits)</td>
<td>(1 bit)</td>
<td>(1 bit)</td>
<td>(1 bit)</td>
<td>(3 bits)</td>
</tr>
</tbody>
</table>

The next sequence number (NESN) bit is used to perform packet acknowledgments. It informs the receiver device about the next sequence number that the transmitting device expects it to send. Packet is retransmitted until the NESN is different from the sequence number (SN) value in the sent packet.

More data bits are used to signal to a device that the transmitting device has more data ready to be sent during the current connection event.

For a detailed description of advertising and data header contents and types refer to the Bluetooth specification [Vol 2], in Section 5 References.

- Length: number of bytes on data field:

**Table 7. Packet length field and valid values**

<table>
<thead>
<tr>
<th>Description</th>
<th>Length field bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertising packet</td>
<td>6 bits, with valid values from 0 to 37 bytes</td>
</tr>
<tr>
<td>Data packet</td>
<td>5 bits, with valid values from 0 to 31 bytes</td>
</tr>
<tr>
<td></td>
<td>8 bits, with valid values from 0 to 255 bytes, with LE data packet length extension</td>
</tr>
</tbody>
</table>

- Data or payload: it is the current transmitted data (advertising data, scan response data, connection establishment data, or application data sent during the connection)
- CRC (24 bits): it is used to protect data against bit errors. It is calculated over the header, length and data fields

**1.3.2 Extended advertising**

On Bluetooth Low Energy specification v4.x, the maximum advertising packet payload is 31 bytes. Each advertising packet payload is sent on three specific channels (37, 38, 39), one packet at a time.

Bluetooth Low Energy v5.x advertising extension capability allows:

- Increasing the number of data in contexts where Bluetooth LE connection is not used (i.e.: beacon scenario, ...)
- Having advertising sent to a deterministic way
- Having different advertising data to be sent.
New extended advertising packets can use the Bluetooth LE 4.x connection channels (0-36) for extending advertising payload up to 255 bytes. The advertising header data is still transmitted on the traditional advertising channels (37, 38, 39) called primary channels. The header field also includes a new data AuxPtr, which contains the channel number (0-36), where the packet, including the advertising payload, is transmitted (it is called the secondary channel).

**Figure 6. Bluetooth LE 5.x extended advertising**

![Bluetooth LE 5.x extended advertising](image)

It is also possible to create a chain of advertising packets on secondary channels in order to transmit more advertising payload data (greater than 255 bytes). Each advertising packet on a secondary channel includes on its AuxPtr the number of the next secondary channel for the next advertising packet on the chain. As a consequence, each chain in the advertising packet chain can be sent on a different secondary channel.

**Figure 7. Advertising packet chain**

![Advertising packet chain](image)

A direct advantage of new extended advertising packets is the capability to send less data on primary channels (37,38,39) so reducing the data on primary channels. Furthermore, the advertising payload is sent on a secondary channel only and no longer on all the 3 primary advertising channels, by reducing the overall duty cycle.

*Note:* The minimum advertising interval has been reduced from 100 ms to 20 ms for non-connectable advertising.

### 1.3.3 Advertising sets

Bluetooth Low Energy v4.x does not vary the advertising payload during the advertising in order to have different data on different advertising packets.

Bluetooth Low Energy v5.x defines advertising sets having an ID used to indicate which set each packet belongs to. Each set has its specific advertising parameters (advertising interval, PDU type) and it can use the primary or secondary channel. The link layer has the ownership to schedule and set the advertising sets defined by the host layer.
1.3.4 Advertising state

Advertising states allow the link layer to transmit advertising packets and also to respond with scan responses to scan requests coming from those devices, which are actively scanning. An advertiser device can be moved to a standby state by stopping the advertising.

Each time a device advertises, it sends the same packet on each of the three advertising channels. This three packet sequence is called an “advertising event”. The time between two advertising events is referred to the advertising interval, which can go from 20 milliseconds to every 10.28 seconds.

An example of advertising packet lists the service UUID that the device implements (general discoverable flag, tx power = 4 dbm, service data = temperature service and 16-bit service UUIDs).

Figure 8. Advertising packet with AD type flags

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Advertising Access Address</th>
<th>Advertising Header</th>
<th>Payload Length</th>
<th>Advertising Address</th>
<th>Flags-LE General Discoverable Flag</th>
<th>TX Power Level = 4 dbm</th>
<th>Service Data Temperature = 20.5°C</th>
<th>16 bit service UUIDs = “Temperature service”</th>
<th>CRC</th>
</tr>
</thead>
</table>

The flag AD type byte contains the following flag bits:

- Limited discoverable mode (bit 0)
- General discoverable mode (bit 1)
- BR/EDR not supported (bit 2, it is 1 on Bluetooth LE)
- Simultaneous LE and BR/EDR to the same device capable (controller) (bit 3)
- Simultaneous LE and BR/EDR to the same device capable (host) (bit 4)

The flag AD type is included in the advertising data if any of the bits is non-zero (it is not included in scan response).

The following advertising parameters can be set before enabling advertising:

- Advertising interval
- Advertising address type
- Advertising device address
- Advertising channel map: which of the three advertising channels should be used
- Advertising filter policy:
  - Process scan/connection requests from the devices in the white-list
  - Process all scan/connection requests (default advertiser filter policy)
  - Process connection requests from all the devices but only scan requests in the white-list
  - Process scan requests from all the devices but only connection requests in the white-list.

A white-list is a list of stored device addresses used by the device controller to filter devices. The white-list content cannot be modified while it is being used. If the device is in advertising state and uses a white-list to filter the devices (scan requests or connection requests), it has to disable the advertising mode to change its white-list.

1.3.5 Scanning state

There are two types of scanning:

- Passive scanning: it allows the advertisement data to be received from an advertiser device
- Active scanning: when an advertisement packet is received, the device can send back a scan request packet, in order to get a scan response from the advertiser. This allows the scanner device to get additional information from the advertiser device.

The following scan parameters can be set:
• Scanning type (passive or active)
• Scan interval: how often the controller should scan
• Scan window: for each scanning interval, it defines how long the device has been scanning
• Scan filter policy: it can accept all the advertising packets (default policy) or only those on the white-list.

Once the scan parameters are set, the device scanning can be enabled. The controller of the scanner devices sends to the upper layers any received advertising packets within an advertising report event. This event includes the advertiser address, advertiser data, and the received signal strength indication (RSSI) of this advertising packet. The RSSI can be used with the transmit power level information included within the advertising packets to determine the path-loss of the signal and identify how far the device is:

Path loss = Tx power – RSSI

### 1.3.6 Connection state

When data to be transmitted are more complex than those allowed by advertising data or a bidirectional reliable communication between two devices is needed, the connection is established.

When an initiator device receives an advertising packet from an advertising device to which it wants to connect, it can send a connect request packet to the advertiser device. This packet includes all the required information needed to establish and handle the connection between the two devices:

• Access address used in the connection in order to identify communications on a physical link
• CRC initialization value
• Transmit window size (timing window for the first data packet)
• Transmit window offset (transmit window start)
• Connection interval (time between two connection events)
• Slave latency (number of times slave can ignore connection events before it is forced to listen)
• Supervision timeout (max. time between two correctly received packets before link is considered lost)
• Channel map: 37 bits (1 = good; 0 = bad)
• Frequency-hop value (random number between 5 and 16)
• Sleep clock accuracy range (used to determine the uncertainty window of the slave device at connection event).

For a detailed description of the connection request packet refer to Bluetooth specifications [Vol 6].

The allowed timing ranges are summarized in Table 1. Bluetooth LE RF channel types and frequencies:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min.</th>
<th>Max.</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmit window size</td>
<td>1.25 milliseconds</td>
<td>10 milliseconds</td>
<td>Multiples of 1.25 milliseconds</td>
</tr>
<tr>
<td>Transmit window offset</td>
<td>0</td>
<td>Connection interval</td>
<td>Multiples of 1.25 milliseconds</td>
</tr>
<tr>
<td>Connection interval</td>
<td>7.5 milliseconds</td>
<td>4 seconds</td>
<td>Multiples of 1.25 milliseconds</td>
</tr>
<tr>
<td>Supervision timeout</td>
<td>100 milliseconds</td>
<td>32 seconds</td>
<td>Multiples of 10 milliseconds</td>
</tr>
</tbody>
</table>

The transmit window starts after the end of the connection request packet plus the transmit window offset plus a mandatory delay of 1.25 ms. When the transmit window starts, the slave device enters the receiver mode and waits for a packet from the master device. If no packet is received within this time, the slave leaves the receiver mode, and it tries one connection interval again later. When a connection is established, a master has to transmit a packet to the slave on every connection event to allow the slave to send packets to the master. Optionally, a slave device can skip a given number of connection events (slave latency).

A connection event is the time between the start of the last connection event and the beginning of the next connection event.

A Bluetooth LE slave device can only be connected to one Bluetooth LE master device, but a Bluetooth LE master device can be connected to several Bluetooth LE slave devices. On the Bluetooth SIG, there is no limit to the number of slaves a master can connect to (this is limited by the specific used Bluetooth LE technology or stack).
1.4 Host controller interface (HCI)

The host controller interface (HCI) layer provides a communication between the host and controller either through software API or by a hardware interface such as: SPI, UART or USB. It comes from standard Bluetooth specifications, with new additional commands for Low Energy-specific functions.

1.5 Logical link control and adaptation layer protocol (L2CAP)

The logical link control and adaptation layer protocol (L2CAP) supports a higher level protocol multiplexing, packet segmentation and reassembly operations, and the conveying of quality of service information.

1.5.1 LE L2CAP connection-oriented channels

L2CAP connection-oriented channels provide support to efficient bulk data transfer with reduced overhead. Service data units (SDUs) are reliably delivered using flow control. Segmentation and reassembly of large SDUs are performed automatically by the L2CAP entity. Multiplexing allows multiple services to be carried out at the same time.

1.6 Attribute protocol (ATT)

The attribute protocol (ATT) allows a device to expose some data, known as attributes, to another device. The device defining the attributes is called the server and the peer device using them is called the client.

An attribute is data with the following components:

- Attribute handle: it is a 16-bit value, which identifies an attribute on a server, allowing the client to reference the attribute in read or write requests
- Attribute type: it is defined by a universally unique identifier (UUID), which determines what the value means. Standard 16-bit attribute UUIDs are defined by Bluetooth SIG
- Attribute value: a (0 ~ 512) octets in length
- Attribute permissions: they are defined by each upper layer that uses the attribute. They specify the required security level for read and/or write access, as well as notification and/or indication. The permissions are not discoverable using the attribute protocol. There are different permission types:
  - Access permissions: they determine which types of requests can be performed on an attribute (readable, writable, readable and writable)
  - Authentication permissions: they determine if attributes require authentication or not. If an authentication error is raised, the client can try to authenticate it by using the security manager and sending back the request
  - Authorization permissions (no authorization, authorization): this is a property of a server, which can authorize a client to access or not set of attributes (client cannot resolve an authorization error).

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>Attribute value</th>
<th>Attribute permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0008</td>
<td>“Temperature UUID”</td>
<td>“Temperature value”</td>
<td>“Read only, no authorization, no authentication”</td>
</tr>
</tbody>
</table>

- “Temperature UUID” is defined by “Temperature characteristic” specification and it is a signed 16-bit integer.

A collection of attributes is called a database that is always contained in an attribute server. Attribute protocol defines a set of method protocols to discover, read and write attributes on a peer device. It implements the peer-to-peer client-server protocol between an attribute server and an attribute client as follows:

- Server role
  - Contains all attributes (attribute database)
  - Receives requests, executes, responds commands
  - Indicates, notifies an attribute value when data change
• Client role
  – Talks with server
  – Sends requests, waits for response (it can access (read), update (write) the data)
  – Confirms indications.

Attributes exposed by a server can be discovered, read, and written by the client, and they can be indicated and notified by the server as described in Table 2. LE PHY key parameters:

<table>
<thead>
<tr>
<th>Protocol data unit (PDU message)</th>
<th>Sent by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>Client</td>
<td>Client asks server (it always causes a response)</td>
</tr>
<tr>
<td>Response</td>
<td>Server</td>
<td>Server sends response to a request from a client</td>
</tr>
<tr>
<td>Command</td>
<td>Client</td>
<td>Client commands something to server (no response)</td>
</tr>
<tr>
<td>Notification</td>
<td>Server</td>
<td>Server notifies client of a new value (no confirmation)</td>
</tr>
<tr>
<td>Indication</td>
<td>Server</td>
<td>Server indicates to client a new value (it always causes a confirmation)</td>
</tr>
<tr>
<td>Confirmation</td>
<td>Client</td>
<td>Confirmation to an indication</td>
</tr>
</tbody>
</table>

1.7 Security manager (SM)

The Bluetooth Low Energy link layer supports both encryption and authentication by using the counter mode with the CBC-MAC (cipher block chaining-message authentication code) algorithm and a 128-bit AES block cipher (AES-CCM). When encryption and authentication are used in a connection, a 4-byte message integrity check (MIC) is appended to the payload of the data channel PDU.

Encryption is applied to both the PDU payload and MIC fields.

When two devices want to encrypt the communication during the connection, the security manager uses the pairing procedure. This procedure allows two devices to be authenticated by exchanging their identity information in order to create the security keys that can be used as the basis for a trusted relationship or a (single) secure connection. There are some methods used to perform the pairing procedure. Some of these methods provide protections against:

• Man-in-the-middle (MiTM) attacks: a device is able to monitor and modify or add new messages to the communication channel between two devices. A typical scenario is when a device is able to connect to each device and act as the other devices by communicating with each of them

• Passive eavesdropping attacks: listening through a sniffing device to the communication of other devices

The pairing on Bluetooth Low Energy specifications v4.0 or v4.1, also called LE legacy pairing, supports the following methods based on the IO capability of the devices: Just Works, Passkey Entry and Out of band (OOB).

On Bluetooth Low Energy specification v4.2, the LE secure connection pairing model has been defined. The new security model main features are:

1. Key exchange process uses the elliptical curve Diffie-Hellman (ECDH) algorithm: this allows keys to be exchanged over an unsecured channel and to protect against passive eavesdropping attacks (secretly listening through a sniffing device to the communication of other devices)

2. A new method called “numeric comparison” has been added to the 3 methods already available with LE legacy pairing

The pairing procedures are selected depending on the device IO capabilities.

There are three input capabilities:

• No input
• Ability to select yes/no
• Ability to input a number by using the keyboard.

There are two output capabilities:

• No output
• Numeric output: ability to display a six-digit number
The following table shows the possible IO capability combinations:

<table>
<thead>
<tr>
<th>Combination of input/output capabilities on a Bluetooth LE device</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No output</strong></td>
</tr>
<tr>
<td>No input</td>
</tr>
<tr>
<td>Yes/No</td>
</tr>
<tr>
<td>Keyboard</td>
</tr>
</tbody>
</table>

**LE legacy pairing**

LE legacy pairing algorithm uses and generates 2 keys:
- Temporary key (TK): a 128-bit temporary key, which is used to generate a short-term key (STK)
- Short-term key (STK): a 128-bit temporary key used to encrypt a connection following pairing

Pairing procedure is a three-phase process.

**Phase 1: pairing feature exchange**

The two connected devices communicate their input/output capabilities by using the pairing request message. This message also contains a bit stating if out-of-band data are available and the authentication requirements. The information exchanged in phase 1 is used to select which pairing method is used for the STK generation in phase 2.

**Phase 2: short-term key (STK) generation**

The pairing devices first define a temporary key (TK), by using one of the following key generation methods:
1. The out-of-band (OOB) method, which uses out-of-band communication (e.g. NFC) for TK agreement. It provides the authentication (MITM protection). This method is selected only if the out-of-band bit is set on both devices, otherwise the IO capabilities of the devices must be used to determine which other method could be used (Passkey Entry or Just Works)
2. Passkey Entry method: user passes six numeric digits as the TK between the devices. It provides the authentication (MITM protection)
3. Just Works: this method does not provide the authentication and protection against man-in-the-middle (MITM) attacks

The selection between the Passkey and Just Works method is done based on the IO capability as defined in the following table.

<table>
<thead>
<tr>
<th>Methods used to calculate the temporary key (TK)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display only</strong></td>
</tr>
<tr>
<td>Display only</td>
</tr>
<tr>
<td>Display Yes/No</td>
</tr>
<tr>
<td>Keyboard only</td>
</tr>
<tr>
<td>No input No output</td>
</tr>
<tr>
<td>Keyboard display</td>
</tr>
</tbody>
</table>

**Phase 3: transport specific key distribution methods used to calculate the temporary key (TK)**

Once phase 2 is completed, up to three 128-bit keys can be distributed by messages encrypted by the STK key:
1. Long-term key (LTK): it is used to generate the 128-bit key used for link layer encryption and authentication
2. Connection signature resolving key (CSRK): a 128-bit key used for the data signing and verification performed at the ATT layer
3. Identity resolving key (IRK): a 128-bit key used to generate and resolve random addresses

**LE secure connections**
LE secure connection pairing methods use and generate one key:

- Long-term key (LTK): a 128-bit key used to encrypt the connection following pairing and subsequent connections

Pairing procedure is a three-phase process:

**Phase 1: pairing feature exchange**

The two connected devices communicate their input/output capabilities by using the pairing request message. This message also contains a bit stating if out-of-band data are available and the authentication requirements. The information exchanged in phase 1 is used to select which pairing method is used in phase 2.

**Phase 2: long-term key (LTK) generation**

Pairing procedure is started by the initiating device which sends its public key to the receiving device. The public key exchange phase is done for all the pairing methods (except the OOB one). Each device generates its own elliptic curve Diffie-Hellman (ECDH) public-private key pair. Each key pair contains a private (secret) key, and a public key. The key pair should be generated only once on each device and may be computed before a pairing is performed.

The following pairing key generation methods are supported:

1. The out-of-band (OOB) method uses out of band communication to set up the public key. This method is selected if the out-of-band bit, in the pairing request/response, is set at least by one device, otherwise the IO capabilities of the devices must be used to determine which other method could be used (Passkey Entry, Just Works or numeric comparison).
2. Just Works: this method is not authenticated, and it does not provide any protection against man-in-the-middle (MITM) attacks.
3. Passkey Entry method: this method is authenticated. User passes six numeric digits. This six-digit value is the base of the device authentication.
4. Numeric comparison: this method is authenticated. Both devices have IO capabilities set to either display Yes/No or keyboard display. The two devices compute six-digit confirmation values that are displayed to the user on both devices: user is requested to confirm if there is a match by entering yes or not. If yes is selected on both devices, pairing is performed with success. This method allows confirmation to the user that their device is connected with the right one, in a context where there are several devices, which could not have different names.

The selection among the possible methods is based on the following table.

<table>
<thead>
<tr>
<th>Initiator/ responder</th>
<th>Display only</th>
<th>Display yes/no</th>
<th>Keyboard only</th>
<th>No input no output</th>
<th>Keyboard display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display only</td>
<td>Just Works</td>
<td>Just Works</td>
<td>Passkey Entry</td>
<td>Just Works</td>
<td>Passkey Entry</td>
</tr>
<tr>
<td>Display yes/no</td>
<td>Just Works</td>
<td>Just Works (LE legacy) Numeric comparison (LE secure connections)</td>
<td>Passkey Entry</td>
<td>Just Works</td>
<td>Passkey Entry (LE legacy) Numeric comparison (LE secure connections)</td>
</tr>
<tr>
<td>Keyboard only</td>
<td>Passkey Entry</td>
<td>Passkey Entry</td>
<td>Passkey Entry</td>
<td>Just Works</td>
<td>Passkey Entry</td>
</tr>
<tr>
<td>No input no output</td>
<td>Just Works</td>
<td>Just Works</td>
<td>Just Works</td>
<td>Just Works</td>
<td>Just Works</td>
</tr>
<tr>
<td>Keyboard display</td>
<td>Passkey Entry</td>
<td>Passkey Entry (LE legacy) Numeric comparison (LE secure connections)</td>
<td>Passkey Entry</td>
<td>Just Works</td>
<td>Passkey Entry (LE legacy) Numeric comparison (LE secure connections)</td>
</tr>
</tbody>
</table>

**Table 13. Mapping of IO capabilities to possible key generation methods**

*Note: If the possible key generation method does not provide a key that matches the security properties (authenticated - MITM protection or unauthenticated - no MITM protection), then the device sends the pairing failed command with the error code “Authentication Requirements”.*

**Phase 3: transport specific key distribution**

The following keys are exchanged between master and slave:
• Connection signature resolving key (CSRK) for authentication of unencrypted data
• Identity resolving key (IRK) for device identity and privacy

When the established encryption keys are stored in order to be used for future authentication, the devices are bonded.

Data signing
It is also possible to transmit authenticated data over an unencrypted link layer connection by using the CSRK key: a 12-byte signature is placed after the data payload at the ATT layer. The signature algorithm also uses a counter, which protects against replay attacks (an external device that catches some packets and sends them later. The receiver device checks the packet counter and discards it since its frame counter is less than the latest received good packet).

1.8 Privacy

A device, which always advertises with the same address (public or static random), can be tracked by scanners. However, this can be avoided by enabling the privacy feature on the advertising device. On a privacy enabled device, private addresses are used. There are two kinds of private addresses:

• Non-resolvable private address
• Resolvable private address

Non-resolvable private addresses are completely random (except for the two most significant bits) and cannot be resolved. Hence, a device using a non-resolvable private address cannot be recognized by those devices which have not been previously paired. The resolvable private address has a 24-bit random part and a hash part. The hash is derived from the random number and from an IRK (identity resolving key). Hence, only a device that knows this IRK can resolve the address and identify the device. The IRK is distributed during the pairing process. Both types of addresses are frequently changed, enhancing the device identity confidentiality. The privacy feature is not used during the GAP discovery modes and procedures but during GAP connection modes and procedures only.

On Bluetooth Low Energy stacks up to v4.1, the private addresses are resolved and generated by the host. In Bluetooth v4.2, the privacy feature has been updated from version 1.1 to version 1.2. On Bluetooth Low Energy stack v4.2, private addresses can be resolved and generated by the controller, using the device identity information provided by the host.

Peripheral
A privacy-enabled peripheral in a non-connectable mode uses non-resolvable or resolvable private addresses. To connect to a central, the undirected connectable mode only should be used if host privacy is used. If controller privacy is used, the device can also use the directed connectable mode. When in connectable mode, the device uses a resolvable private address. Whether non-resolvable or resolvable private addresses are used, they are automatically regenerated after each interval of 15 minutes. The device does not send the device name to the advertising data.

Central
A privacy-enabled central, performing active scanning, uses non-resolvable or resolvable private addresses only. To connect to a peripheral, the general connection establishment procedure should be used if host privacy is enabled. With controller-based privacy, any connection procedure can be used. The central uses a resolvable private address as the initiator’s device address. A new resolvable or non-resolvable private address is regenerated after each interval of 15 minutes.

Broadcaster
A privacy-enabled broadcaster uses non-resolvable or resolvable private addresses. New addresses are automatically generated after each interval of 15 minutes. A broadcaster should not send the name or unique data to the advertising data.

Observer
A privacy-enabled observer uses non-resolvable or resolvable private addresses. New addresses are automatically generated after each interval of 15 minutes.

1.8.1 The device filtering
Bluetooth LE reduces the number of responses from the devices in order to diminish the power consumption, since this implies less transmissions and less interactions between controller and upper layers. The filtering is implemented by a white-list. When the white-list is enabled, those devices, which are not on this list, are ignored by the link layer.
Before Bluetooth 4.2, the device filtering could not be used, while privacy was used by the remote device. Thanks to the introduction of the link layer privacy, the remote device identity address can be resolved before checking whether it is on the white-list or not.

1.9 Generic attribute profile (GATT)

The generic attribute profile (GATT) defines a framework to use the ATT protocol, and it is used for services, characteristics, descriptors discovery, characteristics reading, writing, indications and notifications.

On GATT context, when two devices are connected, there are two device roles:

- GATT client: the device accesses data on the remote GATT server via read, write, notify, or indicates operations
- GATT server: the device stores data locally and provides data access methods to a remote GATT client

It is possible for a device to be a GATT server and a GATT client at the same time.

The GATT role of a device is logically separated from the master, slave role. The master, slave roles define how the Bluetooth LE radio connection is managed, and the GATT client/server roles are determined by the data storage and flow of data.

It is the most common for the slave (peripheral) device to be the GATT server and the master (central) device to be the GATT client, but this is not required.

Attributes, as transported by the ATT, are encapsulated within the following fundamental types:

1. Characteristics (with related descriptors)
2. Services (primary, secondary and include services)

1.9.1 Characteristic attribute type

A characteristic is an attribute type, which contains a single value and any number of descriptors describing the characteristic value that may make it understandable for the user.

A characteristic exposes the type of data that the value represents, if the value can be read or written, how to configure the value to be indicated or notified, and it says what a value means.

A characteristic has the following components:

1. Characteristic declaration
2. Characteristic value
3. Characteristic descriptor(s)
A characteristic declaration is an attribute defined as follows:

### Table 14. Characteristic declaration

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>Attribute value</th>
<th>Attribute permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xNNNN</td>
<td>0x2803</td>
<td>(UUID for characteristic attribute type)</td>
<td>Characteristic value properties (read, broadcast, write, write without response, notify, indicate, …). Determine how characteristic value can be used or how characteristic descriptor can be accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read only, no authentication, no authorization</td>
</tr>
<tr>
<td></td>
<td>Characteristic value attribute handle</td>
<td>Characteristic value UUID (16 or 128 bits)</td>
<td></td>
</tr>
</tbody>
</table>

A characteristic declaration contains the value of the characteristic. This value is the first attribute after the characteristic declaration:

### Table 15. Characteristic value

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>Attribute value</th>
<th>Attribute permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xNNNN</td>
<td>0xuuuu – 16 bits or 128 bits for characteristic UUID</td>
<td>Characteristic value</td>
<td>Higher layer profile or implementation specific</td>
</tr>
</tbody>
</table>

### 1.9.2 Characteristic descriptor type

Characteristic descriptors are used to describe the characteristic value to add a specific “meaning” to the characteristic and make it understandable for the user. The following characteristic descriptors are available:
1. Characteristic extended properties: it allows extended properties to be added to the characteristic
2. Characteristic user description: it enables the device to associate a text string to the characteristic
3. Client characteristic configuration: it is mandatory if the characteristic can be notified or indicated. Client application must write this characteristic descriptor to enable characteristic notifications or indications (provided that the characteristic property allows notifications or indications)
4. Server characteristic configuration: optional descriptor
5. Characteristic presentation format: it allows the characteristic value presentation format to be defined through some fields as format, exponent, unit name space, description in order to correctly display the related value (example temperature measurement value in °C format)
6. Characteristic aggregation format: it allows several characteristic presentation formats to be aggregated

For a detailed description of the characteristic descriptors, refer to Bluetooth specifications.

1.9.3 Service attribute type

A service is a collection of characteristics, which operates together to provide a global service to an applicative profile. For example, the health thermometer service includes characteristics for a temperature measurement value, and a time interval among measurements. A service or primary service can refer to other services that are called secondary services.

A service is defined as follows:

Table 16. Service declaration

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>Attribute value</th>
<th>Attribute permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xNNNN</td>
<td>0x2800 – UUID for “Primary Service” or 0x2801 – UUID for “Secondary Service”</td>
<td>0xuuuu – 16 bits or 128 bits for service UUID</td>
<td>Read only, no authentication, no authorization</td>
</tr>
</tbody>
</table>

A service contains a service declaration and may contain definitions and characteristic definitions. A "service include declaration" follows the service declaration and any other attributes of the service.

Table 17. Include declaration

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>Attribute value</th>
<th>Attribute permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xNNNN</td>
<td>0x2802 (UUID for include attribute type)</td>
<td>Include service attribute handle, End group handle, Service UUID</td>
<td>Read only, no authentication, no authorization</td>
</tr>
</tbody>
</table>

"Include service attribute handle" is the attribute handle of the included secondary service and "end group handle" is the handle of the last attribute within the included secondary service.

1.9.4 GATT procedures

The generic attribute profile (GATT) defines a standard set of procedures allowing services, characteristics, related descriptors to be known and how to use them.

The following procedures are available:
• Discovery procedures (Table 18. Discovery procedures and related response events)
• Client-initiated procedures (Table 19. Client-initiated procedures and related response events)
• Server-initiated procedures (Table 20. Server-initiated procedures and related response events)
### Table 18. Discovery procedures and related response events

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Response events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery all primary services</td>
<td>Read by group response</td>
</tr>
<tr>
<td>Discovery primary service by service UUID</td>
<td>Find by type value response</td>
</tr>
<tr>
<td>Find included services</td>
<td>Read by type response event</td>
</tr>
<tr>
<td>Discovery all characteristics of a service</td>
<td>Read by type response</td>
</tr>
<tr>
<td>Discovery characteristics by UUID</td>
<td>Read by type response</td>
</tr>
<tr>
<td>Discovery all characteristic descriptors</td>
<td>Find information response</td>
</tr>
</tbody>
</table>

### Table 19. Client-initiated procedures and related response events

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Response events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read characteristic value</td>
<td>Read response event</td>
</tr>
<tr>
<td>Read characteristic value by UUID</td>
<td>Read response event</td>
</tr>
<tr>
<td>Read long characteristic value</td>
<td>Read blob response events</td>
</tr>
<tr>
<td>Read multiple characteristic values</td>
<td>Read response event</td>
</tr>
<tr>
<td>Write characteristic value without response</td>
<td>No event is generated</td>
</tr>
<tr>
<td>Signed write without response</td>
<td>No event is generated</td>
</tr>
<tr>
<td>Write characteristic value</td>
<td>Write response event</td>
</tr>
<tr>
<td>Write long characteristic value</td>
<td>Prepare write response</td>
</tr>
<tr>
<td>Reliable write</td>
<td>Execute write response</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 20. Server-initiated procedures and related response events

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Response events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notifications</td>
<td>No event is generated</td>
</tr>
<tr>
<td>Indications</td>
<td>Confirmation event</td>
</tr>
</tbody>
</table>

For a detailed description of the GATT procedures and related response events refer to the Bluetooth specifications in Section 5 References.

### 1.10 Generic access profile (GAP)

The Bluetooth system defines a base profile implemented by all Bluetooth devices called generic access profile (GAP). This generic profile defines the basic requirements of a Bluetooth device. The four GAP profile roles are described in the table below:

### Table 21. GAP roles

<table>
<thead>
<tr>
<th>Role(1)</th>
<th>Description</th>
<th>Transmitter</th>
<th>Receiver</th>
<th>Typical example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcaster</td>
<td>Sends advertising events</td>
<td>M</td>
<td>O</td>
<td>Temperature sensor which sends temperature values</td>
</tr>
</tbody>
</table>
### Role Description

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
<th>Transmitter</th>
<th>Receiver</th>
<th>Typical example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>Receives advertising events</td>
<td>O</td>
<td>M</td>
<td>Temperature display which just receives and displays temperature values</td>
</tr>
<tr>
<td>Peripheral</td>
<td>Always a slave. It is on connectable advertising mode. Supports all LL control procedures; encryption is optional</td>
<td>M</td>
<td>M</td>
<td>Watch</td>
</tr>
<tr>
<td>Central</td>
<td>Always a master. It never advertises. It supports active or passive scan. It supports all LL control procedures; encryption is optional</td>
<td>M</td>
<td>M</td>
<td>Mobile phone</td>
</tr>
</tbody>
</table>

1. 1. **M** = mandatory; **O** = optional

On GAP context, two fundamental concepts are defined:
- **GAP modes**: it configures a device to act in a specific way for a long time. There are four GAP mode types: broadcast, discoverable, connectable and bondable type.
- **GAP procedures**: it configures a device to perform a single action for a specific, limited time. There are four GAP procedure types: observer, discovery, connection, bonding procedures.

Different types of discoverable and connectable modes can be used at the same time. The following GAP modes are defined:

#### Table 22. GAP broadcaster mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Notes</th>
<th>GAP role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast mode</td>
<td>Device only broadcasts data using the link layer advertising channels and packets (it does not set any bit on Flags AD type)</td>
<td>Broadcast data can be detected by a device using the observation procedure</td>
<td>Broadcaster</td>
</tr>
</tbody>
</table>

#### Table 23. GAP discoverable modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Notes</th>
<th>GAP role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-discoverable mode</td>
<td>It cannot set the limited and general discoverable bits on flags AD type</td>
<td>It cannot be discovered by a device performing a general or limited discovery procedure</td>
<td>Peripheral</td>
</tr>
<tr>
<td>Limited discoverable mode</td>
<td>It sets the limited discoverable bit on flags AD type</td>
<td>It is allowed for about 30 s. It is used by devices with which the user has recently interacted. For example, when a user presses a button on the device</td>
<td>Peripheral</td>
</tr>
<tr>
<td>General discoverable mode</td>
<td>It sets the general discoverable bit on flags AD type</td>
<td>It is used when a device wants to be discoverable. There is no limit on the discoverability time</td>
<td>Peripheral</td>
</tr>
</tbody>
</table>

#### Table 24. GAP connectable modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Notes</th>
<th>GAP role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-connectable mode</td>
<td>It can only use ADV_NONCONN_IND or ADV_SCAN_IND advertising packets</td>
<td>It cannot use a connectable advertising packet when it advertises</td>
<td>Peripheral</td>
</tr>
<tr>
<td>Mode</td>
<td>Description</td>
<td>Notes</td>
<td>GAP role</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Direct connectable mode</td>
<td>It uses the ADV_DIRECT advertising packet</td>
<td>It is used from a peripheral device that wants to connect quickly to a central device. It can be used only for 1.28 seconds, and it requires both peripheral and central device addresses</td>
<td>Peripheral</td>
</tr>
<tr>
<td>Undirected connectable mode</td>
<td>It uses the ADV_IND advertising packet</td>
<td>It is used from a device that wants to be connectable. Since ADV_IND advertising packet can include the flag AD type, a device can be in discoverable and undirected connectable mode at the same time. Connectable mode is terminated when the device moves to connection mode or when it moves to non-connectable mode</td>
<td>Peripheral</td>
</tr>
</tbody>
</table>

Table 25. GAP bondable modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Notes</th>
<th>GAP role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-bondable mode</td>
<td>It does not allow a bond to be created with a peer device</td>
<td>No keys are stored from the device</td>
<td>Peripheral</td>
</tr>
<tr>
<td>Bondable mode</td>
<td>Device accepts bonding request from a central device</td>
<td></td>
<td>Peripheral</td>
</tr>
</tbody>
</table>

The following GAP procedures are defined in Table 6. Data packet header content:

Table 26. GAP observer procedure

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Notes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observation procedure</td>
<td>It allows a device to look for broadcaster device data</td>
<td></td>
<td>Observer</td>
</tr>
</tbody>
</table>

Table 27. GAP discovery procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Notes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited discoverable procedure</td>
<td>It is used for discovery of peripheral devices in limited discovery mode</td>
<td>Device filtering is applied based on flag AD type information</td>
<td>Central</td>
</tr>
<tr>
<td>General discoverable procedure</td>
<td>It is used for discovery of peripheral devices in general and limited discovery mode</td>
<td>Device filtering is applied based on flag AD type information</td>
<td>Central</td>
</tr>
<tr>
<td>Name discovery procedure</td>
<td>It is the procedure to retrieve the &quot;Bluetooth Device Name&quot; from connectable devices</td>
<td></td>
<td>Central</td>
</tr>
</tbody>
</table>

Table 28. GAP connection procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Notes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto connection establishment procedure</td>
<td>Allows connection with one or more devices in the directed connectable mode or the undirected connectable mode</td>
<td>It uses whitelists</td>
<td>Central</td>
</tr>
</tbody>
</table>
### Procedure Description Notes Role

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Notes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>General connection establishment procedure</td>
<td>Allows a connection with a set of known peer devices in the directed connectable mode or the undirected connectable mode</td>
<td>It supports private addresses by using the direct connection establishment procedure when it detects a device with a private address during the passive scan</td>
<td>Central</td>
</tr>
<tr>
<td>Selective connection establishment procedure</td>
<td>Establishes a connection with the host selected connection configuration parameters with a set of devices in the whitelist</td>
<td>It uses white-lists and it scans by this white-list</td>
<td>Central</td>
</tr>
<tr>
<td>Direct connection establishment procedure</td>
<td>Establishes a connection with a specific device using a set of connection interval parameters</td>
<td>General and selective procedures use it</td>
<td>Central</td>
</tr>
<tr>
<td>Connection parameter update procedure</td>
<td>Updates the connection parameters used during the connection</td>
<td></td>
<td>Central</td>
</tr>
<tr>
<td>Terminate procedure</td>
<td>Terminates a GAP procedure</td>
<td></td>
<td>Central</td>
</tr>
</tbody>
</table>

**Table 29. GAP bonding procedures**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Notes</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bonding procedure</td>
<td>Starts the pairing process with the bonding bit set on the pairing request</td>
<td></td>
<td>Central</td>
</tr>
</tbody>
</table>

For a detailed description of the GAP procedures, refer to the Bluetooth specifications.

### 1.11 Bluetooth LE profiles and applications

A service collects a set of characteristics and exposes the behavior of these characteristics (what the device does, but not how a device uses them). A service does not define characteristic use cases. Use cases determine which services are required (how to use services on a device). This is done through a profile defining which services are required for a specific use case:

- Profile clients implement use cases
- Profile servers implement services

Standard profiles or proprietary profiles can be used. When using a non-standard profile, a 128-bit UUID is required and must be generated randomly.

Currently, any standard Bluetooth SIG profile (services, and characteristics) uses 16-bit UUIDs. Services, characteristic specifications and UUID assignment can be downloaded from the following SIG web pages:

- [https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx](https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx)
- [https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx](https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx)
1.11.1 Proximity profile example

This section describes the proximity profile target, how it works and required services:

**Target**

- When a device is close, very far, far away:
  - Causes an alert

**How it works**

- If a device disconnects, it causes an alert
- Alert on link loss: «Link Loss» service
  - If a device is too far away
  - Causes an alert on path loss: «Immediate Alert» and «Tx Power» service
- «Link Loss» service
  - «Alert Level» characteristic
  - Behavior: on link loss, causes alert as enumerated
- «Immediate Alert» service
  - «Alert Level» characteristic
  - Behavior: when written, causes alert as enumerated
- «Tx Power» service
  - «Tx Power» characteristic
  - Behavior: when read, reports current Tx Power for connection.

- Use case 1 uses Service A and B
- Use case 2 uses Service B
Bluetooth Low Energy (LE) stack v3.x provides a new architecture, which offers further advantages rather than the previous Bluetooth LE stack v2.x family.

The new architecture provides the following new features with related benefits:

1. Code is more modular and testable in isolation:
   - Test coverage is increased

2. Hardware dependent parts are provided in source form:
   - New sleep timer module external to Bluetooth LE stack (Init API and tick API to be called on user main application)
   - New NVM module external to Bluetooth LE stack (Init API and tick API to be called on user main application).

   **Note:** It makes easy customize or fix bugs

3. Certification targets the protocol part only:
   - It reduces the number of stack versions, since hardware related problems are mostly isolated in other modules
   - It reduces the number of certifications

4. It implements more flexible and robust radio activity scheduler
   - It allows the robustness against late interrupt routines (e.g. Flash writes and/or interrupt disabled by the user)

5. It reduces real time constraint (less code in interrupt handler)
   - System gives more time to applications
Bluetooth Low Energy (Bluetooth LE) stack v3.x is a standard C library, in binary format, which provides a high-level interface to control STMicroelectronics devices Bluetooth Low Energy functionalities. The Bluetooth LE binary library provides the following functionalities:

- Stack APIs for
  - Bluetooth LE stack initialization
  - Bluetooth LE stack application command interface (HCI command prefixed with hci_, and vendor specific command prefixed with aci_)
  - Sleep timer access
  - Bluetooth LE stack state machine handling

- Stack event callbacks
  - Inform user application about Bluetooth LE stack events
  - Sleep timer events

- Interrupt handler for radio IP

In order to get access to the Bluetooth LE stack functionalities, user application is just requested to:

- Call the related stack APIs
- Handle the expected events through the provided stack callbacks
- Link the Bluetooth LE stack binary library to the user application, as described in Figure 12. Bluetooth LE stack reference application.

*Figure 12. Bluetooth LE stack reference application*
1. **API** is a C function defined by the Bluetooth LE stack library and called by user application.
2. A callback is a C function called by the Bluetooth LE stack library and defined by the user application.
3. Driver sources are a set of drivers (header and source files) which handles all the Bluetooth LE device peripherals (ADC, P2C, SPI, timers, watchdog, GPIOs, RTC, UART, ...).
4. Bluetooth LE radio initialization structure and related application configuration files have been modified if compared to Bluetooth LE stack v2.x family
5. Bluetooth LE stack APIx naming for Bluetooth LE stack init, Bluetooth LE stack tick has been modified if compared to Bluetooth LE stack v2.x family
6. New GAP APIs interface for GAP initialization, scanning, connection and advertising APIs/events have been added if compared to Bluetooth LE stack v2.x family
7. New GATT APIs/event framework and interface have been defined if compared to Bluetooth LE stack v2.x family

### 2.1 Bluetooth LE stack library framework

The Bluetooth LE stack library framework allows commands to be sent to Bluetooth LE stack and it also provides definitions of Bluetooth LE event callbacks.

The Bluetooth LE stack APIs utilize and extend the standard HCI data format defined within the Bluetooth specifications.

The provided set of APIs supports the following commands:

- Standard HCI commands for controller as defined by Bluetooth specifications
- Vendor Specific (VS) HCI commands for controller
- Vendor Specific (VS) ACI commands for host (L2CAP, ATT, SM, GATT, GAP)

The reference Bluetooth LE API interface framework is provided within the supported ST Bluetooth LE devices DK software package targeting the related DK platforms (refer to Section 5 References).

The Bluetooth LE stack library framework interface for the BlueNRG-LP device is defined by the following header files:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ble_status.h</td>
<td>Header file for Bluetooth LE stack error codes</td>
<td>Middlewares\ST\Bluetooth_LE\inc</td>
<td></td>
</tr>
<tr>
<td>bluenrg_lp_api.h</td>
<td>Header file for Bluetooth LE stack APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bluenrg_lp_events.h</td>
<td>Header file for Bluetooth LE stack events callbacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bluenrg_lp_gatt.h</td>
<td>Header file for the BlueNRG-LP Bluetooth LE GATT</td>
<td></td>
<td>It provides new GATT structures definition</td>
</tr>
<tr>
<td>stack_user_cfg.h</td>
<td>Bluetooth LE stack configuration header file</td>
<td></td>
<td>It provides the available configuration options for Bluetooth LE stack v3.x</td>
</tr>
<tr>
<td>stack_user_cfg.c</td>
<td>Bluetooth LE stack configuration file</td>
<td>Middlewares\ST\Bluetooth_LE\src</td>
<td>Source file to be included on user application IDE project in order to support the Bluetooth LE modular approach available with Bluetooth LE stack v3.x</td>
</tr>
</tbody>
</table>

**Table 30. Bluetooth LE stack library framework interface**
Note: Bluetooth LE stack v3.x or later provides the capability to enable/disable, at compile time, the following Bluetooth LE stack features based on user specific application scenario:
1. Enable/disable controller privacy
2. Enable/disable LE secure connections
3. Enable/disable master role
4. Enable/disable data length extension (valid only for the device supporting the data length extension feature)
5. Enable/disable LE 2M and LE Coded PHYs features
6. Enable/disable extended advertising and scanning features
7. Enable/disable L2CAP, connection oriented data service feature (L2CAP-COS)

This allows the user to potentially exclude some features from the available Bluetooth LE stack binary library and decrease the overall Flash memory footprint.

Refer to the Bluetooth LE stack preprocessor configuration options defined on file stack_user_cfg.h, in order to identify which are the available and supported combinations.

### Table 31. Bluetooth LE application stack library framework interface

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ble_const.h</td>
<td>It includes the required Bluetooth LE stack header files</td>
<td>Middlewares\ST\BLE_Application\layers_inc</td>
<td>To be included on the user main application</td>
</tr>
<tr>
<td>bluenrg_lp_gap.h</td>
<td>Header file for GAP layer constants</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>bluenrg_lp_gatt_server.h</td>
<td>Header file for GATT server constants</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>bluenrg_lp_hal.h</td>
<td>Header file with HAL</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>bluenrg_lp_l2cap.h</td>
<td>Header file for l2cap</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>bluenrg_lp_stack.h</td>
<td>Header file for Bluetooth LE stack initialization, tick and sleep timer APIs</td>
<td>&quot;&quot;</td>
<td>To be included on the user main application</td>
</tr>
<tr>
<td>hci_const.h</td>
<td>It contains constants for HCI layer</td>
<td>&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>link_layer.h</td>
<td>Header file for link layer constants</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>sm.h</td>
<td>Header file for security manager constants</td>
<td>&quot;&quot;</td>
<td>It is included through ble_const.h header file</td>
</tr>
<tr>
<td>gap_profile.[ch]</td>
<td>Header and source files for generic access profile service (GAP library)</td>
<td>Middlewares\BLE_Application\Profiles</td>
<td></td>
</tr>
<tr>
<td>gatt_profile.[ch]</td>
<td>Header and source files for generic attribute profile service (GATT library)</td>
<td>Middlewares\BLE_Application\Profiles</td>
<td></td>
</tr>
<tr>
<td>att_pwrq.[ch]</td>
<td>Header and source files for ATT prepare write queue implementation library</td>
<td>Middlewares\BLE_Application\Queued_Write</td>
<td></td>
</tr>
</tbody>
</table>

Note: The AES CMAC encryption functionality required by Bluetooth LE stack is available on new standalone binary library: cryptolib\cryptolib.a. This library must also be included on the user application IDE project.

### 2.2 Bluetooth LE stack event callbacks

The Bluetooth LE stack library framework provides a set of events and related callbacks which are used to notify the user application of specific events to be processed.
The Bluetooth LE event callback prototypes are defined on header file bluenrg lp_events.h. All callbacks are defined by default through weak definitions (no check is done on event callback name defined by the user, so the user should carefully check that each defined callbacks is in line with the expected function name).

The user application must define the used events callbacks with application code, inline with specific application scenario.

### 2.3 Bluetooth LE stack init and tick APIs

The Bluetooth LE stack v3.x must be initialized in order to properly configure some parameters inline with specific application scenario.

The following API must be called before using any other Bluetooth LE stack v3.x functionality:

```c
BLE_STACK_Init(&BLE_STACK_InitParams);
```

**BLE_STACK_InitParams** is a variable which contains memory and low level hardware configuration data for the device, and it is defined using this structure:

```c
typedef struct {
    uint8_t* BLEStartRamAddress;
    uint32_t TotalBufferSize;
    uint16_t NumAttrRecord;
    uint8_t MaxNumOfClientProcs;
    uint8_t NumOfLinks;
    uint16_t NumBlockCount;
    uint16_t ATT_MTU;
    uint32_t MaxConnEventLength;
    uint16_t SleepClockAccuracy;
    uint8_t *HotAnaConfigTable;
    uint8_t NumOfAdvDataSet;
    uint8_t NumOfAuxScanSlots;
    uint8_t WhiteListSizeLog2;
    uint16_t L2CAP_MPS;
    uint8_t L2CAP_NumChannels;
} BLE_STACK_InitParams;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEStartRamAddress</td>
<td>Start address of the RAM buffer for GATT database allocated according to TOTAL_BUFFER_SIZE</td>
<td>32-bit aligned RAM area</td>
</tr>
<tr>
<td>TotalBufferSize</td>
<td>TOTAL_BUFFER_SIZE return value, used to check the MACRO correctness</td>
<td>Refer to lMiddlewares\ST\Bluetooth_LE\Inc \bluenrg_lp_stack.h file</td>
</tr>
<tr>
<td>NumAttrRecord</td>
<td>Maximum number of attributes (i.e. the number of characteristics + the number of characteristic values + the number of descriptors + the number of the services) that can be stored in the GATT database</td>
<td>For each characteristic, the number of attributes goes from 2 to 5 depending on the characteristic properties: minimum of 2 (one for declaration and one for the value). Add one more record for each additional property: notify or indicate, broadcast, extended property.</td>
</tr>
<tr>
<td>MaxNumOfClientProcs</td>
<td>Maximum number of concurrent client procedures</td>
<td>This value is less or equal to NumOfLinks</td>
</tr>
<tr>
<td>NumOfLinks</td>
<td>Maximum number of simultaneous connections that the device can support</td>
<td>Valid values are from 1 to 24 (NUM_LINKS used in the calculation of TOTAL_BUFFER_SIZE)</td>
</tr>
<tr>
<td>NumBlockCount</td>
<td>Number of allocated memory blocks for the Bluetooth LE stack</td>
<td>The minimum required value is calculated using a specific macro provided on bluenrg_lp_stack.h file: BLE_STACK_MBLOCKS_CALC()</td>
</tr>
<tr>
<td>ATT_MTU</td>
<td>Maximum supported ATT_MTU size</td>
<td>Supported values range is 23, 247 bytes</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>MaxConnEventLength</td>
<td>Maximum duration of the connection event when the device is in Slave mode in units of 625/256 us (~2.44 us)</td>
<td>&lt;= 4000 (ms)</td>
</tr>
<tr>
<td>SleepClockAccuracy</td>
<td>Sleep clock accuracy</td>
<td>ppm value</td>
</tr>
<tr>
<td>HotAnaConfigTable</td>
<td>Table containing register values to be set before the radio activity</td>
<td></td>
</tr>
<tr>
<td>NumOfAdvDataSet</td>
<td>Maximum number of advertising data set, valid only when advertising extension feature is enabled</td>
<td></td>
</tr>
<tr>
<td>NumOfAuxScanSlots</td>
<td>Maximum number of slots for scanning on the secondary advertising channel, valid only when advertising extension feature is enabled</td>
<td></td>
</tr>
<tr>
<td>WhiteListSizeLog2</td>
<td>Two’s logarithm of the white/resolving list size</td>
<td></td>
</tr>
<tr>
<td>L2CAP_MPS</td>
<td>The maximum size of payload data in octets that the L2CAP layer entity is capable of accepting</td>
<td></td>
</tr>
<tr>
<td>L2CAP_NumChannels</td>
<td>Maximum number of channels in LE Credit Based Flow Control mode</td>
<td></td>
</tr>
</tbody>
</table>

### 2.4 The Bluetooth LE stack v3.x application configuration

During the device initialization phase, after STMicroelectronics Bluetooth LE device powers on, some specific parameters must be defined on the Bluetooth LE device controller registers, in order to define the following configurations:

- Low speed crystal source: external 32 kHz oscillator, internal RO
- SMPS: on or off (if on: 2.2 μH, 1.5 μH or 10 μH SMPS inductor)

The BlueNRG-LP application configuration parameters are defined on file system_bluenrg_lp.c through the following configuration table:

<table>
<thead>
<tr>
<th>Preprocessor option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_HW_LS_XTAL</td>
<td>Low speed crystal source: external 32 kHz oscillator</td>
</tr>
<tr>
<td>CONFIG_HW_LS_RO</td>
<td>Low speed crystal source: internal RO</td>
</tr>
<tr>
<td>CONFIG_HW_SMPS_10uH</td>
<td>Enable SMPS with 10 μH</td>
</tr>
<tr>
<td>CONFIG_HW_SMPS_2_2uH</td>
<td>Enable SMPS with 2.2 μH inductor</td>
</tr>
<tr>
<td>CONFIG_HW_SMPS_1_5uH</td>
<td>Enable SMPS with 1.5 μH inductor</td>
</tr>
<tr>
<td>CONFIG_HW_SMPS_NONE</td>
<td>Disable SMPS</td>
</tr>
<tr>
<td>CONFIG_HW_HSE_TUNE</td>
<td>HSE capacitor configuration: [0-63] as values range</td>
</tr>
</tbody>
</table>

### 2.5 Bluetooth LE stack tick function

The Bluetooth LE stack v3.x provides a special API `BLE_STACK_Tick()`, which must be called in order to process the internal Bluetooth LE stack state machines and when there are Bluetooth LE stack activities ongoing (normally within the main application while loop).

The `BLE_STACK_Tick()` function executes the processing of all host stack layers and it has to be executed regularly to process incoming link layer packets and to process host layers procedures. All stack callbacks are called by this function.

If a low speed ring oscillator is used instead of the LS crystal oscillator, this function also performs the LS RO calibration and hence must be called at least once at every system wake-up in order to keep the 500 ppm accuracy (at least 500 ppm accuracy is mandatory if acting as a master).
Note: No Bluetooth LE stack function must be called while the `BLE_STACK_Tick()` is being run. For example, if a Bluetooth LE stack function may be called inside an interrupt routine, that interrupt must be disabled during the execution of `BLE_STACK_Tick()`.

Example: if a stack function may be called inside UART ISR, the following code should be used:

```c
NVIC_DisableIRQ(UART_IRQn);
BLE_STACK_Tick();
NVIC_EnableIRQ(UART_IRQn);
```
This section provides some information and code examples about how to design and implement a Bluetooth Low Energy application using the Bluetooth LE stack v3.x binary library.

A user implementing a Bluetooth LE stack v3.x application has to go through some basic and common steps:

1. Initialization phase and main application loop
2. Bluetooth LE stack events callbacks setup
3. Services and characteristic configuration (on GATT server)
4. Create a connection: discoverable, connectable modes and procedures
5. Security (pairing and bonding)
6. Service and characteristic discovery
7. Characteristic notification/indications, write, read
8. Basic/typical error conditions description

Note: In the following sections, some user application “defines” are used to simply identify the Bluetooth Low Energy device role (central, peripheral, client and server). Furthermore the BlueNRG-LP device is used as reference device running Bluetooth LE stack v3.x applications. Any specific device dependent part is highlighted whenever it is needed.

### Table 34. User application defines for Bluetooth LE device roles

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATT_CLIENT</td>
<td>GATT client role</td>
</tr>
<tr>
<td>GATT_SERVER</td>
<td>GATT server role</td>
</tr>
</tbody>
</table>

### 3.1 Initialization phase and main application loop

The following main steps are required to properly configure the Bluetooth LE device running a Bluetooth LE stack v3.x application:

1. Initialize the Bluetooth LE device vector table, interrupt priorities, clock: `SystemInit()` API
2. Initialize IOs for power save modes using the `BSP_IO_Init()` API, and the serial communication channel used for I/O communication as debug and utility information: `BSP_COM_Init()` API
3. Initialize the Bluetooth LE stack: `BLE_STACK_Init(&BLE_STACK_InitParams)` API
4. Initialize the timer module: `HAL_VTIMER_Init(&VTIMER_InitStruct)` API
5. Initialize the NVM module: `BLEPLAT_Init()` API
6. Configure Bluetooth LE device public address (if public address is used): `aci_hal_write_config_data()` API
7. Init Bluetooth LE GATT layer: `aci_gatt_srv_init()` API
8. Init Bluetooth LE GAP layer depending on the selected device role: `aci_gap_init("role")` API
9. Set the proper security I/O capability and authentication requirement (if Bluetooth LE security is used)
10. Define the required Services & Characteristics & Characteristics Descriptors if the device is a GATT server
11. Add a while(1) loop calling the Timer module tick API `HAL_VTIMER_Tick()`, Bluetooth LE stack tick API `BTLE_StackTick()`, NVM module tick API `NVMDB_Tick()` and a specific user tick handler where user actions/events are processed. Furthermore, a call to the `HAL_PWR_MNGR_Request()` API is added in order to enable device sleep mode and preserve the Bluetooth LE radio operating modes.

The following pseudocode example illustrates the required initialization steps:
```c
int main(void)
{
    WakeupSourceConfig_TypeDef wakeupIO;
    PowerSaveLevels stopLevel;
    BLE_STACK_InitTypeDef BLE_STACK_InitParams = BLE_STACK_INIT_PARAMETERS;
    
    /* System initialization function */
    if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M) != SUCCESS)
    {
        /* Error during system clock configuration take appropriate action */
        while(1);
    }
    /* Configure IOs for power save modes */
    BSP_IO_Init();
    /* Configure I/O communication channel */
    BSP_COM_Init(BSP_COM_RxDataUserCb);
    
    /* Enable clock for PKA and RNG IPs used by BLE radio */
    LL_AHB_EnableClock(LL_AHB_PERIPH_PKA|LL_AHB_PERIPH_RNG);
    /* Bluetooth LE stack init */
    ret = BLE_STACK_Init(&BLE_STACK_InitParams);
    if (ret != BLE_STATUS_SUCCESS) {
        printf("Error in BLE_STACK_Init() 0x%02x\r\n", ret);
        while(1);
    }
    /* Init Virtual Timer module */
    HAL_VTIMER_InitType VTIMER_InitStruct = {HS_STARTUP_TIME,
        INITIAL_CALIBRATION,
        CALIBRATION_INTERVAL};
    HAL_VTIMER_Init(&VTIMER_InitStruct);
    /* Init NVM module */
    BLEPLAT_Init();
    
    /* Device Initialization: GATT and GAP Init APIs*/
    It could add services and characteristics (if it is a GATT server)/
    It could also initialize its state machine and other specific drivers (
    i.e. leds, buttons, sensors, ..) */
    ret = DeviceInit();
    if (ret != BLE_STATUS_SUCCESS) {
        /* Set wakeup sources if needed through the wakeupIO. Variable */
        /* No wakeup sources: */
        wakeupIO.IO_Mask_High_polarity = 0;
        wakeupIO.IO_Mask_Low_polarity = 0;
        wakeupIO.RTC_enable = 0;
        while(1);
    }
    /* Timer tick */
    HAL_VTIMER_Tick();
    /* BLE stack tick */
    BLE_STACK_Tick();
    /* NVM manager tick */
    NVMDB_Tick();
    /* Application Tick: user application where application
    state machine is handled*/
    APP_Tick();
    /* Power Save management*/
    /* It enables power save modes with wakeup on radio operating timings
    (advertising, connections intervals) */
    HAL_PWR_MNGR_Request(POWER_SAVE_LEVEL_STOP_NOTIMER, wakeupIO, &stopLevel);
    /* while (1) */
}
/* end main() */
```

1. **BLE_STACK_InitParams** variable defines the stack initialization parameters as described on BLE stack event callbacks.

2. **BLE_STACK_Tick()** must be called in order to process stack events.

3. **APP_Tick()** is just an application dependent function, which handles the user application state machine, according to the application working scenario.

4. **HAL_PWR_MNGR_Request(POWER_SAVE_LEVEL_STOP_NOTIMER, wakeupIO, &stopLevel)** enables the device HW power stop mode with no timer: the device is in deep sleep. All the peripherals and clock sources are turned off. Wakeup is possible only from GPIOs (PA0 to PA15 and PB0 to PB11). It is worth noticing that this API with the specified parameters must be called, on application main while loop, in order to allow the Bluetooth LE device to enter sleep mode with wake-up source on Bluetooth LE stack advertising and connection intervals. If not called, the Bluetooth LE device always remains in running power save mode (Bluetooth LE stack does not autonomously enter sleep mode unless this specific API is called). The user application can use the **HAL_PWR_MNGR_Request()** API to select one of the supported Bluetooth LE device HW power save modes and set the related wake-up sources and sleep timeout, when applicable. The **HAL_PWR_MNGR_Request()** API combines the low power requests coming from the application with the radio operating mode, choosing the best low power mode applicable in the current scenario.

   The negotiation between the radio module and the application requests is done to avoid losing data exchanged over-the-air.

5. For more information about the **HAL_PWR_MNGR_Request()** API and BLE device low power modes refer to the related application note in Section 5 References at the end of this document.

6. The last attribute handles reserved for the standard GAP service is 0x000F when no privacy or host-based privacy is enabled on **aci_gap_init()** API, 0x0011 when controller-based privacy is enabled on **aci_gap_init()** API.

### Table 35. GATT, GAP service handles

<table>
<thead>
<tr>
<th>Service</th>
<th>Start handle</th>
<th>End handle</th>
<th>Service UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute profile service</td>
<td>0x0001</td>
<td>0x0008</td>
<td>0x1801</td>
</tr>
<tr>
<td>Generic access profile (GAP) service</td>
<td>0x0009</td>
<td>0x000F</td>
<td>0x1800</td>
</tr>
</tbody>
</table>

### Table 36. GATT, GAP characteristic handles

<table>
<thead>
<tr>
<th>Default services</th>
<th>Characteristic</th>
<th>Attribute handle</th>
<th>Char property</th>
<th>Char value handle</th>
<th>Char UUID</th>
<th>Char value length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Service changed</td>
<td>0x0002</td>
<td>Indicate</td>
<td>0x0003</td>
<td>0x2A05</td>
<td>4</td>
</tr>
<tr>
<td>profile service</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic access</td>
<td>Device name</td>
<td>0x000A</td>
<td>Read</td>
<td>write without response</td>
<td>0x000B</td>
<td>0x2A00</td>
</tr>
<tr>
<td>profile (GAP)</td>
<td>Appearance</td>
<td>0x000C</td>
<td>Read</td>
<td>write without response</td>
<td>0x000D</td>
<td>0x2A01</td>
</tr>
<tr>
<td>service</td>
<td>Peripheral preferred connection parameters</td>
<td>0x000E</td>
<td>Read</td>
<td>write</td>
<td>0x000F</td>
<td>0x2A04</td>
</tr>
</tbody>
</table>
The aci_gap_init() role parameter values are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Role parameter values</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>0x01: Peripheral</td>
<td>The role parameter can be a bitwise OR of any of the supported values (multiple roles simultaneously support)</td>
</tr>
<tr>
<td></td>
<td>0x02: Broadcaster</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x04: Central</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x08: Observer</td>
<td></td>
</tr>
<tr>
<td>Privacy_Type</td>
<td>0x00 for disabling privacy;</td>
<td>Specify whether privacy is enabled or not and which one</td>
</tr>
<tr>
<td></td>
<td>0x01 for enabling privacy;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x02 for enabling controller-based privacy</td>
<td></td>
</tr>
<tr>
<td>device_name_char_len</td>
<td>0x00: public address</td>
<td>It allows the length of the device name characteristic to be indicated</td>
</tr>
<tr>
<td>Identity_Address_Type</td>
<td>0x01: static random address</td>
<td>Specify which address has to be used as identity address</td>
</tr>
</tbody>
</table>

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and event documentation, in Section 5 References.

### 3.1.1 Bluetooth LE addresses

The following device addresses are supported by the Bluetooth LE stack:
- Public address
- Random address
- Private address

Public MAC addresses (6 bytes - 48 bits address) uniquely identify a Bluetooth LE device, and they are defined by the Institute of Electrical and Electronics Engineers (IEEE).

The first 3 bytes of the public address identify the company that issued the identifier and are known as the Organizationally Unique Identifier (OUI). An Organizationally Unique Identifier (OUI) is a 24-bit number that is purchased from the IEEE. This identifier uniquely identifies a company and it allows a block of possible public addresses to be reserved (up to 2^24 coming from the remaining 3 bytes of the public address) for the exclusive use of a company with a specific OUI.

An organization/company can request a new set of 6-byte addresses when at least 95% of the previously allocated block of addresses have been used (up to 2^24 possible addresses are available with a specific OUI).

If the user wants to program their custom MAC address, they have to store it on a specific device Flash location used only for storing the MAC address. Then, at device power-up, it has to program this address on the radio by calling a specific stack API.

The Bluetooth LE API command to set the MAC address is aci_hal_write_config_data()

The command aci_hal_write_config_data() should be sent to Bluetooth LE devices before starting any Bluetooth LE operations (after stack initialization API BLE_STACK_Init()).
The following pseudocode example illustrates how to set a public address:

```c
uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret=aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET, CONFIG_DATA_PUBADD_LEN, bdaddr);
if(ret) PRINTF("Setting address failed.\n")
```

MAC address needs to be stored in the specific Flash location associated to the MAC address during the product manufacturing.

A user can write its application assuming that the MAC address is placed at a known specific MAC Flash location of the device. During manufacturing, the microcontroller can be programmed with the customer Flash image via SWD.

A second step could involve generating the unique MAC address (i.e. reading it from a database) and storing of the MAC address in the known MAC Flash location.

**Figure 13. MAC address storage**

The BlueNRG-LP device does not have a valid preassigned MAC address, but a unique serial number (read only for the user). The unique serial number is a 6-byte value stored at address 0x10001EF0: it is stored as two words (8 bytes) at address 0x10001EF0 and 0x10001EF4 with unique serial number padded with 0xAA55.
The static random address is generated and programmed at every 1st boot of the device on the dedicated Flash area. The value on Flash is the actual value the device uses: each time the user resets the device the stack checks if valid data are on the dedicated Flash area and it uses it (a special valid marker on FLASH is used to identify if valid data are present). If the user performs mass erase, the stored values (including marker) are removed so the stack generates a new random address and stores it on the dedicated Flash.

Private addresses are used when privacy is enabled and according to the Bluetooth low energy specification. For more information about private addresses, refer to Security manager (SM).

3.1.2 Set tx power level

During the initialization phase, the user can also select the transmitting power level using the following APIs:

```c
aci_hal_set_tx_power_level(high, power level)
```

Follow a pseudocode example for setting the radio transmit power at 0 dBm output power:

```c
ret= aci_hal_set_tx_power_level (0, 25);
```

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and event documentation, in Section 5 References.

3.2 Bluetooth LE stack v3.x GATT interface

3.2.1 Bluetooth LE stack v3.x vs Bluetooth LE stack v2.x

The new ATT/GATT component is designed to optimize the memory footprint and usage. To achieve this result, the GATT component avoids allocating the static buffer. The user application allocates such resources and provides them to the stack library on demand. Instead of allocating space for attribute values inside the Bluetooth LE stack buffer, the Bluetooth LE stack asks the application for read and write operations on attribute values. It is then up to the application to decide if it is necessary to store values and how to store and retrieve them. For instance, a characteristic linked to real-time data from a sensor may not need a static buffer to store an attribute value, since data are read on-demand from the sensor. Similarly, a control-point attribute, which controls an external component (e.g. an LED) may not need a static buffer for the attribute value.

While the Bluetooth LE GATT stack v2.x allocates attributes in RAM memory, Bluetooth LE stack v3.x ATT/GATT component avoids, as much as possible, such memory allocation, providing a profile registration mechanism based on C-language structures that can be stored in Flash memory if needed. These structures are also designed to reduce memory allocation and to be cheaper than the v2.x. The components needed to be stored in volatile memory are collected in a structure that sits in 8 bytes of RAM.

The Bluetooth LE stack v3.x ATT/GATT component also supports the Bluetooth robust caching feature. This is handled internally in the stack and it does not need support from the application.

3.2.2 GATT server

The GATT server is in charge of storing the attribute, which compose a profile on a GATT database.

The GATT profile is designed to be used by an application or another profile and defines how to use the contained attributes to obtain some information.

3.2.2.1 Service

A service is a collection of data and associated behaviors to accomplish a function or feature. A service definition may contain included services and characteristics.

There are two types of services:

- **Primary service** is a service that exposes the primary usable functionality of this device
- **Secondary service** is a service that is only intended to be included from a primary service or another secondary service

A service is defined using the following `ble_gatt_srv_def_t C` language structure:
typedef struct ble_gatt_srv_def_s {
    ble_uuid_t uuid;
    uint8_t type;
    uint16_t group_size;
    struct {
        uint8_t incl_srv_count;
        struct ble_gatt_srv_def_s **included_srv_pp;
    } included_srv;
    struct {
        uint8_t chr_count;
        ble_gatt_chr_def_t *chrs_p;
    } chrs;
} ble_gatt_srv_def_t

This structure represents a service with its properties:

- **uuid**: is the 16-bit Bluetooth UUID or 128-bit UUID for the service, known as service UUID
- **type**: indicates if the service is primary (BLE_GATT_SRV_PRIMARY_SRV_TYPE, with value 0x01) or secondary (BLE_GATT_SRV_SECONDARY_SRV_TYPE, with value 0x02)
- **group_size**: optional field, indicates how many attributes can be added to the service group. If it is set to 0 then no size is defined, and no limit is used on the number of attributes that is added to the service. An equal number of handles are reserved, so that new attributes can be added to the service at a later time
- **included_srv**: optional field, is the list of included services
- **chrs**: optional field, is the list of contained characteristics. If one or more characteristics are present then these are registered, with their descriptors if any, at service registration

For instance, consider a GAP profile composed of a primary service with the 16-bit UUID equal to 0x1800, the C structure that defines it is

```c
static ble_gatt_srv_def_t gap_srvc = {
    .type = BLE_GATT_SRV_PRIMARY_SRV_TYPE,
    .uuid = BLE_UUID_INIT_16(0x1800),
    .chrs = {
        .chrs_p = gap_chrs,
        .chr_count = 2U,
    },
};
```

Note: Static variables and global variables (and their fields in case of structures) are initialized to 0 if not explicitly initialized.

To register a service in the GATT DB the **aci_gatt_srv_add_service** function is used:

```c
aci_gatt_srv_add_service(&gap_srvc);
```

while retrieving the assigned attribute handle the **aci_gatt_srv_get_service_handle** function is used:

```c
uint16_t gap_h = aci_gatt_srv_get_service_handle(&gap_srvc);
```

The registered services can be also removed at run time if needed using the **aci_gatt_srv_rm_service** function:

```c
aci_gatt_srv_rm_service(gap_h);
```

Note: The memory used for a service definition structure is kept valid for all the time such service is active.

3.2.2.2 The characteristic

A characteristic is used to expose a device value: for instance, to expose the temperature value. A characteristic is defined using the following **ble_gatt_chr_def_t** C language structure:

```c
typedef struct ble_gatt_chr_def_s {
    uint8_t properties;
    uint8_t min_key_size;
    uint8_t permissions;
    ble_uuid_t uuid;
    struct {
        uint8_t descr_count;
        ble_gatt_descr_def_t *descrs_p;
    } descrs;
    ble_gatt_val_buffer_def_t *val_buffer_p;
} ble_gatt_chr_def_t;
```
This structure represents a characteristic with its properties:

- **properties**: is a bit field and determines how the characteristic value can be used or how the characteristic descriptor can be accessed.
- **min_key_size**: indicates the minimum encryption key size requested to access the characteristic value. This parameter is only used if encryption is requested for this attribute (see `permission` field) and in this case its value must be greater than or equal to 7 and less than or equal to 16.
- **permission**: this is a bit field and indicates how the characteristic can be accessed:
  - BLE_GATT_SRV_PERM_NONE (0x00): indicates no permissions are required to access characteristic value
  - BLE_GATT_SRV_PERM_AUTHEN_READ (0x01): indicates that the reading of the characteristic value requires an authenticated pairing (i.e. with MITM protection enabled)
  - BLE_GATT_SRV_PERM_AUTHOR_READ (0x02): indicates that the application must authorize access to the device services on the link before the reading of the characteristic value can be granted
  - BLE_GATT_SRV_PERM_ENCRY_READ (0x04): indicates that the reading of the characteristic value requires an encrypted link. The minimum encryption key size to access this characteristic value must be specified through the `min_key_size` field
  - BLE_GATT_SRV_PERM_AUTHEN_WRITE (0x08): indicates that the writing of the characteristic value requires an authenticated pairing (i.e. with MITM protection enabled)
  - BLE_GATT_SRV_PERM_AUTHOR_WRITE (0x10): indicates that the application must authorize access to the device services on the link before the writing of the characteristic value can be granted
  - BLE_GATT_SRV_PERM_ENCRY_WRITE (0x20): indicates that writing of the characteristic value requires an encrypted link. The minimum encryption key size to access this characteristic value must be specified through the `min_key_size` field
- **uuid**: this field is a 16-bit Bluetooth UUID or 128-bit UUID that describes the type of characteristic value
- **descrs**: this optional field is the list of characteristic descriptors. If one or more descriptors are present, then these are registered at the time of characteristic registration
- **val_buffer_p**: this is an optional field that if set, points to the allocated buffer storing the characteristic value. If it is not set (e.g. set to NULL) then an event is emitted by the stack to request a read or a write operation on the characteristic value (see `aci_gatt_srv_read_event()` and `aci_gatt_srv_write_event()`).

For instance, consider the device name characteristic of the GAP profile, that has read/write access, no particular security permission and a 16-bit UUID equal to 0x2A00 (min_key_size is not set since encryption is not required):

```c
static ble_gatt_chr_def_t gap_device_name_chr = {
    .properties = BLE_GATT_SRV_CHAR_PROP_READ,
    .permissions = BLE_GATT_SRV_PERM_NONE,
    .uuid = BLE_UUID_INIT_16(0x2A00),
    .val_buffer_p=(ble_gatt_val_buffer_def_t *)&gap_device_name_val_buff,
};
```

To register the defined characteristic the `aci_gatt_srv_add_char` function can be used:

`aci_gatt_srv_add_char (&gap_device_name_chr, gap_p);`

As an alternative, the characteristic can be added to the service at the same time that the service is registered. In this case, the list of characteristics is passed to the stack through the `chrs` field of the `ble_gatt_srv_def_t` structure.

As services and characteristics, there is a function to retrieve the assigned attribute handle (`aci_gatt_srv_get_char_decl_handle`) and to remove a registered characteristic at run time (`aci_gatt_srv_rm_char`).

---

**Note:**

The memory used for a characteristic definition structure is kept valid for all the time such characteristic is active.

### 3.2.2.3 Descriptor

Characteristic descriptors are used to contain the related information about the characteristic value. A standard set of characteristic descriptors are defined to be used by application. The application can also define additional characteristic descriptors as profile specifics.

A characteristic descriptor is defined using the following `ble_gatt_descr_def_t` C language structure:
typedef struct ble_gatt_descr_def_s {
    uint8_t properties;
    uint8_t min_key_size;
    uint8_t permissions;
    ble_uuid_t uuid;
    ble_gatt_val_buffer_def_t *val_buffer_p;
} ble_gatt_descr_def_t;

This structure represents a characteristic descriptor with its properties:

- **properties**: is a bit field and determines how the characteristic descriptor can be accessed. The BLE_GATT_SRV_DESCR_PROP_READ (0x01) bit enables the descriptor reading, while the BLE_GATT_SRV_DESCR_PROP_WRITE (0x02) bit enables the descriptor writing.

- **min_key_size**: indicates the minimum encryption key size requested to access the characteristics descriptor. This parameter is used only if encryption is requested for this attribute (see `permissions` field) and in this case its value must be greater than or equal to 7 and less than or equal to 16.

- **Permission**: this is a bit field and indicates how the characteristic descriptor can be accessed:
  - BLE_GATT_SRV_PERM_NONE, (0x00) indicates no permissions are set to access characteristic descriptor.
  - BLE_GATT_SRV_PERM_AUTHEN_READ, (0x01) indicates that the reading of characteristic descriptor requires prior pairing with authentication (MITM) on.
  - BLE_GATT_SRV_PERM_AUTHOR_READ, (0x02) indicates that the link is authorized before reading the characteristic descriptor.
  - BLE_GATT_SRV_PERM_ENCRY_READ, (0x04) indicates that reading of characteristic descriptor requires prior pairing with encryption.
  - BLE_GATT_SRV_PERM_AUTHEN_WRITE, (0x08) indicates that writing about characteristic descriptor requires prior pairing with authentication (MITM) on.
  - BLE_GATT_SRV_PERM_AUTHOR_WRITE, (0x10) indicates that the link is authorized before writing the characteristic descriptor.
  - BLE_GATT_SRV_PERM_ENCRY_WRITE, (0x20) indicates that writing of characteristic descriptor requires prior pairing with encryption.

- **uuid**: this field is a 16-bit Bluetooth UUID or 128-bit UUID that describes the type of characteristic descriptor.

- **val_buffer_p**: this is an optional field that, if set, points to the allocated buffer storing characteristic descriptor value. If it is not set (e.g. set to NULL) then an event is emitted by the stack to request a read or a write operation on characteristic descriptor value (see `aci_gatt_srv_read_event()` and `aci_gatt_srv_write_event()`).

For instance, to define a descriptor with a read access permission, a 16-bit UUID value set to 0xAABB and no security permissions, the following structure is used:

```c
static ble_gatt_descr_def_t chr descr = {
    .uuid = BLE_UUID_INIT_16(0xAABB),
    .properties = BLE_GATT_SRV_DESCR_PROP_READ,
    .permissions = BLE_GATT_SRV_PERM_NONE,
};
```

To register the defined descriptor in the DB the `aci_gatt_srv_add_char_desc` function is used:

```c
aci_gatt_srv_add_char_desc(&chr descr, chr_handle);
```

where `chr_handle` is the attribute handle of the characteristic that contains this descriptor. Besides, for descriptors there is the function to retrieve its attribute handle (`aci_gatt_srv_get_descriptor_handle`) and to remove the descriptor (`aci_gatt_srv_rm_char_desc`) itself.

The descriptor can also be added together with the related characteristic by specifying the `descrs` field of `ble_gatt_chr_def_t`.

Since the client characteristic configuration descriptor (CCCD) is quite common to be present in a profile, then some helper macros are provided to define it:

- **BLE_GATT_SRV_CCCD_DECLARE** declares CCCD value buffer and descriptor fields. Commonly used when a characteristic has just the CCCD as unique descriptor.
BLE_GATT_SRV_CCCD_DEF_STR_FIELDS: fills the descriptor structure fields for a CCCD. It can be used when a characteristic has more than the CCC descriptor to fill the fields of a descriptor definition array.

Note: The memory used for a characteristic descriptor definition structure is valid for all the time such descriptor is active.

3.2.2.4 Value buffer

The value buffer is a structure that holds the characteristic value and characteristic descriptor values. Such structure stores the buffer information and is kept valid for all the life of the containing structure. The value buffer structure is defined by ble_gatt_val_buffer_def_t type:

```c
typedef struct ble_gatt_val_buffer_def_s {
    uint8_t op_flags;
    uint16_t val_len;
    uint16_t buffer_len;
    uint8_t *buffer_p;
} ble_gatt_val_buffer_def_t;
```

This structure has the following field:

- **op_flags**: this is a bit field that enables a specific behavior when the value is written
  - BLE_GATT_SRV_OP_MODIFIED_EVT_ENABLE_FLAG (0x01): enables the generation of aci_gatt_attribute_modified_event event when the value is changed by the client
  - BLE_GATT_SRV_OP_VALUE_VAR_LENGTH_FLAG (0x02): indicates that the value is a variable length
- **val_len**: stores the value length. Ignored if BLE_GATT_SRV_OP_VALUE_VAR_LENGTH_FLAG bit is not set in the op_flags field
- **buffer_len**: the length of the buffer pointed by buffer_p pointer. For a fixed length characteristic, this is the length of the characteristic value
- **buffer_p**: the pointer to value buffer.

For example, to define a variable length value buffer, with a maximum size of 10 bytes, the following code is used:

```c
uint8_t buffer[10];
ble_gatt_val_buffer_def_t val_buffer = {
    .op_flags = BLE_GATT_SRV_OP_VALUE_VAR_LENGTH_FLAG,
    .buffer_len = 10,
    .buffer_p = buffer,
};
```

If the application needs to fill the value buffer with a 2-byte value (e.g. 0x0101), it can directly address its value through the buffer and set the actual length:

```c
memset(val_buffer.buffer_p, 0x01, 2);
val_buffer.val_len = 2;
```

The stack is not aware of such value update until a remote device sends a read request to retrieve its value.

If the characteristic has a fixed length, ble_gatt_val_buffer_def_t structure can be defined as a constant:

```c
const ble_gatt_val_buffer_def_t val_buffer = {
    .buffer_len = 2,
    .buffer_p = buffer,
};
```

Moreover, if the value cannot be changed (i.e. read only access), then the buffer pointed by buffer_p can be also declared as a constant.

```c
const uint8_t buffer[2] = {0x01, 0x01};
```

If the value is dynamically computed (e.g. temperature) then the value buffer is not needed: if val_buffer_p field of characteristic or descriptor C structure is not set (i.e. set to NULL) then some events are generated to access such value:

- aci_gatt_srv_read_event is generated when a remote device needs to read a characteristic value or descriptor
• `aci_gatt_srv_write_event`, is generated when a remote device needs to write a characteristic value or descriptor.

**Note:** The memory used for value buffer definition is valid for all the time such buffer is active.

### 3.2.2.5 GATT server database APIs

The following paragraph contains the list of functions that are available to manipulate the GATT server database.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aci_gatt_srv_add_service</code></td>
<td>This function adds the provided service to the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_rm_service</code></td>
<td>This function removes the provided service from the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_get_service_handle</code></td>
<td>This function retrieves the attribute handle assigned to the service registered using the provided definition structure</td>
</tr>
<tr>
<td><code>aci_gatt_srv_add_include_service</code></td>
<td>This function adds the provided include service</td>
</tr>
<tr>
<td><code>aci_gatt_srv_rm_include_service</code></td>
<td>This function removes the provided include service from the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_get_include_service_handle</code></td>
<td>This function retrieves the attribute handle assigned to the include service</td>
</tr>
<tr>
<td><code>aci_gatt_srv_add_char</code></td>
<td>This function adds the provided characteristic to the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_rm_char</code></td>
<td>This function removes the provided characteristic from the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_get_char_decl_handle</code></td>
<td>This function retrieves the attribute handle assigned to the characteristic registered using the provided definition structure</td>
</tr>
<tr>
<td><code>aci_gatt_srv_add_char_desc</code></td>
<td>This function adds the provided descriptor to the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_rm_char_desc</code></td>
<td>This function removes the provided descriptor from the database</td>
</tr>
<tr>
<td><code>aci_gatt_srv_get_descriptor_handle</code></td>
<td>This function retrieves the attribute handle assigned to the characteristic descriptor registered using the provided definition structure</td>
</tr>
</tbody>
</table>

### 3.2.2.6 Examples

The following examples are intended to explain how to define a profile.

**GATT profile**

This example illustrates how to implement and register the GATT profile.

**Note:** The complete GATT service is already implemented in `gatt_profile.c`. A more simple implementation is described here.

This profile is composed of a primary service with a 16-bit UUID set to 0x1801 and the service changed characteristic, with the indication property bit set. To support indications the characteristic has the client characteristic configuration descriptor. To declare this descriptor the BLE_GATT_SRV_CCCD_DECLARE macro can be used. This macro has the following parameter:

```
BLE_GATT_SRV_CCCD_DECLARE(NAME, NUM_CONN, PERM, OP_FLAGS)
```

Where:

- **NAME** is the name assigned to identify this CCCD
- **NUM_CONN** is the number of supported concurrent connections for the targeted application
- **PERM**, is the bit field of descriptor permission
- **OP_FLAGS** is the bit field of descriptor value buffer.

Then, for instance, the declaration can be the following:
BLE_GATT_SRV_CCCD_DECLARE(gatt_chr_srv_changed,
   GATT_SRV_MAX_CONN,
   BLE_GATT_SRV_PERM_NONE,
   BLE_GATT_SRV_OP_MODIFIED_EVT_ENABLE_FLAG);

Declare now the characteristic value buffer, since it has to be provided as val_buffer_p of ble_gatt_chr_def_t structure:

```c
uint8_t srv_chgd_buff[4];
const ble_gatt_val_buffer_def_t srv_chgd_val_buff = {
   .buffer_len = 4U,
   .buffer_p = gatt_chr_srv_changed_buff,
};
```

Once the descriptor and the characteristic value buffer are declared then the service changed characteristic can be declared:

```c
static ble_gatt_chr_def_t gatt_chr = {
   .properties = BLE_GATT_SRV_CHAR_PROP_INDICATE,
   .permissions = BLE_GATT_SRV_PERM_NONE,
   .uuid = BLE_UUID_INIT_16(0x2A05),
   .val_buffer_p = &srv_chgd_val_buff,
   .descrs = {
      .descrs_p = &BLE_GATT_SRV_CCCD_DEF_NAME(gatt_chr_srv_changed),
      .descr_count = 1U,
   },
};
```

As stated, this characteristic has the indication bit set in the properties bit field, no security permissions, the UUID set to 0x2A05 and one descriptor.

Now the GATT service can be declared:

```c
static ble_gatt_srv_def_t gatt_srvc = {
   .type = BLE_GATT_SRV_PRIMARY_SRV_TYPE,
   .uuid = BLE_UUID_INIT_16(0x1801),
   .chrs = {
      .chrs_p = &gatt_chr,
      .chr_count = 1,
   },
};
```

To register the whole profile, only the service is registered: all the included characteristics and its descriptors are automatically registered. Use the aci_gatt_srv_add_service to register the service:

```c
aci_gatt_srv_add_service(&gatt_srvc);
```

**Glucose**

Consider the following database:

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>Attribute type</th>
<th>UUID</th>
<th>Properties</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>Primary service</td>
<td>0x1808</td>
<td></td>
<td>Glucose service</td>
</tr>
<tr>
<td></td>
<td>Included service</td>
<td></td>
<td></td>
<td>Included battery service</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>0x2A18</td>
<td>Read, indicate, extended properties</td>
<td>Glucose measurement characteristic</td>
</tr>
<tr>
<td></td>
<td>Descriptor</td>
<td></td>
<td></td>
<td>CCCD</td>
</tr>
<tr>
<td></td>
<td>Descriptor</td>
<td></td>
<td></td>
<td>Extended properties descriptor</td>
</tr>
<tr>
<td>0x1000</td>
<td>Secondary service</td>
<td>0x180F</td>
<td></td>
<td>Battery service</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>0x2A19</td>
<td>Read</td>
<td>Battery level characteristic</td>
</tr>
</tbody>
</table>
Start defining the battery service with its characteristic:

```c
uint8_t battery_level_value;
const ble_gatt_val_buffer_def_t battery_level_val_buff = {
    .buffer_len = 1,
    .buffer_p = &battery_level_value,
};
ble_gatt_chr_def_t batt_level_chr = {
    .properties = BLE_GATT_SRV_CHAR_PROP_READ,
    .permissions = BLE_GATT_SRV_PERM_NONE,
    .uuid = BLE_UUID_INIT_16(0x2A19),
    .val_buffer_p = &battery_level_val_buff,
};
ble_gatt_srv_def_t battery_level_srvc = {
    .type = BLE_GATT_SRV_SECONDARY_SRV_TYPE,
    .uuid = BLE_UUID_INIT_16(0x180F),
    .chrs = {
        .chrs_p = &batt_level_chr,
        .chr_count = 1,
    },
};
```

Define the glucose profile:

```c
uint8_t ext_prop_descr_buff[2];
ble_gatt_val_buffer_def_t ext_prop_descr_val_buff = {
    .buffer_len = 2,
    .buffer_p = ext_prop_descr_buff,
};
BLE_GATT_SRV_CCCD_BUFFER_DECLARE(glucose_mes, MAX_CONN, 0);
ble_gatt_descr_def_t glucose_chr_descrs[] = {
    BLE_GATT_SRV_CCCD_DEF_STR_FIELDS(glucose_mes, MAX_CONN,
        BLE_GATT_SRV_CCCD_PERM_DEFAULT),
},
};
ble_gatt_chr_def_t glucose_mes_chr = {
    .properties = BLE_GATT_SRV_CHAR_PROP_READ | BLE_GATT_SRV_CHAR_PROP_INDICATE |
        BLE_GATT_SRV_CHAR_PROP_EXTENDED_PROP,
    .permissions = BLE_GATT_SRV_PERM_NONE,
    .uuid = BLE_UUID_INIT_16(0x2A18),
    .descrs = {
        .descrs_p = &ble_gatt_srv_cccd_def_name(glucose_chr_descrs),
        .descr_count = 2,
    },
};
static ble_gatt_srv_def_t *incl_srv_pa[1] = {
    &battery_level_srvc,
};
ble_gatt_srv_def_t glucose_srvc = {
    .type = BLE_GATT_SRV_PRIMARY_SRV_TYPE,
    .uuid = BLE_UUID_INIT_16(0x1808),
    .group_size = 0x1000,
};
```

As shown the `glucose_srvc` has set the `group_size` field: this is there to allow the battery service to be registered at 0x1000 attribute handles. This service does not include any characteristic or included services. This is a choice, for this example, to show how to register such elements one by one.

First, register the glucose service:

```c
aci_gatt_srv_add_service(&glucose_srvc);
```

This is a primary service with a 16-bit UUID set to 0x1808.

Register battery service and its characteristic:

```c
aci_gatt_srv_add_service(&battery_level_srvc);
```

This is a secondary service with a 16-bit UUID set to 0x180F that includes a characteristic. Both service and characteristic are registered.
Now it is time to include the battery service in the glucose service and register its characteristic and descriptors. For this purpose the assigned attribute handle is needed to get for the glucose and battery services:

```c
uint16_t glucose_srvc_handle = aci_gatt_srv_get_service_handle(&glucose_srvc);
uint16_t battery_srvc_handle = aci_gatt_srv_get_service_handle(&battery_level_srvc);
```

Include battery into glucose service:

```c
aci_gatt_srv_include_service(glucose_srvc_handle, battery_srvc_handle);
```

**Note:** Included services are added to service before registering any characteristic to the same service. This is needed because the include service attribute is placed after the service declaration attribute and before any characteristic declaration attributes.

To register the glucose measurement characteristic and its descriptors, use the `aci_gatt_srv_add_char()` function as shown below:

```c
aci_gatt_srv_add_char(&glucose_mes_chr, glucose_srvc_handle);
```

The glucose measurement characteristic and the included descriptors are added.

### 3.2.2.7 Server initiated procedures

The so-called server initiated procedures are the ones used to send data to a remote client that is subscribed to receive it. There are two kinds of server initiated procedures.

- **Notification:** this procedure is used when the server is configured to notify a characteristic value to a client without expecting any acknowledgment that the notification was successfully received
- **Indication:** this procedure is used when the server is configured to indicate a characteristic value to a client and to expect an acknowledgment that the indication has been successfully received

For both procedures, an API is provided: the `aci_gatt_srv_notify()`.

#### Table 40. aci_gatt_srv_notify parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16_t</td>
<td>Connection_Handle</td>
<td>The connection handle for which the notification is requested</td>
</tr>
<tr>
<td>uint16_t</td>
<td>Attr_Handle</td>
<td>The attribute handle to notify</td>
</tr>
</tbody>
</table>
| uint8_t | Flags | Notification flags:  
  0x00: sends a notification  
  0x01: sends a flushable notification  
  0x02: sends an indication |
| uint16_t | Val_Length | The length of provided value buffer |
| uint8_t * | Val_p | The pointer to the buffer containing the value to notify |

The `Flags` parameter indicates the kind of message that is sent. For instance, to notify the value of the attribute with handle 0x13 to a client, with a connection handle of 0x0801 then the following code is used.

```c
uint8_t value = 1;
ret = aci_gatt_srv_notify(0x0801, 0x13, 0, 1, &value);
```

If the client has set the notification bit into the CCCD of the characteristic then the notification is sent, otherwise a BLE_STATUS_NOT_ALLOWED error is returned.
3.2.2.8 Attribute value read and write

As described in the previous section, the stack can access the characteristic values and descriptors through their value buffers. If such buffer is not set in the definition structure (val_buffer_p = NULL), then the stack generates an event to ask the application to operate on the related value buffer. There are two events: aci_gatt_srv_read_event() and aci_gatt_srv_write_event(). For queued writes, the stack does not have a queue to store them: if the application wants to support the queued write, it must implement the queue to store each prepare write. For this reason, the stack generates the aci_att_srv_prepare_write_req_event() event for each received prepare write request so that the application can store them. The aci_att_srv_exec_write_req_event() event is generated when an execute write request is received.

aci_gatt_srv_read_event()

This event is generated when the server receives a read operation of an attribute value and the stack does not have direct access to such value. This event must be followed by aci_gatt_srv_resp(), passing the value of the attribute.

<table>
<thead>
<tr>
<th>Table 41. aci_gatt_srv_read_event parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
</tbody>
</table>

aci_gatt_srv_write_event()

This event is generated when the server receives a write operation of an attribute value and it does not have access to the attribute value buffer. This event must be followed by aci_gatt_srv_resp() if Resp_Needed is 1.

<table>
<thead>
<tr>
<th>Table 42. aci_gatt_srv_write_event parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint8_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint8_t*</td>
</tr>
</tbody>
</table>

aci_att_srv_prepare_write_req_event()

This event is generated when the server receives a prepare write request. It carries the received data stored at application level. This event must be followed by aci_gatt_srv_resp(), passing back to the stack the value, which is written by the application.

<table>
<thead>
<tr>
<th>Table 43. aci_att_srv_prepare_write_req_event parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint16_t</td>
</tr>
<tr>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**aci_att_srv_exec_write_req_event()**

This event is generated when the server receives an execute write request. Application must handle it to write or flush all stored prepare write requests, depending on the Flags value. This event must be followed by `aci_gatt_srv_resp()`.

**Table 44. aci_att_srv_exec_write_req_event parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16_t</td>
<td>Connection_Handle</td>
<td>Connection handle identifies the connection</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Flags</td>
<td>• 0x00 – Cancel all prepared writes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0x01 – Immediately write all pending prepared values</td>
</tr>
</tbody>
</table>

**aci_gatt_srv_resp()**

When the previous events are generated by the stack, the application must decide to allow (and execute) or deny the requested operation. To inform the stack about the choice made by the application and pass the requested data (in case of a read request or a prepare write request), a function is provided: `aci_gatt_srv_resp()`. This function is used to close a read or write transaction started by a remote client.

*Note:* This function is executed within 30 seconds from the reception of event otherwise a GATT timeout occurs.

**Table 45. aci_gatt_srv_resp parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16_t</td>
<td>Connection_Handle</td>
<td>Connection handle that identifies the connection</td>
</tr>
<tr>
<td>uint16_t</td>
<td>Attribute_Handle</td>
<td>Attribute handle for which the response command is issued</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Error_Code</td>
<td>The reason why the request has generated an error response (use one of the ATT error codes, see [1], Vol. 3, Part F, Table 3.4)</td>
</tr>
<tr>
<td>uint16_t</td>
<td>Val_Length</td>
<td>Length of the value field</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Val</td>
<td>The response data in the following cases:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• read request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• prepare write request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For other requests, this parameter can be NULL</td>
</tr>
</tbody>
</table>

*Note:* Data pointed by Val is no more needed on function return and can be released.

**aci_gatt_srv_read_event() example**

The following code shows an example of how to implement the `aci_gatt_srv_read_event()` handler.
void aci_gatt_srv_read_event(uint16_t Connection_Handle, 
                         uint16_t Attribute_Handle, 
                         uint16_t Data_Offset) 
{ 
    uint16_t val_len;
    uint8_t attr_error_code;
    uint8_t val_buff[4];
    attr_error_code = BLE_ATT_ERR_NONE;
    if (Attribute_Handle == 0x16) 
    { 
        /** Attribute is mapped on a GPIO value */
        val_len = 1;
        gpio_get(GPIO_01, val_buffer[0]);
    } 
    else if (Attribute_Handle == 0x19) 
    { 
        /** Fill buffer with some custom data */
        val_len = 4;
        memset(val_buffer, 2, 4);
    } 
    else 
    { 
        val_len = 0;
        attr_error_code = BLE_ATT_ERR_UNLIKELY;
    }
    aci_gatt_srv_resp(Connection_Handle, Attribute_Handle, 
                       attr_error_code, val_len, 
                       val_buff);
} 

The code manages the attribute handles 0x16 and 0x19. The handle 0x16 is mapped on a GPIO value while handle 0x19 returns 4 bytes. The aci_gatt_srv_resp() generates the read response with the given data if Error_Code is set to 0, otherwise it sends an error response.

aci_gatt_srv_write_event() example

The following example shows how to manage the aci_gatt_srv_write_event() event. In this example, writing the attribute handle 0x16 changes a GPIO state.
void aci_gatt_srv_write_event(uint16_t Connection_Handle, uint8_t Resp_Needed, uint16_t Attribute_Handle, uint16_t Data_Length, uint8_t Data[]) {
    uint16_t buffer_len;
    uint8_t *buffer, att_err;
    buffer_len = 0;
    buffer = NULL;
    if (Data_Length != 1) {
        att_err = BLE_ATT_ERR_INVAL_ATTR_VALUE_LEN;
    } else if (Attribute_Handle != 0x16) {
        att_err = BLE_ATT_ERR_UNLIKELY;
    } else if ((Data[0] != 0) && (Data[0] != 1)) {
        att_err = BLE_ATT_ERR_VALUE_NOT_ALLOWED;
    } else {
        /** Set GPIO value */
        att_err = BLE_ATT_ERR_NONE;
        gpio_set(GPIO_01, Data[0]);
        if (Resp_Needed == 1U) {
            aci_gatt_srv_resp(Connection_Handle, Attribute_Handle, att_err, buffer_len, buffer);
        }
    }
}

aci_att_srv_prepare_write_req_event() example
The following example shows how to implement the aci_att_srv_prepare_write_req_event() handler. This function uses the ATT_pwrq module to store the received prepare write but any kind of queue that can handle such use case can be used.

void aci_att_srv_prepare_write_req_event(uint16_t Connection_Handle, uint16_t Attribute_Handle, uint16_t Data_Offset, uint16_t Data_Length, uint8_t Data[]) {
    uint8_t att_err;
    tBleStatus ret;
    ret = ATT_pwrq_push(Connection_Handle, Attribute_Handle, Data_Offset, Data_Length, Data);
    if (ret != BLE_STATUS_SUCCESS) {
        att_err = BLE_ATT_ERR_PREP_QUEUE_FULL;
    } else {
        att_err = BLE_ATT_ERR_NONE;
        /* Send response */
        aci_gatt_srv_resp(Connection_Handle, Attribute_Handle, att_err, Data_Length, Data);
    }
}

Note: The aci_gatt_srv_resp() function is called passing the received data. This data is needed to be redirected back to the stack to generate the prepare write response packet. It contains such data and acknowledges the client about the correctness of the received data.

aci_gatt_srv_exec_write_resp() example
The following code shows an example of how to implement an aci_att_srv_exec_write_req_event() handler. All queued prepare write requests are written by write_queued_data() function, as shown below.
The `aci_gatt_srv_resp()` function generates an exec write response or an error based on `att_error` value.

```c
void aci_gatt_srv_exec_write_resp(uint16_t Conn_Handle, uint8_t Exec)
{
    uint8_t att_error;
    att_error = BLE_ATT_ERR_NONE;
    if (Exec == 1U)
    {
        att_error = write_queued_data(Conn_Handle);
    }
    ATT_pwrq_flush(Conn_Handle);
    aci_gatt_srv_resp(Conn_Handle, 0U, att_error, 0U, NULL);
}
```

### 3.2.2.9 GAP and GATT profile components

The ATT/GATT component of the BlueNRG stack does not automatically allocate the GAP and GATT services but it delegates the implementation of these services to the application: this allows the application to customize such profiles based on its needs. Two components are provided as reference to register such profiles: they can be found in `gatt_profile.c` and `gap_profile.c`. These components implement the initialization functions called by the stack to register GATT and GAP profiles: `Gatt_profile_init()` and `Gap_profile_init()`.

`gatt_profile.c` provides a default GATT profile, but it can be customized: for instance if the client supported feature and database hash characteristics are not registered in the GATT service then the robust caching feature is automatically disabled since these characteristics are mandatory for such feature. If the service changed characteristic is removed, the database is intended to be static and then no service change indication can be sent.

*Note:* The database hash characteristics are handled in a special way: no value buffer is allocated by the application, since the hash is generated and allocated internally by stack. `gap_profile.c` implements the default GAP profile with its characteristics. These components provide also some commodity functions to set up characteristic values like the device name.

### 3.2.2.10 ATT_PWRQ component

The scope of this component is to provide the mechanism to store in a FIFO the received prepare write requests. This is an optional component provided in `att_pwrq.c` as a reference.

**ATT_pwrq_init()**

This function initializes the ATT_PWRQ component.

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint16_t</code></td>
<td>queue_length</td>
<td>Queue buffer size</td>
</tr>
<tr>
<td><code>uint8_t*</code></td>
<td>queue_buffer_p</td>
<td>Pointer to the buffer used to store the FIFO</td>
</tr>
</tbody>
</table>

**ATT_pwrq_flush()**

This function removes all the queued writes related to a connection handle.

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint16_t</code></td>
<td>conn_handle</td>
<td>Queue buffer size</td>
</tr>
</tbody>
</table>

**ATT_pwrq_read()**

Read the FIFO elements. Such elements are filtered per connection handle and indexed by a parameter.
3.2.3 SoC vs. network coprocessor

While in the application processor mode, the application resides on the BlueNRG-LP memory; in the network coprocessor mode the application runs outside. The application layer inside the Bluetooth LE device exports Bluetooth LE stack functionality through a serial interface. For this scope the network coprocessor needs to allocate buffers that can hold the database definition structure and values.

3.2.3.1 DTM

An adaptation layer inside the device is needed to use the device as a network coprocessor. The DTM (direct test mode) example application is a way to use the device in a network processor mode. DTM application exposes ACI (application-controller interface) commands and events, so that an application on an external device can use the Bluetooth LE stack through a serial interface.

The interface to the GATT layer exposed by the DTM application is different from the native GATT API. The aci_gatt_nwk.c file implements the adaptation layer between the network co-processor API and the native API. This module defines some buffers used to allocate the memory space needed to define services, characteristics and descriptors. It also allocates the memory needed to store the attribute values that reside in the DTM memory space. It uses a dynamic memory allocator to allocate the requested memory. The allocated structures are inserted in a linked list to search the associated value buffer when a read/write operation is requested by a client.
Some client procedures (e.g. write and write long characteristic procedures) need to temporarily store data in a buffer. This component also allocates the buffer needed by those procedures. These buffers are kept allocated until the procedure completes.

### 3.2.4 GATT client

A device, acting as a client, initiates commands and requests towards the server and can receive responses, indications and notifications sent by the server. The following actions are covered by this role:

- Exchange configuration
- Discover services and characteristics on a server
- Read an attribute value
- Write an attribute value
- Receive notifications and indications by a server
- Send a confirmation of a received indication.

GATT uses the attribute protocol (ATT) to transport data to the form of commands, requests, indications, notifications and confirmations between client and server. Some GATT client procedures generate only one ATT request and wait for the response from the server. The procedure is terminated after the response is received (or a timeout occurs). Other procedures are composed of more than one exchange of request-response ATT packets. The end of a procedure is indicated by the reception of `aci_gatt_proc_complete_event` event. Once a procedure is on-going, no other procedures can be started with the same server.

In the following section, the list of available functions for a GATT client is shown.

<table>
<thead>
<tr>
<th>Table 51. GATT client APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API</strong></td>
</tr>
<tr>
<td><code>aci_gatt_clt_exchange_config</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_disc_all_primary_services</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_disc_primary_service_by_uuid</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_find_included_services</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_disc_all_char_of_service</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_disc_char_by_uuid</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_disc_all_char_desc</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_read</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_read_long</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_read_multiple_char_value</code></td>
</tr>
<tr>
<td>API</td>
</tr>
<tr>
<td>------------------------------------------</td>
</tr>
<tr>
<td><code>aci_gatt_clt_read_using_char_uuid</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_write_without_resp</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_signed_write_without_resp</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_write</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_write_long</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_write_char_reliable</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_notification_event</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_indication_event</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_confirm_indication</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_prepare_write_req</code></td>
</tr>
<tr>
<td><code>aci_gatt_clt_execute_write_req</code></td>
</tr>
</tbody>
</table>

### 3.2.5 Services and characteristic configuration

In order to add a service and its related characteristics, a user application has to define the specific profile to be addressed:

1. Standard profile defined by the Bluetooth SIG organization. The user must follow the profile specification and services, characteristic specification documents in order to implement them by using the related defined Profile, Services and Characteristics 16-bit UUID (refer to Bluetooth SIG web page: [www.bluetooth.org/en-%20us/specification/adopted-specifications](http://www.bluetooth.org/en-%20us/specification/adopted-specifications)).
2. Proprietary, non-standard profile. The user must define their own services and characteristics. In this case, 128-bit UIDS are required and must be generated by profile implementers (refer to UUID generator web page: www.famkruithof.net/uuid/uuidgen).

The following pseudocode describes how to define a service with two characteristics, TX (notification property) and RX (write without response property) with the following UUIDs (128 bits):

**Service UUID:** D973F2E0-B19E-11E2-9E96-0800200C9A66

**TX_Char UUID:** D973F2E1-B19E-11E2-9E96-0800200C9A66

**RX_Char UUID:** D973F2E2-B19E-11E2-9E96-0800200C9A66

```c
/* Service and Characteristic UUIDs */
#define SRVC_UUID
0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe0,0xf2,0x73,0xd9
#define TX_CHR_UUID
0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe0,0xf2,0x73,0xd9
#define RX_CHR_UUID
0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe0,0xf2,0x73,0xd9

/* Define RX_BUFFER_SIZE (20) */

/* Define the client configuration characteristic descriptor */
BLE_GATT_SRV_CCCD_DECLARE(tx, NUM_LINKS, BLE_GATT_SRV_CCCD_PERM_DEFAULT,
BLE_GATT_SRV_OP_MODIFIED_EVT_ENABLE_FLAG);

/* TX (notification), RX (write without response) characteristics definition */
static const ble_gatt_chr_def_t user_chars[] = {
    /* TX characteristic with CCCD */
    .properties = BLE_GATT_SRV_CHAR_PROP_NOTIFY,
    .permissions = BLE_GATT_SRV_PERM_NONE,
    .min_key_size = BLE_GATT_SRV_MAX_ENCRY_KEY_SIZE,
    .uuid = BLE_UUID_INIT_128(TX_CHR_UUID),
    .descrs = {
        .descrs_p = &BLE_GATT_SRV_CCCD_DEF_NAME(tx),
        .descr_count = 1U,
    },
    /* RX characteristic */
    .properties = BLE_GATT_SRV_CHAR_PROP_WRITE | BLE_GATT_SRV_CHAR_PROP_WRITE_NO_RESP,
    .permissions = BLE_GATT_SRV_PERM_NONE,
    .min_key_size = BLE_GATT_SRV_MAX_ENCRY_KEY_SIZE,
    .uuid = BLE_UUID_INIT_128(RX_CHR_UUID),
},

/* Chat Service definition */
static const ble_gatt_srv_def_t user_service = {
    .type = BLE_GATT_SRV_PRIMARY_SRV_TYPE,
    .uuid = BLE_UUID_INIT_128(SRVC_UUID),
    .chrs = {
        .chrs_p = (ble_gatt_chr_def_t *) user_chars,
        .chr_count = 2U,
    },
};

uint16_t TXCharHandle, RXCharHandle;
A service with its characteristics can be added using the following command:

```c
ret= aci_gatt_srv_add_service((ble_gatt_srv_def_t *)&user_service);
```  

Once the service with related characteristics (TX, RX) has been added, the user can get the related TX, RX characteristics handles with the following commands:

```c
TXCharHandle=aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&user_chars[0]);
RXCharHandle=aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&user_chars[1]);
```
For a detailed description of the `aci_gatt_srv_add_service()` API parameters refer to the header file `bluenrg_lp_events.h`.

### 3.3 GAP API interface

Bluetooth LE stack v3.x has redefined the GAP API interface allowing the advertising mode and scanning procedures to be enabled, or a connection to be established between a BLE GAP central (master) device and a Bluetooth LE GAP peripheral (slave) device.

**GAP peripheral mode APIs**

The `aci_gap_set_advertising_configuration()` API allows the advertising parameters to be configured for the legacy advertising or for a given extended advertising set. In particular, it defines the discoverable modes and a type of advertising to be used.

**Table 52. aci_gap_set_advertising_configuration() API : discoverable mode and advertising type selection**

<table>
<thead>
<tr>
<th>API</th>
<th>Discoverable_Mode parameter</th>
<th>Advertising_Event_Properties parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aci_gap_set_advertising_configuration()</code></td>
<td>0x00: not discoverable</td>
<td>0x0001: connectable</td>
</tr>
<tr>
<td></td>
<td>0x01: limited discoverable</td>
<td>0x0002: scannable</td>
</tr>
<tr>
<td></td>
<td>0x02: general discoverable</td>
<td>0x0004: directed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0008: high duty cycle directed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connectable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0010: legacy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0020: anonymous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0040: include TX power</td>
</tr>
</tbody>
</table>

The `aci_gap_set_advertising_data()` API allows the data to be set in advertising PDUs. In particular, the user is requested to specify the length of advertising data (Advertising_Data_Length parameter) and to provide the advertising data (Advertising_Data parameter) inline with the advertising format defined on BLE specifications.

Once the advertising configuration and data have been defined, the user can enable/disable advertising using the `aci_gap_set_advertising_enable()` API (Enable parameter).

**GAP discovery procedure**

The `aci_gap_set_scan_configuration()` API allows the scan parameters to be configured for a given PHY. Once the scan parameters are defined, the user can start a specific discovery procedure by using the `aci_gap_start_procedure()` API, Procedure_Code parameter.

**Table 53. aci_gap_start_procedure() API**

<table>
<thead>
<tr>
<th>API</th>
<th>Procedure_Code parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aci_gap_start_procedure()</code></td>
<td>0x00: LIMITED_DISCOVERY</td>
</tr>
<tr>
<td></td>
<td>0x01: GENERAL_DISCOVERY</td>
</tr>
<tr>
<td></td>
<td>0x02: AUTO_CONNECTION</td>
</tr>
<tr>
<td></td>
<td>0x03: GENERAL_CONNECTION</td>
</tr>
<tr>
<td></td>
<td>0x04: SELECTIVE_CONNECTION</td>
</tr>
<tr>
<td></td>
<td>0x05: OBSERVATION</td>
</tr>
</tbody>
</table>

A GAP discovery procedure can be terminated using the `aci_gap_terminate_proc()` API.
The aci_gap_set_connection_configuration() API allows the connection configuration parameter to be configured for a given PHY to establish a connection with a peer device.

A direct connection with a peer device can be built by the aci_gap_create_connection(). This API specifies the PHY only to be used for the connection, the peer device type (Peer_Address_Type parameter) and the peer address (Peer_Address parameter).

The aci_gap_terminate() API allows an established connection to be terminated by selecting the related handle (Connection_Handle parameter) and the reason to end the connection (reason parameter).

### 3.3.1 Set the discoverable mode and use the direct connection establishment procedure

The following pseudocode example illustrates the specific steps only to be followed to let a GAP peripheral device be a general discoverable mode, and for a GAP central device to directly connect to it through a direct connection establishment procedure.

**Note:** It is assumed that the device public address has been set during the initialization phase as follows:
Once that advertising mode and data have been configured, the GAP peripheral device can enable advertising using the `aci_gap_set_advertising_enable()` API:
static Advertising_Set_Parameters_t Advertising_Set_Parameters[1];

/*GAP Peripheral: enable advertising (and no scan response is sent)*/
void GAP_Peripheral_Enable_Advertising(void)
{
    /* Enable advertising:
        Enable: ENABLE (enable advertising);
        Advertising_Set_Parameters: Advertising_Set_Parameters */
    Advertising_Set_Parameters[0].Advertising_Handle = 0;
    Advertising_Set_Parameters[0].Duration = 0;
    Advertising_Set_Parameters[0].Max_EXTENDED_Advertising_Events = 0;
    //enable advertising
    ret = aci_gap_set_advertising_enable(ENABLE, 1, Advertising_Set_Parameters);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
}

GAP central device must configure the scanning and connection parameters, before connecting to the GAP peripheral device in discoverable mode.

/*GAP Central: configure scanning and connection parameters*/
void GAP_Central_Configure_Connection(void)
{
    /* Configure the scanning parameters:
        Filter_Duplicates: DUPLICATE_FILTER_ENABLED (Duplicate filtering enabled)
        Scanning_Filter_Policy: SCAN_ACCEPT_ALL (accept all scan requests)
        Scanning_PHY: LE_1M_PHY (1Mbps PHY)
        Scan_Type: PASSIVE_SCAN
        Scan_Interval: 0x4000;
        Scan_Window: 0x4000;
    */
    ret = aci_gap_set_scan_configuration(DUPLICATE_FILTER_ENABLED,
                                          SCAN_ACCEPT_ALL,
                                          LE_1M_PHY,
                                          PASSIVE.Scan,
                                          0x4000,
                                          0x4000);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

    /* Configure the connection parameters:
        Initiating_PHY: LE_1M_PHY (1Mbps PHY)
        Conn_Interval_Min: 40 (Minimum value for the connection event interval);
        Conn_Interval_Max: 40 (Maximum value for the connection event interval);
        Conn_Delay: 0 (Slave latency for the connection in a number of connection events);
        Supervision_Timeout: 60 (Supervision timeout for the LE Link); Minimum_CE_Length: 2000 (Minimum length of connection needed for the LE connection);
        Maximum_CE_Length: 2000 (Maximum length of connection needed for the LE connection).
    */
    ret = aci_gap_set_connection_configuration(LE_1M_PHY, 40, 40, 0, 60,
                                              2000, 2000);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
    /* GAP_Central_Configure_Connection()*/

Once the scanning and connection parameters have been configured, the GAP central device can perform the direct connection to the GAP peripheral device using the aci_gap_create_connection() API.
void GAP_Central_Make_Connection(void)
{
    tBDAddr GAP_Peripheral_address = {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
    ret = aci_gap_create_connection(LE_1M_PHY,
        PUBLIC_ADDR, GAP_Peripheral_address);
    if(ret != BLE_STATUS_SUCCESS) PRINTF("Failure.");
}
/* GAP_Central_Make_Connection(void )*/

Note: 1. If ret = BLE_STATUS_SUCCESS is returned, on termination of the GAP procedure, the event callback
    hci_le_enhanced_connection_complete_event() is called, to indicate that a connection has been
    established with the GAP_Peripheral_address (same event is returned on the GAP peripheral device)
2. The connection procedure can be explicitly terminated by issuing the API aci_gap_terminate_proc()
    with proper Procedure_Code parameter value
3. The last two parameters Minimum_CE_Length and Maximum_CE_Length of the
    aci_gap_set_connection_configuration() are the length of the connection event needed for the
    Bluetooth LE connection. These parameters allow the user to specify the amount of time the master has to
    allocate for a single slave so they must be chosen wisely. In particular, when a master connects to more
    slaves, the connection interval for each slave must be equal or a multiple of the other connection intervals
    and the user must not overdo the connection event length for each slave.

3.3.2 Set discoverable mode and use general discovery procedure (active scan)

The following pseudocode example illustrates the specific steps only to be followed to let a GAP peripheral device
be in a general discoverable mode, and for a GAP central device to start a general discovery procedure in order
to discover the devices within its radio range.

Note: It is assumed that the device public address has been set during the initialization phase as follows:

uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret =aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,CONFIG_DATA_PUBADDR_LEN, bdaddr)
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.");

Furthermore, the GAP peripheral device has configured the advertising and related data as described in
Section 3.3.1 Set the discoverable mode and use the direct connection establishment procedure. The GAP
peripheral device can enable scan response data and then the advertising using the
aci_gap_set_advertising_enable() API:
static Advertising_Set_Parameters_t Advertising_Set_Parameters[1];

/* GAP Peripheral: general discoverable mode (scan responses are sent): */
void GAP_Peripheral_Make_Discoverable(void)
{
    tBleStatus ret;
    const char local_name[] = {AD_TYPE_COMPLETE_LOCAL_NAME,'B','l','u','e', 'N','R','G'};
    /* As scan response data, a proprietary 128bits Service UUID is used. This 128bits data cannot be inserted within the advertising packet (ADV_IND) due its length constraints (31 bytes). AD Type description:
    0x11: length
    0x06: 128 bits Service UUID type
    0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x20,0x0c, 0x9a,0x66: 128 bits Service UUID */
    uint8_t ServiceUUID_Scan[18]=
    {0x11,0x06,0x8a,0x97,0xf7,0xc0,0x85, 0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x20,0x0c,0x9a,0x66};
    /* Enable scan response to be sent when GAP peripheral receives scan requests from GAP Central performing general discovery procedure (active scan) */
    aci_gap_set_scan_response_data(18,ServiceUUID_Scan);
    /* Enable advertising: 
    Enable: ENABLE (enable advertising); 
    Number_of_Sets: 1; 
    Advertising_Set_Parameters: Advertising_Set_Parameters */
    Advertising_Set_Parameters[0].Advertising_Handle = 0;
    Advertising_Set_Parameters[0].Duration = 0;
    Advertising_Set_Parameters[0].Max_Extended_Advertising_Events = 0;
    //enable advertising 
    ret = aci_gap_set_advertising_enable(ENABLE, 1,Advertising_Set_Parameters);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");
} /* end GAP_Peripheral_Make_Discoverable() */

The GAP central device must configure the scanning parameters, before starting the GAP general discovery procedure.
/*GAP Central: configure scanning parameters for general discovery procedure*/
void GAP_Central_Configure_General_Discovery_Procedure(void)
{
    tBleStatus ret;
    /* Configure the scanning parameters (active scan):
        Filter_Duplicates: DUPLICATE_FILTER_DISABLED (Duplicate filtering disabled)
        Scanning_Filter_Policy: SCAN_ACCEPT_ALL (accept all scan requests)
        Scanning_PHY: LE_1M_PHY (1Mbps PHY)
        Scan_Type: ACTIVE_SCAN
        Scan_Interval: 0x4000;
        Scan_Window: 0x4000;
    */
    ret=aci_gap_set_scan_configuration(DUPLICATE_FILTER_DISABLED,
                                        SCAN_ACCEPT_ALL,
                                        LE_1M_PHY,
                                        ACTIVE_SCAN,
                                        0x4000,
                                        0x4000);
    if(ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
    ");
    /* end GAP_Central_Configure_General_Discovery_Procedure() */
}

/*GAP Central: start general discovery procedure to discover the GAP peripheral device in discoverable mode */
void GAP_Central_General_Discovery_Procedure(void)
{
    tBleStatus ret;
    /* Start the general discovery procedure using the following parameters:
        Procedure_Code: 0x1 (general discovery procedure)
        PHYs: LE_1M_PHY (1Mbps PHY)*/
    ret =aci_gap_start_procedure(0x01,LE_1M_PHY,0,0);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
    ");
}

The responses of the procedure are given through the event callback
hci_le_extended_advertising_report_event(). The end of the procedure is indicated by
aci_gap_proc_complete_event() event callback with Procedure_Code parameter equal to
GAP_GENERAL_DISCOVERY_PROC (0x1).
void hci_le_extended_advertising_report_event(uint8_t Num_Reports,
                                                      Extended_Advertising_Report_t Advertising_Report[])
{
    /* Advertising_Report contains all the expected parameters.
       User application should add code for decoding the received
       Advertising_Report event databased on the specific evt_type (ADV_IND, SCAN_RSP, ..) */

    /* Example: store the received Advertising_Report fields */
    uint8_t bdaddr[6];
    /* type of the peer address (PUBLIC_ADDR,RANDOM_ADDR) */
    uint8_t bdaddr_type = Advertising_Report[0].Address_Type;

    /* event type (advertising packets types) */
    uint8_t evt_type = Advertising_Report[0].Event_Type;

    /* RSSI value */
    uint8_t RSSI = Advertising_Report[0].RSSI;

    /* address of the peer device found during discovery procedure */
    Osal_MemCpy(bdaddr, Advertising_Report[0].Address, 6);

    /* length of advertising or scan response data */
    uint8_t data_length = Advertising_Report[0].Length_Data;

    /* data_length octets of advertising or scan response data
       formatted are on Advertising_Report[0].Data field to be stored/filtered based on specific user application scenario*/

} /* hci_le_extended_advertising_report_event() */

In particular, in this specific context, the following events are raised on the GAP central
hci_le_extended_advertising_report_event(), as a consequence of the GAP peripheral device in
discernible mode with scan response enabled:

1. Advertising Report event with advertising packet type (evt_type = ADV_IND - 0x0013)
2. Advertising Report event with scan response packet type (evt_type = SCAN_RSP - 0x001B)

Table 55. ADV_IND event type: main fields

<table>
<thead>
<tr>
<th>Event type</th>
<th>Address type</th>
<th>Address</th>
<th>Advertising data</th>
<th>RSSI</th>
</tr>
</thead>
</table>
| 0x0013 (ADV_IND) | 0x00 (public address) | 0x028E1003 412 | 0x02,0x01,0x06,0x08,0x09,0x42
 ,0x6C,0x75,0x65,0x4E,0x52,0x47,0x02,0xA,0xFE,0xCE | 0xCE |

The advertising data are shown as follows (refer to Bluetooth specification version in Section 5 References):

Table 56. ADV_IND advertising data: main fields

<table>
<thead>
<tr>
<th>Flag AD type field</th>
<th>Local name field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02: length of the field 0x01: AD type flags</td>
<td>0x08: length of the field</td>
</tr>
<tr>
<td>0x06: 0x110 (Bit 2: BR/EDR Not supported; bit 1: general discoverable mode)</td>
<td>0x09: complete local name type</td>
</tr>
<tr>
<td></td>
<td>0x42,0x6C,0x75,0x65,0x4E0x</td>
</tr>
<tr>
<td></td>
<td>52,0x47: BlueNRG</td>
</tr>
</tbody>
</table>
Table 57. SCAN_RSP event type

<table>
<thead>
<tr>
<th>Event type</th>
<th>Address type</th>
<th>Address</th>
<th>Scan response data</th>
<th>RSSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x001B (SCAN_RSP)</td>
<td>0x01 (random address)</td>
<td>0x0280E1003412</td>
<td>0x12,0x66,0xA9,0xA0,0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0xA85,0xC0,0xF7,0x97,0x8A,0x06,0x11</td>
<td>0xDA</td>
</tr>
</tbody>
</table>

The scan response data can be interpreted as follows: (refer to Bluetooth specifications):

Table 58. Scan response data

<table>
<thead>
<tr>
<th>Scan response data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12: data length</td>
</tr>
<tr>
<td>0x11: length of service UUID advertising data; 0x06: 128 bits service UUID type;</td>
</tr>
<tr>
<td>0x66,0xA9,0xA0,0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0xA85,0xC0,0xF7,0x97,0x8A:</td>
</tr>
<tr>
<td>128-bit service UUID</td>
</tr>
</tbody>
</table>

3.4 Bluetooth LE stack events and event callbacks

Whenever there is a Bluetooth LE stack event to be processed, the Bluetooth LE stack library notifies this event to the user application through a specific event callback. An event callback is a function defined by the user application and called by the Bluetooth LE stack, while an API is a function defined by the stack and called by the user application. The Bluetooth LE stack event callback prototypes are defined on file `bluenrg_lp_events.h`. Weak definitions are available for all the event callbacks in order to have a definition for each event callback. As a consequence, based on their own application scenario, the user has to identify the required device event callbacks to be called and the related application specific actions to be done.

When a Bluetooth LE application is implemented, the most common and widely used Bluetooth LE stack events are those related to the discovery, connection and terminate procedures, services, characteristics, characteristics descriptors discovery procedures and attribute notification/ indication events on a GATT client, attribute writes/ reads events on a GATT server.

Table 59. Bluetooth LE stack: main event callbacks

<table>
<thead>
<tr>
<th>Event callback</th>
<th>Description</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>hci_disconnection_complete_event()</td>
<td>A connection is terminated</td>
<td>GAP central/peripheral</td>
</tr>
<tr>
<td>hci_le_enhanced_connection_complete_event()</td>
<td>Indicates to both of the devices forming the connection that a new connection has been established. This event is raised when the Bluetooth LE stack supports the extended advertising/scanning features</td>
<td>GAP central/peripheral (Bluetooth LE stack v3.x)</td>
</tr>
<tr>
<td>aci_gatt_clt_notification_event()</td>
<td>Generated by the GATT client when a server notifies any attribute on the client</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_indication_event()</td>
<td>Generated by the GATT client when a server indicates any attribute on the client</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gap_pass_key_req_event()</td>
<td>Generated by the security manager to the application when a passkey is required for pairing.</td>
<td>GAP central/peripheral</td>
</tr>
<tr>
<td>Event callback</td>
<td>Description</td>
<td>Where</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>aci_gap_pairing_complete_event()</td>
<td>Generated when the pairing process has completed successfully or a pairing procedure timeout has occurred or the pairing has failed</td>
<td>GAP central/peripheral</td>
</tr>
<tr>
<td>aci_gap_bond_lost_event()</td>
<td>Event generated when a pairing request is issued, in response to a slave security request from a master which has previously bonded with the slave. When this event is received, the upper layer has to issue the command aci_gap_allow_rebond() to allow the slave to continue the pairing process with the master</td>
<td>GAP peripheral</td>
</tr>
<tr>
<td>aci_gatt_clt_read_by_group_type_resp_event()</td>
<td>The Read-by-group type response is sent in reply to a received Read-by-group type request and contains the handles and values of the attributes that have been read</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_read_by_type_resp_event()</td>
<td>The Read-by-type response is sent in reply to a received Read-by-type request and contains the handles and values of the attributes that have been read</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_proc_complete_event()</td>
<td>A GATT procedure has been completed</td>
<td>GATT client</td>
</tr>
<tr>
<td>hci_le_extended_advertising_report_event()</td>
<td>Event given by the GAP layer to the upper layers when a device is discovered during scanning as a consequence of one of the GAP procedures started by the upper layers. This event is raised when the BLE stack supports the extended advertising/scanning features</td>
<td>GAP central (BLE stack v3.x)</td>
</tr>
</tbody>
</table>

For a detailed description of the Bluetooth LE event, and related formats refer to the Bluetooth LE stack APIs and events documentation in Section 5 References.

The following pseudocode provides an example of event callbacks handling some of the described Bluetooth LE stack events (disconnection complete event, connection complete event, GATT attribute modified event, GATT notification event):
void hci_disconnection_complete_event(uint8_t Status, uint16_t Connection_Handle, uint8_t Reason)
{
    /* Add user code for handling Bluetooth LE disconnection complete event based on application scenario. */
}

/* This event callback indicates the end of a connection procedure.
* NOTE: If the Bluetooth LE v3.x stack includes the extended advertising/scanning features, the
* hci_le_enhanced_connection_complete_event() is raised.
* /
void hci_le_enhanced_connection_complete_event(uint8_t Status,
                                             uint16_t Connection_Handle,
                                             uint8_t Role,
                                             uint8_t Peer_Address_Type,
                                             uint8_t Peer_Address[6],
                                             uint8_t Local_Resolvable_Private_Address[6],
                                             uint8_t Peer_Resolvable_Private_Address[6],
                                             uint16_t Conn_Interval,
                                             uint16_t Conn_Latency,
                                             uint16_t Supervision_Timeout,
                                             uint8_t Master_Clock_Accuracy);
{
    /* Add user code for handling BLE connection complete event based on application scenario.
    NOTE: Refer to header file Library\Bluetooth_LE\inc\bluenrg1_events.h for a complete description of the event callback parameters. */
    
    /* Store connection handle */
    connection_handle = Connection_Handle;
    ...
}

#if GATT_SERVER
/* This event callback indicates that an attribute has been written from a peer device. */
void aci_gatt_srv_write_event(uint16_t Connection_Handle,
                              uint8_t Resp_Needed,
                              uint16_t Attribute_Handle,
                              uint16_t Data_Length,
                              uint8_t Data[]);
{
    /* Add user code for handling attribute modification event based on application scenario.
    NOTE: Refer to header file bluenrglp_events.h for a complete description of the event callback parameters. */
    ...
}
#endif /* GATT_SERVER */

#if GATT_CLIENT
/* This event callback indicates that an attribute notification has been received from a peer device. */
void aci_gatt_notification_event(uint16_t Connection_Handle, uint16_t Attribute_Handle,
                                 uint8_t Attribute_Value_Length, uint8_t Attribute_Value[])
{
    /* Add user code for handling attribute notification event based on application scenario.
    NOTE: Refer to header bluenrglp_events.h for a complete description of the event callback parameters*/
    ...
}
#endif /* GATT_CLIENT */
3.5 Security (pairing and bonding)

This section describes the main functions to be used in order to establish a pairing between two devices (authenticate the device identity, encrypt the link and distribute the keys to be used on the next reconnections).

To successfully pair with a device, IO capabilities have to be correctly configured, depending on the IO capability available on the selected device.

`aci_gap_set_io_capability(io_capability)` should be used with one of the following `io_capability` values:

- 0x00: 'IO_CAP_DISPLAY_ONLY'
- 0x01: 'IO_CAP_DISPLAY_YES_NO'
- 0x02: 'KEYBOARD_ONLY'
- 0x03: 'IO_CAP_NO_INPUT_NO_OUTPUT'
- 0x04: 'IO_CAP_KEYBOARD_DISPLAY'

**PassKey Entry example with 2 Bluetooth LE devices: Device_1, Device_2**

The following pseudocode example illustrates only the specific steps to be followed to pair two devices by using the PassKey entry method.

As described in Table 2. LE PHY key parameters, Device_1, Device_2 have to set the IO capability in order to select PassKey entry as a security method.

In this particular example, "Display Only" on Device_1 and "Keyboard Only" on Device_2 are selected, as follows:

```c
/*Device_1: */
tBleStatus ret;
ret = aci_gap_set_io_capability(IO_CAP_DISPLAY_ONLY);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");

/*Device_2: */
tBleStatus ret;
ret = aci_gap_set_io_capability(IO_CAP_KEYBOARD_ONLY);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");
```

Once the IO capability is defined, the `aci_gap_set_authentication_requirement()` should be used to set all the security authentication requirements the device needs (MITM mode (authenticated link or not), OOB data present or not, use fixed pin or not, enabling bonding or not).

The following pseudocode example illustrates only the specific steps to be followed to set the authentication requirements for a device with: "MITM protection, no OOB data, don’t use fixed pin": this configuration is used to authenticate the link and to use a not fixed pin during the pairing process with PassKey Method.

```c
ret = aci_gap_set_authentication_requirement(BONDING, /*bonding is enabled */
                                         MITM_PROTECTION_REQUIRED,
                                         SC_IS_SUPPORTED, /*Secure connection supported but optional */
                                         KEYPRESS_IS_NOT_SUPPORTED,
                                         7, /* Min encryption key size */
                                         16, /* Max encryption key size */
                                         0x01, /* fixed pin is not used */
                                         0x123456, /* fixed pin */
                                         0x00 /* Public Identity address type */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");
```

Once the security IO capability and authentication requirements are defined, an application can initiate a pairing procedure as follows:

1. By using `aci_gap_slave_security_req()` on a GAP peripheral (slave) device (it sends a slave security request to the master):
• Or by using the `aci_gap_send_pairing_req()` on a GAP central (master) device.

Since the no fixed pin has been set, once the pairing procedure is initiated by one of the two devices, Bluetooth LE device calls the `aci_gap_pass_key_req_event()` event callback (with related connection handle) to ask the user application to provide the password to be used to establish the encryption key. Bluetooth LE application has to provide the correct password by using the `aci_gap_pass_key_resp(conn_handle, passkey)` API.

When the `aci_gap_pass_key_req_event()` callback is called on Device_1, it should generate a random pin and set it through the `aci_gap_pass_key_resp()` API, as follows:

```c
void aci_gap_pass_key_req_event(uint16_t Connection_Handle)
{
  tBleStatus ret;
  uint32_t pin;
  /*Generate a random pin with an user specific function */
  pin = generate_random_pin();
  ret= aci_gap_pass_key_resp(Connection_Handle,pin);
  if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
}
```

Since the Device_1, I/O capability is set as “Display Only”, it should display the generated pin in the device display. Since Device_2, I/O capability is set as “Keyboard Only”, the user can provide the pin displayed on Device_1 to the Device_2 though the same `aci_gap_pass_key_resp()` API, by a keyboard.

Alternatively, if the user wants to set the authentication requirements with a fixed pin 0x123456 (no pass key event is required), the following pseudocode can be used:

```c
tBleStatus ret;
ret= aci_gap_set_auth_requirement(BONDING, /* bonding is enabled */
  MITM_PROTECTION_REQUIRED,
  SC_IS_SUPPORTED, /* Secure connection supported but optional */
  KEYPRESS_IS_NOT_SUPPORTED,
  7, /* Min encryption key size */
  16, /* Max encryption key size */
  0x00, /* fixed pin is used*/
  0x123456, /* fixed pin */
  0x00 /* Public Identity address */
);  
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```
### 3.6 Service and characteristic discovery

This section describes the main functions allowing a GAP central device to discover the GAP peripheral services and characteristics, once the two devices are connected. The sensor profile demo services and characteristics, with related handles, are used as reference services and characteristics on the following pseudocode examples. Furthermore, it is assumed that a GAP central device is connected to a GAP peripheral device running the sensor demo profile application. The GAP central device uses the service and discovery procedures to find the GAP peripheral sensor profile demo service and characteristics.

<table>
<thead>
<tr>
<th>Service</th>
<th>Characteristic</th>
<th>Service / characteristic handle</th>
<th>Characteristic value handle</th>
<th>Characteristic client descriptor configuration handle</th>
<th>Characteristic format handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration service</td>
<td>NA</td>
<td>0x0010</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Free Fall characteristic</td>
<td>0x0011</td>
<td>0x0012</td>
<td>0x0013</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Acceleration characteristic</td>
<td>0x0014</td>
<td>0x0015</td>
<td>0x0016</td>
<td>NA</td>
</tr>
<tr>
<td>Environmental service</td>
<td>NA</td>
<td>0x0017</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Temperature characteristic</td>
<td>0x0018</td>
<td>0x0019</td>
<td>NA</td>
<td>0x001A</td>
</tr>
<tr>
<td></td>
<td>Pressure characteristic</td>
<td>0x001B</td>
<td>0x001C</td>
<td>NA</td>
<td>0x001D</td>
</tr>
</tbody>
</table>
For detailed information about the sensor profile demo, refer to the SDK user manual and the sensor demo source code available within the SDK software package (see Section 5 References).

A list of the service discovery APIs with related description is as follows:

<table>
<thead>
<tr>
<th>Discovery service API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aci_gatt_clt_disc_all_primary_services()</td>
<td>This API starts the GATT client procedure to discover all primary services on the GATT server. It is used when a GATT client connects to a device and it wants to find all the primary services provided on the device to determine what it can do.</td>
</tr>
<tr>
<td>aci_gatt_clt_disc_primary_service_by_uuid()</td>
<td>This API starts the GATT client procedure to discover a primary service on the GATT server by using its UUID. It is used when a GATT client connects to a device and it wants to find a specific service without the need to get any other services.</td>
</tr>
<tr>
<td>aci_gatt_clt_find_included_services()</td>
<td>This API starts the procedure to find all included services. It is used when a GATT client wants to discover secondary services once the primary services have been discovered.</td>
</tr>
</tbody>
</table>

The following pseudocode example illustrates the **aci_gatt_clt_disc_all_primary_services()** API:

```c
/* GAP Central starts a discovery all services procedure: conn_handle is the connection handle returned on hci_le_extended_advertising_report_event() event callback */
if (aci_gatt_clt_disc_all_primary_services(conn_handle) != BLE_STATUS_SUCCESS) {
    PRINTF("Failure.\n");
}
```

The responses of the procedure are given through the **aci_gatt_clt_read_by_group_type_resp_event()** event callback. The end of the procedure is indicated by **aci_gatt_clt_proc_complete_event()** event callback() call.

```c
/* This event is generated in response to a Read By Group Type Request: refer to aci_gatt_clt_disc_all_primary_services() */
void aci_gatt_clt_read_by_group_type_resp_event(uint16_t Conn_Handle, uint8_t Attr_Data_Length, uint8_t Data_Length, uint8_t Att_Data_List[]);
{
    /* Conn_Handle: connection handle related to the response; 
    Attr_Data_Length: the size of each attribute data; 
    Data_Length: length of Attribute_Data_List in octets; 
    Att_Data_List: Attribute Data List as defined in Bluetooth LE Specifications.
     A sequence of attribute handle, end group handle, attribute value tuples: 
     (2 octets for Attribute Handle, 2 octets End Group Handle, 
     (Attribute_Data_Length - 4 octets) for 
     Attribute Value). 
    */

    /* Add user code for decoding the Att_Data_List field and getting the services attribute handle, end group handle and service uuid */
    /* aci_gatt_clt_read_by_group_type_resp_event() */
```

In the context of the sensor profile demo, the GAP central application should get three read by group type response events (through related **aci_gatt_clt_read_by_group_type_resp_event()** event callback), with the following callback parameter values.

First read by group type response event callback parameters:
Connection_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x06 (length of each discovered service data: service handle, end group handle, service uuid);
Data_Length: 0x0C (length of Attribute_Data_List in octets);
Att_Data_List: 0x0C bytes as follows:

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>End group handle</th>
<th>Service UUID</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0x0008</td>
<td>0x1801</td>
<td>Attribute profile service. Standard 16-bit service UUID</td>
</tr>
<tr>
<td>0x0009</td>
<td>0x000F</td>
<td>0x1800</td>
<td>GAP profile service. Standard 16-bit service UUID.</td>
</tr>
</tbody>
</table>

Second read by group type response event callback parameters:

Conn_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x14 (length of each discovered service data: service handle, end group handle, service uuid);
Data_Length: 0x14 (length of Attribute_Data_List in octets);
Att_Data_List: 0x14 bytes as follows:

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>End group handle</th>
<th>Service UUID</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0010</td>
<td>0x0016</td>
<td>0x02366E80CF3A11E19AB40002A5D5C51B</td>
<td>Acceleration service 128-bit service proprietary UUID</td>
</tr>
</tbody>
</table>

Third read by group type response event callback parameters:

Connection_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x14 (length of each discovered service data: service handle, end group handle, service uuid);
Data_Length: 0x14 (length of Attribute_Data_List in octets);
Att_Data_List: 0x14 bytes as follows:

<table>
<thead>
<tr>
<th>Attribute handle</th>
<th>End group handle</th>
<th>Service UUID</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0017</td>
<td>0x001D</td>
<td>0x42821A40E47711E282D0002A5D5C51B</td>
<td>Environmental service 128-bit service proprietary UUID</td>
</tr>
</tbody>
</table>

In the context of the sensor profile demo, when the discovery all primary service procedure completes, the aci_gatt_clt_proc_complete_event() event callback is called on GAP central application, with the following parameters:

Conn_Handle: 0x0801 (connection handle);
Error_Code: 0x00
3.7 Characteristic discovery procedures and related GATT events

A list of the characteristic discovery APIs with associated description is as follows:

<table>
<thead>
<tr>
<th>Discovery service API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aci_gatt_ctl_disc_all_char_of_service ()</td>
<td>This API starts the GATT procedure to discover all the characteristics of a given service</td>
</tr>
<tr>
<td>aci_gatt_ctl_disc_char_by_uuid ()</td>
<td>This API starts the GATT procedure to discover all the characteristics specified by a UUID</td>
</tr>
<tr>
<td>aci_gatt_ctl_disc_all_char_desc ()</td>
<td>This API starts the procedure to discover all characteristic descriptors on the GATT server</td>
</tr>
</tbody>
</table>

In the context of the BLE sensor profile demo, follow a simple pseudocode illustrating how a GAP central application can discover all the characteristics of the acceleration service (refer to Table 1. Bluetooth LE RF channel types and frequencies second read by group type response event callback parameters):

```c
uint16_t service_handle = 0x0010;
uint16_t end_group_handle = 0x0016;

/*GAP Central starts a discovery all the characteristics of a service procedure: conn_handle is the connection handle returned on hci_le_advertising_report_event()eventcallback */
if(aci_gatt_disc_all_char_of_service(conn_handle,
    service_handle,/* Service handle */
    end_group_handle/* End group handle */
) != BLE_STATUS_SUCCESS)
{
    PRINTF("Failure.
");
}
```

The responses of the procedure are given through the `aci_att_read_by_type_resp_event()` event callback. The end of the procedure is indicated by `aci_gatt_proc_complete_event()` event callback call.

```c
void aci_att_read_by_type_resp_event(uint16_t Connection_Handle,
    uint8_t Handle_Value_Pair_Length,
    uint8_t Data_Length,
    uint8_t Handle_Value_Pair_Data[])
{
    /*
    Connection_Handle: connection handle related to the response;
    Handle_Value_Pair_Length: size of each attribute handle-value Pair;
    Data_Length: length of Handle_Value_Pair_Data in octets.
    Handle_Value_Pair_Data: Attribute Data List as defined in Bluetooth Core specifications. A sequence of handle-value pairs: [2 octets for Attribute Handle, (Handle_Value_Pair_Length - 2 octets) for Attribute Value].
    */
    /* Add user code for decoding the Handle_Value_Pair_Data field and get the characteristic handle, properties, characteristic value handle, characteristic UUID*/

    /* aci_att_read_by_type_resp_event() */
}
```

In the context of the Bluetooth LE sensor profile demo, the GAP central application should get two read type response events (through related `aci_att_read_by_type_resp_event()` event callback), with the following callback parameter values.

First read by type response event callback parameters:
Table 66. First read by type response event callback parameters

<table>
<thead>
<tr>
<th>Characteristic handle</th>
<th>Characteristic properties</th>
<th>Characteristic value handle</th>
<th>Characteristic UUID</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0011</td>
<td>0x10 (notify)</td>
<td>0x0012</td>
<td>0xE23E78A0CF4A11E18FFC002A5D5C51B</td>
<td>Free fall characteristic 128-bit proprietary UUID</td>
</tr>
</tbody>
</table>

Table 67. Second read by type response event callback parameters

<table>
<thead>
<tr>
<th>Characteristic handle</th>
<th>Characteristic properties</th>
<th>Characteristic value handle</th>
<th>Characteristic UUID</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0014</td>
<td>0x12 (notify and read)</td>
<td>0x0015</td>
<td>0x340A1B80CF4B11E1AC360002A5D5C51B</td>
<td>Acceleration characteristic 128-bit proprietary UUID</td>
</tr>
</tbody>
</table>

In the context of the sensor profile demo, when the discovery all primary service procedure completes, the `aci_gatt_proc_complete_event()` event callback is called on GAP central application, with the following parameters:

Connection_Handle: 0x0801 (connection handle);
Error_Code: 0x00.

Similar steps can be followed in order to discover all the characteristics of the environment service (Table 1. Bluetooth LE RF channel types and frequencies).

### 3.8 Characteristic notification/indications, write, read

This section describes the main functions to get access to Bluetooth LE device characteristics.
Table 68. Characteristic update, read, write APIs

<table>
<thead>
<tr>
<th>Discovery service API</th>
<th>Description</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>aci_gatt_srv_notify</td>
<td>If notifications (or indications) are enabled on the characteristic, this API sends a notification (or indication) to the client.</td>
<td>GATT server</td>
</tr>
<tr>
<td>aci_gatt_clt_read</td>
<td>It starts the procedure to read the attribute value.</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_write</td>
<td>It starts the procedure to write the attribute value (when the procedure is completed, a GATT procedure complete event is generated).</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_write_without_resp()</td>
<td>It starts the procedure to write a characteristic value without waiting for any response from the server.</td>
<td>GATT client</td>
</tr>
<tr>
<td>aci_gatt_clt_confirm_indication()</td>
<td>It confirms an indication. This command has to be sent when the application receives a characteristic indication.</td>
<td>GATT client</td>
</tr>
</tbody>
</table>

In the context of the sensor profile demo, the GAP central application should use a simple pseudocode in order to configure the free fall and the acceleration characteristic client descriptor configuration for notification:

tBleStatus ret;
uint16_t handle_value = 0x0013;
/*Enable the free fall characteristic client descriptor configuration */
ret = aci_gatt_clt_write(conn_handle,
handle_value /* handle for free fall client descriptor configuration */
0x02, /* attribute value length */
0x0001, /* attribute value: 1 for notification */
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");
handle_value = 0x0016;
/*Enable the acceleration characteristic client descriptor configuration for notification */
ret = aci_gatt_clt_write (conn_handle, handle_value /* handle for acceleration client descriptor configuration */
0x02, /* attribute value length */
0x0001, /* attribute value: 1 for notification */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");

Once the characteristic notification has been enabled from the GAP central, the GAP peripheral can notify a new value for the free fall and acceleration characteristics as follows:

tBleStatus ret;
uint8_t val = 0x01;
uint16_t charac_handle = 0x0017;
/*GAP peripheral notifies free fall characteristic to GAP central*/
ret = aci_gatt_srv_notify (connection_handle, /*connection handle*/
charac_handle, /* free fall characteristic handle*/
0, /*updated type: notification*/
0x01, /* characteristic value length */
&val /* characteristic value */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");

/*Set the mems acceleration values on three axis x,y,z on buff array*/
/*GAP peripheral notifies acceleration characteristic to GAP Central*/
ret = aci_gatt_srv_notify (connection_handle, /* connection handle */
charac_handle, /* acceleration characteristic handle*/
0, /* updated type: notification */
0x06, /* characteristic value length */
buff /* characteristic value */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.
");
On GAP central, the `aci_gatt_clt_notification_event()` event callback is raised on reception of the characteristic notification (acceleration or free fall) from the GAP peripheral device. Follow a pseudocode of the `aci_gatt_clt_notification_event()` callback:

```c
void aci_gatt_clt_notification_event(uint16_t Connection_Handle,
    uint16_t Attribute_Handle,
    uint8_t Attribute_Value_Length,
    uint8_t Attribute_Value[])
{
    /* aci_gatt_clt_notification_event event callback parameters:
        Connection_Handle: connection handle related to the response;
        Attribute_Handle: the handle of the notified characteristic;
        Attribute_Value_Length: length of Attribute_Value in octets;
        Attribute_Value: the current value of the notified characteristic.
    */
    /* Add user code for handling the received notification based on the
        application scenario.
    */
}
/* aci_gatt_clt_notification_event */
```

### 3.9 Basic/typical error condition description

On the Bluetooth LE stack v3.x API framework, the `tBleStatus` type is defined in order to return the Bluetooth LE stack error conditions. The error codes are defined within the header file "ble_status.h". When a stack API is called, it is recommended to get the API return status and to monitor it in order to track potential error conditions.

BLE_STATUS_SUCCESS (0x00) is returned when the API is successfully executed. For a list of error conditions associated to each ACI API refer to Bluetooth LE stack APIs and event documentation, in Section 5 References.

### 3.10 Simultaneously master, slave scenario

The Bluetooth LE stack v3.x supports multiple roles simultaneously. This allows the same device to act as master on one or more connections (up to eight connections are supported), and to act as a slave on another connection. The following pseudocode describes how a Bluetooth LE stack device can be initialized to support central and peripheral roles simultaneously:

```c
uint8_t role = GAP_PERIPHERAL_ROLE | GAP_CENTRAL_ROLE;
ret = aci_gap_init(role, 0, 0x07, &service_handle,
    &dev_name_char_handle, &appearance_char_handle);
```

A simultaneous master and slave test scenario can be targeted as follows:
Figure 14. Bluetooth LE simultaneous master and slave scenario

Step 1. One Bluetooth LE device (called Master&Slave) is configured as central and peripheral by setting role as GAP_PERIPHERAL_ROLE | GAP_CENTRAL_ROLE on GAP_Init() API. Let’s also assume that this device also defines a service with a characteristic.

Step 2. Two devices (called Slave_A, Slave_B) are configured as peripheral by setting role as GAP_PERIPHERAL_ROLE on GAP_Init() API. Both Slave_A and Slave_B define the same service and characteristic as Master&Slave device.

Step 3. One device (called Master) is configured as central by setting role as GAP_CENTRAL_ROLE on GAP_Init() API.
Step 4. Both Slave_A and Slave_B devices enter discovery mode by using the following APIs:

- aci_gap_set_advertising_configuration(). This API defines the advertising configuration with the following main parameters:
  - Discoverable_mode = GAP_MODE_GENERAL_DISCOVERABLE
  - Advertising_Event_Properties = ADV_PROP_CONNECTABLE|ADV_PROP_SCANNABLE|ADV_PROP_LEGACY,
  - Primary_Advertising_Interval_Min = 0x20
  - Primary_Advertising_Interval_Max = 0x100
- aci_gap_set_advertising_enable(). This API allows advertising to be enabled as follows:

  ```c
  static Advertising_Set_Parameters_t Advertising_Set_Parameters[1];
  Advertising_Set_Parameters[0].Advertising_Handle = 0;
  Advertising_Set_Parameters[0].Duration = 0;
  Advertising_Set_Parameters[0].Max_Extended_Advertising_Events = 0;
  ret = aci_gap_set_advertising_enable(ENABLE, 1,Advertising_Set_Parameters);
  ```
- aci_gap_set_advertising_data(). This API defines the advertising data with the following parameters:
  - Advertising_Handle = 0;
  - Operation = ADV_COMPLETE_DATA'
  - Advertising_Data_Length = sizeof(adv_data);
  - Advertising_Data_Length = adv_data
  - Where adv_data is defined as follows:

  ```c
  static uint8_t adv_data[] = {0x02,AD_TYPE_FLAGS,
   FLAG_BIT_LE_GENERAL_DISCOVERABLE_MODE|FLAG_BIT_BR_EDR_NOT_SUPPORTED,6,AD_TYPE_COMPLETE_LOCAL_NAME,
   0x08,0x74,0x65,0x73,0x74};
  ```

Step 5. Master&Slave device configures the scanning and connection parameters before performing a discovery procedure, by using the following APIs:

- aci_gap_set_scan_configuration(). This API defines the scanning parameters:
  - Filter_Duplicates: 0x0;
  - Scanning_Filter_Policy: 0x0 (accept all);
  - Scanning_PHY: LE_1M_PHY (1 Mbps PHY);
  - Scan_Type: PASSIVE_SCAN;
  - Scan_Interval: 0x10;Scan_Window: 0x10
- aci_gap_set_connection_configuration(). This API defines the connection parameters:
  - Initiating_PHY = LE_1M_PHY (1 Mbps PHY);
  - Conn_Interval_Min = 0x6c;
  - Conn_Interval_Max = 0x6c;
  - Conn_Latency = 0;
  - Supervision_Timeout = 0xc80;
  - Minimum_CE_Length = 0x000c;
  - Maximum_CE_Length = 0x000c
Step 6. Master&Slave device performs a discovery procedure in order to discover the peripheral devices Slave_A and Slave_B:
The general discovery procedure is started by calling the API aci_gap_start_procedure() with the parameters:
– Procedure_Code = 0x01 /* GENERAL_DISCOVERY */
– PHYs=LE_1M_PHY /* 1 Mbps PHY */
The two devices are discovered through the advertising report events notified with the
hci_le_extended_advertising_report_event() event callback.
Once the two devices are discovered, Master&Slave device starts two connection procedures (as central) to connect, respectively, to Slave_A and Slave_B devices:

/*
  Connect to Slave_A:Slave_A address type and address have been found during the discovery procedure through the Advertising Report events.
*/
ret = aci_gap_create_connection(LE_1M_PHY, "Slave_A address type", "Slave_A address");

/*
  Connect to Slave_B:Slave_B address type and address have been found during the discovery procedure through the Advertising Report events.
*/
ret = aci_gap_create_connection(LE_1M_PHY, "Slave_B address type", "Slave_B address");

Step 7. Once connected, Master&Slave device enables the characteristics notification, on both of them, using the aci_gatt_clt_write API. Slave_A and Slave_B devices start the characteristic notification by using the aci_gatt_srv_notify API.

Step 8. At this stage, Master&Slave device enters discovery mode (acting as peripheral). During initialization sequence, defines the advertising configuration data and advertising data with local name 'Test' = [0x08,0x74,0x65,0x73,0x74]. Master&Slave enters in discovery mode by enabling advertising as follows:

aci_gap_set_advertising_enable(ENABLE, 1, Advertising_Set_Parameters);

Since Master&Slave device also acts as a central device, it receives the notification event related to the characteristic values notified from, respectively, Slave_A and Slave_B devices.

Step 9. Once Master&Slave device enters discovery mode, it also waits for the connection request coming from the other Bluetooth LE device (called Master) configured as GAP central. Master device starts discovery procedure to discover the Master&Slave device after configuring the scan parameters by using the aci_gap_set_scan_configuration() API.
The general discovery procedure is started as follows:

ret = aci_gap_start_procedure(Procedure_Code = 0x01, PHYs = 0x01, Duration = 0; Period=0);

Step 10. Once the Master&Slave device is discovered, Master device starts a connection procedure to connect to it (after configuring the scan parameters using the aci_gap_set_scan_configuration() API).

/* Master device connects to Master&Slave device: Master&Slave address type and address have been found during the discovery procedure through the Advertising Report events */
ret = aci_gap_create_connection(Initiating_PHY = 0x01, Peer_Address_Type = "Master&Slave address type", Peer_Address = "Master&Slave address");

Master&Slave device is discovered through the advertising report events notified with the hci_le_extended_advertising_report_event() event callback.
Step 11. Once connected, Master device enables the characteristic notification on Master&Slave device using the `aci_gatt_clt_write` API.

Step 12. At this stage, Master&Slave device receives the characteristic notifications from both Slave_A, Slave_B devices, since it is a GAP central and, as GAP peripheral, it is also able to notify these characteristic values to the Master device.

3.11 Bluetooth Low Energy privacy 1.2

Bluetooth LE stack v3.x supports the Bluetooth low energy privacy 1.2. Privacy feature reduces the ability to track a specific Bluetooth LE by modifying the related Bluetooth LE address frequently. The frequently modified address is called the private address and the trusted devices are able to resolve it.

In order to use this feature, the devices involved in the communication need to be previously paired: the private address is created using the devices IRK exchanged during the previous pairing/bonding procedure.

There are two variants of the privacy feature:
1. Host-based privacy private addresses are resolved and generated by the host
2. Controller-based privacy private addresses are resolved and generated by the controller without involving the host after the Host provides the controller device identity information.

When controller privacy is supported, device filtering is possible since address resolution is performed in the controller (the peer's device identity address can be resolved prior to checking whether it is in the white list).

3.11.1 Controller-based privacy and the device filtering scenario

On Bluetooth LE stack v2.x, the `aci_gap_init()` API supports the following options for the `privacy_type` parameter:

- 0x00: privacy disabled
- 0x01: host privacy enabled
- 0x02: controller privacy enabled.

When a slave device wants to resolve a resolvable private address and be able to filter on private addresses for reconnection with bonded and trusted devices, it must perform the following steps:
1. Enable privacy controller on `aci_gap_init()`: use 0x02 as `privacy_type` parameter.
2. Connect, pair and bond with the candidate trusted device using one of the allowed security methods: the private address is created using the devices IRK.
3. Get the bonded device identity address and type using the `aci_gap_get_bonded_devices()` API.
4. Add the bonded device identity address and type to the Bluetooth LE device controller whitelist and to the list of address translations used to resolve resolvable private addresses in the controller, by using the `aci_gap_configure_white_and_resolving_list(0x01|0x02);` API.
5. The device configures the undirected connectable mode by calling the `aci_gap_set_advertising_configuration()` API with `Advertising_Filter_Policy = ADV_WHITE_LIST_FOR_ALL` (allow scan request from whitelist only, allow connect request from whitelist only).

Note: A set of test scripts allowing the described privacy controller and device filtering scenario to be executed, which are provided within the BlueNRG GUI SW package (see Section 5 References). These scripts can be run using the BlueNRG GUI and they can be taken as reference to implement a firmware application using the privacy controller and device filtering feature.

6. When a bonded master device performs a connection procedure for reconnection to the slave device, the slave device is able to resolve and filter the master address and connect with it.

3.11.2 Resolving addresses

After a reconnection with a bonded device, it is not strictly necessary to resolve the address of the peer device to encrypt the link. In fact, Bluetooth LE stack automatically finds the correct LTK to encrypt the link.

However, there are some cases where the peer's address must be resolved. When a resolvable privacy address is received by the device, it can be resolved by the host or by the controller (i.e. link layer).

Host-based privacy
If controller privacy is not enabled, a resolvable private address can be resolved by using `aci_gap_resolve_private_addr()`. The address is resolved if the corresponding IRK can be found among the stored IRKs of the bonded devices. A resolvable private address may be received when Bluetooth LE devices are in scanning, through `hci_le_extended_advertising_report_event()/hci_le_advertising_report_event()`, or when a connection is established, through `hci_le_extended_connection_complete_event()/hci_le_connection_complete_event()`.  

**Controller-based privacy**

If the resolution of addresses is enabled at link layer, a resolving list is used when a resolvable private address is received. To add a bonded device to the resolving list, the `aci_gap_configure_white_and_resolving_list()` has to be called. This function searches for the corresponding IRK and adds it to the resolving list.

When privacy is enabled, if a device has been added to the resolving list, its address is automatically resolved by the link layer and reported to the application without the need to explicitly call any other function. After a connection with a device, the `hci_le_enhanced_connection_complete_event()` is returned. This event reports the identity address of the device, if it has been successfully resolved.

When scanning, the `hci_le_extended_advertising_report_event()` contains the identity address of the device in advertising if that device uses a resolvable private address and its address is correctly resolved. In that case, the reported address type is 0x02 or 0x03. If no IRK can be found that can resolve the address, the resolvable private address is reported. If the advertiser uses directed advertisement, the resolved private address is reported through the `hci_le_extended_advertising_report_event()` or through the `hci_le_direct_advertising_report_event()` if it has been unmasked and the scanner filter policy is set to 0x02 or 0x03.

### 3.12 ATT_MTU and exchange MTU APIs, events

ATT_MTU is defined as the maximum size of any packet sent between a client and a server:

- **default ATT_MTU value:** 23 bytes

This determines the current maximum attribute value size when the user performs characteristic operations (notification/write max. size is ATT_MTU-3).

The client and server may exchange the maximum size of a packet that can be received using the exchange MTU request and response messages. Both devices use the minimum of these exchanged values for all further communications:

```c
#include "bluenrg_lp_api.h"

/* ble status */

typedef enum
{
    BLE_STATUS_SUCCESS = 0,
    BLE_STATUS_ERROR = 1
} tBleStatus;

/* Client */

tBleStatus aci_gatt_clt_exchange_config(uint16_t Connection_Handle);

/* Server */

// In response to an exchange MTU request, the aci_att_exchange_mtu_resp_event() callback is triggered on both devices:

void aci_att_exchange_mtu_resp_event(uint16_t Connection_Handle, uint16_t Server_RX_MTU);
```

**Server_RX_MTU** specifies the ATT_MTU value agreed between the server and client.

### 3.13 LE data packet length extension APIs and events

On Bluetooth LE specification v4.2, packet data unit (PDU) size has been increased from 27 to 251 bytes. This allows data rate to be increased by reducing the overhead (header, MIC) needed on a packet. As a consequence, it is possible to achieve: faster OTA FW upgrade operations, more efficiency due to less overhead.

The Bluetooth LE stack v3.x supports LE data packet length extension features and related APIs, events:

- **HCI LE APIs (API prototypes on bluenrg_lp_api.h)**
  - `hci_le_set_data_length()`
  - `hci_le_read_suggested_default_data_length()`
  - `hci_le_write_suggested_default_data_length()`
  - `hci_le_read_maximum_data_length()`

- **HCI LE events (events callbacks prototypes on bluenrg_lp_events.h)**
  - `hci_le_data_length_change_event()`
hci_le_set_data_length() API allows the user's application to suggest maximum transmission packet size (TxOctets) and maximum packet (TxTime) transmission time to be used for a given connection:

```c
uint16_t txOctets,
```

The supported TxOctets value is in the range [27-251] and the TxTime is provided as follows: (TxOctets + 14)*8. Once hci_le_set_data_length() API is performed after the device connection, if the connected peer device supports LE data packet length extension feature, the following event is raised on both devices:

```c
MaxTxOctets,
MaxTxTime,
MaxRxOctets,
MaxRxTime
```

This event notifies the host of a change to either the maximum link layer payload length or the maximum time of link layer data channel PDUs in either direction (TX and RX). The values reported (MaxTxOctets, MaxTxTime, MaxRxOctets, MaxRxTime) are the maximum values that are actually used on the connection following the change.

### 3.14 No packet retry feature

Bluetooth LE stack v3.x provides the capability to disable the standard Bluetooth LE link layer retransmission mechanism for characteristic notifications that are not acknowledged by the link layer of the peer device. This feature is supported only on notifications which are within the maximum allowed link layer packet length.

When standard Bluetooth Low Energy protocol is used, no packets can be lost, since an unlimited number of retries is applied by the protocol. In case of a weak link with lots of errors and retries, the time taken to deliver a certain number of packets can increase with the number of errors. If the "no packet retry feature" is applied, the corrupted packets are not retried by the protocol and, therefore, the time to deliver the selected number of packets is the same, but the number of lost packet moves from 0 to something proportional to the error rates. No packet retry feature can be enabled when a notification is sent by setting the parameter

```c
Flags = 0x01 on aci_gatt_srv_notify() API:
```

Refer to the aci_gatt_srv_notify() API description for detailed information about API usage and its parameter values.

### 3.15 Bluetooth LE radio activities and Flash operations

During Flash erase or write operations, execution from Flash could be stalled and so critical activities like radio interrupt may be delayed. This could lead to a loss of connection and/or incorrect radio behavior. It is worth noticing that Bluetooth LE v3.x implements a more flexible and robust radio activity scheduler which enhances the overall robustness against late interrupt routines.

In order to prevent this possible delay and impact on radio activities, Flash erase and write operations could be synchronized with the scheduled Bluetooth LE radio activities through the

```c
aci_hal_end_of_radio_activity_event() callback.
```

The aci_hal_end_of_radio_activity_event() callback is called when the device completes a radio activity and provides information when a new radio activity is performed. Provided information includes the type of radio activity and absolute time in system ticks when a new radio activity is scheduled. Application can use this information to schedule user activity synchronous to selected radio activities.

Let us assume a Bluetooth LE application starts advertising and it also performs write operations on Flash. The

```c
aci_hal_end_of_radio_activity_event() callback is used to register the Next_Advertising_SysTime
```

(1) time when next advertising event is programmed:
void aci_hal_end_of_radio_activity_event(uint8_t Last_State,
    uint8_t Next_State,
    uint32_t Next_State_SysTime)
{
    if (Next_State == 0x01) /* 0x01: Advertising */
    {
        /* Store time of next programmed advertising */
        Next_Advertising_SysTime = Next_State_SysTime;
    }
}

A FlashRoutine() performs the Flash write operation only if there is enough time for this operation before next scheduled radio activity.

### 3.16 Bluetooth LE 2 Mbps and Coded Phy

The following APIs allow the Host to specify its preferred values for the transmitter PHY and receiver PHY to be used for all subsequent connections over the LE transport.

```c
BleStatus hci_le_set_default_phy(uint8_t ALL_PHYS, uint8_t TX_PHYS, uint8_t RX_PHYS);
```

The following API allows PHY preferences to be set for the connection identified by the Connection_Handle.

```c
tBleStatus hci_le_set_phy(uint16_t Connection_Handle,
    uint8_t ALL_PHYS,
    uint8_t TX_PHYS,
    uint8_t RX_PHYS,
    uint16_t PHY_options);
```

Refer to APIs html documentation for detailed description about APIs and related parameters.

### 3.17 Bluetooth LE extended advertising/scanning

A set of HCI LE standard APIs for extended advertising/scanning is supported by Bluetooth LE stack v3.x. Refer to related html documentation for APIs and parameters description.

Furthermore new GAP APIs for configuring advertising modes and scanning procedures allow extended advertising/scanning feature to be supported.

An example about how to set advertising configuration to enable extended advertising is as follows:
/* Advertising_Handle used to identify an advertising set */
#define ADVERTISING_HANDLE 0
/* Type of advertising event that is being configured:
  0x0001: Connectable
  0x0002: Scannable
  0x0004: Directed
  0x0008: High Duty Cycle Directed Connectable
  0x0010: Legacy (if this bit is not set, extended advertising is used)
  0x0020: Anonymous
  0x0040: Include TX Power
  */
#define ADVERTISING_EVENT_PROPERTIES 0x01 /* Connectable advertising event */
/* PHY on which the advertising packets are transmitted */
#define ADV_PHY LE_1M_PHY
/* Advertising data: ADType flags + manufacturing data */
static uint8_t adv_data[] = {
  0x02, 0x01, 0x06, # ADType Flags for discoverability
  0x08, # Length of next AD data
  0xFF, /* Manufacturing data */
  0x53, 0x54, 0x4d, 0x69, 0x63, 0x72, 0x6f /* STMicro */
};
/* Set advertising configuration for extended advertising. */
ret = aci_gap_set_advertising_configuration(ADVERTISING_HANDLE,
  GAP_MODE_GENERAL_DISCOVERABLE, ADVERTISING_EVENT_PROPERTIES,
  160, 160,
  ADV_CH_ALL,
  0, NULL, /* No peer address */
  ADV_NO_WHITE_LIST_USE,
  0, /* 0 GHz */
  ADV_PHY, /* Primary advertising PHY */
  0, /* 0 skips */
  ADV_PHY, /* Secondary advertising PHY */
  0, /* 0 skips */
  0, /* 0 skips */
  0 /* No scan request notifications */
);
/* Set the advertising data */
ret = aci_gap_set_advertising_data(ADVERTISING_HANDLE, ADV_COMPLETE_DATA, sizeof(adv_data), adv_data);
/* Define advertising set (at least one advertising must be set) */
Advertising_Set_Parameters_t Advertising_Set_Parameters[1];
Advertising_Set_Parameters[0].Advertising_Handle = 0;
Advertising_Set_Parameters[0].Duration = 0;
Advertising_Set_Parameters[0].Max_Extended_Advertising_Events = 0;
/* Enable advertising */
ret = aci_gap_set_advertising_enable(ENABLE, 1, Advertising_Set_Parameters);
4 Bluetooth LE stack v3.x scheduler

On Bluetooth LE stack v3.x the concept of “anchor period” has been removed. Bluetooth LE stack v2.x “anchor period” is a periodic time interval where the periodic “master slots” (i.e. master connect events, scan events and advertise events) are allocated by the current device in such a way to systematically avoid collisions. Collisions can still occur between “master slots” and “slave slots” (i.e. slave connect events), since the periodicity of the slave slots is decided by the peer device (i.e. not under the control of the current device).

Use of “anchor period” has the advantage of minimizing the number of events skipped because of collisions (thus also increasing the performance in terms of network throughput), but it implies a large penalty in terms of flexibility, since “master slots” require to have a periodicity, which has to be an integer multiple of the current anchor period in order to be allocated (otherwise the start procedure command typically returns an error code).

If reduced flexibility in terms of time slot allocation is acceptable until Bluetooth Low Energy specification v4.2, it becomes not sustainable on Bluetooth Low Energy Specification v5.x, where many new asynchronous events have been introduced (due to the advertising extension feature), that could not be handled through the anchor period approach.

For this reason, on Bluetooth LE stack v3.x, a new scheduler has been introduced where all time slots are treated as complete asynchronous slots (thus removing all the constraints in terms of time slots periodicity with respect to the anchor period).

The removal of the concept of the “anchor period” increases the flexibility for time slots allocation, since the requested slot periodicity never causes a start procedure to be rejected, but, it also implies a penalty in terms of probability of collision occurrence (and potentially in terms of overall throughput performance). With the new scheduler, whenever a collision occurs, a priority mechanism is implemented which makes only one colliding time slot to be served at a time. Priority is dynamically assigned by the new scheduler based on a “fairness” principle (i.e. all events must be served), and in such a way that supervision timeout should never be exceeded, thus causing a connection to be dropped.
## References

<table>
<thead>
<tr>
<th>Name</th>
<th>Title/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth specifications</td>
<td>Specification of the Bluetooth system (v4.0, v4.1, v4.2, v5.x)</td>
</tr>
<tr>
<td>STSW-BNRGUI</td>
<td>BlueNRG GUI SW package</td>
</tr>
</tbody>
</table>
This section lists the standard acronyms and abbreviations used throughout the document.

### Table 70. List of acronyms

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI</td>
<td>Application command interface</td>
</tr>
<tr>
<td>ATT</td>
<td>Attribute protocol</td>
</tr>
<tr>
<td>Bluetooth LE</td>
<td>Bluetooth low energy</td>
</tr>
<tr>
<td>BR</td>
<td>Basic rate</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy check</td>
</tr>
<tr>
<td>CSRK</td>
<td>Connection signature resolving key</td>
</tr>
<tr>
<td>EDR</td>
<td>Enhanced data rate</td>
</tr>
<tr>
<td>DK</td>
<td>Development kits</td>
</tr>
<tr>
<td>EXTI</td>
<td>External interrupt</td>
</tr>
<tr>
<td>GAP</td>
<td>Generic access profile</td>
</tr>
<tr>
<td>GATT</td>
<td>Generic attribute profile</td>
</tr>
<tr>
<td>GFSK</td>
<td>Gaussian frequency shift keying</td>
</tr>
<tr>
<td>HCI</td>
<td>Host controller interface</td>
</tr>
<tr>
<td>IFR</td>
<td>Information register</td>
</tr>
<tr>
<td>IRK</td>
<td>Identity resolving key</td>
</tr>
<tr>
<td>ISM</td>
<td>Industrial, scientific and medical</td>
</tr>
<tr>
<td>LE</td>
<td>Low energy</td>
</tr>
<tr>
<td>L2CAP</td>
<td>Logical link control adaptation layer protocol</td>
</tr>
<tr>
<td>L2CAP-COS</td>
<td>Logical link control adaptation layer protocol - connection oriented services</td>
</tr>
<tr>
<td>LTK</td>
<td>Long-term key</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller unit</td>
</tr>
<tr>
<td>MITM</td>
<td>Man-in-the-middle</td>
</tr>
<tr>
<td>NA</td>
<td>Not applicable</td>
</tr>
<tr>
<td>NESN</td>
<td>Next sequence number</td>
</tr>
<tr>
<td>OOB</td>
<td>Out-of-band</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol data unit</td>
</tr>
<tr>
<td>RF</td>
<td>Radio frequency</td>
</tr>
<tr>
<td>RSSI</td>
<td>Received signal strength indicator</td>
</tr>
<tr>
<td>SIG</td>
<td>Special interest group</td>
</tr>
<tr>
<td>SM</td>
<td>Security manager</td>
</tr>
<tr>
<td>SN</td>
<td>Sequence number</td>
</tr>
<tr>
<td>USB</td>
<td>Universal serial bus</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally unique identifier</td>
</tr>
<tr>
<td>WPAN</td>
<td>Wireless personal area networks</td>
</tr>
</tbody>
</table>
## Revision history

Table 71. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-Aug-2020</td>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
## Contents

1 Bluetooth Low Energy technology ................................................. 2
   1.1 Bluetooth LE stack architecture ............................................. 2
   1.2 Physical layer ........................................................................ 3
      1.2.1 LE 2M and LE Coded physical layers ................................. 4
      1.2.2 LE 2M PHY ................................................................ 4
      1.2.3 LE Coded PHY .............................................................. 5
   1.3 Link layer (LL) ...................................................................... 5
      1.3.1 Bluetooth LE packets ....................................................... 6
      1.3.2 Extended advertising ..................................................... 8
      1.3.3 Advertising sets ........................................................... 9
      1.3.4 Advertising state .......................................................... 10
      1.3.5 Scanning state ............................................................ 10
      1.3.6 Connection state .......................................................... 11
   1.4 Host controller interface (HCI) ..................................................... 12
   1.5 Logical link control and adaptation layer protocol (L2CAP) ................. 12
      1.5.1 LE L2CAP connection-oriented channels .......................... 12
   1.6 Attribute protocol (ATT) .......................................................... 12
   1.7 Security manager (SM) ............................................................. 13
   1.8 Privacy .................................................................................. 16
      1.8.1 The device filtering ........................................................ 16
   1.9 Generic attribute profile (GATT) .................................................. 17
      1.9.1 Characteristic attribute type ............................................. 17
      1.9.2 Characteristic descriptor type ......................................... 18
      1.9.3 Service attribute type .................................................... 19
      1.9.4 GATT procedures ....................................................... 19
   1.10 Generic access profile (GAP) ..................................................... 20
   1.11 Bluetooth LE profiles and applications ........................................... 23
      1.11.1 Proximity profile example .............................................. 24
2 Bluetooth Low Energy (LE) stack v3.x ............................................. 25
   2.1 Bluetooth LE stack library framework ........................................ 27
3 Design an application using the Bluetooth LE stack v3.x ........................................... 32

3.1 Initialization phase and main application loop ................................................................. 32
  3.1.1 Bluetooth LE addresses ................................................................................................. 35
  3.1.2 Set tx power level ........................................................................................................ 37

3.2 Bluetooth LE stack v3.x GATT interface ........................................................................ 37
  3.2.1 Bluetooth LE stack v3.x vs Bluetooth LE stack v2.x ..................................................... 37
  3.2.2 GATT server ................................................................................................................. 37
  3.2.3 SoC vs. network coprocessor ....................................................................................... 51
  3.2.4 GATT client .................................................................................................................. 52
  3.2.5 Services and characteristic configuration ..................................................................... 53

3.3 GAP API interface ............................................................................................................. 55
  3.3.1 Set the discoverable mode and use the direct connection establishment procedure ....... 56
  3.3.2 Set discoverable mode and use general discovery procedure (active scan) ................. 59

3.4 Bluetooth LE stack events and event callbacks ............................................................. 63

3.5 Security (pairing and bonding) ......................................................................................... 66

3.6 Service and characteristic discovery ............................................................................... 68

3.7 Characteristic discovery procedures and related GATT events ......................................... 71

3.8 Characteristic notification/indications, write, read .......................................................... 72

3.9 Basic/typical error condition description .......................................................................... 74

3.10 Simultaneously master, slave scenario .......................................................................... 74

3.11 Bluetooth Low Energy privacy 1.2 .................................................................................. 78
  3.11.1 Controller-based privacy and the device filtering scenario ........................................ 78
  3.11.2 Resolving addresses .................................................................................................... 78

3.12 ATT_MTU and exchange MTU APIs, events ............................................................... 79

3.13 LE data packet length extension APIs and events ...................................................... 79

3.14 No packet retry feature ................................................................................................. 80

3.15 Bluetooth LE radio activities and Flash operations .................................................... 80
3.16 Bluetooth LE 2 Mbps and Coded Phy.................................................. 81
3.17 Bluetooth LE extended advertising/scanning....................................... 81
4 Bluetooth LE stack v3.x scheduler ......................................................... 83
5 References ....................................................................................... 84
6 List of acronyms and abbreviations ..................................................... 85
Revision history .................................................................................... 86
List of tables

Table 1. Bluetooth LE RF channel types and frequencies ................................................. 4
Table 2. LE PHY key parameters ................................................................................. 5
Table 3. Advertising data header content ...................................................................... 7
Table 4. Advertising packet types .................................................................................. 7
Table 5. Advertising event type and allowable responses .............................................. 8
Table 6. Data packet header content .......................................................................... 8
Table 7. Packet length field and valid values ................................................................. 8
Table 8. Connection request timing intervals ................................................................ 11
Table 9. Attribute example ......................................................................................... 12
Table 10. Attribute protocol messages ......................................................................... 13
Table 11. Combination of input/output capabilities on a Bluetooth LE device ............... 14
Table 12. Methods used to calculate the temporary key (TK) ....................................... 14
Table 13. Mapping of IO capabilities to possible key generation methods ................... 15
Table 14. Characteristic declaration ......................................................................... 18
Table 15. Characteristic value ...................................................................................... 18
Table 16. Service declaration ...................................................................................... 19
Table 17. Include declaration ...................................................................................... 19
Table 18. Discovery procedures and related response events ......................................... 20
Table 19. Client-initiated procedures and related response events .............................. 20
Table 20. Server-initiated procedures and related response events ............................... 20
Table 21. GAP roles ................................................................................................... 20
Table 22. GAP broadcaster mode ................................................................................ 21
Table 23. GAP discoverable modes ......................................................................... 21
Table 24. GAP connectable modes ............................................................................ 21
Table 25. GAP bondable modes ............................................................................... 22
Table 26. GAP observer procedure .......................................................................... 22
Table 27. GAP discovery procedures ....................................................................... 22
Table 28. GAP connection procedures .................................................................... 22
Table 29. GAP bonding procedures .......................................................................... 23
Table 30. Bluetooth LE stack library framework interface ......................................... 27
Table 31. Bluetooth LE application stack library framework interface ...................... 28
Table 32. Bluetooth LE stack v3.x initialization parameters ....................................... 29
Table 33. Application configuration preprocessor options .......................................... 30
Table 34. User application defines for Bluetooth LE device roles ................................ 32
Table 35. GATT, GAP service handles ...................................................................... 34
Table 36. GATT, GAP characteristic handles ............................................................ 34
Table 37. aci_gap_init() role parameter values ............................................................ 35
Table 38. GATT server database APIs ...................................................................... 42
Table 39. Example database ...................................................................................... 43
Table 40. aci_gatt_srv_notify parameters ................................................................. 45
Table 41. aci_gatt_srv_read_event parameters ........................................................... 46
Table 42. aci_gatt_srv_write_event parameters ......................................................... 46
Table 43. aci_att_srv_prepare_write_req_event parameters ....................................... 46
Table 44. aci_att_srv_exec_write_req_event parameters ............................................ 47
Table 45. aci_gatt_srv_resp parameters ..................................................................... 47
Table 46. ATT_pwrq_init parameters ...................................................................... 50
Table 47. ATT_pwrq_flush parameter ..................................................................... 50
Table 48. ATT_pwrq_read parameters ..................................................................... 51
Table 49. ATT_pwrq_pop parameters ..................................................................... 51
Table 50. ATT_pwrq_push parameters ..................................................................... 51
Table 51. GATT client APIs .................................................................................... 52
Table 52. aci_gap_set_advertising_configuration() API : discoverable mode and advertising type selection . 55
| Table 53. | aci_gap_start_procedure() API | 55 |
| Table 54. | aci_gap_terminate_proc() API | 56 |
| Table 55. | ADV_IND event type: main fields | 62 |
| Table 56. | ADV_IND advertising data: main fields | 62 |
| Table 57. | SCAN_RSP event type | 63 |
| Table 58. | Scan response data | 63 |
| Table 59. | Bluetooth LE stack: main event callbacks | 63 |
| Table 60. | Bluetooth LE sensor profile demo services and characteristic handle | 68 |
| Table 61. | Service discovery procedures APIs | 69 |
| Table 62. | First read by group type response event callback parameters | 70 |
| Table 63. | Second read by group type response event callback parameters | 70 |
| Table 64. | Third read by group type response event callback parameters | 70 |
| Table 65. | Characteristics discovery procedures APIs | 71 |
| Table 66. | First read by type response event callback parameters | 72 |
| Table 67. | Second read by type response event callback parameters | 72 |
| Table 68. | Characteristic update, read, write APIs | 73 |
| Table 69. | References | 84 |
| Table 70. | List of acronyms | 85 |
| Table 71. | Document revision history | 86 |
**List of figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Bluetooth Low Energy technology enabled coin cell battery devices</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Bluetooth Low Energy stack architecture</td>
<td>3</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Link layer state machine</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Packet structure</td>
<td>6</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Packet structure with LE data packet length extension feature</td>
<td>7</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Bluetooth LE 5.x extended advertising</td>
<td>9</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Advertising packet chain</td>
<td>9</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Advertising packet with AD type flags</td>
<td>10</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Example of characteristic definition</td>
<td>18</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Client and server profiles</td>
<td>24</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Bluetooth LE stack v3.x architecture</td>
<td>25</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Bluetooth LE stack reference application</td>
<td>26</td>
</tr>
<tr>
<td>Figure 13</td>
<td>MAC address storage</td>
<td>36</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Bluetooth LE simultaneous master and slave scenario</td>
<td>75</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2020 STMicroelectronics – All rights reserved