PM0269

Programming manual

Bluetooth® Low Energy stack v3.x programming guidelines

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® Low Energy stack v3.x family APIs and related event callbacks.

The document describes the Bluetooth® Low Energy stack v3.x 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® Low Energy (Bluetooth® LE) stack v3.x family:

- BlueNRG-LP device
- BlueNRG-LPS device (Bluetooth® Low Energy stack v3.1 or later)

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

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

For more information about the supported devices and the Bluetooth® Low Energy specifications, refer to Section 1.1 References at the end of this document.

The manual is structured as follows:

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

This document applies to the BlueNRG-LP, BlueNRG-LPS MCUs, which are Arm®-based devices.

For information on Bluetooth®, refer to the www.bluetooth.com website.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.1 References

Table 1. Reference documents

Reference	Description
Bluetooth specifications	Specification of the Bluetooth system (v5.x)
DS13282	BlueNRG-LP datasheet
STSW-BNRGUI	BlueNRG GUI SW package
STSW-BNRGLP-DK	BlueNRG-LP, BlueNRG-LPS SW package
DS13819	BlueNRG-LPS datasheet

1.2 List of acronyms and abbreviations

This section lists the standard acronyms and abbreviations used throughout the document.

Table 2. List of acronyms

Term	Meaning
ACI	Application command interface
AoA	Angle of arrival
AoD	Angle of departure
ATT	Attribute protocol
Bluetooth® LE	Bluetooth® Low Energy
BIS	Broadcast Isochronous Streams
BIG	Broadcast Isochronous Groups
BR	Basic rate
CIG	Connected Isochronous Group
CIS	Connected Isochronous Stream
CRC	Cyclic redundancy check
CSRK	Connection signature resolving key
CTE	Constant tone extension
EATT	Enhanced ATT
EDR	Enhanced data rate
DK	Development kits
EXTI	External interrupt
GAP	Generic access profile

Term	Meaning
GATT	Generic attribute profile
GFSK	Gaussian frequency shift keying
HCI	Host controller interface
IFR	Information register
IRK	Identity resolving key
ISOAL	Isochronous Adaptation Layer
ISM	Industrial, scientific, and medical
LE	Low energy
L2CAP	Logical link control adaptation layer protocol
L2CAP-COS	Logical link control adaptation layer protocol - connection oriented services
LTK	Long-term key
MCU	Microcontroller unit
MITM	Man-in-the-middle
NA	Not applicable
NESN	Next sequence number
OOB	Out-of-band
PDU	Protocol data unit
RF	Radio frequency
RSSI	Received signal strength indicator
SIG	Special interest group
SM	Security manager
SN	Sequence number
SW	Software
USB	Universal serial bus
UUID	Universally unique identifier
WPAN	Wireless personal area networks

2 Bluetooth® Low Energy technology

The Bluetooth® Low Energy 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 standardBluetooth® enabled devices.

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

2.1 Bluetooth® LE stack architecture

Bluetooth® LE technology has been formally adopted by the Bluetooth® core specification version 4.0 (Section 1.1 References). This version of the Bluetooth® standard supports two systems of wireless technology:

- Basic rate
- Bluetooth® LE

The Bluetooth® LE 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® LE technology are:

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

In particular, Bluetooth® LE 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® LE stack consists of two components:

- Controller
- Host

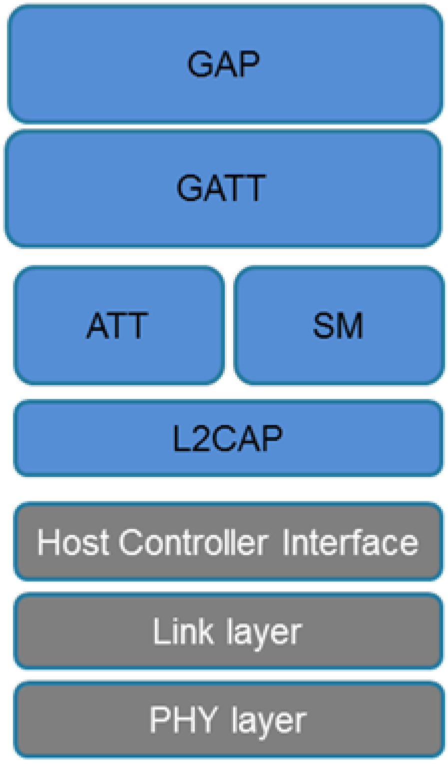
The controller includes the physical layer and the link layer.

Host includes the logical link control and adaptation protocol (L2CAP), the security manager (SM), the attribute protocol (ATT), the 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® LE stack architecture**



2.2 Physical layer

The LE 1M 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® Low Energy system uses 40 RF channels (0-39), with 2 MHz spacing. These RF channels have frequencies centered at:

$$2402 + k * 2MHz, \ where \ k = 0, \ldots, 39 \quad (1)$$

RF channels can be divided into two groups:

1. Primary advertising channels that use three fixed RF channels (37, 38 and 39) for:
    a. legacy advertising and connection
    b. initial packet of extended advertising
2. General purpose channels that use all other RF channels.

Table 3. **Bluetooth® Low Energy LE RF channel types and frequencies**

Channel index	RF center frequency	Channel type
37	2402 MHz	Primary Advertising

Channel index	RF center frequency	Channel type
0	2404 MHz	General Purpose
1	2406 MHz	General Purpose
….	….	General Purpose
10	2424 MHz	General Purpose
38	2426 MHz	Primary Advertising
11	2428 MHz	General Purpose
12	2430 MHz	General Purpose
….	….	General Purpose
36	2478 MHz	General Purpose
39	2480 MHz	Primary Advertising

Bluetooth® Low Energy 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 nonadaptive 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.

2.2.1 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® LE specifications v4.x:

- LE 2M
- LE Coded

Standard HCI APIs are defined on Bluetooth® Low Energy 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® Low Energy specification v5.x is still mandatory.*

2.2.2 LE 2M PHY

Main characteristics:

- 2 Msym/s modulation
- Uncoded

There are several application use cases demanding a higher throughput:

- Over-the-air firmware upgrade procedure
- Sports, fitness, medical applications use cases require to collect a significant 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.

2.2.3 LE Coded PHY

Main characteristics:

- 1 Msym/s modulation
  - Same as LE 1M
- Payload can be coded with two different rates:
  - 500 kb/s (S = 2)
  - 125 kb/s (S = 8)

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 four 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 two further steps to the bit stream processing:

1. FEC encoding, which generates two further bits for each bit
2. Pattern mapper, which converts each bit from the previous step in P symbols depending on two coding schemes:

- S= 2: no change is done. This doubles the range (approximately)
- S= 8: each bit is mapped to 4 bits. This leads to a quadruple range (approximately)

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 4. LE PHY key parameters

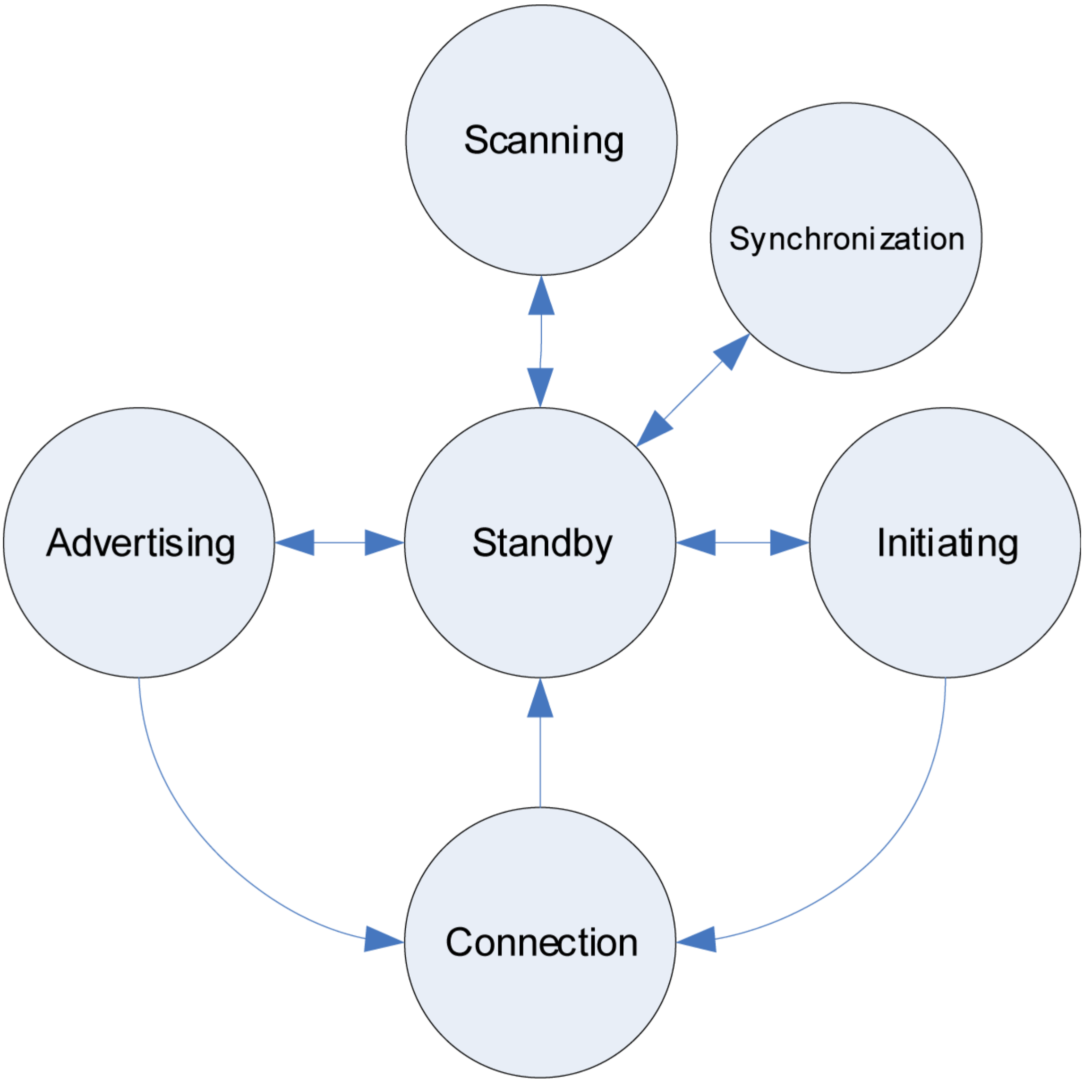
	LE 1M	LE 2M	LE Coded (S=2)	LE Coded (S=8)
Symbol rate	1 Ms/s	2 Ms/s	1 Ms/s	1 Ms/s
Data rate	1 Mbps	2 Mbps	500 Kbps	125 Kbps
Error detection	3 bytes CRC	3 bytes CRC	3 bytes CRC	3 bytes CRC
Error correction	No	No	FEC	FEC
Range increase	1	0.8	2	4
Bluetooth® LE specification 5.x requirement type	Mandatory	Optional	Optional	Optional

2.3 Link layer (LL)

The link layer (LL) defines how two devices can use a radio to transmitt information between each other.

The link layer defines a state machine with five states:

Figure 3. **LL state machine**



- 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: if this state is entered from an initiating state, the device is in master role. If connection state is entered from advertising state, the device is in slave role. Master communicates with slave and defines the timing of transmissions.
- Synchronization: the device listens to the periodic advertising.

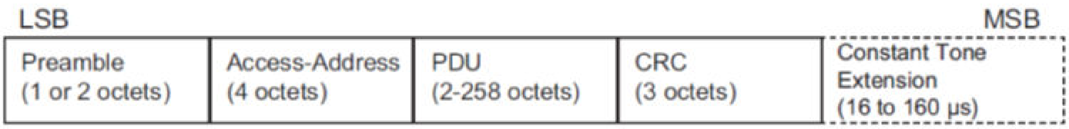
2.3.1 Bluetooth® Low Energy packet

The Bluetooth® Low Energy packet format is different for uncoded and coded PHYs.

The format for uncoded PHYs (LE 1M and LE 2M) is shown in Figure 4. 2MB .

The Bluetooth® Low Energy data packet structure is described below.

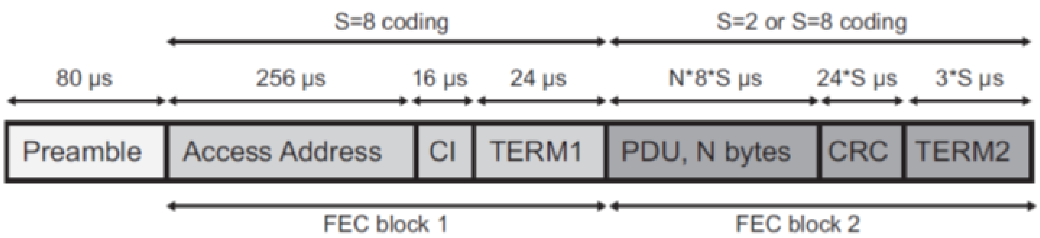
Figure 4. 2MB



- Preamble: start of the packet. 1 octet for LE 1M PHY, 2 octets for LE 2M PHY.
- Access address: 32 bits. It is different for each connection.
- PDU: it can be an advertising physical channel PDU or a data physical channel PDU.
- CRC: 24 bits of control code calculated over the PDU.
- Constant tone extension: optional. It consists of a constantly modulated series of unwhitened 1s.

The packet format for LE Coded PHY is shown in Figure 5. Coded PHY.

Figure 5. Coded PHY

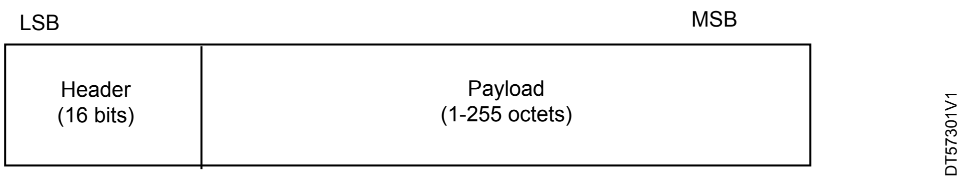


- Preamble: 80 symbols, not coded
- Access address: 32 bits. It is different for each connection
- CI: coding indicator. To select between 2 kinds of coded configurations: S=2 or S=8
- PDU: it can be an advertising physical channel PDU or a data physical channel PDU
- CRC: 24 bits of control code calculated over the PDU
- TERM1 and TERM2: termination sequences of FEC blocks to bring encoder to its original state.

The content of the PDU depends on the type of packet: advertising physical channel PDU or data physical channel PDU.

The format of advertising physical channel PDU is shown in Figure 6. Advertising physical channel PDU

Figure 6. Advertising physical channel PDU



- The header is 16 bits and contains information on the PDU type and the length of the payload
- The payload filed is specific to the PDU type.

The PDU type contained in the advertising header can be one of types described in the Table 5. PDU advertising header.

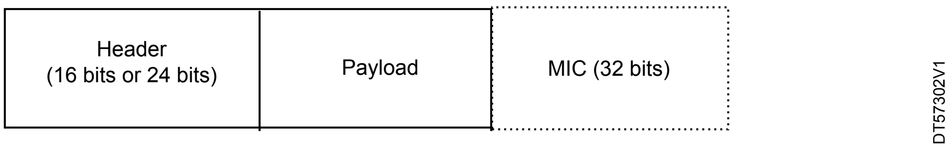
Table 5. PDU advertising header

Packet Type	Packet Name	Description	Advertising Physical Channel	Permitted PHY			
				LE 1M	LE 2M	LE Coded	Notes
0000b	ADV_IND	Connectable undirected advertising	Primary	X	-	-	Used by an advertiser when it wants another device to connect to it. The device can be scanned by a scanning device, or enter connection state as a slave on reception of a connection request.
0001b	ADV_DIRECT_IND	Connectable directed advertising	Primary	X	-	-	Used by an advertiser when it wants to connect to a particular device. The ADV_DIRECT_IND packet contains advertiser's address and initiator address only.
0010b	ADV_NONCONN_IND	Nonconnectable undirected advertising	Primary	X	-	-	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.
0011b	SCAN_REQ	Scan request	Primary	X	-	-	Used by a device in scanning state to request additional information from the advertiser on primary channel using legacy PDUs.
	AUX_SCAN_REQ	Auxiliary scan request	Secondary	X	X	X	Used by a scanner to request additional info on secondary advertising physical channels.
0100b	SCAN_RSP	Scan response	Primary	X	-	-	Used by an advertiser device to provide additional information to a scan device after reception of a SCAN_REQ.
0101b	CONNECT_IND	Connection request	Primary	X	-	-	Sent by an initiating device to a device in connectable/discoverable mode on primary advertising channel.
	AUX_CONNECT_REQ	Auxiliary connection request	Secondary	X	X	X	Sent by an initiating device to a device in connectable/discoverable mode on secondary advertising channel.
0110b	ADV_SCAN_IND	Scannable undirected advertising	Primary	X	-	-	Used by an advertiser which wants to allow a scanner to require more information from it. The device cannot connect, but it is discoverable to advertise data and scans response data.
0111b	ADV_EXT_IND	Extended Advertising PDU	Primary	X	-	X	Extended advertising packet which usually refers to AUX_ADV_IND packets on secondary advertising channel.
	AUX_ADV_IND	Auxiliary advertising	Secondary	X	X	X	Packet containing advertising information on secondary advertising channel.

Packet Type	Packet Name	Description	Advertising Physical Channel	Permitted PHY			
				LE 1M	LE 2M	LE Coded	Notes
0111b	AUX_SCAN_RSP	Auxiliary scan response	Secondary	X	X	X	Sent by an advertiser upon reception of an AUX_SCAN_REQ.
	AUX_SYNC_IND	Auxiliary sync PDU	Periodic	X	X	X	Packet used in periodic advertising.
	AUX_CHAIN_IND	Auxiliary chaining PDU	Secondary and Periodic	X	X	X	PDU used to add additional data.
1000b	AUX_CONNECT_RSP	Auxiliary connection response	Secondary	X	X	X	PDU used to respond to an AUX_CONNECT_REQ

Data physical channel PDUs has the format shown in Figure 7. Data physical channel PDUs.

Figure 7. Data physical channel PDUs



- The header contains the PDU type, the length of the payload and other information like sequence numbers. It is 16-bit long but can be increased to 24 bits if CTE info is included (i.e. if constant tone extension for direction finding is used)
- The payload can be up to 251 octets
- The MIC is included in encrypted link layer connection if payload length is not zero.

2.3.2 Extended advertising

On Bluetooth® LE 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® LE v5.x advertising extension capability allows:

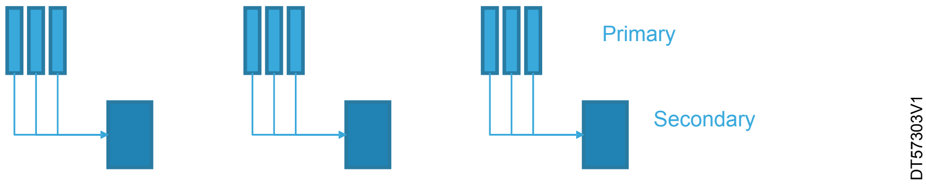
- To extend the data length in connectionless scenarios (that is, beacon)
- to have multiple sets of advertising data to be sent
- to have advertising sent in a deterministic way

New extended advertising packets can use the Bluetooth® LE 4.x connection channels (0-36) for extending advertising payload up to 255 bytes. Initial advertising and legacy PDUs are transmitted on 3 RF channels (37, 38, 39), known as “primary advertising physical channel”. 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).

Most of the communication is made on 37 RF channels, the “secondary advertising physical channel”.

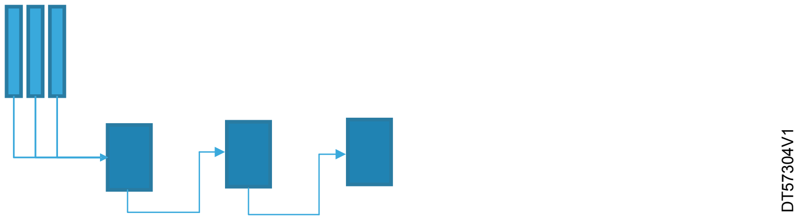
ADV_EXT_IND new packet is used and it that can be sent on the primary adv phy channel. If needed, it contains the pointer to an auxiliary packet on the secondary adv phy channel: most of the info is on the auxiliary packet.

Figure 8. **Bluetooth® LE 5.x extended advertising**



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. Consequently, each chain in the advertising packet chain can be sent on a different secondary channel.

Figure 9. **Advertising packet chain**



A direct advantage of the 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 three primary advertising channels, reducing the overall duty cycle.

Some restrictions are applied to allowed channels:

- Legacy advertising PDUs and new ADV_EXT_IND are allowed on the primary adv phy channel only
- New advertising PDUs (except for ADV_EXT_IND) are allowed on the secondary adv phy channel only.

Some restrictions are also applied to allowed PHYs:

- Legacy advertising PDUs can only use LE 1M PHY
- New advertising PDUs can use one of the three PHYs (LE 1M, LE 2M, LE Coded), but LE 2M is not allowed on a primary adv phy channel. So, ADV_EXT_IND can only be sent on LE 1M and LE Coded.

Any combination of PHYs is possible between primary and secondary channels. However, some combination may not have a real use case.

Common use cases are: LE 1M -> LE 1M, LE 1M -> LE 2M, LE Coded -> LE Coded

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

2.3.3 Advertising sets

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

It does not provide enough flexibility to interleave different advertising data with a power-efficient mechanism.

Bluetooth® Low Energy v5.x defines advertising sets having an ID used to indicate which set each packet belongs to.

Each set has an ID (SID, Set ID) used to indicate which set each packet belongs to.

- The SID is an identifier of the set that is sent over-the-air
- Since an advertiser can have more than one advertising set, this number can be used by the scanner to distinguish between different advertising sets from the same advertiser
- This SID is used (together with the DID, that is, Data ID) for duplicate filtering

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 of scheduling and transmitting the adverting sets defined by the host layer.

2.3.4 Advertising state

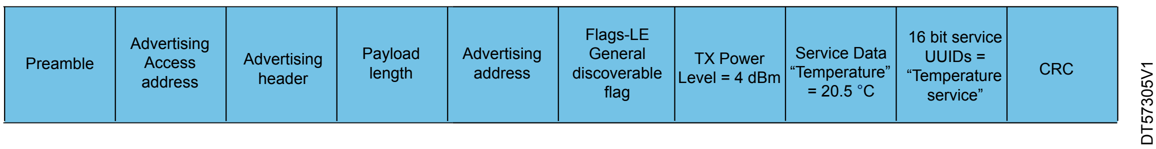
Advertising states allow the link layer to transmit advertising packets and also to respond with scan responses to scan requests coming from those devices, which are actively scanning.

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. These three packet sequences are 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 an 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 10. Advertising packet with AD type flags



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® Low Energy)
- 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 nonzero (it is not included in a 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 an 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.

2.3.5 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

2.3.6 Connection state

When data to be transmitted are more complex than those allowed by advertising data or a bidirectional reliable communication between the 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 handles 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 the 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 3. Bluetooth® Low Energy LE RF channel types and frequencies:

Table 6. Connection request timing intervals

Parameter	Min.	Max.	Note
Transmit window size	1.25 milliseconds	10 milliseconds	-
Transmit window offset	0	Connection interval	Multiples of 1.25 milliseconds
Connection interval	7.5 milliseconds	4 seconds	Multiples of 1.25 milliseconds
Supervision timeout	100 milliseconds	32 seconds	Multiples of 10 milliseconds

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).

2.3.6.1 *Extended scan*

Since extended advertising uses new packets and new PHYs, these changes are reflected on scan procedures.

Scanning on the primary channel is possible using LE 1M, to find:

- Legacy events
- Extended adverting events, possibly switching to other PHYs on secondary advertising physical channel

Scanning on the primary channel is possible using LE Coded, to find:

- Extended advertising events, possibly switching to other PHYs on the secondary advertising physical channel.

2.3.7 Periodic advertising and periodic advertising sync transfer

Bluetooth® specification v5.0 defines periodic advertising that uses deterministic scheduling to allow a device to synchronize its scanning with the advertising of another device.

A new synchronization mode is defined by the generic access profile, which allows the periodic advertising synchronization establishment procedure to be performed and to synchronize with the advertising.

Periodic advertisements use a new link layer PDU called AUX_SYNC_IND. The required information (timing and timing offset) needed to synchronize with the periodic advertising packets is sent to a field, called SyncInfo, included in AUX_ADV_IND PDUs.

The periodic advertising synchronization procedure has a cost in terms of energy and some devices could not be in the conditions to perform this procedure.

A new periodic advertising sync transfer (PAST) procedure has been defined in order to allow a device A, which receives periodic advertising packets from device B, to pass the acquired synchronization information to another device C, which is connected to the device A. As consequence, the device C is able to receive the periodic advertising packets directly from device B without the need to scan for AUX_ADV_IND PDUs, which would consume too much energy.

2.3.8 Randomized advertising

Bluetooth® Low Energy uses three primary advertising channels. The PDUs sent onto these channels form an advertising event.

In order to reduce the possible packet collisions, the time between two consecutive advertising events must have a random delay from 0 to 10 ms.

While in BBluetooth® core specification v5.0 advertising uses the three primary advertising channels in strict order (that is, 37, 38, 39) starting from Bluetooth® v5.1 this order is no longer required. This means that the PDUs may not be sent to a fixed order and now it is even possible to randomize the sequence of used primary advertising channels. It has been proven that a random selection of the advertising channel further reduces the probability of packet collisions, with significative benefits on a crowded network.

2.3.9 Bluetooth® Low Energy power control

The Bluetooth® Low Energy power control feature allows to dynamically update the transmission power level on a specific connection.

This feature provides a way to reduce the overall power consumption on the transmitter device by dynamically changing the transmitting power level within the current connection.

Further the LE power control feature allows the receiver device to dynamically monitor the signal strength within a connection, and to request a power level change to the transmitter device in order to have the required signal strength keeping the requested range and optimize the power consumption.

The capability to tune the transmitter power level to the optimal level also improves the coexistence with all the other 2.4 GHz wireless networks.

2.4 Host controller interface (HCI)

The host controller interface (HCI) layer provides a communication between the host and the 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.

2.5 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.

2.5.1 LE L2CAP connection-oriented channels

L2CAP connection-oriented channels provide support for 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.

2.6 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 7. **Attribute example**

Attribute handle	Attribute type	Attribute value	Attribute permissions
0x0008	“Temperature UUID”	“Temperature value”	“Read only, no authorization, no authentication”

- “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 changes
- 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 4. LE PHY key parameters:

Table 8. Attribute protocol messages

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

2.7 Security manager (SM)

TheBluetooth® 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 the 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 three methods already available with LE legacy pairing

The paring 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 9. **Combination of input/output capabilities on a Bluetooth® LE device**

	No output	Display
No input	No input, no output	Display only
Yes/No	No input, no output	Display yes/no
Keyboard	Keyboard only	Keyboard display

LE legacy pairing

LE legacy pairing algorithm uses and generates two 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 is 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 (for example, 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 10. **Methods used to calculate the temporary key (TK)**

	Display only	Display yes/no	Keyboard only	No input, no output	Keyboard display
Display only	*Just Works*	*Just Works*	*Passkey Entry*	*Just Works*	*Passkey Entry*
Display Yes/No	*Just Works*	*Just Works*	*Passkey Entry*	Just Works	*Passkey Entry*
Keyboard only	*Passkey Entry*	*Passkey Entry*	*Passkey Entry*	*Just Works*	*Passkey Entry*
No input No output	*Just Works*	*Just Works*	*Just Works*	*Just Works*	*Just Works*
Keyboard display	*Passkey Entry*	*Passkey Entry*	*Passkey Entry*	*Just Works*	*Passkey Entry*

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 is 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 receiving device replies with its public key. 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 OOB 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 11. Mapping of IO capabilities to possible key generation methods

Initiator/ responder	Display only	Display yes/no	Keyboard only	No input no output	Keyboard display
Display only	*Just Works*	*Just Works*	*Passkey Entry*	*Just Works*	*Passkey Entry*
Display yes/no	*Just Works*	*Just Works* (LE legacy) Numeric comparison (LE secure connections)	*Passkey Entry*	*Just Works*	*Passkey Entry* (LE legacy) Numeric comparison (LE secure connections)
Keyboard only	*Passkey Entry*	*Passkey Entry*	*Passkey Entry*	*Just Works*	*Passkey Entry*
No input no output	*Just Works*	*Just Works*	*Just Works*	*Just Works*	*Just Works*
Keyboard display	*Passkey Entry*	*Passkey Entry* (LE legacy) Numeric comparison (LE secure connections)	*Passkey Entry*	*Just Works*	*Passkey Entry* (LE legacy) Numeric comparison (LE secure connections)

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).

2.8 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® Low Energy 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 should only 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.

2.8.1 The device filtering

Bluetooth® LE reduces the number of responses from the devices in order to diminish the power consumption, since this implies fewer transmissions and fewer 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.

2.9 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 a 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® Low Energy 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)

2.9.1 Characteristic attribute type

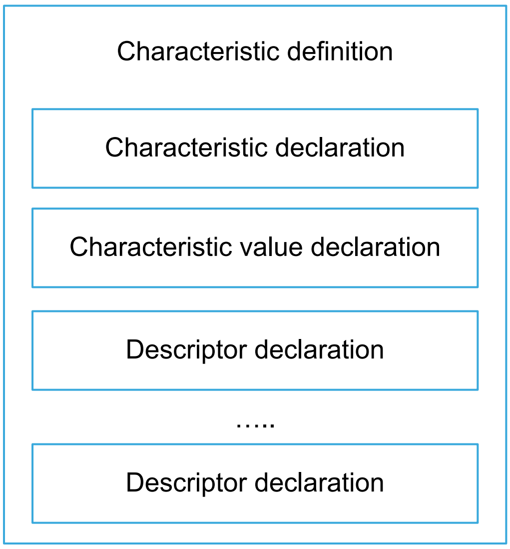
A characteristic is an attribute type, which contains a single value and any number of descriptors describing the characteristic value that may make it understandable for the user.

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)

Figure 11. **Example of characteristic definition**



DT57306V1

A characteristic declaration is an attribute defined as follows:

Table 12. **Characteristic declaration**

Attribute handle	Attribute type	Attribute value	Attribute permissions
0xNNNN	0x2803 (UUID for characteristic attribute type)	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	Read only, no authentication, no authorization
		Characteristic value attribute handle	
		Characteristic value UUID (16 or 128 bits)	

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

Table 13. **Characteristic value**

Attribute handle	Attribute type	Attribute value	Attribute permissions
0xNNNN	0xuuuu–16 bits or 128 bits for characteristic UUID	Characteristic value	Higher layer profile or implementation specific

2.9.2 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 display the related value (example temperature measurement value in ^{o}C format)
6. Characteristic aggregation format: it allows several characteristic presentation fthe ormats to be aggregated

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

2.9.3 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 14. Service declaration

Attribute handle	Attribute type	Attribute value	Attribute permissions
0xNNNN	0x2800–UUID for “Primary Service” or 0x2801–UUID for “Secondary Service”	0xuuuu–16 bits or 128 bits for service UUID	Read only, no authentication, no authorization

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 15. Include declaration

Attribute handle	Attribute type	Attribute value			Attribute permissions
0xNNNN	0x2802 (UUID for include attribute type)	Include service attribute handle	End group handle	Service UUID	Read only, no authentication, no authorization

“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.

2.9.4 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 16. Discovery procedures and related response events)
- Client-initiated procedures (Table 17. Client-initiated procedures and related response events)
- Server-initiated procedures (Table 18. Server-initiated procedures and related response events)

Table 16. **Discovery procedures and related response events**

Procedure	Response events
Discovery all primary services	Read by group response
Discovery primary service-by-service UUID	Find by type value response
Find included services	Read by type response event
Discovery all characteristics of a service	Read by type response
Discovery characteristics by UUID	Read by type response
Discovery all characteristic descriptors	Find information response

Table 17. **Client-initiated procedures and related response events**

Procedure	Response events
Read characteristic value	Read response event
Read characteristic value by UUID	Read response event
Read-long characteristic value	Read blob response events
Read-multiple characteristic values	Read response event
Write characteristic value without response	No event is generated
Signed write without response	No event is generated
Write characteristic value	Write response event
Write long characteristic value	Prepare write response Execute write response
Reliable write	Prepare write response Execute write response

Table 18. **Server-initiated procedures and related response events**

Procedure	Response events
Notifications	No event is generated
Indications	Confirmation event

For a detailed description of the GATT procedures and related response events, refer to the Bluetooth® specifications in Section 1.1 References.

2.9.5 GATT caching

The Bluetooth® LE service and characteristic discovery procedures allow a GATT client to know all the GATT server services and characteristics stored on the server attributes database table and to use them through ATT procedures (read, write, etc.). These procedures are time and power consuming.

Some devices often do not change their attributes table in terms of new services and characteristics, but they only change the characteristic and descriptor values.

Some other devices could add new services/characteristics during their life.

The only way to inform a GATT client about a possible change on the GATT server attributes database table is the service change indication characteristic, which allows a GATT server to send a specific indication to a bonded connected GATT client, which, in its turns, can perform a Service/Characteristic discovery to get the updated table.

This mechanism does not provide any synchronization between client and server and it could lead to a situation where a GATT client could start ATT procedures on the server attribute database before receiving the service change indication from the server.

For connected and not bonded devices, the only safe mechanism to be aligned to the possible change of the GATT server attribute table is to perform a time and power consuming service discovery procedure at each connection.

Bluetooth® specifications v5.1 defines two new characteristics:

1. Database hash
2. Client-supported features

These characteristics allow a client to understand if something has been changed on the server GATT attributes table, even if there is no bonding between the two devices.

The GATT client updates a flag on the client-supported features characteristic on the server to inform the server that it supports the database hash characteristic.

The database hash characteristic stores a 128-bit value, which is an AES-CMAC hash calculated from the server attribute table. Any change in the server GATT attribute database structure results in a different hash value. The server has the responsibility to keep the database hash value always up to date.

The database hash characteristic allows the client to ask the server if something has been changed on its attributes database: each time a GATT client connects to a server, it performs immediately a read of this characteristic in order to establish if a change or not has been performed on the server attribute table.

The GATT client may cache the read database hash value to verify if a change on the database structure occurred since the last connection. This allows the client to perform the service and characteristic discovery procedures only if a change has occurred since the last discovery, enhancing the user experience in terms of timing and saving power.

In addition to these two characteristics, the concept of robust caching has been defined in order to synchronize client and server views of the attributes table. Basically, when client tries to read the GATT server attributes table before receiving the service change indication, the client may get inconsistent data if the GATT server attributes table content has changed. In this case, the server can send a new ATT error code (database out-of-sync) to inform the client of this inconsistency.

When robust caching is enabled, a client can be in two states, from the server point of view:

1. Change-aware state
2. Change-unaware state

After connection, the state of a client without a trusted relationship is change-aware. Instead, if the client is bonded, the state is the same as on the previous connection, unless the database has changed. In this case, the client is considered change-unaware.

Server ignores all the ATT commands from a client that are change-unaware and if it receives an ATT request it sends an ATT error response with an error code set to the database out of sync.

Some events can change the state of the client to change-aware:

1. Server receives an ATT confirmation for a service changed indication it has previously sent
2. Server sends the database out of sync ATT error response to an ATT request from client and then receives another ATT request from the client.

2.10 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 19. GAP roles

Role(1)	Description	Transmitter	Receiver	Typical example
Broadcaster	Sends advertising events	M	O	Temperature sensor, which sends temperature values
Observer	Receives advertising events	O	M	Temperature display, which just receives and displays temperature values
Peripheral	Always a slave. It is on connectable advertising mode. Supports all LL control procedures; encryption is optional	M	M	Watch
Central	Always a master. It never advertises. It supports active or passive scan. It supports all LL control procedures; encryption is optional	M	M	Mobile phone

1. 1. M = mandatory; O = optional

On a 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 20. GAP broadcaster mode

Mode	Description	Notes	GAP role
Broadcast mode	Device only broadcasts data using the link layer advertising channels and packets (it does not set any bit on flags AD type)	Broadcast data can be detected by a device using the observation procedure	Broadcaster

Table 21. GAP discoverable modes

Mode	Description	Notes	GAP role
Non-discoverable mode	It cannot set the limited and general discoverable bits on flags AD type	It cannot be discovered by a device performing a general or limited discovery procedure	Peripheral
Limited discoverable mode	It sets the limited discoverable bit on flags AD type	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	Peripheral
General discoverable mode	It sets the general discoverable bit on flags AD type	It is used when a device wants to be discoverable. There is no limit on the discoverability time	Peripheral

Table 22. GAP connectable modes

Mode	Description	Notes	GAP role
Non-connectable mode	It can only use ADV_NONCONN_IND or ADV_SCAN_IND advertising packets	It cannot use a connectable advertising packet when it advertises	Peripheral
Direct connectable mode	It uses the ADV_DIRECT advertising packet	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	Peripheral
Undirected connectable mode	It uses the ADV_IND advertising packet	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	Peripheral

Table 23. GAP bondable modes

Mode	Description	Notes	GAP role
Non-bondable mode	It does not allow a bond to be created with a peer device	No keys are stored from the device	Peripheral
Bondable mode	Device accepts bonding request from a central device	-	Peripheral

The following GAP procedures are defined in Section 2.3.1 Bluetooth® Low Energy packet:

Table 24. GAP observer procedure

Procedure	Description	Notes	Role
Observation procedure	It allows a device to look for broadcaster device data	-	Observer

Table 25. GAP discovery procedures

Procedure	Description	Notes	Role
Limited discoverable procedure	It is used for discovery of peripheral devices in limited discovery mode	Device filtering is applied based on flag AD type information	Central
General discoverable procedure	It is used for discovery of peripheral devices in general and limited discovery mode	Device filtering is applied based on flag AD type information	Central
Name discovery procedure	It is the procedure to retrieve the “Bluetooth® Device Name” from connectable devices	-	Central

Table 26. GAP connection procedures

Procedure	Description	Notes	Role
Auto connection establishment procedure	Allows connection with one or more devices in the directed connectable mode or the undirected connectable mode	It uses whitelists	Central

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

Table 27. **GAP bonding procedures**

Procedure	Description	Notes	Role
Bonding procedure	Starts the pairing process with the bonding bit set on the pairing request	-	Central

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

2.11 Direction finding

Classic location applications (proximity, beacons...) using Bluetooth® LE are based on the simple concept of received signal strength (RSSI) measurements. This establishes if two devices are in the range of each other and estimates the related distance.

Bluetooth® specifications v5.1 define a new feature, which allows the direction of a received Bluetooth® LE packet to be identified with a high degree of accuracy.

There are two methods:

1. Angle of arrival (AoA)
2. Angle of departure (AoD).

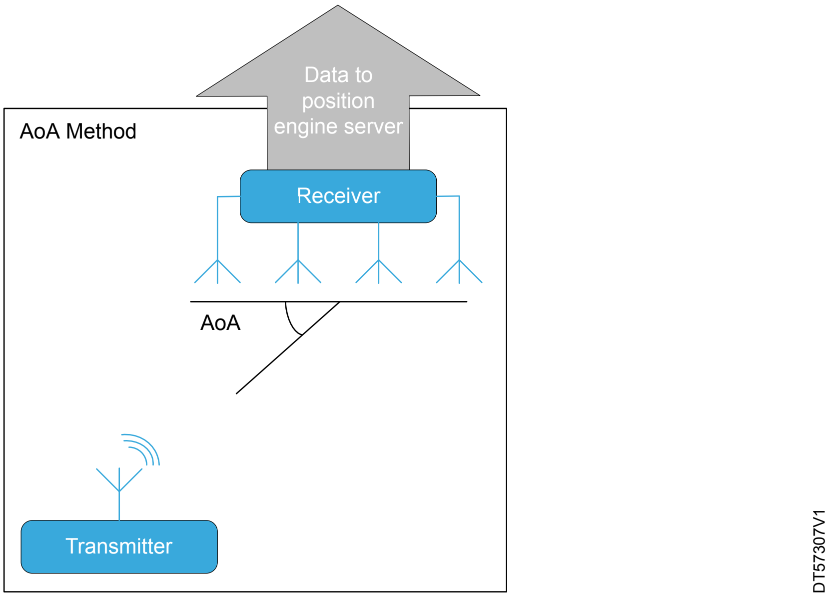
Both methods require one of the two communicating devices to have an array of multiple antennas (on receiving device with AoA, on transmitting device with AoD).

2.11.1 Direction finding with angle of arrival (AoA)

In AoA method, a transmitter (tag) device to which direction is being determined, sends a direction-finding signal using a single antenna. The receiver device (locator) has multiple array antennas, through which it detects a signal phase difference due to the difference in distance from each of the antenna in the array to the transmitting antenna. The receiver device takes IQ samples data of the signal while switching between the active antenna in the array. The receiver device is able to calculate the relative signal direction by applying specific algorithms to the IQ samples data.

This feature allows several application scenarios to be addressed as real-time asset tracking.

Figure 12. Angle of arrival (AoA)



On a receiver (locator) device, an antenna array is needed, and the angle calculation of several tags requires a consistent processing power. The transmitter (tag) device requires a single antenna, and it must only support the capability to send the information required to perform IQ data sampling.

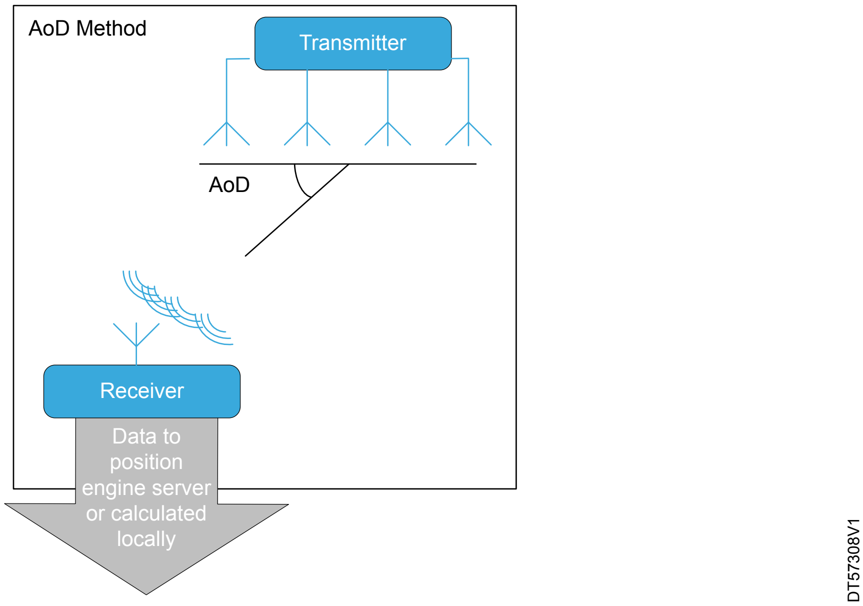
2.11.2 Direction finding with angle of departure (AoD)

In the AoD method, a transmitter device (typically an anchor device) to which direction is being determined, sends a special direction-finding signal using an antenna array. The receiver device (for example, a mobile phone) has a single antenna.

As the multiple signals from the transmitting device cross the array antenna in the receiver device, the latter takes the IQ samples. The receiver device is able to calculate the relative signal direction by applying specific algorithms to the IQ samples.

This feature allows several application scenarios to be addressed as an indoor positioning system.

Figure 13. Angle of departure (AoD)



The transmitter device needs to send a multi-antenna location signal using an antenna array. It must support the capability to send the information required to perform an IQ data sampling.

The receiver device needs a single antenna and the angle calculation of the received signal may require an FPU (float processing unit) and some memory resources.

2.11.3 In-phase and quadrature (IQ)

Direction finding uses in-phase and quadrature (IQ) sampling to measure the phase of radio waves incident upon an antenna at a specific time.

In the AoA approach, the sampling procedure is done on each antenna in the array, one at a time, and in a specific sequence depending on the design of the array.

The sampled data is then provided to the host through the host controller interface (HCI). The IQ sample should then be used with some specific algorithms to calculate the angle of the incident wave. These algorithms are not defined on the Bluetooth® LE core specification.

The IQ sampling process has involved some changes at a link layer (LL) and the host controller interface (HCI) level. At link layer level, a new field called constant tone extension is added to the LL packet. This field provides a constant frequency signal against which IQ sampling can be performed. This field includes a sequence of 1s, and it is not subjected to the whitening process and is not used in the CRC calculation.

Constant tone extension can be used in both connectionless and connection-oriented scenarios:

1. connectionless use: the periodic advertising feature is needed (since deterministic timing in the sampling process is important) and CTE is appended to AUX_SYNC_IND PDUs.
2. connection-oriented use: new PDUs LL_CTE_REQ and LL_CTE_RSP are defined.

2.12 Enhanced Attribute protocol

On ATT protocol, a sequential transaction model is used. Once a ATT transaction is started, it must be completed with a response or confirm packet. Otherwise, a timeout is generated after 30 seconds.

Furthermore, the MTU related to the L2CAP layer has a one to one correspondence with the MTU value related to the ATT layer. As consequence, the L2CAP cannot interleave packets coming from different applications and different ATT packets.

ATT protocol over Bluetooth® Low Energy connections is also based on the L2CAP Basic mode, which has no flow control. As consequence, the ATT transactions cannot be considered as a reliable mechanism: not acknowledged packets, such as notifications, may get lost on the receiving side, without no evidence.

Another limitation is linked to the ATT MTU size (which can negotiated through the `ATT_Exchange_MTU` request) and response PDUs, as soon as 2 devices connect with no capability to change the established value. This prevents that a second application from requesting an increased ATT MTU size on the same connection.

Enhanced Attribute protocol (EATT) is an improvement of the Attribute protocol (ATT) which provides the following main new features:

- Concurrent transactions of L2CAP packets related to ATT packets coming from different applications are possible over distinct enhanced ATT bearers, which are based on a new L2CAP mode called L2CAP Enhanced Credit Based Flow Control Mode providing flow control and reliable communication.
- Capability to change the ATT Maximum Transmission Unit (MTU) during a connection (the MTU values at ATT and L2CAP layer can be independently configured by reducing latency problems on scenarios where some applications share the Bluetooth® Low Energy stack with other applications).

The new features in the EATT + L2CAP Enhanced Credit Based Flow Control Mode determine a consistent improvement in terms of latency, when different applications use the Bluetooth® Low Energy stack at the same time. Since L2CAP packets can be interleaved with each other, this reduces the case where application usage of the stack would prevent usage of other application. This implies a benefit on latency and therefore on user experience.

Some mechanisms are available to know if a connected device supports the new EATT feature:

- A new characteristic Server Supported Feature is defined. It must be included if the device supports the EATT feature. A client device can just read this characteristic value to discover if the EATT is supported (bit 0 of the first octet of the characteristic value equal to 1 means that EATT is supported).
- The Client Supported Features characteristic bit 1 indicates whether or not the Enhanced ATT Bearer is supported by the client. Bit 2 indicates whether or not a new ATT packet called Multiple Handle Value Notifications is supported.

2.13 L2CAP enhanced credit-based flow control

The new credit-based flow control method has been defined as follows:

- When a data transfer must start, the transmitter device can obtain the receiver capacity: the transmitter device can get the number of PDUs that the receiver device could obtain without losing any data.
- The transmitter device has a counter, which is initialized to the receiver capacity.
- The counter value is decreased when a PDU is sent from the transmitter to the receiver device.
- When the value reaches 0, the transmitter device stops sending PDUs until the receiver device reads and handle some of the received PDUs.
- The receiver device sends back the number of read PDUs. The counter is set to this value and the transmitter device could restart the PDUs sending.

The enhanced attribute protocol uses the L2CAP enhanced credit-based flow control mode. It also allows the ATT MTU size to be dynamically modified. It can be used only over an encrypted connection.

Two types of L2CAP channels are available:

- Enhanced ATT Bearer based on the enhanced credit-based flow control method used by EATT
- Unenhanced ATT Bearer, which is used by ATT

2.14 Bluetooth® Low Energy isochronous channels

The Bluetooth® Low Energy isochronous channels feature has been defined in order to send time-bound data to one or more devices for allowing time-synchronized processing.

The Bluetooth® Low Energy isochronous channels feature allows multiple sink devices receiving data from a specific source device, rendering it simultaneously. Data has a time-limited validity period, after which data is declared as expired. Expired data that has not yet been transmitted is discarded. The receiver devices only obtain valid data, according to age and latency rules.

A new Bluetooth® Low Energy isochronous physical channel has been defined in order to support both connection-oriented or connectionless (broadcast scenarios towards several devices) isochronous channels.

This new physical channel uses frequency hopping to set the timing of the first packet, which then determines the anchor point for the timing of next packets.

The new Bluetooth® Low Energy isochronous physical channel is supported on all Bluetooth® Low Energy PHYs.

Connection-oriented isochronous channels use the Connected Isochronous Stream (CIS) logical transport and support bidirectional communication.

A CIS stream defines a point-to point isochronous communication between two connected devices.

A flushing period is defined for the CIS logical transport. Any packet that has not been transmitted within the flushing period is discarded.

CIS streams are members of groups named Connected Isochronous Groups (CIG), each of which may contain multiple CIS instances.

Connectionless isochronous channels use Broadcast Isochronous Streams (BIS) and only support uni-directional communication. A BIS can stream identical copies of data to multiple receiver devices. BIS streams are members of groups named Broadcast Isochronous Groups (BIG), each of which may contain multiple BIS instances.

The capability of the Bluetooth® Low Energy isochronous channels has a wide range of application use cases, in particular on audio sharing domains:

- Personal audio sharing groups of friends: sharing simultaneously music on one smartphone through Bluetooth®headphones.
- Public assisted hearing: broadcasting a dialogue to a specific audience.
- Public television: listening to the television through Bluetooth® Low Energy headphones.
- Multilanguage flight announcements: getting specific information through the preferred language via Bluetooth® Low Energy headphones.

2.15 Bluetooth® Low Energy connection subrating

Some application scenarios, like audio applications, require the capability to switch quickly from a low duty cycle connection to a high duty cycle connection. The Bluetooth® Low Energy connection subrating capability addresses this scenario.

The connection subrating feature makes possible to indicate that only a specific subset of connection events has to be actively used by the connected devices. The radio is not used on all the other connection events. As a consequence, the connection subrating mechanism provides both a connection interval, which establishes the connection events rate, and an effective connection interval, which establishes how often the connection intervals are actually used when the subrating parameters are used.

Connection subrating is also based on continuation events: when a subrated connection is actively in use (nonzero length link layer packet), a specific number of next standard connection events can still be selected. The subrating feature does not impact the application data exchange.

2.16 Bluetooth® Low Energy channel classification

The central device performs the channel classification procedure on the Bluetooth® Low Energy specification up to version 5.2.

On Bluetooth® Low Energy version 5.3, the channel classification procedure is performed by both the peripheral device and the central device:

- The peripheral device can now report its channel classifications to the connected central device.
- The central device can use such data to update the channel map used on adaptive frequency hopping (AFH).

As consequence, the central device uses channels in the AFH procedure that are appropriate for both devices, improving communication reliability and throughput.

2.17 Bluetooth® Low Energy 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

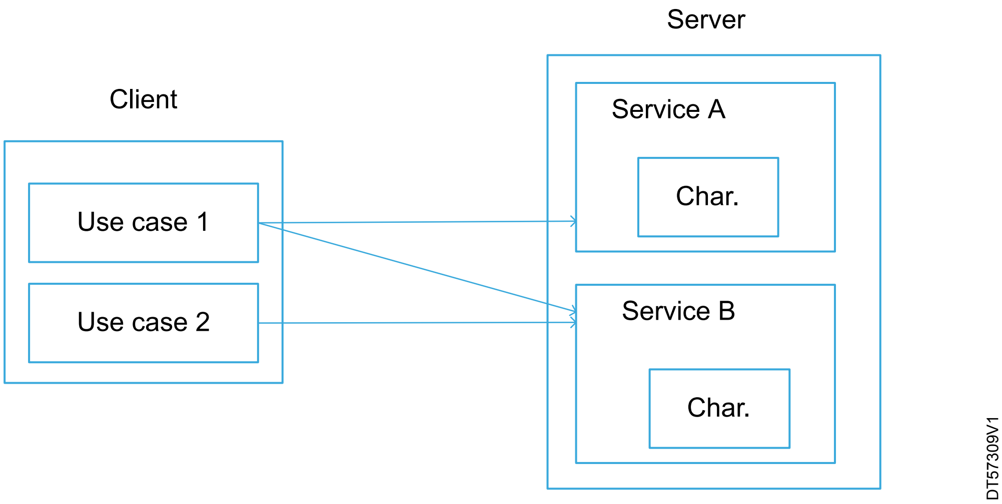
Standard profiles or proprietary profiles can be used. When using a non-standard profile, a 128-bit UUID is required and must be generated randomly.

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

- https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx

- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx

Figure 14. **Client and server profiles**



2.17.1 Proximity profile example

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

Target

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

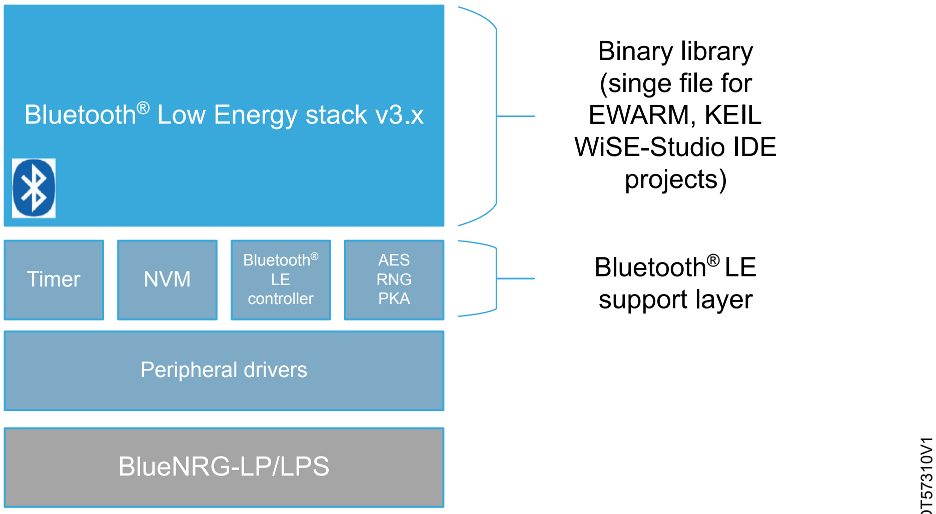
How it works

- 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 counted
- «Immediate Alert» service
  - «Alert Level» characteristic
  - Behavior: when written, causes alert as counted
- «Tx Power» service
  - «Tx Power» characteristic
  - Behavior: when read, reports current Tx power for connection.

3 Bluetooth® Low Energy stack v3.x

Bluetooth® Low Energy stack v3.x provides a new architecture, which offers further advantages rather than the previous Bluetooth® Low Energy stack v2.x family.

Figure 15. Bluetooth® Low Energy stack v3.x architecture



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® Low Energy stack (Init API and tick API to be called on user main application)
   – New NVM module external to Bluetooth® Low Energy 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 (for example, 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 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® Low Energy binary library provides the following functionalities:

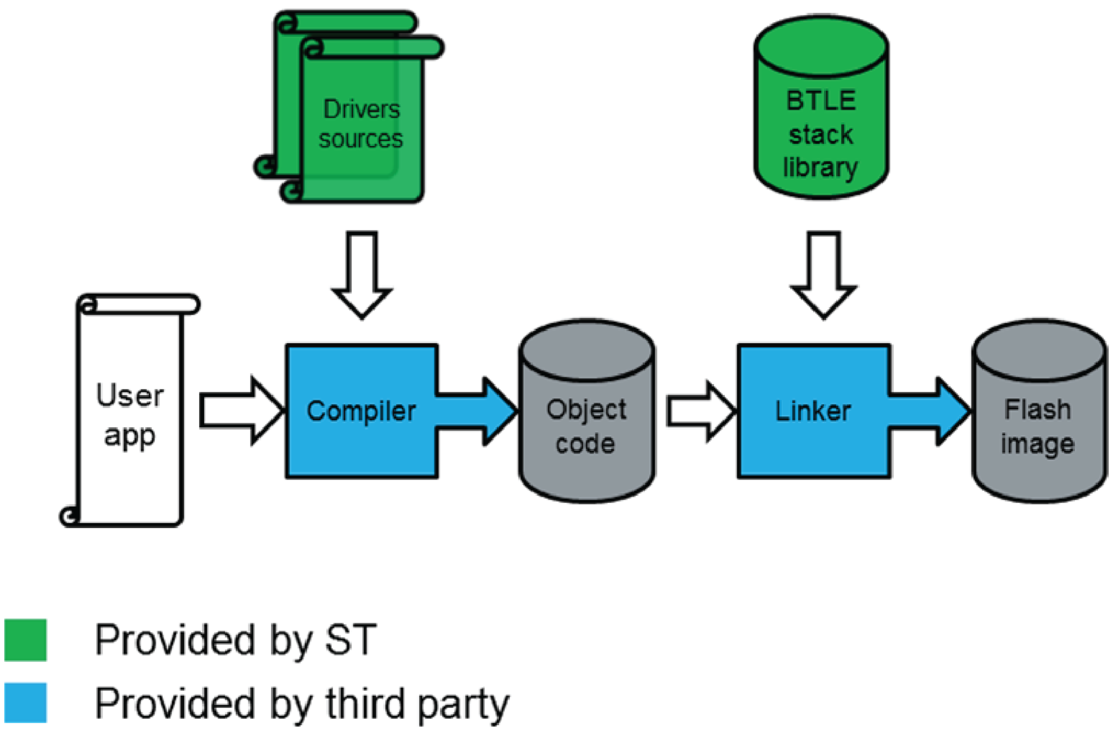
- Stack APIs for
  – Bluetooth® Low Energy stack initialization
  – Bluetooth® Low Energy stack application command interface (HCI command prefixed with hci_, and vendor-specific command prefixed with aci_)
  – Sleep timer access
  – Bluetooth® Low Energy stack state machine handling

- Stack event callbacks
  - Inform user application aboutBluetooth® Low Energy stack events
  - Sleep timer events
- Interrupt handler for radio IP

In order to get access to the Bluetooth® Low Energy stack functionalities, a user application is just requested to:

- Call the related stack APIs
- Handle the expected events through the provided stack callbacks
- Link the Bluetooth® Low Energy stack binary library to the user application, as described in Figure 16. Bluetooth® LE stack reference application.

Figure 16. **Bluetooth® LE stack reference application**



Note:

1. *API is a C function defined by the Bluetooth® Low Energy stack library and called by the user application.*
2. *A callback is a C function called by the Bluetooth® Low Energy 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® Low Energy device peripherals (ADC, I^2C, SPI, timers, watchdog, GPIOs, RTC, UART, ...).*
4. *Bluetooth® Low Energy radio initialization structure and related application configuration files have been modified if compared to the Bluetooth® Low Energy stack v2.x family*
5. *Bluetooth® Low Energy stack APIx naming for Bluetooth® Low Energy stack init, Bluetooth® Low Energy stack tick has been modified if compared to Bluetooth® Low Energy 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® Low Energy stack v2.x family*
7. *New GATT APIs/event framework and interface have been defined if compared to Bluetooth® Low Energy stack v2.x family*

3.1 Bluetooth® Low Energy stack library framework

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

The Bluetooth® Low Energy stack APIs use 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® Low Energy API interface framework is provided within the supported ST Bluetooth® Low Energy devices DK software package targeting the related DK platforms (refer to Section 1.1 References).

The Bluetooth® Low Energy stack library framework interface for the BlueNRG-LP/BlueNRG-LPS devices is defined by the following header files:

Table 28. Bluetooth® Low Energy stack library framework interface (BlueNRG-LP, BlueNRG-LPS STSW-BNRGLP-DK)

File	Description	Location	Notes
ble_status.h	Header file for Bluetooth® Low Energy stack error codes	Middlewares\ST\Bluetooth_LE\inc	-
bluenrg_lp_api.h	Header file for Bluetooth® Low Energy stack APIs	"	-
bluenrg_lp_events.h	Header file for Bluetooth® Low Energy stack events callbacks	"	-
bluenrg_lp_gatt.h	Header file for the Bluetooth® Low Energy GATT	"	It provides new GATT structures definition
stack_user_cfg.h	Bluetooth® Low Energy stack configuration header file	"	It provides the available configuration options for Bluetooth® Low Energy stack v3.x
stack_user_cfg.c	Bluetooth® Low Energystack configuration file	Middlewares\ST\Bluetooth_LE\src	Source file to be included on user application IDE project in order to support the Bluetooth® Low Energy modular approach available with Bluetooth® Low Energy stack v3.x

Note: *Bluetooth® Low Energy stack v3.x provides the capability to enable/disable, at compile time, the following Bluetooth® Low Energy stack features based on a user-specific application scenario:*

1. Enable/disable controller privacy
2. Enable/disable LE secure connections
3. Enable/disable master capability
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)

8. Enable/disable constant tone extension (where applicable; from Bluetooth® Low Energy v3.1 or later)
9. Enable/disable LE Power Control and Path Loss Monitoring (from Bluetooth® Low Energy v3.1 or later)
10. Enable/disable the connection capability (from Bluetooth® Low Energy stack v3.1a or later). It configures support to connections:
    – If connection option is disabled, connections are not supported; the device is a broadcaster only if master capability option is disabled, or a broadcaster and observer only if master capability option is enabled.
    – If connection option is enabled, connections are supported; device can only act as broadcaster or peripheral if the master capability option is disabled, or any role (broadcaster, observer, peripheral, and central) if master capability option is enabled.
11. Enable/disable the enhanced ATT (from Bluetooth® Low Energy v3.2 or later)
12. Enable/disable the connection subrating (from Bluetooth® Low Energy v3.2 or later)
13. Enable/disable the channel classification (from Bluetooth® Low Energy v3.2 or later)
14. Enable/disable the broadcast isochronous streams (from Bluetooth® Low Energy v3.2 or later)
15. Enable/disable the connected isochronous streams (from Bluetooth® Low Energy v3.2 or later)

From Bluetooth® Low Energy stack v3.1a or later, the master capability option is linked to the new connection option as follows:

1. Observer disabled (master capability option disabled) or enabled (master capability option enabled) if connection option is disabled
2. Observer and Central disabled (master capability option disabled) or enabled (master capability option enabled) if connection option is enabled.

The modular configuration options allow the user to exclude some features from the available Bluetooth® Low Energy stack binary library and decrease the overall flash memory.

The following table provides the list of modular options to be added or not in order to address some typical Bluetooth® Low Energy configurations:

1. Broadcaster only: send advertising PDU and scan response PDU; receive and process scan request PDU.
2. Broadcaster + observer only: send advertising PDU, scan request PDU and scan response PDU; receive and process advertising PDU, scan request PDU and scan response PDU.
3. Basic: all modular options OFF expects the capability to connect as a peripheral device (connection capability enabled).
4. Basic + DLE: all modular options OFF expect with the connection and the data length extension capabilities.
5. Full: all the modular options ON.

Table 29. Modular configurations option combination examples

-	Broadcaster only (1)	Broadcaster and observer only(1) (2)	Basic	Basic + DLE	Full
Controller Privacy	NO	NO	NO	NO	YES
Secure Connections	NO	NO	NO	NO	YES
Central/Observer capability	NO	YES	NO	NO	YES
Controller Data Length Extension	NO	NO	NO	YES	YES
Controller 2M, coded PHY	NO	NO	NO	NO	YES
Controller extended advertising	NO	NO	NO	NO	YES
L2CP Cos	NO	NO	NO	NO	YES
Controller Periodic Advertising	NO	NO	NO	NO	YES
Controller CTE (Direction Finding)	NO	NO	NO	NO	YES
Controller LE power control	NO	NO	NO	NO	YES
Connection (3)(4)	NO	NO	YES	YES	YES
Channel classification	NO	NO	NO	NO	YES
Connection subrating	NO	NO	NO	NO	YES
Enhanced ATT (EATT)	NO	NO	NO	NO	YES
Broadcast Isochronous Stream (BIS)	NO	NO	NO	NO	YES
Connected Isochronous Stream (CIS)	NO	NO	NO	NO	YES

1. *Broadcaster only or broadcaster + observer-only roles could be also extended with other modular options and related features except for the following ones:*
   - *Secure Connections*
   - *L2CAP COS*
   - *Power Control*
2. *Observer-only role is not supported.*
3. *It is supported on Bluetooth® LE stack v3.1a or later. This implies that broadcaster only (connection capability = OFF) or broadcaster + observer only (connection capability = OFF and master capability = ON) and possible extensions could be supported only from Bluetooth® LE stack v3.1a or later.*
4. *ACI_GAP_INIT(), Role parameter should be set according to the connection support and master role modular options (using the available parameter values or combination of such values: broadcaster, peripheral, observer, central).*

Some predefined configurations are available on the BlueNRG-LP, BlueNRG-LPS STSW-BNRGLP-DK development framework, by selecting some preprocessor options:

- BLE_STACK_FULL_CONF: all the described features are enabled
- BLE_STACK_BASIC_CONF: all the described features except the connection option are disabled
- BLE_STACK_SLAVE_DLE_CONF: only connection and data length extension features are enabled

The following Bluetooth® Low Energy stack preprocessor configuration options defined on file stack_user_cfg.h are used to activate/disactivate (1: ENABLED; 0: DISABLED) each modular configuration option:

- CONTROLLER_MASTER_ENABLED: Central/observer (1) or only observer role (0) depending on the CONNECTION_ENABLED value (1/0)
- CONTROLLER_PRIVACY_ENABLED: Controller privacy
- SECURE_CONNECTIONS_ENABLED: LE Secure connections feature
- CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED: Data length extension feature
- CONTROLLER_2M_CODED_PHY_ENABLED: LE 2M + Coded PHYs
- CONTROLLER_EXT_ADV_SCAN_ENABLED: Extended advertising and scanning feature
- L2CAP_COS_ENABLED: L2CAP COS
- CONTROLLER_PERIODIC_ADV_ENABLED: Periodic advertising and synchronizer
- CONTROLLER_CTE_ENABLED: Constant tone extension

- CONTROLLER_POWER_CONTROL_ENABLED: LE power control
- CONNECTION_ENABLED: connection capability
- CONTROLLER_CHAN_CLASS_ENABLED: LE Channel Classification feature
- CONTROLLER_BIS_ENABLED : BIS feature
- EATT_ENABLED: EATT feature
- CONTROLLER_CIS_ENABLED : CIS feature
- CONNECTION_SUBRATING_ENABLED : Connection Subrating feature

Note: *On BlueNRG-LP, BlueNRG-LPS, STSW-BNRGLP-DK SW package, starting from Bluetooth® Low Energy stack v3.1 or later, if a user needs to define a specific set of configuration options, he can follow these steps:*

1. *Define the modular configuration options in a header file named custom_ble_stack_conf.h to be placed on application Inc folder*
2. *Add the BLE_STACK_CUSTOM_CONF preprocessor option on application project.*

Follow some examples of a specific set of configuration options, which could be defined through the custom_ble_stack_conf.h.

Broadcaster only configuration

```
#define CONTROLLER_MASTER_ENABLED (0U)
#define CONTROLLER_PRIVACY_ENABLED (0U)
#define SECURE_CONNECTIONS_ENABLED (0U)
#define CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED (0U)
#define CONTROLLER_2M_CODED_PHY_ENABLED (0U)
#define CONTROLLER_EXT_ADV_SCAN_ENABLED (0U)
#define L2CAP_COS_ENABLED (0U)
#define CONTROLLER_PERIODIC_ADV_ENABLED (0U)
#define CONTROLLER_CTE_ENABLED (0U)
#define CONTROLLER_POWER_CONTROL_ENABLED (0U)
#define CONNECTION_ENABLED (0U)
#define CONTROLLER_CHAN_CLASS_ENABLED (0U)
#define CONTROLLER_BIS_ENABLED (0U)
#define EATT_ENABLED  (0U)
#define CONNECTION_SUBRATING_ENABLED  (0U)
#define CONTROLLER_CIS_ENABLED  (0U)
```

Note: *Broadcaster capability is always supported and it cannot be turned off.*

Broadcaster + Observer only configuration

```
#define CONTROLLER_MASTER_ENABLED (1U)
#define CONTROLLER_PRIVACY_ENABLED (0U)
#define SECURE_CONNECTIONS_ENABLED (0U)
#define CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED (0U)
#define CONTROLLER_2M_CODED_PHY_ENABLED (0U)
#define CONTROLLER_EXT_ADV_SCAN_ENABLED (0U)
#define L2CAP_COS_ENABLED (0U)
#define CONTROLLER_PERIODIC_ADV_ENABLED (0U)
#define CONTROLLER_CTE_ENABLED (0U)
#define CONTROLLER_POWER_CONTROL_ENABLED (0U)
#define CONNECTION_ENABLED (0U)
#define CONTROLLER_CHAN_CLASS_ENABLED   (0U)
#define CONTROLLER_BIS_ENABLED  (0U)
#define EATT_ENABLED  (0U)
#define CONNECTION_SUBRATING_ENABLED   (0U)
#define CONTROLLER_CIS_ENABLED     (0U)
```

Peripheral device with legacy security (aka: basic configuration already available through the BLE_STACK_BASIC_CONF preprocessor option)

```
#define CONTROLLER_MASTER_ENABLED (0U)
#define CONTROLLER_PRIVACY_ENABLED (0U)
#define SECURE_CONNECTIONS_ENABLED (0U)
#define CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED (0U)
#define CONTROLLER_2M_CODED_PHY_ENABLED (0U)
#define CONTROLLER_EXT_ADV_SCAN_ENABLED (0U)
#define L2CAP_COS_ENABLED (0U)
#define CONTROLLER_PERIODIC_ADV_ENABLED (0U)
#define CONTROLLER_CTE_ENABLED (0U)
#define CONTROLLER_POWER_CONTROL_ENABLED (0U)
#define CONNECTION_ENABLED (1U)
#define CONTROLLER_CHAN_CLASS_ENABLED   (0U)
#define CONTROLLER_BIS_ENABLED    (0U)
#define EATT_ENABLED   (0U)
#define CONNECTION_SUBRATING_ENABLED  (0U)
#define CONTROLLER_CIS_ENABLED  (0U)
```

Note: *When CONNECTION_ENABLED = 0 (connection capability is not supported), a user shall perform the following modifications on his user application:*

- *On the user IDE project, the*

  ```
  BLE_STACK_CUSTOM_CONF
  ```

  preprocessor option is defined and a related custom_ble_stack_config.h header file with

  ```
  CONNECTION_ENABLED = 0
  ```

  is used.
- *On the user IDE project, the gap_profile.c, gatt_profile.c files are excluded from the application building.*
- *The aci_gatt_srv_init() is not anymore called (since no characteristics are added).*
- *The aci_gap_init(), role parameter is defined as GAP_BROADCASTER_ROLE (only broadcaster) or GAP_BROADCASTER_ROLE|GAP_OBSERVER_ROLE (only broadcaster, observer) depending on CONTROLLER_MASTER_ENABLED value (0/1).*
- *The Gap_profile_set_dev_name() is not anymore called.*
- *The aci_gap_set_advertising_configuration(), Discoverable_Mode parameter is defined as GAP_MODE_BROADCAST.*
- *The Advertising_Data parameter on aci_gap_set_advertising_data() does not set the general discoverable mode flag, since it is not supported on broadcaster mode.*

Table 30. Bluetooth® Low Energy application stack library framework interface (BlueNRG-LP, BlueNRG-LPS STSW-BNRGLP-DK)

File	Description	Location	Notes
ble_const.h	It includes the requiredBluetooth® Low Energy stack header files	Middlewares\ST\BLE_Application\layers_inc	To be included on the user-main application
gap_profile.[ch]	Header and source files for generic access profile service (GAP) library	Middlewares\BLE_Application\Profiles	-
gatt_profile.[ch]	Header and source files for generic attribute profile service (GATT) library	Middlewares\BLE_Application\Profiles	-
att_pwrq.[ch]	Header and source files for ATT prepare write queue implementation library	Middlewares\BLE_Application\Queued_Write	-
eatt_pwrq.[ch]	Header and source files for EATT Prepare Write Queue implementation	Middlewares\BLE_Application\Queued_Write	-

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

3.2 Bluetooth® Low Energy stack event callbacks

The Bluetooth® Low Energy 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® Low Energy event callback prototypes are defined on a header file bluenrg_lp_events.h. All callbacks are defined by default through weak definitions (no check is done on an event callback name defined by the user, so the user should carefully check that each defined callback is in line with the expected function name).

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

3.3 Bluetooth® Low Energy stack init and tick APIs

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

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

```
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:

```
typedef struct {
    uint8_t* BLEStartRamAddress ;
    uint32_t TotalBufferSize ;
    uint16_t NumAttrRecord ;
    uint8_t MaxNumOfClientProcs;
    uint8_t NumOfLinks;
    uint8_t NumOfEATTChannels;
    uint16_t NumBlockCount ;
    uint16_t ATT_MTU;
    uint32_t MaxConnEventLength;
    uint16_t SleepClockAccuracy;
    uint8_t NumOfAdvDataSet;
    uint8_t NumOfAuxScanSlots;
    uint8_t NumOfSyncSlots;
    uint8_t WhiteListSizeLog2;
    uint16_t L2CAP_MPS;
    uint8_t L2CAP_NumChannels;
    uint8_t CTE_MaxNumAntennaIDs;
    uint8_t CTE_MaxNumIQSamples;
    uint16_t isr0_fifo_size;
    uint16_t isr1_fifo_size;
    uint16_t user_fifo_size;
    }
```

Table 31. Bluetooth® Low Energy stack v3.x initialization parameters

Name	Description	Value
BLEStartRamAddress	Start address of the RAM buffer for GATT database allocated according to BLE_STACK_TOTAL_BUFFER_SIZE macro	32-bit aligned RAM area
TotalBufferSize	BLE_STACK_TOTAL_BUFFER_SIZE macro return value, used to check the MACRO correctness.	Refer to \Middlewares\ST\Bluetooth_LE\Inc\bluenrg_lp_stack.h file

Name	Description	Value
	It defines the total RAM reserved to manage all the data stack according to the number of radio tasks, number of attributes, number of concurrent GATT client procedures, number of memory blocks allocated for packets, number of advertising sets, number of auxiliary scan slots, white list size, number of L2CAP connection oriented channels supported).	
NumAttrRecord	Maximum number of attributes (that is, the number of characteristic declarations + the number of characteristic values + the number of descriptors + the number of the services) that can be stored in the GATT database	Minimum number is 2 attributes for each characteristic.
MaxNumOfClientProcs	Maximum number of concurrent client procedures	This value is less or equal to NumOfLinks
NumOfLinks	Maximum number of simultaneous radio tasks that the device can support. A radio task is an internal link layer state machine, which handles a specific radio activity (connection, advertising, scanning).	Radio controller supports up to 128 simultaneous radio tasks, but the actual usable max. value depends on the available device RAM (NUM_LINKS used in the calculation of BLE_STACK_TOTAL_BUFFER_SIZE)
NumOfEATTChannels	Maximum number of simultaneous EATT active channels	0 to L2CAP channels -1
NumBlockCount	Number of allocated memory blocks for the Bluetooth® LE stack packet allocations.	The minimum required value is calculated using a specific macro provided on bluenrg_lp_stack.h file: BLE_STACK_MBLOCKS_CALC()
ATT_MTU	Maximum supported ATT_MTU size	Supported values range is from 23 to 1020 bytes
MaxConnEventLength	Maximum duration of the connection event when the device is in slave mode in units of 625/256 us (~2.44 us)	<= 4000 (ms)
SleepClockAccuracy	Sleep clock accuracy	ppm value
NumOfAdvDataSet	Maximum number of advertising data set, valid only when advertising extension feature is enabled	Refer to Table 1
NumOfAuxScanSlots	Maximum number of slots for scanning on the secondary advertising channel, valid only when advertising extension feature is enabled	Refer to Table 1
NumOfSyncSlots	Maximum number of slots to be synchronized, valid only when Periodic Advertising and Synchronizing Feature is enabled.	0 if Periodic Advertising is disabled. [1 - NumOfLinks-1] if Periodic Advertising is enabled
WhiteListSizeLog2	Two's logarithm of the white/resolving list size	-
L2CAP_MPS	The maximum size of payload data in octets that the L2CAP layer entity can accept	Supported values range is from 0 to 1024 bytes
L2CAP_NumChannels	Maximum number of channels in LE credit based flow control mode	Supported values range is from 0 to 255 bytes
CTE_MaxNumAntennaIDs	Maximum number of antenna IDs in the antenna pattern used in CTE connection oriented mode.	[0x02-0x4B] if the direction finding feature is enabled and supported from the device.

Name	Description	Value
		0: if the direction finding feature is not supported from the device.
CTE_MaxNumIQSamples	Maximum number of IQ samples in the buffer used in CTE connection oriented mode.	[0x09-0x52] if the direction finding feature is enabled and supported from the device. 0: if the direction finding feature is supported from the device
isr0_fifo_size	Size of the internal FIFO used for critical controller events produced by the ISR (for example, rx data packets)	Default value: 256
isr1_fifo_size	Size of the internal FIFO used for noncritical controller events produced by the ISR (for example, advertising or IQ sampling reports)	Default value: 768
user_fifo_size	Size of the internal FIFO used for controller and host events produced outside the ISR	Default value: 1024

Note: A specific tool named "Radio Init Wizard" is provided on the BlueNRG-LP, BlueNRG-LPS device software development kit (STSW_BNRGLP_DK). The Radio Init Wizard allows the described parameter values to be configured according to the application needs, and to get an overview about the associated required RAM for the Bluetooth® Low Energy stack. User can also evaluate the impact of each parameter on an overall RAM memory footprint. This tunes the value of each radio initialization parameter (that is, NumOfLinks, ATT_MTU, ...), according to the available device RAM memory.

3.4 The Bluetooth® Low Energy stack v3.x application configuration

During the device initialization phase, after STMicroelectronics Bluetooth® Low Energy device powers on, some specific parameters must be defined on the Bluetooth® Low Energy 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/BlueNRG-LPS application configuration parameters are defined on a file system_bluenrg_lp.c through the following configuration table:

Table 32. Application configuration preprocessor options

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

3.5 Bluetooth® Low Energy stack tick function

The Bluetooth® Low Energy stack v3.x provides a special API `BLE_STACK_Tick()`, which must be called in order to process the internal Bluetooth® Low Energy stack state machines and when there are Bluetooth® Low Energy 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® Low Energy stack function must be called while the `BLE_STACK_Tick()` is being run. For example, if a Bluetooth® Low Energy 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:

```
NVIC_DisableIRQ(UART_IRQn);
BLE_STACK_Tick();
NVIC_EnableIRQ(UART_IRQn);
```

4 Designing an application with the Bluetooth® Low Energy stack v3.x

This section provides some information and code examples about how to design and implement a Bluetooth® Low Energy application using the Bluetooth® Low Energy stack v3.x binary library.

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

1. Initialization phase and main application loop.
2. Bluetooth® Low Energy 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 applications "defines" identify the Bluetooth® Low Energy device role (central, peripheral, client, and server). Furthermore, the BlueNRG-LP device is used as a reference device running Bluetooth® Low Energy stack v3.x applications. Any specific device-dependent part is highlighted whenever it is needed.

Table 33. User application defines for Bluetooth® Low Energy device roles

Define	Description
GATT_CLIENT	GATT client role
GATT_SERVER	GATT server role

4.1 Initialization phase and main application loop

The following main steps are required to properly configure the Bluetooth® Low Energy device running a Bluetooth® Low Energy stack v3.x application on STSW-BNRGLP-DK SW package supporting BlueNRG-LP, BlueNRG-LPS devices:

1. Initialize the Bluetooth® Low Energy 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® Low Energy controller and timer module: `BLECNTR_InitGlobal() and HAL_VTIMER_Init(&VTIMER_InitStruct)` APIs;
4. Initialize the NVM, PKA, RNG and AES modules: `BLEPLAT_Init(), PKAMGR_Init(), RNGMGR_Init() and AESMGR_Init()` APIs
5. Initialize the Bluetooth® Low Energy stack: `BLE_STACK_init(&BLE_STACK_InitParams)` API
6. Configure Bluetooth® Low Energy device public address (if public address is used): `aci_hal_write_config_data()` API
7. Init Bluetooth® Low Energy GATT layer: `aci_gatt_srv_init()` API
8. Init Bluetooth® Low Energy 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® Low Energy 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® Low Energy 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® Low Energy radio operating modes.

The following pseudocode example illustrates the required initialization steps on STSW-BNRGLP-DK SW package supporting BlueNRG-LP, BlueNRG-LPS devices:

```
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 Bluetooth® LE radio */
     LL_AHB_EnableClock(LL_AHB_PERIPH_PKA|LL_AHB_PERIPH_RNG);


     BLECNTR_InitGlobal();

     /* 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();
     if (PKAMGR_Init() == PKAMGR_ERROR)
     {
         while(1);
     }
     if (RNGMGR_Init() != RNGMGR_SUCCESS)
     {
         while(1);
     }
     /* Init the AES block */

     AESMGR_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);
     }
     /* 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) {
        while(1);
     }
     /* 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;
     wakeupIO.LPU_enable = 0;

     while(1)
     {
         /* Timer tick */
         HAL_VTIMER_Tick();
         /* Bluetooth® LE 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
         (adverting, connections intervals) */
         HAL_PWR_MNGR_Request(POWER_SAVE_LEVEL_STOP_NOTIMER,wakeupIO, &stopLevel);
     }/* while (1) */
```

```
} /* end main() */
```

Note:

1. *`BLE_STACK_InitParams` variable defines the stack initialization parameters as described on Section 3.2 Bluetooth® Low Energy 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 (refer to the BlueNRG-LP, BlueNRG-LPS datasheets on reference section). It is worth noticing that this API with the specified parameters must be called, on an 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 hardware 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 Bluetooth® LE device low power modes, refer to the related application note in Section 1.1 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 34. GATT, GAP service handles

Service	Start handle	End handle	Service UUID
Attribute profile service	0x0001	0x0008	0x1801
Generic access profile (GAP) service	0x0009	0x000F	0x1800

Table 35. GATT, GAP characteristic handles

Default services	Characteristic	Attribute handle	Char property	Char value handle	Char UUID	Char value length (bytes)
Attribute profile service	-	0x0001	-	-	-	-
-	Service changed	0x0002	Indicate	0x0003	0x2A05	4
-	Client Supported Features	0x0005	Read, Write	0x0006	0x2B29	1
-	Database Hash GATT	0x0007	Read	0x0008	0x2B2A	16
Generic access profile (GAP) service	-	0x0009	-	-	-	-
-	Device name	0x000A	Read|write without response| write| authenticated signed writes	0x000B	0x2A00	8

Default services	Characteristic	Attribute handle	Char property	Char value handle	Char UUID	Char value length (bytes)
-	Appearance	0x000C	Read|write without response| write| authenticated signed writes	0x000D	0x2A01	2
-	Peripheral preferred connection parameters	0x000E	Read| write	0x000F	0x2A04	8
-	Central address resolution(1)	0x0010	Readable without authentication or authorization. Not writable	0x00011	0x2AA6	1

1. *It is added only when controller-based privacy (0x02) is enabled on aci_gap_init() API.*

The `aci_gap_init()` role parameter values are as follows:

Table 36. aci_gap_init() role parameter values

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

For a complete description of this API and related parameters, refer to the Bluetooth® Low Energy stack APIs and event documentation, in Section 1.1 References.

4.1.1 Bluetooth® Low Energy addresses

The following device addresses are supported by the Bluetooth® Low Energy stack:

- Public address
- Random address
- Private address

Public MAC addresses (6 bytes - 48 bits address) uniquely identify a Bluetooth® Low Energy device, and they are defined by the institute of electrical and electronics engineers (IEEE).

The first three 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 three 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® Low Energy 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® Low Energydevices before starting any Bluetooth® Low Energy operations (after stack initialization API `BLE_STACK_Init()`).

The following pseudocode example illustrates how to set a public address:

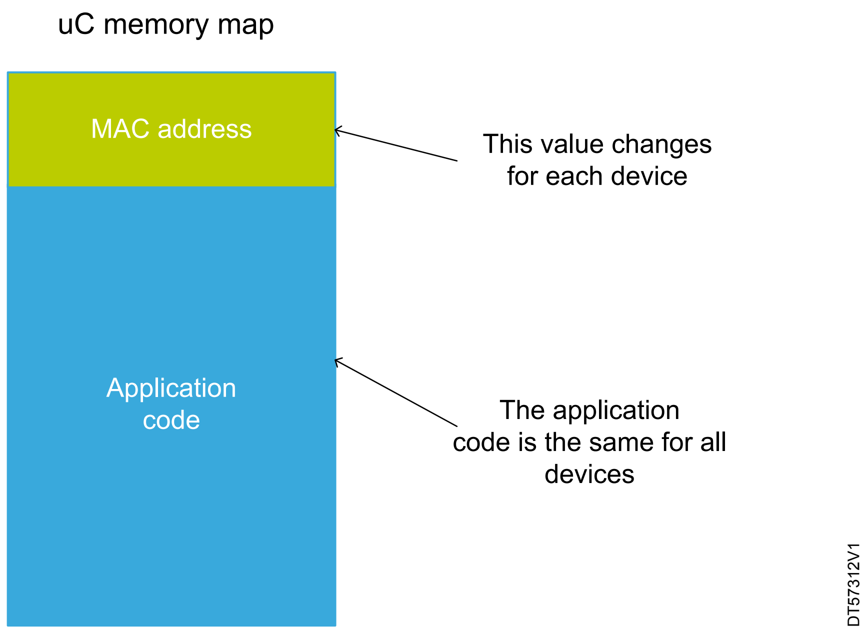
```
uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret=aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,CONFIG_DATA_PUBAD DR_LEN, bdaddr);
if(ret) PRINTF("Setting address failed.\n")}
```

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 (that is, reading it from a database) and storing of the MAC address in the known MAC flash location.

Figure 17. **MAC address storage**



The BlueNRG-LP/BlueNRG-LPS devices do not have a valid preassigned MAC address, but a unique serial number (read only for the user). The unique serial number is an 8-bytes value stored at address 0x10001EF0: it is stored as two words at address 0x10001EF0 and 0x10001EF4.

The following utility APIs get access to the BlueNRG-LP, BlueNRG-LPS unique serial ID:

```
uint32_t UID_word0;
uint32_t UID_word1;

/* Get first serial ID 4 bytes at serial id base address (0x10001EF0) */
UID_word0 = LL_GetUID_Word0();

/* Get last serial ID 4 bytes at serial id base address + 4 */
UID_word1 = LL_GetUID_Word1();
```

The static random address is generated and programmed at every first boot of the device on the dedicated flash area. The value on flash is the actual value that the device uses: each time the user resets the device the stack checks if valid data is on the dedicated flash area and it uses it (a special valid marker on flash is used to identify if valid data is 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 Section 2.7 Security manager (SM).

4.1.2 Set tx power level

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

```
aci_hal_set_tx_power_level(En_High_Power,PA_Level);
```

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

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

The En_High_Power parameter allows the selection of one of two possible power configurations. In each configuration, a different SMPS output voltage level is set. The PA_Level parameter is used to select the output level of the power amplifier with a granularity of around 1 dB. This output level depends on the SMPS output voltage.

The Table 37 and Table 38 provide the expected TX output power for each level in the two configurations on the device.

Table 37. TX power level with En_High_Power = 0 (SMPS voltage @ 1.4 Volts)

PA level	Expected TX power (dBm)
0	OFF (1)
1	-21
2	-20
3	-19
4	-17
5	-16
6	-15
7	-14
8	-13
9	-12
10	-11
11	-10
12	-9
13	-8
14	-7
15	-6
16	-6
17	-4
18	-3
19	-3
20	-2
21	-2
22	-1
23	-1
24	0
25	0
26	1

PA level	Expected TX power (dBm)
27	2
28	3
29	4
30	5
31	6

1. *When PA level is set to 0, PA is turned off. However, a very low output power may still be measured on the RF pin.*

Table 38. TX power level with En_High_Power = 1 (SMPS voltage @ 1.9 Volts)

PA level	Expected TX power (dBm)
0	OFF (1)
1	-19
2	-18
3	-17
4	-16
5	-15
6	-14
7	-13
8	-12
9	-11
10	-10
11	-9
12	-8
13	-7
14	-6
15	-5
16	-4
17	-3
18	-3
19	-2
20	-1
21	0
22	1
23	2
24	3
25	8
26	8
27	8
28	8
29	8
30	8
31	8

1. When PA level is set to 0, PA is turned off. However, a very low output power may still be measured on the RF pin.

Note: *When using an SMPS OFF configuration, the output power levels depend on the voltage applied to the VFBSD pin. In this case, the user must fill the look-up table in miscutil.c file (under Middlewares\ST\hal\Src directory) with the real-measured values. Filling the look-up table with the correct values is important because it is used by the Bluetooth® stack to correctly convert dBm to PA level and vice versa.*

4.2 Bluetooth® LE stack v3.x GATT interface

4.2.1 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.

4.2.2 GATT server

The GATT server is in charge of storing the attribute, which composes 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.

4.2.2.1 *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

```
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:

`aci_gatt_srv_add_service(&gap_srvc);`

while retrieving the assigned attribute handle the `aci_gatt_srv_get_service_handle`

function is used: `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: `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.*

4.2.2.2 *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:

```
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):

```
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.

4.2.2.3 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 `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 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:

```
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:

`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.

4.2.2.4 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:

```
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:

```
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:

```
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.

```
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.

```
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.

4.2.2.5 GATT server database APIs

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

Table 39. GATT server database APIs

API	Description
`aci_gatt_srv_add_service`	This function adds the provided service to the database
`aci_gatt_srv_rm_service`	This function removes the provided service from the database
`aci_gatt_srv_get_service_handle`	This function retrieves the attribute handle assigned to the service registered using the provided definition structure
`aci_gatt_srv_add_include_service`	This function adds the provided include service
`aci_gatt_srv_rm_include_service`	This function removes the provided include service from the database
`aci_gatt_srv_get_include_service_handle`	This function retrieves the attribute handle assigned to the include service
`aci_gatt_srv_add_char`	This function adds the provided characteristic to the database
`aci_gatt_srv_rm_char`	This function removes the provided characteristic from the database. All the included attributes (characteristic value and descriptors) are removed accordingly
`aci_gatt_srv_get_char_decl_handle`	This function retrieves the attribute handle assigned to the characteristic registered using the provided definition structure
`aci_gatt_srv_add_char_desc`	This function adds the provided descriptor to the database
`aci_gatt_srv_rm_char_desc`	This function removes the provided descriptor from the database
`aci_gatt_srv_get_descriptor_handle`	This function retrieves the attribute handle assigned to the characteristic descriptor registered using the provided definition structure

4.2.2.6 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:

```
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:

```
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:

```
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:

`aci_gatt_srv_add_service(&gatt_srvc);`

Glucose

Consider the following database:

Table 40. **Example database**

Attribute handle	Attribute type	UUID	Properties	Note
0x0001	Primary service	0x1808	-	Glucose service
-	Included service	-	-	Included battery service
-	Characteristic	0x2A18	Read, indicate, extended properties	Glucose measurement characteristic
-	Descriptor	-	-	CCCD
-	Descriptor	-	-	Extended properties descriptor
0x1000	Secondary service	0x180F	-	Battery service
-	Characteristic	0x2A19	Read	Battery level characteristic

Start defining the battery service with its characteristic:

```
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:

```
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 = 2U,
},
};

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:

`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:

`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:

```
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:

```
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:

`aci_gatt_srv_add_char(&glucose_mes_chr, glucose_srvc_handle);`

The glucose measurement characteristic and the included descriptors are added.

4.2.2.7 *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 41. aci_gatt_srv_notify parameters

Type	Parameter	Description
uint16_t	Connection_Handle	The connection handle for which the notification is requested
uint16_t	Attr_Handle	The attribute handle to notify
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.

```
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.

4.2.2.8 *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 42. aci_gatt_srv_read_event parameters

Type	Parameter	Description
uint16_t	Connection_Handle	The connection handle where the read request is received
uint16_t	Attr_Handle	The attribute handle to read
uint16_t	Data_Offset	The offset where to start reading value

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 43. aci_gatt_srv_write_event parameters

Type	Parameter	Description
uint16_t	Connection_Handle	The connection handle where the write request is received
uint8_t	Resp_Needed	If the value is 1, a call to `aci_gatt_serv_resp()` is required. This happens for ATT requests. ATT commands do not need a response (this parameter is set to 0)
uint16_t	Attribute_Handle	The attribute handle to write
uint16_t	Data_Length	The length of data to write
uint8_t *	Data	The data to write

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 44. aci_att_srv_prepare_write_req_event parameters

Type	Parameter	Description
uint16_t	Connection_Handle	Connection handle that identifies the connection
uint16_t	Attribute_Handle	Attribute handle for which the write request is received
uint16_t	Data_Offset	The offset where to start writing value
uint16_t	Data_Length	Length of the data field
uint8_t *	Data	The data to write

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 45. **aci_att_srv_exec_write_req_event parameters**

Type	Parameter	Description
uint16_t	`Connection_Handle`	Connection handle identifies the connection
uint8_t	`Flags`	• 0x00 – Cancel all prepared writes • 0x01 – Immediately write all pending prepared values

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 46. **aci_gatt_srv_resp parameters**

Type	Parameter	Description
uint16_t	`Connection_Handle`	Connection handle that identifies the connection
uint16_t	`Attribute_Handle`	Attribute handle for which the response command is issued
uint8_t	`Error_Code`	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)
uint16_t	`Val_Length`	Length of the value field
uint8_t *	`Val`	The response data in the following cases: • read request • prepare write request For other requests, this parameter can be NULL

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_Handleg is that , 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.

```
void aci_gatt_srv_exec_write_resp(uint16_t Conn_Handle,
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);
}
```

4.2.2.9 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 also provide some commodity functions to set up characteristic values like the device name.

4.2.2.10 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 47. ATT_pwrq_init parameters

Type	Parameter	Description
uint16_t	`queue_length`	Queue buffer size
uint8_t *	`queue_buffer_p`	Pointer to the buffer used to store the FIFO

ATT_pwrq_flush()

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

Table 48. ATT_pwrq_flush parameter

Type	Parameter	Description
uint16_t	`conn_handle`	Queue buffer size

ATT_pwrq_read()

Read the FIFO elements. Such elements are filtered per connection handle and indexed by a parameter.

Table 49. ATT_pwrq_read parameters

Type	Parameter	Description
uint16_t	conn_handle	The connection handle to filter
uint16_t	idx	The index of the entry to read
ble_gatt_clt_write_ops_t *	wr_ops_p	Returning pointer to the structure that holds the prepared write information

ATT_pwrq_pop()

Extract an entry from the FIFO. The entry is filtered per connection and attribute handles.

Table 50. ATT_pwrq_pop parameters

Type	Parameter	Description
uint16_t	conn_handle	The connection handle to filter
uint16_t	attr_handle	The attribute handle to filter
ble_gatt_clt_write_ops_t *	wr_ops_p	Returning pointer to the structure that holds the prepared write information

ATT_pwrq_push()

Push the data of a prepare write in the FIFO.

Table 51. ATT_pwrq_push parameters

Type	Parameter	Description
uint16_t	conn_handle	The connection handle from where the connection handle is received
uint16_t	attr_handle	The attribute handle to write
uint16_t	data_offset	The offset from where to start writing the data
uint16_t	data_length	Length of data to write
uint8_t *	data	Pointer to data to write

4.2.3 SoC vs. network coprocessor

While in the application processor mode, the application resides on the device memory; in the network coprocessor mode the application runs outside. The application layer inside the Bluetooth® Low Energy device exports Bluetooth® Low Energy 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.

4.2.3.1 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® Low Energy 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 temporary store data in a buffer. This component also allocates the buffer needed by those procedures. These buffers are kept allocated until the procedure is complete.

4.2.4 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 52. GATT client APIs

API	Description
aci_gatt_clt_exchange_config	This procedure is used to set the ATT MTU to the maximum possible value that can be supported by both devices. This procedure can be initiated once during a connection
aci_gatt_clt_disc_all_primary_services	This function is used to discover all the primary services on a server
aci_gatt_clt_disc_primary_service_by_uuid	This function is used to discover a specific primary service on a server when only the service UUID is known
aci_gatt_clt_find_included_services	This function is used to find include service declarations within a service definition on a server. The service is identified by the service handle range
aci_gatt_clt_disc_all_char_of_service	This function is used to find all characteristic declarations within a service definition on a server when only the service handle range is known
aci_gatt_clt_disc_char_by_uuid	This function is used to discover service characteristics on a server when only the service handle ranges are known and the characteristic UUID is known
aci_gatt_clt_disc_all_char_desc	This function is used to find all the characteristic descriptor attribute handles and attribute types within a characteristic definition when only the characteristic handle range is known. The characteristic specified is identified by the characteristic handle range
aci_gatt_clt_read	This function is used to read an attribute value from a server when the client knows the attribute handle
aci_gatt_clt_read_long	This function is used to read a long attribute value from a server when the client knows the attribute handle
aci_gatt_clt_read_multiple_char_value	This function is used to read multiple characteristic values from a server when the client knows the characteristic value handles
aci_gatt_clt_read_using_char_uuid	This function is used to read a characteristic value from a server when the client only knows the characteristic UUID and does not know the handle of the characteristic

API	Description
aci_gatt_clt_write_without_resp	This function is used to write an attribute value to a server when the client knows the attribute handle and the client does not need an acknowledgment that the write was successfully performed. This function only writes the first (ATT_MTU – 3) octets of an attribute value
aci_gatt_clt_signed_write_without_resp	This function is used to write an attribute value to a server when the client knows the attribute handle and the ATT bearer is not encrypted. This function is only used if the attribute properties authenticated bit is enabled and the client and server device share a bond
aci_gatt_clt_write	This function is used to write an attribute value to a server when the client knows the attribute handle. This function only writes the first (ATT_MTU – 3) octets of an attribute value
aci_gatt_clt_write_long	This function is used to write an attribute value to a server when the client knows the attribute handle, but the length of the attribute value is longer than can be sent using the aci_gatt_clt_write() function
aci_gatt_clt_write_char_reliable	This function is used to write a characteristic value to a server when the client knows the characteristic value handle, and assurance is required that the correct characteristic value is going to be written by transferring the characteristic value to be written in both directions before the write is performed. This function can also be used when multiple values must be written, in order, in a single operation.
aci_gatt_clt_notification_event	This event is generated when a server is configured to notify a characteristic value to a client without expecting any attribute protocol layer acknowledgment that the notification was successfully received
aci_gatt_clt_indication_event	This event is generated when a server is configured to indicate a characteristic value to a client and expects an attribute protocol layer acknowledgment that the indication was successfully received. To confirm it the aci_gatt_clt_confirm_indication function is used
aci_gatt_clt_confirm_indication	This function is used to generate a handle value confirmation to the server to indicate that the handle value indication is received.
aci_gatt_clt_prepare_write_req	This function is used to request the server to prepare to write the value of an attribute. The application can send more than one prepare write request to a server, which queues and sends a response for each handle value pair
aci_gatt_clt_execute_write_req	This function is used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client

4.2.5 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).
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

```
/* 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,0xb1,0xe1,0xf2,0x7
3,0xd9
#define RX_CHR_UUID 0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe2,0xf2,0x7
3,0xd9
#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:

```
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:

```
TXCharHandle=aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&chat_chars[0]);
```

```
RXCharHandle=aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&chat_chars[1]);
```

For a detailed description of the `aci_gatt_srv_add_service()` API parameters refer to the header file `bluenrg_lp_events.h`.

4.3 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 Bluetooth® LE 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 53. aci_gap_set_advertising_configuration() API : discoverable mode and advertising type selection

API	Discoverable_Mode parameter	Advertising_Event_Properties parameter
aci_gap_set_advertising_configuration()	0x00: not discoverable 0x01: limited discoverable 0x02: general discoverable	0x0001: connectable 0x0002: scannable 0x0004: directed 0x0008: high duty cycle directed connectable 0x0010: legacy 0x0020: anonymous 0x0040: include TX power

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 Bluetooth® LE specifications.

The content of the buffer containing the advertising/scan response data is directly accessed by the stack, hence it should not change when device is advertising.

An aci_hal_adv_scan_resp_data_update_event is received to inform the application that the buffer is no more used by the stack. This can happen after a new advertising buffer is provided to the stack through aci_gap_set_advertising_data or when advertising has been terminated. It is possible to change the content of the advertising buffer, while it is used by the stack, without any unwanted effect (e.g. corrupted data over-the-air) only when the change is atomic, e.g. a single byte or word is modified.

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 54. aci_gap_start_procedure() API

API	Procedure_Code parameter
aci_gap_start_procedure()	0x00: LIMITED_DISCOVERY 0x01: GENERAL_DISCOVERY 0x02: AUTO_CONNECTION 0x03: GENERAL_CONNECTION 0x04: SELECTIVE_CONNECTION 0x05: OBSERVATION

A GAP discovery procedure can be terminated using the aci_gap_terminate_proc() API.

Table 55. aci_gap_terminate_proc() API

API	Procedure_Code parameter
aci_gap_terminate_proc()	0x00: LIMITED_DISCOVERY 0x01: GENERAL_DISCOVERY

API	Procedure_Code parameter
	0x02: AUTO_CONNECTION 0x03: GENERAL_CONNECTION 0x04: SELECTIVE_CONNECTION 0x05: OBSERVATION

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).

4.3.1 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:

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

/*GAP Peripheral: configure general discoverable mode and set advertising data
*/

void GAP_Peripheral_Configure_Advertising(void )
{
    tBleStatus ret;

    /* Define advertising data content: set AD type flags and complete local name */
    static uint8_t adv_data[] = {0x02,AD_TYPE_FLAGS, FLAG_BIT_LE_GENERAL_DISCOVERABLE_MODE|FLAG_BIT_BR_
EDR_NOT_SUPPORTED,13, AD_TYPE_COMPLETE_LOCAL_NAME, 'B','l','u','e','N','R','G','X','T','e','s',' t'};


    /* Configure the General Discoverable mode, connectable and scannable (legacy advertising):
    Advertising_Handle: 0
    Discoverable_Mode: GAP_MODE_GENERAL_DISCOVERABLE (general discovery mode)
    Advertising_Event_Properties ADV_PROP_CONNECTABLE|ADV_PROP_SCANNABLE|ADV_PROP_LEGACY(connectable, s
cannable and legacy)
    Primary_Advertising_Interval_Min: 100;
    Primary_Advertising_Interval_Max: 100
    Primary_Advertising_Channel_Map: ADV_CH_ALL (channels 37,38,39)
    Peer_Address_Type: 0;
    Peer_Address[6]: NULL;
    Advertising_Filter_Policy: ADV_NO_WHITE_LIST_USE (no whitelist);
    Advertising_Tx_Power: 0;
    Primary_Advertising_PHY: 0;
    Secondary_Advertising_Max_Skip: 0;
    Secondary_Advertising_PHY: 0;
    Advertising_SID: 0;
    Scan_Request_Notification_Enable: 0.
    */

    ret = aci_gap_set_advertising_configuration(0,
                                                GAP_MODE_GENERAL_DISCOVERABLE,
                                                ADV_PROP_CONNECTABLE
                                                ADV_PROP_SCANNABLE|ADV_PROP_LEGACY,
                                                100, 100,
                                                ADV_CH_ALL,
                                                0,
                                                NULL,
                                                ADV_NO_WHITE_LIST_USE,
                                                0, 1, 0, 1, 0,0);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

```
    /* Set the advertising data */
    ret = aci_gap_set_advertising_data(0, ADV_COMPLETE_DATA, sizeof(adv_data), adv_data);
    if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

} /* end GAP_Peripheral_Configure_Advertising() */
```

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 advertsing);
       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_Latency: 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.

```
/*GAP Central: direct connection establishment procedure to connect to the
GAP Peripheral in discoverable mode
*/

void GAP_Central_Make_Connection(void)

{
    /*Start the direct connection establishment procedure to the GAP peripheral device:
    Initiating_PHY:LE_1M_PHY(1 Mbps PHY)
    Peer_Address_Type: PUBLIC_ADDR (public address);
    Peer_Address: {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
    */

    tBDAddr GAP_Peripheral_address = {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};

    /* direct connection to GAP Peripheral device */
    ret = aci_gap_create_connection(LE_1M_PHY,
                                PUBLIC_ADDR, GAP_Peripheral_address);
    if(ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

}/* 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.*

4.3.2 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.\n");
```

Furthermore, the GAP peripheral device has configured the advertising and related data as described in Section 4.3.1 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
    */
    static uint8_t ServiceUUID_Scan[18]=
    {0x11,0x06,0x8a,0x97,0xf7,0xc0,0x85,
    0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x2,0x0c,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 advertsing);
    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.\n");

} /* 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.\n");

}/* end GAP_Central_Configure_General_Discovery_Procedure() */


/*GAP Central: start general discovery procedure to discover the GAP peripheral device in dis coverable
 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.\n");
}
```

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)`.

```
/* This callback is called when an advertising report is received */
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 */
    int8_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 discoverable 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 56. ADV_IND event type: main fields

Event type	Address type	Address	Advertising data	RSSI
0x0013 (ADV_IND)	0x00 (public address)	0x0280E1003412	0x02,0x01,0x06,0x08,0x09,0x42,0x6C,0x75,0x65,0x4E,0x52,0x47,0x02,0x 0A,0xFE	0xCE

The advertising data are shown as follows (refer to Bluetooth® specification version in Section 1.1 References):

Table 57. ADV_IND advertising data: main fields

Flag AD type field	Local name field
0x02: length of the field 0x01: AD type flags 0x06: 0x110 (Bit 2: BR/EDR Not supported; bit 1: general discoverable mode)	0x08: length of the field 0x09: complete local name type 0x42,0x6C,0x75,0x65,0x4E0x52,0x47: BlueNRG

Table 58. SCAN_RSP event type

Event type	Address type	Address	Scan response data	RSSI
0x001B (SCAN_RS P)	0x01 (random address)	0x0280E1003412	0x12,0x66,0x9A,0x0C, 0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0x85,0xC0,0xF7,0x97,0x8A,0x06,0x11	0xDA

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

Table 59. Scan response data

Scan response data
0x12: data length
0x11: length of service UUID advertising data; 0x06: 128 bits service UUID type;
0x66,0x9A,0x0C,0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0x85,0xC0,0xF7,0x97,0x8A: 128-bit service UUID

4.4 Bluetooth® Low Energy stack events and event callbacks

Whenever there is a Bluetooth® Low Energy stack event to be processed, the Bluetooth® Low Energy 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 theBluetooth® Low Energy stack, while an API is a function defined by the stack and called by the user application. The Bluetooth® Low Energy 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® Low Energy application is implemented, the most common and widely used Bluetooth® Low Energy 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 60. Bluetooth® Low Energy stack: main event callbacks

Event callback	Description	Where
`hci_disconnection_complete_event()`	A connection is terminated	GAP central/ peripheral
`hci_le_enhanced_connection_complete_event()`	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 advertsing/scanning features	GAP central/ peripheral (Bluetooth® LE stack v3.x)
`aci_gatt_clt_notification_event()`	Generated by the GATT client when a server notifies any attribute on the client	GATT client
`aci_gatt_clt_indication_event()`	Generated by the GATT client when a server indicates any attribute on the client	GATT client
`aci_gap_pass_key_req_event()`	Generated by the security manager to the application when a passkey is required for pairing. When this event is received, the application has to respond with the aci_gap_pass_key_resp() API	GAP central/ peripheral
`aci_gap_pairing_complete_event()`	Generated when the pairing process has completed successfully or a pairing procedure timeout has occurred or the pairing has failed	GAP central/ peripheral

Event callback	Description	Where
aci_gap_bond_lost_event()	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	GAP peripheral
aci_gatt_clt_read_by_group_type_resp_event()	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	GATT client
aci_gatt_clt_read_by_type_resp_event()	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	GATT client
aci_gatt_clt_proc_complete_event()	A GATT procedure has been completed	GATT client
hci_le_extended_advertising_report_event()	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 Bluetooth® LE stack supports the extended advertsing/scanning features	GAP central (Bluetooth® LE stack v3.x)

For a detailed description of the Bluetooth® Low Energy event, and related formats refer to the Bluetooth® Low Energy stack APIs and events documentation in References.

The following pseudocode provides an example of event callbacks handling some of the described Bluetooth® Low Energy stack events (disconnection complete event, connection complete event, GATT attribute modified event , GATT notification event):

```
/* This event callback indicates the disconnection from a peer device.
It is called in the Bluetooth® LE radio interrupt context.
*/
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 scen
ario.
     */
}/* end hci_disconnection_complete_event() */

/* 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 Bluetooth® Low Energy 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;
    …
}/* end hci_le_enhanced_connection_complete_event() */
```

```
#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 callbac
k parameters.
*/
...
} /* end aci_gatt_srv_write_event () */

#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*/
    …
} /* end aci_gatt_notification_event() */
#endif /* GATT_CLIENT */
```

4.5 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 must be correctly configured, depending on the IO capabilily 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® Low Energy 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 4, Device_1, Device_2 must 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:

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

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

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.

```
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.\n");
```

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):

```
tBleStatus ret;
ret= aci_gap_slave_security_req(conn_handle,
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

- 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 paring procedure is initiated by one of the two devices, Bluetooth® Low Energy 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® Low Energy 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:

```
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:

```
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
                                                    type */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

Note:

1. *When the pairing procedure is started by calling the described APIs (`aci_gap_slave_security_req()` or `aci_gap_send_pairing_req()`) and the value `ret= BLE_STATUS_SUCCESS` is returned, on termination of the procedure, an `aci_gap_pairing_complete_event()` event callback is called to indicate the pairing status on the callback `Status` parameter:*
   - *0x00: pairing success*
   - *0x01: pairing timeout*
   - *0x02: pairing failed*

   The `reason` parameter provides the pairing failed reason code in case of failure (0 if status parameter returns success or timeout).
2. *When two devices get paired, the link is automatically encrypted during the first connection. If bonding is also enabled (keys are stored for a future time), when the two devices get connected again, the link can be simply encrypted (with no need to perform again the pairing procedure). User applications can simply use the same APIs, which do not perform the paring process but just encrypt the link:*
   - *`aci_gap_slave_security_req )` on the GAP peripheral (slave) device or*
   - *`aci_gap_send_pairing_req()` on the GAP central (master) device.*
3. *If a slave has already bonded with a master, it can send a slave security request to the master to encrypt the link. When receiving the slave security request, the master may encrypt the link, initiate the pairing procedure, or reject the request. Typically, the master only encrypts the link, without performing the pairing procedure. Instead, if the master starts the pairing procedure, it means that for some reason, the master lost its bond information, so it has to start the pairing procedure again. As a consequence, the slave device calls the `aci_gap_bond_lost_event()`event callback to inform the user application that it is no longer bonded with the master it was previously bonded with. Then, the slave application can decide to allow the security manager to complete the pairing procedure and re-bond with the master by calling the command `aci_gap_allow_rebond()`, or just close the connection and inform the user of the security issue.*

4.6 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 61. Bluetooth® Low Energy sensor profile demo services and characteristic handle

Service	Characteristic	Service / characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
Acceleration service	NA	0x0010	NA	NA	NA
	Free Fall characteristic	0x0011	0x0012	0x0013	NA
	Acceleration characteristic	0x0014	0x0015	0x0016	NA
Environmental service	NA	0x0017	NA	NA	NA
	Temperature characteristic	0x0018	0xx0019	NA	0x001A
	Pressure characteristic	0x001B	0xx001C	NA	0x001D

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 1.1 References).

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

Table 62. Service discovery procedures APIs

Discovery service API	Description
aci_gatt_clt_disc_all_primary_services()	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
aci_gatt_clt_disc_primary_service_by_uuid()	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
aci_gatt_clt_find_included_services()	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

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

```
/*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.

```
/* 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 63. **First read by group type response event callback parameters**

Attribute handle	End group handle	Service UUID	Notes
0x0001	0x0008	0x1801	Attribute profile service. Standard 16-bit service UUID
0x0009	0x000F	0x1800	GAP profile service. Standard 16-bit service UUID.

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 64. **Second read by group type response event callback parameters**

Attribute handle	End group handle	Service UUID	Notes
0x0010	0x0016	0x02366E80CF3A11E19AB4 0002A5D5C51B	Acceleration service 128-bit service proprietary UUID

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 65. Third read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x0017	0x001D	0x42821A40E47711E282D00 002A5D5C51B	Environmental service 128-bit service proprietary UUID

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
```

4.7 Characteristic discovery procedures and related GATT events

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

Table 66. Characteristics discovery procedures APIs

Discovery service API	Description
`aci_gatt_ctl_disc_all_char_of_service ()`	This API starts the GATT procedure to discover all the characteristics of a given service
`aci_gatt_ctl_disc_char_by_uuid ()`	This API starts the GATT procedure to discover all the characteristics specified by a UUID
`aci_gatt_ctl_disc_all_char_desc ()`	This API starts the procedure to discover all characteristic descriptors on the GATT server

In the context of theBluetooth® Low Energy 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 3. Bluetooth® Low Energy LE RF channel types and frequencies second read by group type response event callback parameters):

```
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.\n");
}
```

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.

```
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® Low Energy 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:

```
conn_handle : 0x0801 (connection handle);
Handle_Value_Pair_Length: 0x15 length of each discovered
characteristic data: characteristic handle, properties,
characteristic value handle, characteristic UUID;
Data_Length: 0x16(length of the event data);
Handle_Value_Pair_Data: 0x15 bytes as follows:
```

Table 67. **First read by type response event callback parameters**

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x0011	0x10 (notify)	0x0012	0xE23E78A0CF4A11E18FFC0002A5D5C51B	Free fall characteristic 128-bit characteristic proprietary UUID

Second read by type response event callback parameters:

```
conn_handle : 0x0801 (connection handle);
Handle_Value_Pair_Length: 0x15 length of each discovered
characteristic data: characteristic handle, properties,
characteristic value handle, characteristic UUID;
Data_Length: 0x16(length of the event data);
Handle_Value_Pair_Data: 0x15 bytes as follows:
```

Table 68. **Second read by type response event callback parameters**

Characteristic handle	Characteristic properties	Characteristi c value handle	Characteristic UUID	Note
0x0014	0x12 (notify and read)	0x0015	0x340A1B80CF4B11E1AC360002A5D5C51 B	Acceleration characteristic 128-bit characteristic proprietary UUID

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 3. Bluetooth® Low Energy LE RF channel types and frequencies).

4.8 Characteristic notification/indications, write, read

This section describes the main functions to get access to Bluetooth® Low Energy device characteristics.

Table 69. Characteristic update, read, write APIs

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

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.\n");
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.\n");
```

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.\n");

tBleStatus ret;
uint8_t buff[6];
uint16_t charac_handle = 0x004;

/*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.\n");
```

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:

```
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 */
```

4.9 Basic/typical error condition description

On the Bluetooth® Low Energy stack v3.x API framework, the `tBleStatus` type is defined in order to return the Bluetooth® Low Energy 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® Low Energy stack APIs and event documentation, in Section 1.1 References.

4.10 Simultaneously master, slave scenario

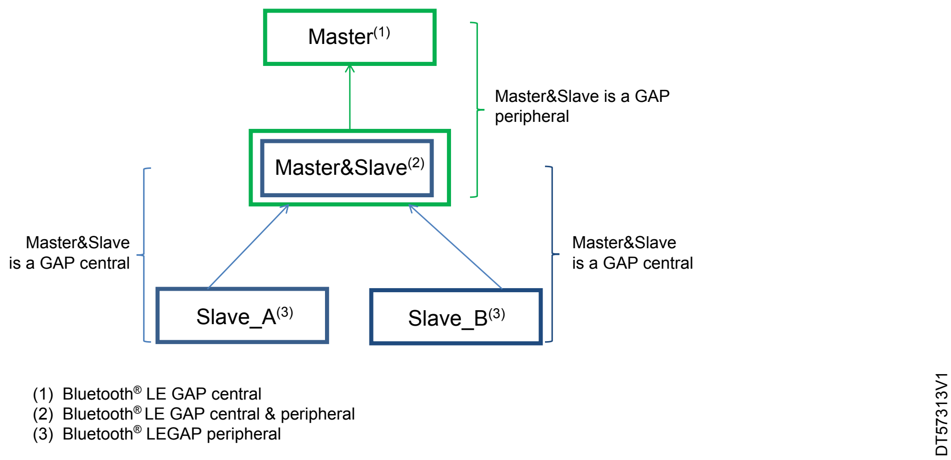
The Bluetooth® LE stack v3.x supports multiple roles simultaneously. This allows the same device to act as master on one or more connections (up to eight connections are supported), and to act as a slave on another connection.

The following pseudocode describes how a Bluetooth® LE stack device can be initialized to support central and peripheral roles simultaneously:

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

A simultaneous master and slave test scenario can be targeted as follows:

Figure 18. 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:

```
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:

```
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_extendede_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_Baddress type and address have been found during the discovery pr
ocedure 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 `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; Perio
d=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 addr
ess have been found during the discovery procedure through the Advertising Report ev
ents */
ret= aci_gap_create_connection(Initiating_PHY = 0x01, Peer_Address_Type= "Master&Sl
ave 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.

4.11 Bluetooth® Low Energy privacy 1.2

Bluetooth® Low Energy stack v3.x supports the Bluetooth® Low Energy privacy 1.2.

Privacy feature reduces the ability to track a specific Bluetooth® Low Energy by modifying the related Bluetooth® Low Energy address frequently. The frequently modified address is called the private address and the trusted devices are able to resolve it.

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).

4.11.1 Controller-based privacy and the device filtering scenario

On Bluetooth® Low Energy 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® Low Energy 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).

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.

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 1.1 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.

4.11.2 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® Low Energy 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® Low Energy 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.

4.12 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:

```
tBleStatus aci_gatt_clt_exchange_config(uint16_t Connection_Handle);
```

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.

4.13 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:

```
tBleStatus hci_le_set_data_length(uint16_t Connection_Handle,
                                  uint16_t TxOctets,
                                  uint16_t TxTime);
```

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:

```
hci_le_data_length_change_event(uint16_t Connection_Handle,
                                uint16_t MaxTxOctets,
                                uint16_t MaxTxTime,
                                uint16_t MaxRxOctets,
                                uint16_t 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.

4.14 No packet retry feature

Bluetooth® Low Energy stack v3.x provides the capability to disable the standard Bluetooth® Low Energy 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 that are within the maximum allowed link layer packet length.

When a 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 many 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.

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

```
tBleStatus aci_gatt_srv_notify(uint16_tConn_Handle,
                               uint16_t Attr_Handle,
                               uint8_t Flags,
                               uint16_t Value_Length,
                               uint8_t Value[]);
```

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

4.15 Bluetooth® Low Energy 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® Low Energy 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® Low Energy radio activities through the `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® Low Energy application starts advertising and it also performs write operations on flash. The `aci_hal_end_of_radio_activity_event()`callback is used to register the `Next_Advertising_SysTime ()` 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.

4.16 Bluetooth® Low Energy 2 Mbit/s 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.

```
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.

```
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);
```

The following API allows the current PHY to be read:

```
tBleStatus hci_le_read_phy(uint16_t Connection_Handle,
                                    uint8_t *TX_PHY,
                                    uint8_t *RX_PHY);
```

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

4.17 Bluetooth® Low Energy extended advertising/scanning

A set of HCI LE standard APIs for extended advertising/scanning is supported by Bluetooth® Low Energystack v3.x.

Refer to related html documentation for APIs and parameters description.

Furthermore, new GAP APIs for configuring advertising modes and scanning procedures allow an extended advertising/scanning feature to be supported. Refer to Section 4.3 GAP API interface for more details and examples.

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 adverting 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,ADVERTISIN
G_EVENT_PROPERTIES,
                                            160, 160,
                                            ADV_CH_ALL,
                                            0,NULL, /* No peer address */
                                            ADV_NO_WHITE_LIST_USE,
                                            0,/* 0 dBm */
                                            ADV_PHY,/* Primary advertising PHY */
                                            0, /* 0 skips */
                                            ADV_PHY,/* Secondary advertising PHY */
                                            0, /* SID */
                                            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);
```

4.17.1 Events for extended adv and scan

The following events are now available:

- `hci_le_extended_advertising_report_event()`:
  This event indicates that one or more Bluetooth® devices have responded to an active scan or have broadcast advertisements that were received during a passive scan.
  It is generated if the extended advertising and scan is supported. Otherwise, the `hci_le_advertising_report_event()` is generated.
- `hci_le_enhanced_connection_complete_event()`:
  This event indicates that a new connection has been created.
  It is already available to support controller privacy (if it is enabled). On Bluetooth® Low Energy stack v3.x, this event is also generated if the extended advertising and scan is supported. Otherwise, the `hci_le_connection_complete_event()` is generated.

4.18 Periodic advertising and periodic advertising sync transfer

Bluetooth® specification v5.0 defines periodic advertising that uses deterministic scheduling to allow a device to synchronize its scanning with the advertising of another device.

A new synchronization mode is defined by the generic access profile, which allows the periodic advertising synchronization establishment procedure to be performed and to synchronize with the advertising.

Periodic advertisements use a new link layer PDU called AUX_SYNC_IND. The required information (timing and timing offset) needed to synchronize with the periodic advertising packets is sent to a field, called SyncInfo, included in AUX_ADV_IND PDUs.

The periodic advertising synchronization procedure has a cost in terms of energy and some devices could not be in the conditions to perform this procedure.

A new periodic advertising sync transfer (PAST) procedure has been defined in order to allow a device A, which receives periodic advertising packets from device B, to pass the acquired synchronization information to another device C, which is connected to the device A. As consequence, the device C is able to receive the periodic advertising packets directly from device B without the need to scan for AUX_ADV_IND PDUs, which would consume too much energy.

4.18.1 Periodic advertising mode

The periodic advertising mode provides a method for a device to send advertising data at periodic and deterministic intervals. This mode applies to the broadcaster role only.

A device in the periodic advertising mode sends periodic advertising events at the interval and using the frequency hopping sequence specified in the periodic advertising synchronization information.

A device entering periodic advertising mode shall also enter periodic advertising synchronization mode for at least long enough to complete one extended advertising event.

HCI commands

`hci_le_set_periodic_advertising_parameters();`

`hci_le_set_periodic_advertising_data();`

`hci_le_set_periodic_advertising_enable().`

Note: Refer to Bluetooth® Low Energy APIs html documentation for detailed commands description.

GAP commands

GAP commands for periodic advertising in the broadcaster role are just formal wrappers to the HCI commands.

A mechanism analogous to the one already existing for advertising data is used.

In system-on-chip (SoC) mode the buffer pointer is provided by the application and transferred to the controller by the `ll_set_periodic_advertising_data_ptr` and freed by the `aci_hal_adv_scan_resp_data_update_event`.

In network mode the buffer shall be managed at DTM level both for the standard and GAP level commands:

`aci_gap_set_periodic_advertising_configuration();`

`aci_gap_set_periodic_advertising_enable();`

`aci_gap_set_periodic_advertising_data();`

`aci_gap_set_periodic_advertising_data_nwk().`

Note: Refer to Bluetooth® Low Energy APIs html documentation for detailed commands description.

4.18.2 Periodic advertising synchronizability mode

The periodic advertising synchronizability mode provides a method for a device to send synchronization information about a periodic advertising train using advertisements. This mode applies to the Broadcaster role only.

A device in the periodic advertising synchronizability mode sends synchronization information for a periodic advertising train in non-connectable and non-scannable extended advertising events. The advertising interval used is unrelated to the interval between the periodic advertising events.

A device is not in periodic advertising synchronizability mode unless it is also in periodic advertising mode. It may leave, and possibly re-enter, periodic advertising synchronizability mode while remaining in periodic advertising mode.

Selection between periodic advertising mode and periodic advertising synchronizability mode is achieved by explicitly enabling/disabling (regular) advertising and periodic advertising. While regular advertising and periodic advertising are both enabled, the device is in synchronizability mode. If regular advertising is disabled, device exits periodic advertising synchronizability mode.

4.18.3 Periodic advertising synchronization establishment procedure

The periodic advertising synchronization establishment procedure provides a method for a device to receive periodic advertising synchronization information and to synchronize to a periodic advertising train. This procedure applies to the observer, peripheral and central roles

A device performing the periodic advertising synchronization establishment procedure scans for non-connectable and non-scannable advertising events containing synchronization information about a periodic advertising train or accepts periodic advertising synchronization information over an existing connection by taking part in the Link Layer Periodic Advertising Sync Transfer Procedure defined in [Vol 6] part B, section 5.1.13.

When a device receives synchronization information for a periodic advertising train, it may listen for periodic advertising events at the intervals and using the frequency hopping sequence specified in the periodic advertising synchronization information.

HCI commands

```
hci_le_periodic_advertising_create_sync();
hci_le_periodic_advertising_create_sync_cancel();
hci_le_periodic_advertising_terminate_sync();
hci_le_set_periodic_advertising_receive_enable();
hci_le_add_device_to_periodic_advertiser_list();
hci_le_remove_device_from_periodic_advertiser_list();
hci_le_clear_periodic_advertiser_list();
hci_le_read_periodic_advertiser_list_size();
hci_le_set_periodic_advertising_sync_transfer_parameters();
hci_le_set_default_periodic_advertising_sync_transfer_parameters()
```

Note: Refer to Bluetooth® LE APIs html documentation for detailed commands description.

GAP commands

GAP commands for synchronizing are just formal wrappers to the HCI commands.

```
aci_gap_periodic_advertising_create_sync();
aci_gap_periodic_advertising_create_sync_cancel();
aci_gap_periodic_advertising_terminate_sync();
```

```
aci_gap_set_periodic_advertising_receive_enable();
aci_gap_add_device_to_periodic_advertiser_list();
aci_gap_remove_device_from_periodic_advertiser_list();
aci_gap_clear_periodic_advertiser_list();
aci_gap_read_periodic_advertiser_list_size();
aci_gap_set_periodic_advertising_sync_transfer_parameters();
aci_gap_set_default_periodic_advertising_sync_transfer_parameters().
```

Note: Refer to Bluetooth® Low Energy APIs html documentation for detailed commands description.

Events

The events involved on this mode are the following:

```
hci_le_periodic_advertising_sync_established_event()
hci_le_periodic_advertising_report_event()
hci_le_periodic_advertising_sync_lost_event()
hci_le_periodic_advertising_sync_transfer_received_event().
```

4.18.4 Periodic advertising synchronization transfer procedure

The periodic advertising synchronization transfer procedure provides a method for a device to send synchronization information about a periodic advertising train over an existing connection. This procedure applies the Peripheral and Central roles only.

A device performing the periodic advertising synchronization transfer procedure shall initiate the link layer periodic advertising sync transfer procedure defined in [Vol 6] part B, section 5.1.13.

HCI commands

```
hci_le_periodic_advertising_sync_transfer();
hci_le_periodic_advertising_set_info_transfer().
```

Note: Refer to Bluetooth® Low Energy APIs html documentation for detailed commands description.

GAP commands

```
aci_gap_periodic_advertising_sync_transfer();
aci_gap_periodic_advertising_set_info_transfer().
```

Note: Refer to Bluetooth® Low Energy APIs html documentation for detailed commands description.

4.18.5 LE power control and path loss monitoring

The following commands and events handle the LE power control and path loss monitoring features provided by the Bluetooth® Low Energy stack v3.1 or later.

```
/* Read the current and maximum transmit power levels of the local Controller on a connection */
tBleStatus hci_le_enhanced_read_transmit_power_level(uint16_t Connection_Handle,
                                                     uint8_tPHY,
                                                     int8_t*Current_Transmit_Power_Level,
                                                     int8_t *Max_Transmit_Power_Level);
/* Read the transmit power level used by the remote Controller on a connection*/
tBleStatus hci_le_read_remote_transmit_power_level(uint16_t Connection_Handle,
                                                   uint8_t PHY);
/* Set the path loss threshold reporting parameters for a connection */
tBleStatus hci_le_set_path_loss_reporting_parameters(uint16_t Connection_Handle,
                                                     uint8_t High_Threshold,
                                                     uint8_t High_Hysteresis,
                                                     uint8_t Low_Threshold,
                                                     uint8_t Low_Hysteresis,
                                                     uint16_t Min_Time_Spent);
/* Enable or disable path loss reporting for a connection */
tBleStatus hci_le_set_path_loss_reporting_enable(uint16_t Connection_Handle,
                                                 uint8_t Enable);
/* Enable or disable the reporting of transmit power level changes in the
 local and remote Controllers for a connection */
tBleStatus hci_le_set_transmit_power_reporting_enable(uint16_t Connection_Handle,
                                                      uint8_t Local_Enable,
                                                      uint8_t Remote_Enable);
/* Enable or disable the LE Power Control feature and procedure for a given PHY on the later established connections
 */
tBleStatus aci_hal_set_le_power_control(uint8_t Enable,
                                        uint8_t PHY,
                                        int8_t RSSI_Target,
```

```
                                          uint8_t RSSI_Hysteresis,
                                          int8_t Initial_TX_Power,
                                          uint8_t RSSI_Filtering_Coefficient);
/* Report a path loss threshold crossing on a connection */
void hci_le_path_loss_threshold_event(uint16_t Connection_Handle,
                                      uint8_t Current_Path_Loss,
                                      uint8_t Zone_Entered);
/* Report the transmit power level on a connection */
void hci_le_transmit_power_reporting_event(uint8_t Status,
                                           uint16_t Connection_Handle,
                                           uint8_t Reason,
                                           uint8_t PHY,
                                           int8_t Transmit_Power_Level,
                                           uint8_t Transmit_Power_Level_Flag,
                                           int8_t Delta);
```

Note: Refer to Bluetooth® Low Energy APIs html documentation for a detailed description of commands and events.

4.19 Direction finding commands and events

The BlueNRG-LPS device supports both methods used for the direction finding feature (angle of arrival and angle of departure) by managing:

- the constant tone extension (CTE) inside a packet
- the antenna switching mechanism for both AoA and AoD.

The BlueNRG-LPS device supports both methods used for the direction finding feature (angle of arrival and angle of departure) by managing:

BlueNRG-LPS can append a new field called constant tone extension to the link layer packets: this field provides a constant frequency signal against which IQ sampling can be performed, in line with Bluetooth® core specification v5.1.

To support some features of the direction finding, the controller needs to support antenna switching. BlueNRG-LPS is able to control an external antenna switch by using a control signal called ANTENNA_ID. This signal is a 7-bit antenna identifier (ANTENNA_ID[6:0]) indicating the antenna number, to be used during a certain time slot, and it is provided in real time by the internal sequencer. With a 7-bit identifier, the maximum number of antennas that can be controlled is 128.

In a AoD transmitter or in a AoA receiver, the radio needs to switch antenna during the CTE field of the packet. For this purpose, the ANTENNA_ID signal can be enabled on some I/Os, by programming them in the associated alternate function. A specific command `aci_hal_set_antenna_switch_parameters()` is provided inside the software development kit for more convenience to perform the proper ANTENNA_ID signal configuration.

This signal needs to be provided to an external antenna switching circuit, where ANTENNA_ID[0] is the least significant bit and ANTENNA_ID[6] the most significant bit of the antenna identifier to be used.

Bluetooth® core specification v5.1 define new HCI commands and events to control constant tone extension and IQ sampling. In particular, the antenna switching pattern is controlled by the Antenna_IDs parameter of the following HCI commands:

- `hci_le_set_connectionless_cte_transmit_parameters()`
- `hci_le_set_connectionless_iq_sampling_enable()`
- `hci_le_set_connection_cte_transmit_parameters()`
- `hci_le_set_connection_cte_receive_parameters()`

Each antenna ID specified in the pattern corresponds to the ANTENNA_ID number generated by the BlueNRG-LPS internal sequencer, which is sent out on PB[6:0].

The Table 70. Direction finding commands and events provides the list of HCI commands and events, which handles the packet transmission and reception with the proper CTE information. These commands allow the configuration of several CTE aspects as follows:

- CTE length
- CTE type
- Length of the antenna switching pattern
- Antenna IDs
- Slot duration.

Table 70. **Direction finding commands and events**

Output parameter	Command/event and parameters
tBleStatus	**aci_hal_set_antenna_switch_parameters**(uint8_t Antenna_IDs, uint8_t Antenna_ID_Shift, uint8_t Default_Antenna_ID, uint8_t RF_Activity_Enable)
tBleStatus	**hci_le_set_connectionless_cte_transmit_parameters** (uint8_t Advertising_Handle, uint8_t CTE_Length, uint8_t CTE_Type, uint8_t CTE_Count, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[])
tBleStatus	**hci_le_set_connectionless_cte_transmit_enable**(uint8_t Advertising_Handle, uint8_t CTE_Enable)
tBleStatus	**hci_le_set_connectionless_iq_sampling_enable** (uint16_t Sync_Handle, uint8_t Sampling_Enable, uint8_t Slot_Durations, uint8_t Max_Sampled_CTEs, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[])
tBleStatus	**hci_le_set_connection_cte_receive_parameters** (uint16_t Connection_Handle, uint8_t Sampling_Enable, uint8_t Slot_Durations, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[])
tBleStatus	**hci_le_set_connection_cte_transmit_parameters** (uint16_t Connection_Handle, uint8_t CTE_Type, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[])
tBleStatus	**hci_le_connection_cte_request_enable** (uint16_t Connection_Handle, uint8_t Enable, uint16_t CTE_Request_Interval, uint8_t Requested_CTE_Length, uint8_t Requested_CTE_Type)
tBleStatus	**hci_le_connection_cte_response_enable** (uint16_t Connection_Handle, uint8_t Enable)
tBleStatus	**hci_le_read_antenna_information** (uint8_t *Supported_Switching_Sampling_Rates, uint8_t *Num_Antenna, uint8_t *Max_Switching_Pattern_Length, uint8_t *Max_CTE_Length)
tBleStatus	**hci_le_transmitter_test_v3** (uint8_t TX_Channel, uint8_t Test_Data_Length, uint8_t Packet_Payload, uint8_t PHY, uint8_t CTE_Length, uint8_t CTE_Type, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[])
tBleStatus	**hci_le_transmitter_test_v4** (uint8_t TX_Channel, uint8_t Test_Data_Length, uint8_t Packet_Payload, uint8_t PHY, uint8_t CTE_Length, uint8_t CTE_Type, uint8_t Switching_Pattern_Length, uint8_t Antenna_IDs[], int8_t Transmit_Power_Level)
void	**hci_le_connectionless_iq_report_event** (uint16_t Sync_Handle, uint8_t Channel_Index, uint16_t RSSI, uint8_t RSSI_Antenna_ID, uint8_t CTE_Type, uint8_t Slot_Durations, uint8_t Packet_Status, uint16_t Periodic_Event_Counter, uint8_t Sample_Count, Samples_t Samples[])
void	**hci_le_connection_iq_report_event** (uint16_t Connection_Handle, uint8_t RX_PHY, uint8_t Data_Channel_Index, uint16_t RSSI, uint8_t RSSI_Antenna_ID, uint8_t CTE_Type, uint8_t Slot_Durations, uint8_t Packet_Status, uint16_t Connection_Event_Counter, uint8_t Sample_Count, Samples_t Samples[])
void	**hci_le_cte_request_failed_event** (uint8_t Status, uint16_t Connection_Handle)

Refer to the Bluetooth® Low Energy v3.1a or later commands and event documentation for a detailed description. Direction finding features are supported only on the BlueNRG-LPS device (not on the BlueNRG-LP devices).

Bluetooth® specifications allow the direction-finding enabled packets to be used in both connectionless and connection-oriented communication, as described on next Section 4.19.1 Connectionless scenario and Section 4.19.2 Connection-oriented scenario.

4.19.1 Connectionless scenario

In a connectionless scenario, the CTE field is sent inside advertising packets in a periodic advertising train. To send such type of advertising packets, the host needs to follow these steps:

1. Configure extended advertising with `aci_gap_set_advertising_configuration()`.
2. Set advertising data, if needed, with `aci_gap_set_advertising_data()`. Data can be changed also while advertising is enabled.
3. Configure periodic advertising on the same advertising handle with `aci_gap_set_periodic_advertising_configuration()`.
4. Set periodic advertising data, if needed, with `aci_gap_set_periodic_advertising_data()`. Data can be changed also while periodic advertising is enabled.
5. Configure CTE transmit parameters with `hci_le_set_connectionless_cte_transmit_parameters()`. The type of CTE, the number of packets with CTE and, in case of AoD, also the switching pattern must be specified.
6. Enable the transmission of the CTE with `hci_le_set_connectionless_cte_transmit_enable()`.
7. Enable periodic advertising with `aci_gap_set_periodic_advertising_enable()`. The periodic advertising actually starts after enabling extended advertising.
8. Enable extended advertising with `aci_gap_set_advertising_enable()`.

On the scanner side, to receive the periodic advertising and extract the IQ samples from the CTE field, the host must execute the following steps:

1. Configure extended scanning with `aci_gap_set_scan_configuration()`.
2. Start an extended scanning procedure with `aci_gap_start_procedure()`.
3. Synchronize with the periodic advertising train by using `aci_gap_periodic_advertising_create_sync()`.
4. Stop scanning procedure, if no longer needed, with `aci_gap_terminate_proc()`.
5. Enable IQ sampling with `hci_le_set_connectionless_iq_sampling_enable()`. The switching pattern needs to be specified for AoA CTE.
6. Process the IQ samples received with `hci_le_connectionless_iq_report_event()`.

Note:
- *To receive the Connectionless IQ Report events on the scanner, they must be enabled also through the `hci_le_set_event_mask()` command.*
- *Some other events, which may be needed in some of the steps on the scanner, are not enabled by default. It is suggested to enable at least:*
  - *LE Extended Advertising Report event*
  - *LE Periodic Advertising Sync Established event*
  - *LE Periodic Advertising Report event*
  - *LE Periodic Advertising Sync Lost event*
- *The IQ samples algorithms are not defined by Bluetooth Core specifications*
- *Reception of IQ reports can be disabled with `hci_le_set_connectionless_iq_sampling_enable()` when no longer needed.*

4.19.2 Connection-oriented scenario

In a connection-oriented scenario, both peripheral and central can send a request (the LL_CTE_REQ PDU) to the peer device to send a packet (the LL_CTE_RSP PDU) containing a CTE field.

Assuming there is a connection between the two devices, these are the steps that the host needs to follow.

On the device that wants to receive the packets with CTE field:

1. Configure CTE receive parameters with `hci_le_set_connection_cte_receive_parameters()`. Slot duration for IQ sampling and the antenna switching pattern need to be specified only if the device wants to request an AoA CTE type.

2. Enable the automatic sending of requests with `hci_le_connection_cte_request_enable()`. With this function, the CTE type (AoA/AoD) must be specified.
3. Process the IQ samples received with the `hci_le_connection_iq_report_event()`

On the device that wants to support the sending of a CTE field during the connection:

1. Configure CTE transmit parameters with `hci_le_set_connection_cte_transmit_parameters()`. The type of CTE must be specified, together with the antenna pattern to be used in case of AoD.
2. Enable the CTE response with `hci_le_connection_cte_response_enable()`. From this moment, the controller automatically responds to CTE requests from the peer.

Note:
- *To receive the connection IQ report events, they must be enabled through the `hci_le_set_event_mask()` command.*
- *Bluetooth core specifications do not define tThe IQ sample algorithms.*
- *Reception of IQ reports can be disabled with `hci_le_connection_cte_request_enable()` when no longer needed.*

4.20 Enhanced ATT commands and events

The following tables detail the new enhanced ATT commands and events added by Bluetooth® Low Energy stack v3.2 or later.

Refer to the APIs and events html documentation for a detailed description of the APIs and the related parameters.

Table 71. EATT commands

Command name
aci_gatt_clt_read_multiple_var_len_char_value
aci_gatt_srv_multi_notify
aci_gatt_eatt_clt_confirm_indication
aci_gatt_eatt_clt_disc_all_char_desc
aci_gatt_eatt_clt_disc_all_char_of_service
aci_gatt_eatt_clt_disc_all_primary_services
aci_gatt_eatt_clt_disc_char_by_uuid
aci_gatt_eatt_clt_disc_primary_service_by_uuid
aci_gatt_eatt_clt_execute_write_req
aci_gatt_eatt_clt_find_included_services
aci_gatt_eatt_clt_prepare_write_req
aci_gatt_eatt_clt_read
aci_gatt_eatt_clt_read_long
aci_gatt_eatt_clt_read_multiple_char_value
aci_gatt_eatt_clt_read_multiple_var_len_char_value
aci_gatt_eatt_clt_read_using_char_uuid
aci_gatt_eatt_clt_write
aci_gatt_eatt_clt_write_char_reliable
aci_gatt_eatt_clt_write_long
aci_gatt_eatt_clt_write_without_resp
aci_gatt_eatt_srv_init
aci_gatt_eatt_srv_multi_notify
aci_gatt_eatt_srv_notify
aci_gatt_eatt_srv_resp

Table 72. EATT events

Event name
aci_att_clt_read_multiple_var_len_resp_event
aci_eatt_clt_exec_write_resp_event
aci_eatt_clt_find_by_type_value_resp_event
aci_eatt_clt_find_info_resp_event
aci_eatt_clt_prepare_write_resp_event
aci_eatt_clt_prepare_write_resp_event
aci_eatt_clt_read_blob_resp_event
aci_eatt_clt_read_by_group_type_resp_event
aci_eatt_clt_read_by_type_resp_event
aci_eatt_clt_read_multiple_resp_event
aci_eatt_clt_read_multiple_var_len_resp_event
aci_eatt_clt_read_resp_event
aci_eatt_srv_exec_write_req_event
aci_eatt_srv_prepare_write_req_event
aci_gatt_eatt_clt_disc_read_char_by_uuid_resp_event
aci_gatt_eatt_clt_error_resp_event
aci_gatt_eatt_clt_indication_event
aci_gatt_eatt_clt_notification_event
aci_gatt_eatt_clt_proc_complete_event
aci_gatt_eatt_proc_timeout_event
aci_gatt_eatt_srv_attribute_modified_event
aci_gatt_eatt_srv_confirmation_event
aci_gatt_eatt_srv_read_event
aci_gatt_eatt_srv_write_event

4.21 L2CAP enhanced credit flow APIs and events

The following tables detail the new L2CAP credit flow commands and events added by Bluetooth® Low Energy stack v3.2.

Table 73. L2CAP enhanced credit flow commands

Command name	Short description
aci_l2cap_ecfc_connection_req	L2CAP_CREDIT_BASED_CONNECTION_REQ packets are sent to create and configure up to five L2CAP channels between two devices. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 3, Part A, L2CAP, Sec. 4.25
aci_l2cap_ecfc_connection_resp	Response packet used to respond to incoming L2CAP_CREDIT_BASED_CONNECTION_REQ packet. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 3, Part A, L2CAP, Sec. 4.26

Command name	Short description
aci_l2cap_ecfc_reconfigure_req	Request the reconfiguration of MTU or MPS values related to a set of channels with respect to values used when channels were created or last reconfigured. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 3, Part A, L2CAP, Sec. 4.27
aci_l2cap_ecfc_reconfigure_resp	Response packet used to respond to incoming L2CAP_CREDIT_BASED_RECONFIGURATION_REQ packet. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 3, Part A, L2CAP, Sec. 4.28

Table 74. L2CAP enhanced credit flow events

Event name	Short description
aci_l2cap_cos_connection_event	Event that indicates an incoming L2CAP Connection request related to either LE Credit Based Flow Control Mode or Enhanced Credit Based Flow Control Mode.
aci_l2cap_ecfc_reconfiguration_event	Event that indicates an incoming L2CAP Reconfiguration request related to a set of channels handled through the Enhanced Credit Based Flow Control model.

Note: *On Bluetooth® Low Energy stack v3.2, the `aci_l2cap_cfc_connection_event` is replaced by `aci_l2cap_cos_connection_event`.*

4.22 Bluetooth® Low Energy isochronous channels APIs and events

The following tables detail the new HCI commands and events related to ISOAL, BIG/BIS and CIG/CIS commands added byBluetooth® Low Energy stack v3.2 for supporting the isochronous channels feature.

Table 75. ISOAL HCI commands

Command name	Short description
hci_le_iso_read_test_counters	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.113
hci_le_iso_receive_test	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.112
hci_le_iso_test_end	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.114
hci_le_iso_transmit_test	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.111
hci_le_read_iso_link_quality	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.116
hci_le_read_iso_tx_sync	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.96
hci_le_remove_iso_data_path	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.110
hci_le_setup_iso_data_path	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.109
hci_tx_iso_data	Function to send isochronous data to the Controller

Table 76. **BIG/BIS HCI commands**

Command name	Short description
hci_le_big_create_sync	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.106
hci_le_big_terminate_sync	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.107
hci_le_create_big	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.103
hci_le_create_big_test	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.104
hci_le_terminate_big	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.105

Table 77. **CIG/CIS HCI commands**

Command name	Short description
hci_le_accept_cis_request	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.101
hci_le_create_cis	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.99
hci_le_reject_cis_request	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.102
hci_le_remove_cig	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.100
hci_le_set_cig_parameters	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.97
hci_le_set_cig_parameters_test	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.98

Table 78. **BIG/BIS HCI events**

Command name	Short description
hci_le_create_big_complete_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.27.
hci_le_big_sync_established_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.29.
hci_le_big_sync_lost_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.30.
hci_le_biginfo_advertising_report_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.31.
hci_le_terminate_big_complete_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.28.
hci_le_cis_established_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.25.
hci_le_cis_request_event	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.26.

4.23 New Bluetooth® Low Energy stack v3.2 or later HCI APIs and events

Table 79. New HCI commands related to some link layer controller features

Command name	Short description
hci_le_read_buffer_size_v2	Returns the size of the HCI buffers used by the LE Controller to buffer data that is to be transmitted. [v2] command adds support for ISO data packets. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.2.
hci_le_set_data_related_address_changes	Used to specify circumstances when the Controller shall refresh any Resolvable Private Address used by the advertising set identified by the Advertising_Handle parameter, whether or not the address timeout period has been reached. This command may be used while advertising is enabled. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.122.
hci_le_set_host_feature	The HCI_LE_Set_Host_Feature command is used to set or clear a bit controlled by the Host in the Link Layer Feature Set stored in the Controller. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.115.
hci_read_afh_channel_assessment_mode	Command to read the value for the AFH_Channel_Assessment_Mode parameter, that controls whether the Controller’s channel assessment scheme is enabled or disabled. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.3.53.
hci_write_afh_channel_assessment_mode	Command to read the value for the AFH_Channel_Assessment_Mode parameter, that controls whether the Controller’s channel assessment scheme is enabled or disabled. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.3.54.
hci_le_set_default_subrate	Command used by the Host to set the initial values for the acceptable parameters for subrating requests. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.123.
hci_le_subrate_request	Command used by a Central or a Peripheral to request a change to the subrating factor and/or other parameters applied to an existing connection using the Connection Subrate Update procedure. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.124.
hci_read_connection_accept_timeout	Command to read the value for the Connection_Accept_Timeout configuration. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.3.13.
hci_write_connection_accept_timeout	Command to write the value for the Connection_Accept_Timeout configuration. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.3.14.
hci_le_request_peer_sca	The HCI_LE_Request_Peer_SCA command requests the Sleep Clock Accuracy of the peer device.

Command name	Short description
	Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.8.108.

Table 80. **New HCI events related to some link layer controller features**

Event name	Short description
`hci_le_request_peer_sca_complete_event`	Event that indicates the completion of HCI_LE_Request_Peer_SCA command. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.31.
`hci_le_subrate_change_event`	Event that indicates completion of the Connection Subrate Update procedure and that some parameters of the connection have changed. Refer to the Bluetooth® Low Energy specification v.5.3, Vol. 4, Part E, HCI, Sec. 7.7.65.35.

5 Bluetooth® LE stack v3.x scheduler

This section aims to provide the user the fundamentals of the new Bluetooth® LE stack v3.x time scheduler.

The goal is to help the user to set up connections, scanning, and advertising procedures in multilink scenarios so that the measured performance (for example, throughput, latency, robustness against connection drops) is as much coherent as possible with the expected one.

Advantages/disadvantages with respect to the `Bluetooth® LE stack v2.x time scheduler are also be described.

All link layer (LL) **radio tasks** (that is, connection, advertising, or scanning) can be represented in a time base through sequences of slots (that is time windows where associated events occur).

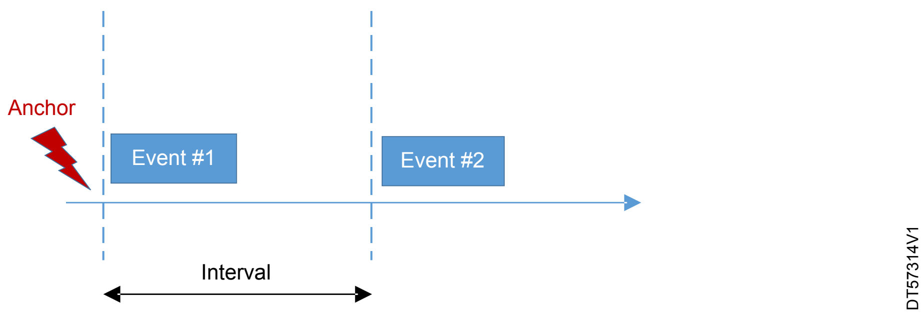
Each slot occurrence of a radio task can be represented through:

- an **anchor**, that is the exact time the slot is expected to start;
- a **length**, that is the expected time duration of the slot.

Radio tasks can be classified as:

- **periodic**, where slots are repeated at fixed time **intervals** (for example, connect events)
- **asynchronous**, where:
  - Slots are NOT repeated (usually referred as one-shot slots)
  - Slots are repeated but NOT at fixed time intervals (for example, advertising events)

Figure 19. Example of periodic radio task



Since multiple radio tasks can be concurrently active on a device, the associated slots can overlap in time. In the following, a slot overlap is referred as a **collision**.

In case of collision, only one of the colliding slots is granted access to the air and is referred as a **scheduled slot**. Colliding slots that do not have access to the air is referred as **skipped slots**.

The **time scheduler** is an SW module that provides the following functionalities:

- At the time, a new radio task is started by the host, it computes the very first anchor of the associated slot sequence (that is, the exact time the first Rx/Tx event associated, with the radio task, is expected to “grab the air”)
- At the time a slot ends, according to the rules specified by the Bluetooth® standard (that is, in case of a master connection event, at the time, the CE_Length is exceeded):
  1. it computes the next anchor (if any) of the associated radio task.
  2. In case of multiple radio tasks currently active on the device, it computes, per each radio task, the next anchor (if any) that is, in the future, with respect to the current time.
  3. In case of multiple radio tasks currently active on the device, it selects which radio task has to be served next, based on a chronological order.
  4. In case the first slot (in chronological order) collides with other slots, it schedules one of the colliding slots based on a priority mechanism.

Number of radio tasks

When a Bluetooth® activity is started, for example a new advertising set or a connection, usually a single radio task is needed. However, there are some Bluetooth® activities, which need more than one radio task.

Table 81. **Bluetooth® LE stack activity and radio tasks**

Bluetooth® LE stack activity	Number of radio tasks
Advertising set, legacy advertising	1
Advertising set, extended advertising	2
Scan, extended ad and scan disabled	1
Scan, extended adv and scan enabled	1 + NumOfAuxScanSlots
Connection	1
Periodic advertising	1
Periodic advertising synchronization	1

In particular, the number of radio tasks for each advertising set is two if extended advertising PDUs are used. Moreover, if an extended scan is enabled through the modular configuration (CONTROLLER_EXT_ADV_SCAN_ENABLED), the scan requires a number of slots equal to 1 + NumOfAuxScanSlots, which is a parameter of the initialization structure passed to BLE_STACK_Init() function. The NumOfAuxScanSlots is equal to the number of radio tasks that can be allocated simultaneously to scan on the secondary advertising physical channel. Once a packet on the primary advertising physical channel is received, which points to a packet on the secondary advertising physical channel, a radio task is allocated and, if served, it triggers a scan on the secondary advertising physical channel. In a presence of several advertisers using extended advertising events, the higher the number of auxiliary scan slots is, the higher the chance to receive extended advertising events.

The total number of radio tasks that can be allocated is specified through NumOfLinks field of the initialization structure BLE_STACK_InitTypeDef. In addition to this constraint, the number of simultaneous periodic advertising synchronization slots is limited by the NumOfSyncSlots field.

5.1 Limitations of the Bluetooth® LE stack v2.x time scheduler

The Bluetooth® LE stack v2.x time scheduler was built around the concept of the *anchor period*.

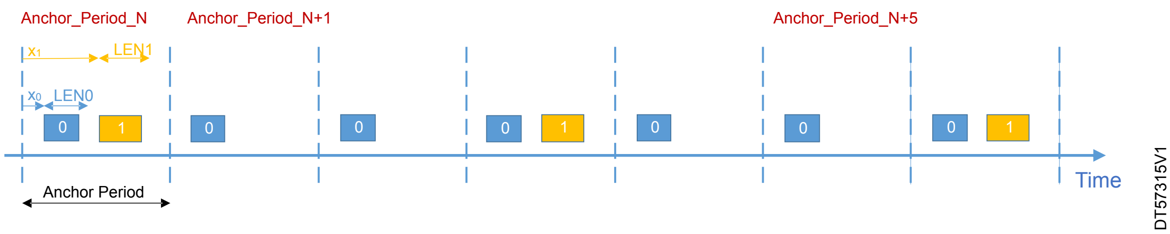
The *anchor period* is the minimum time interval used to represent all the periodic “master” radio tasks running on a device. Note that we refer to “master” radio tasks to indicate Bluetooth® 4.2 standard advertising, scanning, and master connection events (slave connection events are excluded).

According to this definition, the time base is represented as a repetition of anchor periods, where all master radio tasks must have an interval that is an integer multiple of the anchor period.

The anchor period is started at the time the first “master” radio task is started by the host and it is equal to the radio task interval. The anchor period can be decreased (that is, if a new master radio task is started with an interval that is an integer submultiple of the current anchor period), but it cannot be increased after being decreased.

In the following picture, there is a representation of two master radio task allocations in an anchor period. Here radio task #0 has the same interval as the anchor period while radio task #1 has an interval that is three times the anchor period.

Figure 20. **Example of two radio tasks allocation in an anchor period**



Here is a list of the main advantages of the Bluetooth® Low Energy stack v2.x time scheduler:

1. The scheduler forces the application to choose suitable parameters for new slots in order to avoid collisions with existing slots. If new *master* slots cannot be allocated because they would collide with other slots, the scheduler just returns an error. In this condition, where no collisions can occur between master slots, the maximum throughput is guaranteed.
2. Collisions can occur only with slave connect slots. Here the priority mechanism used by the time scheduler is a simple round-robin approach

Here is a list of the main limitations of the Bluetooth® Low Energy stack v2.x time scheduler:

1. Since the sum of the allocated master radio task lengths must fit within the anchor period, there is a limit to the number of concurrent master radio tasks.
2. Since new master radio tasks must have an interval that is a multiple integer of the existing anchor period, the user has a very reduced flexibility in choosing the periodicity of a new link layer slot (that is, connect, advertising, and scan intervals).
3. Once reduced (due to the start of a new radio task), the anchor period cannot be increased anymore, thus resulting in an additional limitation to the number of concurrent radio tasks.
4. Asynchronous radio tasks are not compatible with the anchor period approach because they are not periodic.

Almost all the link layer features introduced by the Bluetooth 5.0 standard (for example, advertising extensions) are characterized by asynchronous events that are not compatible with the anchor period approach of the Bluetooth® Low Energy stack v2.x time scheduler; this is the main reason that led to the implementation of a new time scheduler.

The probability of having multi slots collisions is also proportional to the number of concurrent asynchronous radio tasks, which makes the simple round-robin mechanism of the Bluetooth® Low Energy stack v2.x time scheduler no more appropriate to guarantee adequate performance.

5.2 The new Bluetooth® Low Energy stack v3.x time scheduler

To support to all new link layer features introduced by the Bluetooth® 5.0 standard as well as to allow the user to implement application scenarios with different QoS requirements, a new Bluetooth® Low Energy stack v3.x time scheduler removes the main constraints of the previous implementation.

In the Bluetooth® Low Energy stack v3.x time scheduler, all radio tasks are treated as they were asynchronous, that is they are excluded from the anchor period representation.

One significant advantage with respect to the Bluetooth® Low Energy stack v2.x time scheduler is that an LL activity, started by the host, is never rejected because of its periodicity (that is, connection, advertising, or scan interval) or event duration (that is, connection CE_Length, scan window, etc.).

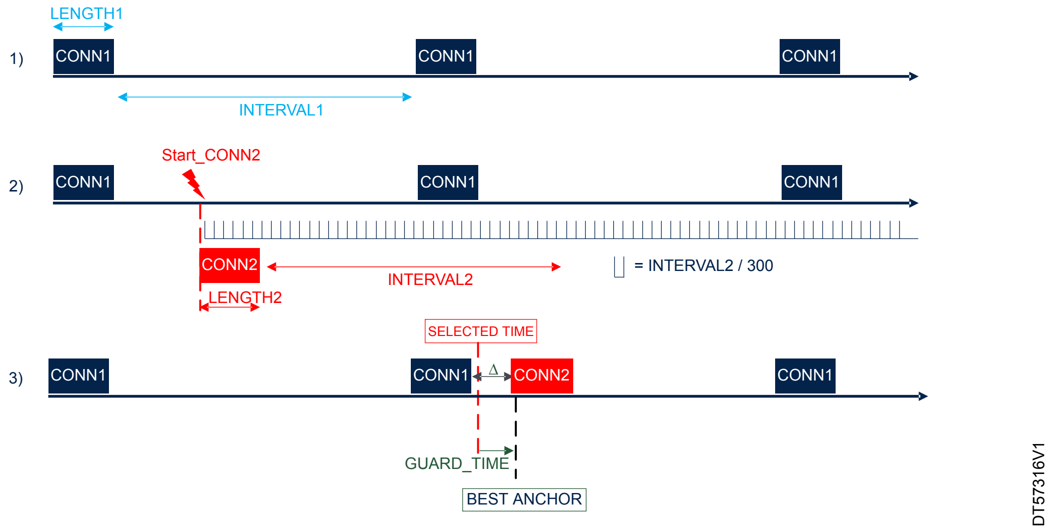
The main drawback of this new approach is that master task slots (that is, Bluetooth® connection, advertising, and scanning events) can now potentially collide each other. For this reason, the users could occasionally experience reduced performances (for example, reduced throughput). Performance can be increased if certain guidelines are followed (refer to Section 5.4 User guidelines).

5.3 The prescheduler

The prescheduler is a software module of the Bluetooth® Low Energy stack v3.x time scheduler responsible for the selection of the “best anchor” in a newly started master connection.

The best anchor is the start time of the first connection slot that minimizes the collisions with existing slots, for an observation period that is 10 times the connection interval of the newly started connection.

Figure 21. Prescheduler operation example



The above image provides an example of prescheduler operation:

1. The device is already running a radio task (`CONN1`), with its own slot length and interval.
2. At `Start_CONN2` time, the host issues a command to start a new connection (that is, master role) and the prescheduler is requested to select the best anchor.
   a. A grid of potential best anchors is created, starting at `Start_CONN2` time and ending at

$$(Start\_CONN2 + 10 \times INTERVAL2)$$

   b. Points in the grid are spaced in time:

$$\frac{INTERVAL2}{300}$$

   c. The number of collisions with existing slots is computed per each point in the grid.
3. A point in the grid is selected (in the picture, `SELECTED_TIME`) that:
   a. minimizes the collisions with existing slots,
   b. minimizes as much as possible the time gap with existing slots (for a more efficient usage of the bandwidth).
4. The best anchor (`BEST_ANCHOR` in the picture) is equal to (`SELECTED_TIME` + `GUARD_TIME`), where *`GUARD_TIME`* is an estimation of the maximum time spent by the time scheduler to reschedule a new radio task (that is, to program the associated next slot anchor) after the current scheduled slot is ended.

The prescheduler selection of the best anchor takes a time that increases linearly with the number of currently active radio tasks. This time interval is in the range of 100 ms (with eight radio tasks currently active) to 1 second (with 128 radio tasks currently active).

The advertising radio tasks are not considered by the prescheduler operations. These are usually not periodic tasks, except for periodic advertising slots, which, however, are not considered by the prescheduler since their duration can vary significantly.

Scanning is also not considered by the prescheduler because it is handled as a background operation, with a low priority. A scan window (that is, the time the device is continuously receiving on a specific channel) is divided into scan chunks with a duration of 10 ms. Whenever a scan chunk is expected to collide with another slot, the scan chunk is shortened to try to avoid the collision. Since, by design, the duration of a scan chunk cannot be reduced below a certain minimum value (that is hardcoded in the firmware), the scan slot can still collide with other slots.

5.4 User guidelines

As already described, while the Bluetooth® Low Energy stack v2.x prescheduler always avoids collisions among master radio tasks by adding constraints to the allocation of new slots, the Bluetooth® Low Energy stack v3.x prescheduler approach aims instead at minimizing the probability of collisions with existing radio tasks, without preventing to allocate slots, which are expected to collide. For this reason, even if the stack never forbids allocations of radio slots, the user experiences bad performances if connection parameters are not properly chosen.

Here are some basic recommendations to maximize the performance of the Bluetooth® Low Energy stack v3.x in scenarios where the device can be connected as a central to multiple peripherals.

1. Choose the same connection interval (`Conn_Interval`) for all the master connections such that:

$$\text{Conn\_interval} > \Sigma_i(\text{CE\_Lengthi} + \text{guard\_time} + \frac{Conn\_Interval}{300})$$

Where:

- `CE_Lengthi` is the `CE_Length` of the i-th connection. It is always a multiple of 625 µs.
- `guard_time` is equal to 350 µs.

Note:
- *`Conn_Interval/300` is included in the formula to consider the worst-case impact of the time granularity used when calculating the best anchor point.*
- *The minimum `CE_Length` parameter provided by the application is ignored. The prescheduler only uses the maximum `CE_Length`.*
- *The maximum connection interval parameter provided by the application is ignored. The prescheduler only uses the minimum `CE_Length`.*
- *The actual value of `CE_Length`, used by the stack, can differ from the one passed by the application: the minimum value used by the stack depends on the support to the Data Length Extension (enabled through modular configuration), and by the initiating PHY. These values are reported in the table below.*

Table 82. Minimum CE_Length

Data length extension	Initating PHY	Minimum CE Length (ms)
DISABLED	1M/2M	1.25
ENABLED	1M/2M	5
DISABLED	CODED	5.625
ENABLED	CODED	34.375

2. Choose a combination of `CE_Length` and `Conn_Interval` that allows the required throughput.

The throughput can be calculated with the following formula:

$$\text{Throughput} = \frac{num\_bits\_CE}{Conn\_Interval\_ms}$$

where `num_bits_CE` is the number of bits transmitted in the same connection event and `Conn_Interval_ms` is the connection interval in units of 625 µs.

The number of bits, transmitted in the same connection event, depends on the maximum length of the link layer PDU and on the number of packets per connection event, which, in turn, depends on the `CE_length` parameter.

If `Conn_Interval` is given, the following formula gives the number of packets to be transmitted per connection event to achieve the desired throughput:

$$num\_packets\_CE = \left\lceil \frac{(Throughput \times Conn\_Interval\_ms)}{(bytes\_per\_packets \times 8)} \right\rceil$$

where:

- `Throughput` is in kbps
- `Conn_Interval_ms` is in ms
- `bytes_per_packets` is the number of bytes contained in an LLn payload

The maximum length of an LL PDU that can be sent or received is established during the connection by using the data length update procedure. The LL payload length can be set either through the `hci_le_write_suggested_default_data_length()` command or through `hci_le_set_data_length()` command. The length of an LL PDU payload is 27 octets by default and can be increased up to 251 octets if the data length extension feature is supported (in the Bluetooth® Low Energy stack 3.x, this is done through the `CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED` macro).

The duration of an LL PDU is reported in the table below and it includes transmission of the MIC (message integrity check) field. `Payload_length` does not include MIC.

Table 83. LL PDU duration (including MIC)

PHY	Empty LL PDU duration (µs)	Max LL PDU duration (µs)
LE_1M	80	(payload_length + 14) x 8
LE_2M	44	(payload_length + 15) x 4
LE_CODED (S=8)	720	976 + payload_length x 64

A connection event is made by several packets exchanged between master and slave. The CE length can be calculated with this formula:

$$CE\_Length = \left\lceil num\_packets\_CE \times \frac{(TX\_PDU\_Duration + RX\_PDU\_Duration + 2 \times T\_IFS)}{625} \right\rceil$$

Where:

- `TX_PDU_Duration` is the value in microseconds needed to transmit the data PDU (see Table 83). If this value is not known, the maximum possible value should be used.
- `RX_PDU_Duration` is the value in microseconds to receive a data PDU (see Table 83). If this value is not known, the maximum possible value should be used.
- `T_IFS` is equal to 150 µs.
- `CE_Length` is in a unit of 625 µs and it is rounded to the next integer value.

If the resulting value of the CE length is greater than the connection interval or, more generally, the condition (2) is not satisfied, the connection interval needs to be increased and a number of packets per connection event needs to be recalculated. If the connection interval cannot be increased, the desired throughput cannot be reached.

5.4.1 Guidelines example

It is requested to have 100 kbps of unidirectional throughput (LL data throughput), with a connection interval of 50 ms. If the data length extension is enabled (LL payload up to 251 octets), data is sent only in one direction (empty packet on RX), and PHY is `LE_1M`:

$$num\_packets\_CE = \frac{(100 \times 50)}{(251 \times 8)} = 3$$

$$CE\_Length = 3 \times \frac{(2120 + 80 + 300)}{625} = 12$$

`CE_Length_ms` = 7.5 ms

The CE length is less than the connection interval, meaning that an increase is not needed.

3. The `Conn_Interval` should be chosen to be less than the maximum latency required by all the connections.

In case a master connection can tolerate a bigger latency than `Conn_Interval`, the user may choose, for that connection, a connection interval that is an integer multiple of `Conn_Interval`.

5.4.2 Three master connections example

If we suppose that the application needs to start three master connections with a different throughput and latency requirements, for example:

- **Application 1**:
  - Throughput = 10 kbps, bidirectional
  - Latency = 100 ms

- **Application 2**:
  - Throughput = 20 kbps, bidirectional
  - Latency = 100 ms
- **Application 3**:
  - Throughput = 30 kbps, bidirectional
  - Latency = 200 ms

The maximum connection interval value that guarantees the required latency to all master connections is equal to 100 ms:

$$Connection\_Interval \leq 100ms \tag{2}$$

Now, we have to verify whether the 100 ms of connection interval is also appropriate to contain all master connections `CE_Lengths`.

If the data length extension is not enabled and PHY is 1 Mbps:

- Application 1 has to transmit (10 x 100)/(27 x 8) = 5 data PDUs (27 bytes each) per each connection event: *CE_Lenght1 = 5 x ((27+14)x8 + (27+14)x8 + 2x150)/625 = 8* (that is, 5 ms).
- Application 2 has to transmit (20 x 100)/(27 x 8) = 10 data PDUs (27 bytes each) per each connection event: *CE_Lenght2 = 10 x ((27+14)x8 + (27+14)x8 + 2x150)/625 = 16,* (that is, 10 ms).
- Application 3 has to transmit (30 x 100)/(27 x 8) = 14 data PDUs (27 bytes each) per each connection event: *CE_Lenght3 = 14 x ((27+14)x8 + (27+14)x8 + 2x150)/625 = 22,* (that is, 13.75 ms).

The minimum connection interval that fits all connections `CE_Lengths` is:

`Conn_Interval_Min_allowed` = *SUM(Max_CE_Length(i)/2 + GUARD) = (4 + 1) + (8 + 1) + (11 + 1) = 26* (that is, 32.5 ms).

Where `GUARD` = 1, assuming the maximum number of radio tasks supported by the device is not greater than 25.

Since *Conn_Interval_Min_allowed ≤ 100 ms*, a connection interval of 100 ms can be used for all the master connections to ensure the appropriate performance.

5.5 The priority mechanism

At the time a slot ends, the time scheduler is responsible to compute the next slot (and consequently the next radio task) to be served.

In the case of multiple radio tasks currently active on the device, the next slot in chronological order could collide with other slots. In this case, the next slot selection is based on a priority mechanism.

The priority mechanism consists in:

1. Assigning priorities to each radio task based on the following formula:

$$priority = \min(priority\_\max, priority\_\min + \log 2(skipped\_slots + 1))$$

where:

- *priority_max* and *priority_min* are constants that are specific of each radio task (that is, different for advertising, scanning, and connection radio tasks)
- *skipped_slots* indicates the current number of consecutive slots, associated to a specific radio task that have been skipped due to collisions
  - According to this definition, the priority associated to a radio task is a dynamic value that is computed every time the associated slot is either served or skipped.

2. Selecting, among the colliding radio tasks, the highest priority radio task.

5.6 Interactions with the ISR robustness mechanism

The ISR robustness mechanism has been implemented to make the firmware robust against delays in the execution of the interrupt service routine (ISR).

A delayed ISR is typically a consequence of the radio interrupt that has been disabled by the application for too long time and, if not properly recovered, can lead to an unpredictable behavior of the stack.

In particular, the most critical situations from the LL protocol perspective are the so-called back-to-back operations, that is, events (receptions or transmissions) that must trigger other events within a very short time spacing (that is, 150 µs).

A typical example of a back-to-back operation is when the device receives a scan request that triggers the transmission of a scan response. In this time-critical situation, a delayed execution of the ISR, associated with the reception of the scan request, can cause the hardware registers, associated with the transmission of the scan response, not to be properly set at the time the transmission occurs, thus causing an unpredictable behavior (for example, a malformed packet is transmitted).

With the ISR robustness mechanism, the firmware can detect the situation where the ISR is served too late with respect to the associated radio event. Whenever this specific condition is detected by the firmware:

1. The associated back-to-back radio event is skipped
2. The Bluetooth® Low Energy stack v3.x time scheduler is called that selects the next radio task to be served
3. A hardware error event is pushed to the application to inform that something wrong happened.

The application is not expected to handle the hardware error event in a specific way but, upon receiving it, the user gets an indication that the application design was not at 100% correct in terms of masking time of the radio interrupt.

Revision history

Table 84. **Document revision history**

Date	Revision	Changes
03-Aug-2020	1	Initial release.
29-Jan-2021	2	Updated Physical layer, LE 2M PHY, LE Coded PHY, Link layer (LL), Bluetooth LE packet, Extended advertising, Advertising sets, Bluetooth LE addresses, Bluetooth LE 2 Mbps and Coded Phy, Bluetooth® Low Energy stack v3.x scheduler. Added Extended scan, Events for extended adv & scan, The new Bluetooth® Low Energy stack v3.x time scheduler, The pre-scheduler, Guidelines for the users, Example of a scenario with three master connections, The priority mechanism and Interactions with the ISR robustness mechanism. Updated Table 1
06-May-2021	3	Added Periodic advertising and periodic advertising sync transfer, Randomized advertising, GATT caching, Bluetooth LE power control, Periodic advertising modes, Periodic Advertising mode, Periodic Advertising Synchronizability mode, Periodic Advertising Synchronization Establishment procedure, LE power control and path loss monitoring. Updated Figure 1, Bluetooth LE stack library framework, Bluetooth LE stack init and tick APIs and Bluetooth LE stack init and tick APIs and GAP API interface. Added Table 2, Table 1 and Table 2 Minor text changes.
07-Jun-2021	4	Updated Initialization phase and main application loop
06-Apr-2022	5	Updated Introduction, Bluetooth LE stack library framework, Bluetooth LE stack init and tick APIs, The Bluetooth LE stack v3.x application configuration, Initialization phase and main application loop and List of acronyms and abbreviations. Updated Figure 1. Added Direction finding commands and events, Direction finding with angle of arrival (AoA), Direction finding with angle of departure (AoD), In-phase and quadrature (IQ), Direction finding commands and events and References.
29-May-2023	6	Added the following sections: • Section 2.12 Enhanced Attribute protocol • Section 2.13 L2CAP enhanced credit-based flow control • Section 2.14 Bluetooth® Low Energy isochronous channels • Section 2.15 Bluetooth® Low Energy connection subrating • Section 2.16 Bluetooth® Low Energy channel classification • Section 4.20 Enhanced ATT commands and events • Section 4.21 L2CAP enhanced credit flow APIs and events • Section 4.22 Bluetooth® Low Energy isochronous channels APIs and events • Section 4.23 New Bluetooth® Low Energy stack v3.2 or later HCI APIs and events • Section 1 General information Applied minor changes to all the document.

Date	Revision	Changes
		Updated the following sections: • Section 4.1 Initialization phase and main application loop • Section 1.2 List of acronyms and abbreviations • Section 1.1 References
13-Nov-2023	7	Updated the following: • Section 5.4 User guidelines • Section 5.3 The prescheduler • Section 4 Designing an application with the Bluetooth® Low Energy stack v3.x • Section 5.5 The priority mechanism

Contents

List of tables

List of figures