PM0271

Programming manual

Guidelines for Bluetooth® Low Energy stack programming on STM32WB/ STM32WBA MCUs

Introduction

This document provides programming guidelines for developers to use when developing Bluetooth® Low Energy (BLE) applications using STM32WB and STM32WBA BLE stack APIs, and related event callbacks.

The document describes the STM32WB and STM32WBA Bluetooth® Low Energy stack library framework, API interfaces, and event callbacks. These allow access to the BLE functions provided by the STM32WB and STM32WBA system-on-chip.

It covers some fundamental concepts of BLE technology. These associate STM32WB and STM32WBA BLE stack APIs, parameters, and related event callbacks to the BLE protocol stack features. The user must have a basic knowledge of the BLE technology and its main features.

For more information about the STM32WB and STM32WBA series and the BLE specifications, refer to Section 7: Reference documents at the end of this document.

The STM32WB and STM32WBA are very low power BLE single-mode network processors. They are compliant with Bluetooth® specification v6.0 and support client or server role.

The manual is structured as follows:

- Fundamentals of BLE technology
- STM32WB and STM32WBA BLE stack library APIs and the event callback overview
- How to design an application using the STM32WB and STM32WBA library APIs and event callbacks. Some examples are given using a "switch case" event handler rather than using the event callbacks framework.

1 General information

This document applies to the STM32WB series dual-core Arm®-based microcontrollers and to the STM32WBA series single-core Arm®-based microcontrollers.

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

2 Bluetooth® Low Energy technology

The Bluetooth® Low Energy (BLE) 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 was 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 a fraction of the power of standard Bluetooth® products only and enable devices with coin cell batteries to be wireless connected to standard Bluetooth® 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 BLE stack architecture

Bluetooth® Low Energy technology has been formally adopted by the Bluetooth® core specification version 4.0 (in Section 7: Reference documents).

The BLE 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 BLE technology are:

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

In particular, BLE 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 BLE technology is designed to address two alternative implementations:

- Smart device
- Smart ready device

Smart devices support the BLE standard only. It is used for applications in which low power consumption and coin cell batteries are the key point (as sensors).

Smart ready devices support both BR/EDR/HS and BLE standards (typically a mobile or a laptop device).

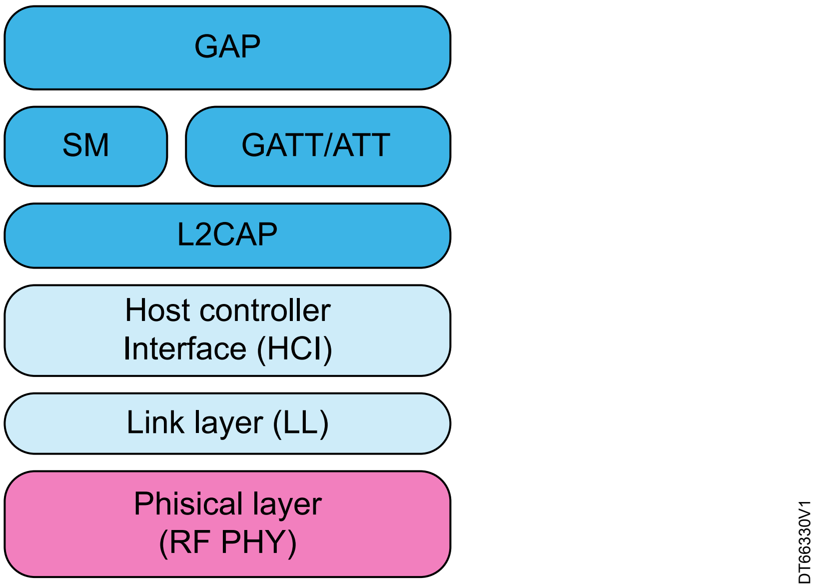
The BLE stack consists of two components:

- Controller
- Host

The controller includes the physical layer and the link layer.

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

Figure 2. BLE stack architecture



The Bluetooth® specifications v4.1, v4.2, v5.0, v5.1, v5.2, v5.3, v5.4, and v6.0 have been released with new supported features:

- **STM32WB and STM32WBA current features supported on v4.1:**
  - Multiple roles simultaneously support
  - Support simultaneous advertising and scanning
  - Support being server of up to two clients simultaneously
  - Privacy V1.1
  - Low duty cycle directed advertising
  - Connection parameters request procedure
  - 32 bits UUIDs
  - L2CAP connection oriented channels
- **STM32WB and STM32WBA current features supported on V4.2:**
  - LE data length extension
  - Address resolution
  - LE privacy 1.2
  - LE secure connections
- **STM32WB and STM32WBA current feature supported on V5.0:**
  - LE 2M PHY
  - LE CODED PHY (applies only to the STM32WBA series)
- **STM32WB and STM32WBA current feature supported on V5.1:**
  - GATT caching
- **STM32WB and STM32WBA current feature supported on V5.2:**
  - Enhanced ATT
- **STM32WB and STM32WBA current feature supported on V5.4:**
- – Encrypted advertising data
  - LE GATT security levels characteristics

2.2 Physical layer

The physical layer is a 1 Mbps adaptive frequency-hopping Gaussian frequency shift keying (GFSK) radio or 2Mbit/s 2-level Gaussian frequency shift keying (GFSK). 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.

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

$$(2.402 + 0.002 * k)\ GHz,\ where\ \ 0 \leq\ k\ \ \leq 39 \quad (1)$$

There are two channel types:

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

Table 1. BLE RF channel types and frequencies

Channel index	RF center frequency	Channel type
0 (channel 37)	2402 MHz	Advertising channel
1 (channel 0)	2404 MHz	Data channel
2 (channel 1)	2406 MHz	Data channel
….	….	Data channel
11 (channel 10)	2424 MHz	Data channel

Channel index	RF center frequency	Channel type
12 (channel 38)	2426 MHz	Advertising channel
13 (channel 11)	2428 MHz	Data channel
14 (channel 12)	2430 MHz	Data channel
….	….	Data channel
38 (channel 36)	2478 MHz	Data channel
39 (channel 39)	2480 MHz	Advertising channel

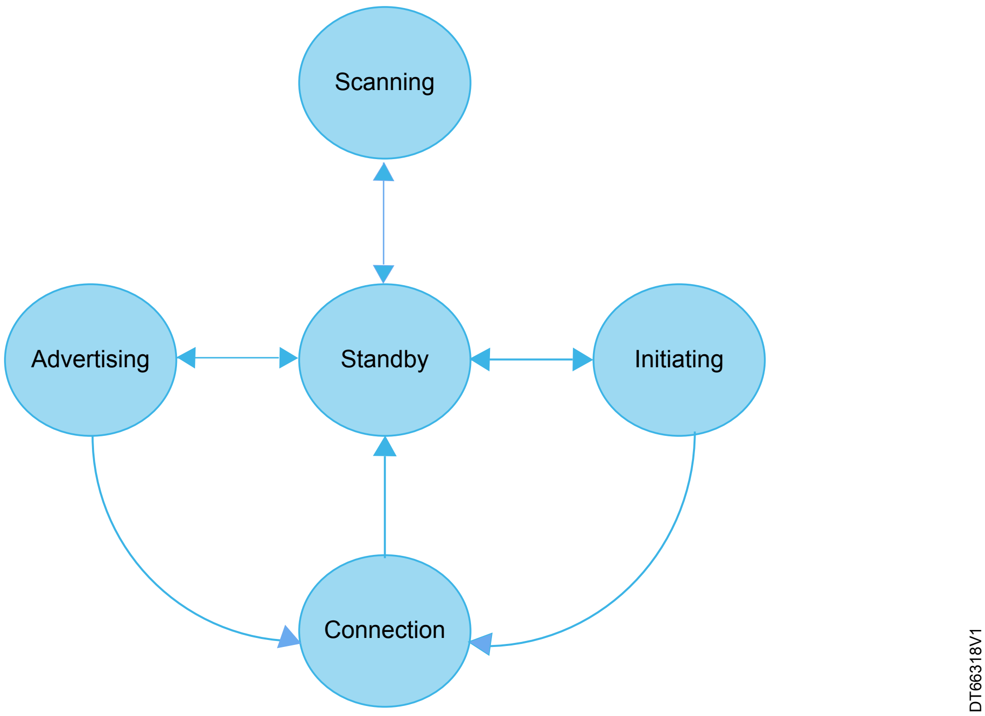
BLE is an adaptive frequency hopping (AFH) technology that can only use a subset of all the available frequencies in order to avoid all frequencies used by other no-adaptive technologies. This allows moving from a bad channel to a known good channel by using a specific frequency hopping algorithm, which determines the next good channel to be used.

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. Link layer state machine



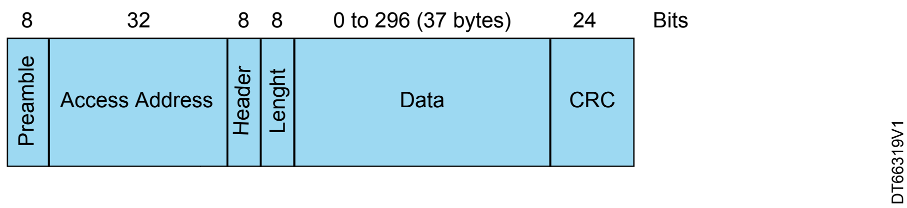
- Standby: the device does not transmit or receive packets
- Advertising: the device broadcasts advertisements in advertising channels (it is called an advertiser device)
- Scanning: the device looks for advertiser devices (it is called a scanner device)
- Initiating: the device initiates connection to the advertiser device
- Connection: the initiator device is in client role: it communicates with the device in the server role and it defines timings of transmissions
- Advertiser device is in server role: it communicates with a single device in client role

2.3.1 BLE packets

A packet is a labeled data that is transmitted by one device and received by one or more other devices.

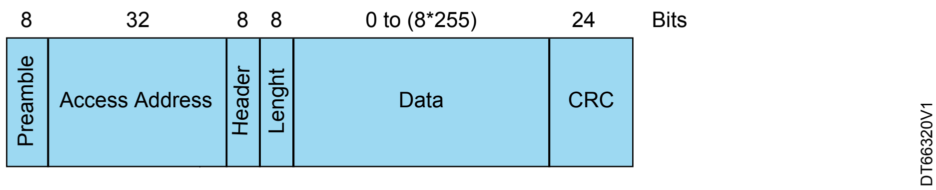
The BLE data packet structure is described below.

Figure 4. Packet structure



The BLE specification v4.2 defines the LE data packet length extension feature which extends the link layer PDU of LE from 27 to 251 bytes of data payload.

Figure 5. Packet structure with LE data packet length extension feature



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

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

Table 2. Advertising data header content

Advertising packet type	Reserved	Tx address type	Rx address type
(4 bits)	(2 bits)	(1 bit)	(1 bit)

- The advertising packet type is defined as shown in Table 3:

Table 3. **Advertising packet types**

Packet type	Description	Notes
ADV_IND	Connectable undirected advertising	Used by an advertiser when it wants another device to connect to it. Device can be scanned by a scanning device, or go into a connection as a peripheral device on connection request reception.
ADV_DIRECT_IND	Connectable directed advertising	Used by an advertiser when it wants a particular device to connect to it. The ADV_DIRECT_IND packet contains only advertiser's address and initiator address.
ADV_NONCONN_IND	Non-connectable undirected advertising	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. Device simply sends advertising packets on related channels, but it does not want to be connectable or scanned by any other device.
ADV_SCAN_IND	Scannable undirected advertising	Used by an advertiser which wants to allow a scanner to require more information from it. The device cannot connect, but it is discoverable for advertising data and scan response data.
SCAN_REQ	Scan request	Used by a device in scanning state to request addition information to the advertiser.
SCAN_RSP	Scan response	Used by an advertiser device to provide additional information to a scan device.
CONNECT_REQ	Connection request	Sent by an initiating device to a device in connectable/discoverable mode.

The advertising event type determines the allowable responses:

Table 4. **Advertising event type and allowable responses**

Advertising event type	Allowable response	
	SCAN_REQ	CONNECT_REQ
ADV_IND	YES	YES
ADV_DIRECT_IND	NO	YES
ADV_NONCONN_IND	NO	NO
ADV_SCAN_IND	YES	NO

- Data packet header:

Table 5. **Data packet header content**

Link layer identifier	Next sequence number	Sequence number	More data	Reserved
(2 bits)	(1 bit)	(1 bit)	(1 bit)	(3 bits)

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

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

For a detailed description of advertising and data header contents and types refer to the Bluetooth® specification [Vol 2], in Section 7: Reference documents.

- Length: number of bytes on data field

Table 6. **Packet length field and valid values**

Packet	Length field bits
Advertising packet	6 bits, with valid values from 0 to 37 bytes
Data packet	5 bits, with valid values from 0 to 31 bytes 8 bits, with valid values from 0 to 255 bytes, with LE data packet length extension

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

2.3.2 Advertising state

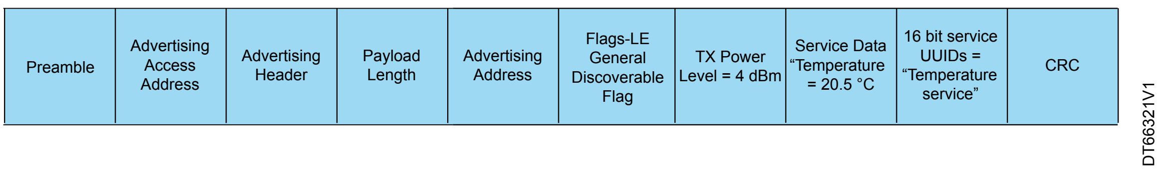
Advertising states allow link layer to transmit advertising packets and also to respond with scan responses to scan requests coming from those devices, which are actively scanning.

An advertiser device can be moved to a standby state by stopping the advertising.

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

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

Figure 6. **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 BLE)
- 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 are non-zero (it is not included in scan response).

The following advertising parameters can be set before enabling advertising:

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

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

2.3.3 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, 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 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

2.3.4 Connection state

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

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

- Access address used in the connection in order to identify communications on a physical link
- CRC initialization value
- Transmit window size (timing window for the first data packet)
- Transmit window offset (transmit window start)
- Connection interval (time between two connection events)
- Server latency (number of times server 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 server device at connection event)

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

The allowed timing ranges are summarized in Table 7. Connection request timing intervals :

Table 7. 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 server device enters in receiver mode and waits for a packet from the client device. If no packet is received within this time, the peripheral leaves receiver mode, and it tries one connection interval again later. When a connection is established, a client has to transmit a packet to the server on every connection event to allow peripheral to send packets to the central. Optionally, a server device can skip a given number of connection events (server latency).

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

2.4 Host controller interface (HCI)

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

2.5 Logical link control and adaptation layer protocol (L2CAP)

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

2.6 Attribute protocol (ATT)

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

An attribute is a 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 security level required 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, client can try to authenticate it by using the security manager and send back the request
  – Authorization permissions (no authorization, authorization): this is a property of a server which can authorize a client to access or not to a set of attributes (client cannot resolve an authorization error)

Table 8. 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 protocol to discover, read and write attributes on a peer device. It implements the peer-to-peer client-server protocol between an attribute server and an attribute client as follows:

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

Attributes exposed by a server can be discovered, read, and written by the client, and they can be indicated and notified by the server as described in Table 9. Attribute protocol messages:

Table 9. 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 new value (no confirmation)
Indication	Server	Server indicates to client new value (it always causes a confirmation)
Confirmation	Client	Confirmation to an indication

2.7 Security manager (SM)

The BLE link layer supports 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 basis for a trusted relationship or a (single) secure connection. There are some methods used to perform the pairing procedure. Some of these methods provide protections against

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

The pairing on BLE 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 BLE specification v4.2, the LE secure connection pairing model has been defined. The new security model main features are:

1. Key exchange process uses the elliptical curve Diffie-Hellman (ECDH) algorithm: this allows keys to be exchanged over an unsecured channel and to protect against passive eavesdropping attacks (secretly listening through a sniffing device to the communication of other devices)
2. A new method called “numeric comparison” has been added to the 3 methods already available with LE legacy pairing

The 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 10. Combination of input/output capabilities on a BLE 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 2 keys:

- Temporary key (TK): a 128-bit temporary key which is used to generate short-term key (STK)
- Short-term key (STK): a 128-bit temporary key used to encrypt a connection following pairing

Pairing procedure is a three-phase process.

Phase 1: pairing feature exchange

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

Phase 2: short-term key (STK) generation

The pairing devices first define a temporary key (TK), by using one of the following key generation methods

1. The out-of-band (OOB) method, which uses out of band communication (for instance, NFC) for TK agreement. It provides 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 authentication (MITM protection)
3. Just works: this method does not provide authentication and protection against man-in-the-middle (MITM) attacks

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

Table 11. **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 the phase 2 is completed, up to three 128-bit keys can be distributed by messages encrypted with the STK key:

1. Long-term key (LTK): it is used to generate the 128-bit key used for Link Layer encryption and authentication
2. Connection signature resolving key (CSRK): a 128-bit key used for the data signing and verification performed at the ATT layer
3. Identity resolving key (IRK): a 128-bit key used to generate and resolve random addresses

LE secure connections

LE secure connection pairing methods use and generate one key:

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

Pairing procedure is a three-phase process:

Phase 1: pairing feature exchange

The two connected devices communicate their input/output capabilities by using the pairing request message. This message also contains a bit stating if out-of-band data are available and the authentication requirements. The information exchanged in phase 1 is used to select which pairing method is used on 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 which uses out of band communication to set up the public key. This method is selected if the out-of-band bit in the pairing request/response is set at least by one device, otherwise the IO capabilities of the devices must be used to determine which other method could be used (Passkey entry, Just Works or numeric comparison)
2. Just Works: this method is not authenticated, and it does not provide any protection against man-in-the-middle (MITM) attacks
3. Passkey entry method: this method is authenticated. User passes six numeric digits. This six-digit value is the base of the device authentication
4. Numeric comparison: this method is authenticated. Both devices have IO capabilities set to either display Yes/No or keyboard display. The two devices compute a 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 user that his device is connected with the proper 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 12. 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 central and peripheral:

- 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. When the number of bonded devices allowed is exceeded, the previous bonded devices information is deleted and only the new one is saved.

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 which can simply capture some packets and send them later as they are, without any understanding of packet content: the receiver device simply checks the packet counter and discards it since its frame counter is less than the latest received good packet).

2.8 Privacy

A device that always advertises with the same address (public or static random), can be tracked by scanners. 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 BLE stacks up to v4.1, the private addresses are resolved and generated by the host. In Bluetooth® v 4.2, the privacy feature has been updated from version 1.1 to version 1.2. On BLE stack v 4.2, 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 non-connectable mode uses non-resolvable or resolvable private addresses.

To connect to a central, the undirected connectable mode only should be used if host privacy is used. If the 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 or every time the advertising is re-enabled. 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 or every time the advertising is re-enabled. 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 Device filtering

BLE allows a way to reduce the number of responses from the devices in order to reduce power consumption, since this implies less transmissions and less interactions between controller and upper layers. The filtering is implemented by a filter accept list. When the filter accept list is enabled, those devices, which are not in this list, are ignored by the link layer.

Before Bluetooth® v 4.2, the device filtering could not be used, while privacy was used by the remote device. Thanks to the introduction of link layer privacy, the remote device identity address can be resolved before checking whether it is in the filter accept list or not.

By setting the "Filter_Duplicates" mode to 1, the user can activates advertising filtering at LL level. It works as described below.

For STM32WBx, the LL maintains two sets of four buffers each: one for four adversing indications addresses and the other for four scan responses addresses.

When an advertising indication packet is received, its address (6 bytes) is compared to the four stored ones. If it matches with one of the four addresses, the packet is discarded. If it does not match, the indication is reported to upper layers and its address is stored in the buffers while the oldest address is removed from the buffers.

The same process respectively applies to the scan responses.

For STM32WBAx, the duplicate filter list has a depth of 10 devices.

2.9 Generic attribute profile (GATT)

The generic attribute profile (GATT) defines a framework for using the ATT protocol, and it is used for services, characteristics, descriptors discovery, characteristics reading, writing, indication and notification.

On GATT context, when two devices are connected, there are two devices 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 central, peripheral role. The central, peripheral roles define how the BLE radio connection is managed, and the GATT client/server roles are determined by the data storage and flow of data.

As consequence, a peripheral device has to be the GATT server and a central device has not to be the GATT client.

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

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

2.9.1 Characteristic attribute type

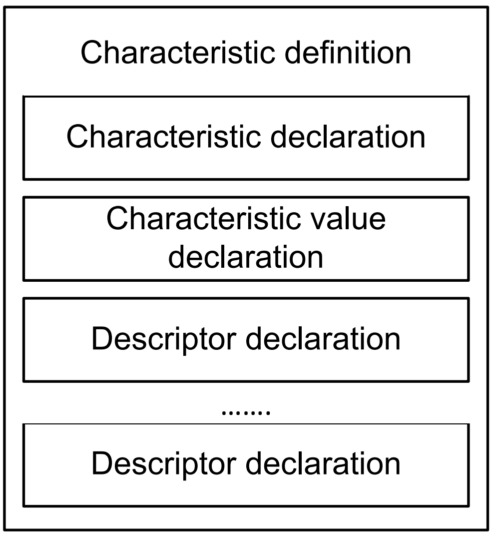
A characteristic is an attribute type which contains a single value and any number of descriptors describing the characteristic value that may make it understandable by the user.

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 7. **Example of characteristic definition**



A characteristic declaration is an attribute defined as shown in Table 13:

Table 13. Characteristic declaration

<table>
<tr><th>Attribute handle</th><th>Attribute type</th><th>Attribute value</th><th>Attribute permissions</th></tr>
<tr><td rowspan="3">0xNNNN</td><td rowspan="3">0x2803
(UUID for characteristic attribute type)</td><td>Characteristic value properties (read, broadcast, write, write without response, notify, indicate, …). Determine how characteristic value can be used or how characteristic descriptor can be accessed</td><td rowspan="3">Read only,
no authentication, no authorization</td></tr>
<tr><td>Characteristic value attribute handle</td></tr>
<tr><td>Characteristic value UUID (16 or 128 bits)</td></tr>
</table>

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

Table 14. 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 making it understandable by 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 notification or indication (provided that the characteristic property allows notification or indication)
4. Server characteristic configuration: optional descriptor
5. Characteristic presentation format: it allows the characteristic value presentation format to be defined through some fields as format, exponent, unit name space, description in order to correctly display the related value (example temperature measurement value in ºC format)
6. Characteristic aggregation format: It allows several characteristic presentation formats to be aggregated.

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

2.9.3 Service attribute type

A service is a collection of characteristics which operate 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 other services that are called secondary services.

A service is defined as follows:

Table 15. 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 includes declaration follows the service declaration and any other attributes of the server.

Table 16. 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 discovered and how to use them.

The following procedures are available:

- Discovery procedures (Table 17. Discovery procedures and related response events)
- Client-initiated procedures (Table 18. Client-initiated procedures and related response events)
- Server-initiated procedures (Table 19. Server-initiated procedures and related response events)

Table 17. 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 characteristics descriptors	Find information response

Table 18. 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 19. Server-initiated procedures and related response events

Procedure	Response events
Notifications	No event is generated
Indications	Confirmation event

For a detailed description about the GATT procedures and related responses events refer to the Bluetooth® specifications in Section 7: Reference documents.

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 20. 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 peripheral. It is on connectable advertising mode. Supports all LL control procedures; encryption is optional	M	M	Watch
Central	Always a central. 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 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 modes 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 procedures 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 21. 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)	Broadcasts data can be detected by a device using the observation procedure	Broadcaster

Table 22. **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 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 23. **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 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 devices 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 24. **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 Table 25. GAP observer procedure:

Table 25. **GAP observer procedure**

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

Table 26. GAP discovery procedures

Procedure	Description	Notes	Role
Limited discoverable procedure	It is used for discovery 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 peripheral devices in general ad 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

In order to implement the name discovery GAP procedure, the user can do the following:

- Call the ACI_GAP_CREATE_CONNECTION command
- Wait for HCI_LE_CONNECTION_COMPLETE_EVENT
- Call the ACI_GATT_READ_USING_CHAR_UUID command with following parameters:
  - Start_Handle = 0x0001
  - End_Handle = 0xFFFF
  - UUID_Type = 1
  - UUID = DEVICE_NAME_UUID

Note: This can replace the former ACI_GAP_START_NAME_DISCOVERY_PROC command that is no more supported.

Table 27. GAP connection procedures

Procedure	Description	Notes	Role
Auto connection establishment procedure	Allows the connection with one or more devices in the directed connectable mode or the undirected connectable mode	It uses white lists	Central
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	Establish a connection with the host selected connection configuration parameters with a set of devices in the white list	It uses white lists and it scans by this white list	Central
Direct connection establishment procedure	Establish 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 28. 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 BLE profiles and applications

A service collects a set of characteristics and exposes the behaviour 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 which defines which services are required for a specific use case:

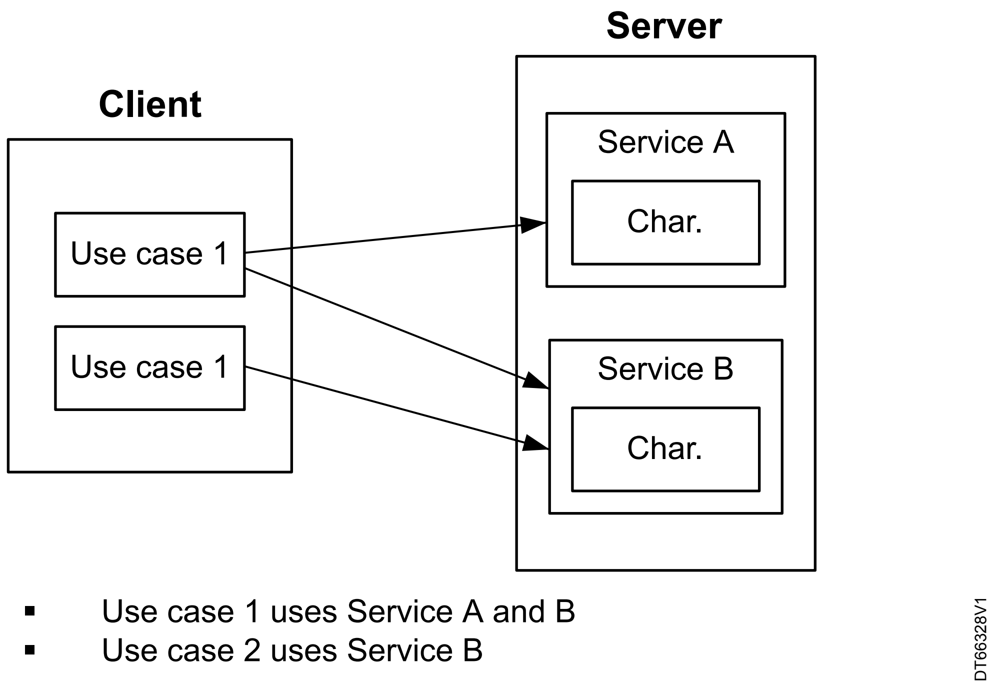
- Profile clients implement use cases
- Profile servers implement services

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, characteristics specification 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 8. Client and server profiles



2.11.1 Proximity profile example

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

Target

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

How it works

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

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

3 STM32WB and STM32WBA Bluetooth® Low Energy stacks

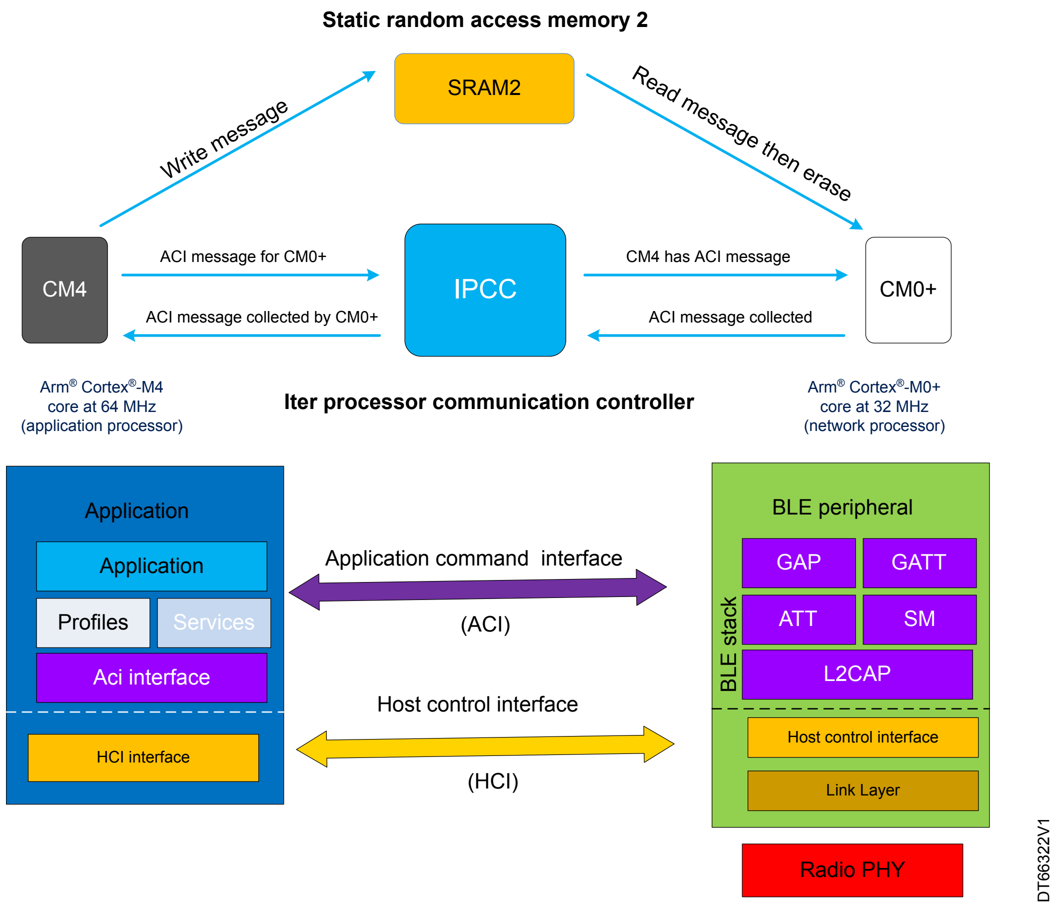
3.1 STM32WB BLE stack architecture and interface

STM32WB devices are network co-processors, which provide high-level interface to control its Bluetooth® Low Energy functionalities. This interface is called ACI (application command interface). STM32WB devices embed on Arm Cortex-M0, respectively and securely, the Bluetooth® smart protocol stack. As a consequence, no BLE library is required on the external micro-controller Arm Cortex-M4. The Inter Process Communication Controller (IPCC) interface communication protocol allows Cortex-M4 micro-controller to send and receive ACI commands to microcontroller Cortex-M0 co-processor. Current secure BLE stack is based on standard C library, in binary format. Before sending any BLE command, the Cortex-M4 shall first send the system command SHCI_C2_BLE_Init() to the Cortex-M0 to start the BLE stack. Refer to AN5289 for more description of the system command and BLE startup flow.

The BLE binary library provides the following functionalities:

- Stack APIs for BLE stack initialization, BLE stack application command interface (HCI command prefixed with hci_, and vendor specific command prefixed with aci_), Sleep timer access and BLE stack state machines handling
- Stack event callbacks inform user application about BLE stack events and sleep timer events
- Provides interrupt handler for radio IP

Figure 9. STM32WB stack architecture and interface between secure Arm Cortex-M0 and Arm Cortex-M4



3.2 STM32WBA BLE stack architecture and interface

STM32WBA devices are microcontrollers based on a single core (Arm®Cortex®-M33) that also use the application command interface (ACI interface).

STM32WBA BLE binaries libraries provide the following functionalities:

- Stack APIs for BLE stack initialization, BLE stack application command interface (HCI command prefixed with hci_, and vendor specific command prefixed with aci_), Sleep timer access, and BLE stack state machines handling
- Stack event callbacks inform the user application about BLE stack events. In order to handle these events, the User application places these events in FIFO before processing them.

3.3 BLE stack library framework

The BLE stack library framework allows commands to be sent to the STM32WB and STM32WBA SoC BLE stacks and it also provides definitions of BLE event callbacks. The BLE 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 ACI interface framework is provided within STM32WB and STM32WBA kits software package (refer to Section 7: Reference documents). The ACI interface framework contains the code that is used to send ACI commands between controller and host. It also provides definitions of device events. The ACI framework interface is defined by the following header files:

Table 29. BLE application stack library framework interface

File	Description
ble_hci_le.h	HCI library functions prototypes and error code definition.
ble_events.h	Header file that contains commands and events for STM32WB and STM32WBA FW stacks
ble_gatt_aci.h	Header file for GATT server definition
ble_l2cap_aci.h	Header file with L2CAP commands for STM32WB and STM32WBA FW stacks
ble_gap_aci.h	Header file for STM32WB and STM32WBA GAP layer
ble_hal_aci.h	Header file with HCI commands for STM32WB and STM32WBA FW stacks
ble_types.h	Header file with ACI definitions for STM32WB and STM32WBA FW stacks

4 Design an application using the STM32WB and STM32WBA BLE stacks

This section provides information and code examples about how to design and implement a Bluetooth® Low Energy application on STM32WB and STM32WBA devices using the BLE stack binary library.

User implementing a BLE application on STM32WB and STM32WBA devices has to go through some basic and common steps:

1. Initialization phase and main application loop
2. Services and characteristic configuration (on GATT server)
3. Create a connection: discoverable, connectable modes and procedures
4. STM32WB and STM32WBA events and events callback setup
5. Security (pairing and bonding)
6. NVM information for GATT and security record.
7. Service and characteristic discovery
8. Characteristic notification/indications, write, read
9. End to end Rx flow control using GATT
10. Basic/typical error conditions description

Note: In the following sections, some user applications "defines" are used to simply identify the device Bluetooth® Low Energy role (central, peripheral, client, and server).

To have more detail on BLE stack initialization and BLE stack NVM usage, refer to AN5289.

Table 30. User application defines for Bluetooth® Low Energy device roles

Define	Description
GATT_CLIENT	GATT client role
GATT_SERVER	GATT server role

Note:

1. *When performing the GATT_Init() and GAP_Init() APIs, STM32WB stack always adds two standard services: Attribute Profile Service (0x1801) with service changed characteristic and GAP service (0x1800) with device name and appearance characteristics.*
2. *The last attribute handles reserved for the standard GAP service is 0x000B when no privacy or host-based privacy is enabled on aci_gap_init() API, 0x000D when controller-based privacy is enabled on aci_gap_init() API.*

Table 31. GATT, GAP default services

Default services	Start handle	End handle	Service UUID
Attribute profile service	0x0001	0x0004	0x1801
Generic access profile (GAP) service	0x0005	0x000B	0x1800

Table 32. GATT, GAP default characteristics

Default services	Characteristic	Attribute handle	Char property	Char value handle	Char UUID	Char value length (bytes)
Attribute profile service	-	-	-	-	-	-
-	Service changed	0x0002	Indicate	0x0003	0x2A05	4
Generic access profile (GAP) service	-	-	-	-	-	-

Default services	Characteristic	Attribute handle	Char property	Char value handle	Char UUID	Char value length (bytes)
-	Device name	0x0006	Read | write without response | write | authenticated signed writes	0x0007	0x2A00	8
-	Appearance	0x0008	Read | write without response | write | authenticated signed writes	0x0009	0x2A01	2
-	Peripheral preferred connection parameters	0x000A	Read | write	0x000B	0x2A04	8
-	Central address resolution(1)	0x000C	Readable without authentication or authorization Not writable	0x000D	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 33. 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)
enable_Privacy	0x00 for disabling privacy 0x01 for enabling privacy 0x02 for enabling controller-based host privacy	-
device_name_char_len	-	Allows the length of the device name characteristic to be indicated

For a complete description of this API and related parameters refer to the BLE stack APIs and event documentations, in Section 7: Reference documents.

4.1 BLE addresses

The following device addresses are supported from the STM32WB and STM32WBA devices:

- Public address
- Random address
- Private address

Public MAC addresses (6 bytes- 48-bits address) uniquely identifies a BLE device, and they are defined by the Institute of electrical and electronics engineers (IEEE).

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

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

The public address remains static and unique and it is read only for the user.

If the user wants to program his own custom MAC address, a specific public address can be set by the application with a valid preassigned MAC address defined in the OTP.

The ACI command to set the MAC address is ACI_HAL_WRITE_CONFIG_DATA (opcode 0xFC0C) with command parameters as follows:

- Offset: 0x00 (0x00 identify the BTLE public address, for instance MAC address)
- Length: 0x06 (length of the MAC address)
- Value: 0xaabbccddeeff (48-bit array for MAC address)

The command ACI_HAL_WRITE_CONFIG_DATA should be sent after each power-up or reset.

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")}
```

The STM32WB and STM32WBA devices do not have a valid preassigned MAC address, but a unique serial number (read only for the user) that can be retrieved by unique device ID register (96 bits).

The following pseudocode example illustrates how to set a random address (valid only when GAP host is present):

```
uint8_t randaddr[] = {0xCC, 0xBB, 0xAA, 0xAA, 0xBB, 0xCC};
ret=aci_hal_write_config_data(CONF_DATA_RANDOM_ADDRESS_OFFSET,0x06,randaddr);
if(ret)PRINTF("Setting address failed.\n")}
```

Private addresses are used when privacy is enabled and according to the BLE specification. For more information about private addresses, refer to Section 2.7: Security manager (SM).

4.2 Set tx power level

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

`aci_hal_set_tx_power_level(En_High_Power, PA_Level)`

Follow a pseudocode example for setting the radio transmit power in Standard power with -20dBm output power for STM32WBA and -17dBm for STM32WB:

`ret= aci_hal_set_tx_power_level (0,4);`

For a complete description of this API and related parameters, refer to the BLE stack APIs and event documentation, and specifically the application note *STM32WB Bluetooth® Low Energy (BLE) wireless interface* (AN5270), referenced in Section 7: Reference documents.

4.3 Services and characteristics configuration

A service must be configured with a dedicated handle, attribute type, UUID, and permissions.

As already mentioned in Section 2.9: Generic attribute profile (GATT), it can have a lot of different characteristics, for this reason, it can have a lot of different handles.

A characteristic should always be attached/dependent to a service.

There are 4 possible different “TYPE” of characteristics with each its own handle, that could be selected one by one by customer:

- Characteristic extended properties
- Characteristic declaration attribute
- Characteristic value attribute
- Client characteristics configuration descriptor (CCCD).

The UUID for any CCCD is always the standard 16-bit UUIDCCCD (0x2902)

In order to add a service and 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 its 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).

3. By default two services are present and it is mandatory to include them along with dedicated characteristics as explained below:
   – The Generic access service:
     ◦ Service UUID 0x1800 along with its three mandatory characteristics:
       • Characteristic: Device name. UUID 0x2A00.
       • Characteristic: Appearance. UUID 0x2A01.
       • Characteristic: Peripheral preferred connection parameters. UUID 0x2A04.
   – The Generic attribute service.
     ◦ UUID 0x1801 along with one optional characteristic:
       • Characteristic: Service Changed. UUID 0x2A05.

A service can be added using the following command:

```
aci_gatt_add_service(uint8_t Service_UUID_Type,
                     Service_UUID_t *Service_UUID,
                     uint8_t Service_Type,
                     uint8_t Max_Attribute_Records,
                     uint16_t *Service_Handle);
```

This command returns the pointer to the service handle (`Service_Handle`), which is used to identify the service within the user application. A characteristic can be added to this service using the following command:

```
aci_gatt_add_char(uint16_t Service_Handle,
                  uint8_t Char_UUID_Type,
                  Char_UUID_t *Char_UUID,
                  uint8_t Char_Value_Length,
                  uint8_t Char_Properties,
                  uint8_t Security_Permissions,
                  uint8_t GATT_Evt_Mask,
                  uint8_t Enc_Key_Size,
                  uint8_t Is_Variable,
                  uint16_t *Char_Handle);
```

This command returns the pointer to the characteristic handle (`Char_Handle`), which is used to identify the characteristic within the user application.

The following pseudocode example illustrates the steps to be followed to add a service and two associated characteristic to a proprietary, non-standard profile.

```
/* Service and characteristic UUIDs variables.*/
Service_UUID_t service_uuid;
Char_UUID_t char_uuid;


tBleStatus Add_Server_Services_Characteristics(void)
{
    tBleStatus ret = BLE_STATUS_SUCCESS;
    /*
    The following 128bits UUIDs have been generated from the random UUID
    generator:
    D973F2E0-B19E-11E2-9E96-0800200C9A66: Service 128bits UUID
    D973F2E1-B19E-11E2-9E96-0800200C9A66: Characteristic_1 128bits UUID
    D973F2E2-B19E-11E2-9E96-0800200C9A66: Characteristic_2 128bits UUID
    */
    /*Service 128bits UUID */
    const uint8_t uuid[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe0,0xf2,0x73,0xd9};
    /*Characteristic_1 128bits UUID */
    const uint8_t charUuid_1[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe1,0xf2,0x73,0xd9};
    /*Characteristic_2 128bits UUID */
    const uint8_t charUuid_2[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe2,0xf2,0x73,0xd9};
    Osal_MemCpy(&service_uuid.Service_UUID_128, uuid, 16);
    /* Add the service with service_uuid 128bits UUID to the GATT server
    database. The service handle Service_Handle is returned.
```

```
    */
    ret = aci_gatt_add_service(UUID_TYPE_128, &service_uuid, PRIMARY_SERVICE,
                               6, &Service_Handle);
    if(ret != BLE_STATUS_SUCCESS) return(ret);
    Osal_MemCpy(&char_uuid.Char_UUID_128, charUuid_1, 16);

    /* Add the characteristic with charUuid_1128bitsUUID to the service
        Service_Handle. This characteristic has 20 as Maximum length of the
        characteristic value, Notify properties(CHAR_PROP_NOTIFY), no security
        permissions(ATTR_PERMISSION_NONE), no GATT event mask (0), 16 as key
        encryption size, and variable-length characteristic (1).
        The characteristic handle (CharHandle_1) is returned.
        */
    ret = aci_gatt_add_char(Service_Handle, UUID_TYPE_128, &char_uuid, 20,
                            CHAR_PROP_NOTIFY, ATTR_PERMISSION_NONE, 0,16, 1,
                            &CharHandle_1);
    if (ret != BLE_STATUS_SUCCESS) return(ret);
    Osal_MemCpy(&char_uuid.Char_UUID_128, charUuid_2, 16);

    /* Add the characteristic with charUuid_2 128bits UUID to the service
       Service_Handle. This characteristic has 20 as Maximum length of the
       characteristic value, Read, Write and Write Without Response properties,
       no security permissions(ATTR_PERMISSION_NONE), notify application when
       attribute is written (GATT_NOTIFY_ATTRIBUTE_WRITE) as GATT event mask ,
       16 as key encryption size, and variable-length characteristic (1). The
       characteristic handle (CharHandle_2) is returned.
    */
     ret = aci_gatt_add_char(Service_Handle, UUID_TYPE_128, &char_uuid, 20,
                         CHAR_PROP_WRITE|CHAR_PROP_WRITE_WITHOUT_RESP,
                         ATTR_PERMISSION_NONE, GATT_NOTIFY_ATTRIBUTE_WRITE,
                         16, 1, &&CharHandle_2);
    if (ret != BLE_STATUS_SUCCESS)return(ret) ;
}/*end Add_Server_Services_Characteristics() */
```

4.4 Create a connection: discoverable and connectable APIs

In order to establish a connection between a BLE GAP central device and a BLE GAP peripheral device, the GAP discoverable/connectable modes and procedures can be used as described in Table 34. GAP mode APIs, Table 35. GAP discovery procedure APIs and Table 36. Connection procedure APIs and by using the related BLE stack APIs provided.

GAP peripheral discoverable and connectable modes APIs

Different types of discoverable and connectable modes can be used as described by the following APIs:

Table 34. GAP mode APIs

API	Supported advertising event types	Description
aci_gap_set_discoverable()	0x00: connectable undirected advertising (default) 0x02: scannable undirected advertising 0x03: non-connectable undirected advertising	Sets the device in general discoverable mode. The device is discoverable until the device issues the `aci_gap_set_non_discoverable()` API.
aci_gap_set_limited_discoverable()	0x00: connectable undirected advertising (default) 0x02: scannable undirected advertising 0x03: non-connectable undirected advertising	Sets the device in limited discoverable mode. The device is discoverable for a maximum period of TGAP (lim_adv_timeout) = 180 seconds. The advertising can be disabled at any time by calling `aci_gap_set_non_discoverable()` API.
aci_gap_set_non_discoverable()	NA	Sets the device in non- discoverable mode. This command disables the LL advertising and sets the device in standby state.
aci_gap_set_direct_connectable()	NA	Sets the device in direct connectable mode. The device is directed connectable mode only for 1.28 seconds. If no connection is established within this duration, the device enters non-discoverable mode and advertising has to be enabled again explicitly.
aci_gap_set_non_connectable()	0x02: scannable undirected advertising 0x03: non-connectable undirected advertising	Puts the device into non- connectable mode.
aci_gap_set_undirect_connectable ()	NA	Puts the device into undirected connectable mode.

Table 35. GAP discovery procedure APIs

API	Description
aci_gap_start_limited_discovery_proc()	Starts the limited discovery procedure. The controller is commanded to start active scanning. When this procedure is started, only the devices in limited discoverable mode are returned to the upper layers.
aci_gap_start_general_discovery_proc()	Starts the general discovery procedure. The controller is commanded to start active scanning.

Table 36. Connection procedure APIs

API	Description
aci_gap_start_auto_connection_establish_proc()	Starts the auto connection establishment procedure. The devices specified are added to the white list of the controller and a create connection call is made to the controller by GAP with the initiator filter policy set to “use whitelist to determine which advertiser to connect to”.
aci_gap_create_connection()	Starts the direct connection establishment procedure. A create connection call is made to the controller by GAP with the initiator filter policy set to “ignore whitelist and process connectable advertising packets only for the specified device”.
aci_gap_start_general_connection_establish_proc()	Starts a general connection establishment procedure. The device enables scanning in the controller with the scanner filter policy set to “accept all advertising packets” and from the scanning results, all the devices are sent to the upper layer using the event `callback hci_le_advertising_report_event()`.
aci_gap_start_selective_connection_establish_proc()	It starts a selective connection establishment procedure. The GAP adds the specified device addresses into white list and enables scanning in the controller with the scanner filter policy set to “accept packets only from devices in white list”. All the devices found are sent to the upper layer by the event callback `hci_le_advertising_report_event()`.
aci_gap_terminate_gap_proc()	Terminate the specified GAP procedure.

4.4.1 Set discoverable mode and use direct connection establishment procedure

The following pseudocode example illustrates only the specific steps to be followed to let a GAP peripheral device be in general discoverable mode, and for a GAP central device direct 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_PUBADDR_OFFSET,CONFIG_DATA_PUBAD DR_LEN, bdaddr);
if(ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");

/*GAP Peripheral: general discoverable mode (and no scan response is sent)
*/

void GAP_Peripheral_Make_Discoverable(void )
{
 tBleStatus ret;
const charlocal_name[]=
{AD_TYPE_COMPLETE_LOCAL_NAME,'S','T','M','3','2','W','B','x','5','T','e','s','t'};/* disable
scan response: passive scan */ hci_le_set_scan_response_data (0,NULL);

 /* disable scan response: passive scan */
 hci_le_set_scan_response_data (0,NULL);

 /* Put the GAP peripheral in general discoverable mode:
    Advertising_Type: ADV_IND(undirected scannable and connectable);
    Advertising_Interval_Min: 100;
    Advertising_Interval_Max: 100;
    Own_Address_Type: PUBLIC_ADDR (public address: 0x00);
    Adv_Filter_Policy: NO_WHITE_LIST_USE (no whit list is used);
    Local_Name_Lenght: 14
    Local_Name: STM32WBx5Test;
    Service_Uuid_Length: 0 (no service to be advertised); Service_Uuid_List: NULL;
    peripheral_Conn_Interval_Min: 0 (peripheral connection internal minimum value);
    peripheral_Conn_Interval_Max:  0 (peripheral connection internal maximum value).
    */

  ret = aci_gap_set_discoverable(ADV_IND, 100, 100, PUBLIC_ADDR,
                            NO_WHITE_LIST_USE,
                            sizeof(local_name),
                            local_name,
                            0, NULL, 0, 0);
  if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

```
} /* end GAP_Peripheral_Make_Discoverable() */

/*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 in general discoverable mode using the
   following connection parameters:
   LE_Scan_Interval: 0x4000;
   LE_Scan_Window: 0x4000;
   Peer_Address_Type: PUBLIC_ADDR (GAP peripheral address type: public
   address);
   Peer_Address: {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
   Own_Address_Type:
   PUBLIC_ADDR (device address type);
   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 (peripheral 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).

  */

   tBDAddr GAP_Peripheral_address = {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
   ret= aci_gap_create_connection(0x4000, 0x4000, PUBLIC_ADDR,
                                  GAP_Peripheral_address,PUBLIC_ADDR, 40,
                                  40,
                                  0, 60, 2000 , 2000);
   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_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_gap_proc()`.*
3. *The last two parameters `Minimum_CE_Length` and `Maximum_CE_Length` of the `aci_gap_create_connection()` are the length of the connection event needed for the BLE connection. These parameters allows user to specify the amount of time the central has to allocate for a single peripheral so they must be wisely chosen. In particular, when a central connects to more peripherals, the connection interval for each peripheral must be equal or a multiple of the other connection intervals and user must not overdo the connection event length for each peripheral. Refer to Section 6: BLE multiple connection timing strategy for detailed information about the timing allocation policy.*

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

The following pseudocode example illustrates only the specific steps to be followed to let a GAP Peripheral device be in general discoverable mode, and for a GAP central device start a general discovery procedure in order to discover 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");

/* 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,'S','T','M','3','2','W','B','x','5',}; /* As scan
response data, a proprietary 128bits Service UUID is used.
     This 128bits data cannot be inserted within the advertising packet
     (ADV_IND) due its length constraints (31 bytes).
     AD Type description:
     0x11: length
     0x06: 128 bits Service UUID type
     0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x20,0x0c,
     0x9a,0x66: 128 bits Service UUID
   */
   uint8_t ServiceUUID_Scan[18]=
{0x11,0x06,0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x2,0x0c,0x9a,0x66};
/* Enable scan response to be sent when GAP peripheral receives scan
   requests from GAP Central performing general
   discovery procedure(active scan) */

 hci_le_set_scan_response_data(18,ServiceUUID_Scan);
 /* Put the GAP peripheral in general discoverable mode:
   Advertising_Type: ADV_IND (undirected scannable and connectable); Advertising_Interval_Min
: 100;
   Advertising_Interval_Max: 100;
   Own_Address_Type: PUBLIC_ADDR (public address: 0x00); Advertising_Filter_Policy: NO_WHITE_
LIST_USE (no whit list is used);
   Local_Name_Length: 8
   Local_Name: STM32WB;
   Service_Uuid_Length: 0 (no service to be advertised); Service_Uuid_List: NULL;
   peripheral_Conn_Interval_Min: 0 (peripheral connection internal minimum value); peripheral
_Conn_Interval_Max: 0 (peripheral connection internal maximum value).
  */
  ret = aci_gap_set_discoverable(ADV_IND, 100, 100, PUBLIC_ADDR,
                                NO_WHITE_LIST_USE,sizeof(local_name),
                                local_name, 0, NULL, 0, 0);
 if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

} /* end GAP_Peripheral_Make_Discoverable() */

/*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(active scan) using the following
   parameters:
   LE_Scan_Interval: 0x4000;
   LE_Scan_Window: 0x4000;
   Own_address_type: 0x00 (public device address);
   Filter_Duplicates: 0x00 (duplicate filtering disabled);
*/
ret =aci_gap_start_general_discovery_proc(0x4000,0x4000,0x00,0x00);
if (ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");
}
```

The responses of the procedure are given through the event callback `hci_le_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 (0x2)`.

```
 /* This callback is called when an advertising report is received */
 void hci_le_advertising_report_event(uint8_t Num_Reports,
                                     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 ;

      /* 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*/

      /* RSSI value (/!\ Be aware that RSSI value position is after Data parameter with
      possible different variable length)*/
      uint8_t RSSI = (int8_t)*(uint8_t*)(adv_report_data + le_advertising_event -
      >Advertising_Report[0].length_data);

} /* hci_le_advertising_report_event() */
```

In particular, in this specific context, the following events are raised on the GAP central `hci_le_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)
2. `Advertising Report` event with scan response packet type (evt_type =SCAN_RSP)

Table 37. ADV_IND event type

Event type	Address type	Address	Advertising data	RSSI
0x00 (ADV_IND)	0x00 (public address)	0x0280E1003 412	0x02, 0x01, 0x06, 0x08, 0x0A, 0x53, 0x54, 0x4D, 0x33, 0x32, 0x57, 0x42, 0x78, 0x35	0xCE

The advertising data can be interpreted as follows (refer to Bluetooth® specification version in Section 7: Reference documents):

Table 38. **ADV_IND advertising data**

Flags AD type field	Local name field	Tx power level
0x02: length of the field 0x01: AD type flags 0x06: 0x110 (Bit 2: BR/EDR Not supported; bit 1: general discoverable mode)	0x09: length of the field 0x0A: complete local name type 0x53, 0x54, 0x4D, 0x33, 0x32, 0x57, 0x48, 0x78, 0x35: STM32WB	0x02: length of the field 0x0A: Tx power type 0x08: power value

Table 39. **SCAN_RSP event type**

Event type	Address type	Address	Scan response data	RSSI
0x04 (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 40. **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

4.5 BLE stack events and event callbacks

In order to handle ACI events in its application, the user can choose between two different methods:

- Use nested "switch case" event handler
- Use event callbacks framework

Based on its own application scenario, the user has to identify the required device events to be detected and handled and the application specific actions to be done as consequence of such events.

When implementing a BLE application, the most common and widely used device events are the ones related to the discovery, connection, terminate procedures, services and characteristics discovery procedures, attribute modified events on a GATT server and attribute notification/ indication events on a GATT client.

Table 41. **BLE stack: main events callbacks**

Event callback	Description	Where
hci_disconnection_complete_event()	A connection is terminated	GAP central/ peripheral
hci_le_connection_complete_event()	Indicates to both of the devices forming the connection that a new connection has been established	GAP central/ peripheral
aci_gatt_attribute_modified_event()	Generated by the GATT server when a client modifies any attribute on the server, if event is enabled	GATT server
aci_gatt_notification_event()	Generated by the GATT client when a server notifies any attribute on the client	GATT client
aci_gatt_indication_event()	Generated by the GATT client when a server indicates any attribute on the client	GATT

Event callback	Description	Where
		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
aci_gap_bond_lost_event()	Event generated when a pairing request is issued, in response to a peripheral security request from a central which has previously bonded with the peripheral. When this event is received, the upper layer has to issue the command aci_gap_allow_rebond() to allow the peripheral to continue the pairing process with the central	GAP peripheral
aci_att_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_att_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_proc_complete_event()	A GATT procedure has been completed	GATT client
hci_le_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	GAP central

For a detailed description about the BLE events, and related formats, refer to the STM32WB and STM32WBA Bluetooth® LE stack APIs and events documentation, in Section 7: Reference documents.

The following pseudocode provides an example of events callbacks handling some of the described BLE 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 BLE radio interrupt context.
*/
void hci_disconnection_complete_event(uint8_t Status,
                                      uint16_t Connection_Handle,
                                      uint8_t Reason)
{
    /* Add user code for handling BLE disconnection complete event based on
       application scenario.
    */
}/* end hci_disconnection_complete_event() */

/* This event callback indicates the end of a connection procedure.
*/
void hci_le_connection_complete_event(uint8_t Status,
                                      uint16_t Connection_Handle,
                                      uint8_t Role,
                                      uint8_t Peer_Address_Type,
                                      uint8_t Peer_Address[6],
                                      uint16_t Conn_Interval,
                                      uint16_t Conn_Latency,
                                      uint16_t Supervision_Timeout,
                                      uint8_t central_Clock_Accuracy)

{
    /* Add user code for handling BLE connection complete event based on
```

```
       application scenario.
    */

/* Store connection handle */
 connection_handle = Connection_Handle;
 …
}/* end hci_le_connection_complete_event() */
```

```
#if GATT_SERVER

/* This event callback indicates that an attribute has been modified from a
   peer device.
*/
void aci_gatt_attribute_modified_event(uint16_t Connection_Handle,
                                       uint16_t Attr_Handle,
                                       uint16_t Offset,
                                       uint8_t Attr_Data_Length,
                                       uint8_t Attr_Data[])
{
    /* Add user code for handling attribute modification event based on
       application scenario.
    */
    ...
} /* end aci_gatt_attribute_modified_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 modification event based on
    application scenario.
    */
…
} /* end aci_gatt_notification_event() */
#endif /* GATT_CLIENT */
```

4.6 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 next re-connections).

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

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

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

PassKey Entry example with two STM32WB or STM32WBA 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 11. Methods used to calculate the temporary key (TK), Device_1, Device_2 have to set the IO capability in order to select PassKey entry as a security method.

On 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 are 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:

- By using `aci_gap_peripheral_security_req()` on a GAP peripheral device (it sends a peripheral security request to the central):

```
tBleStatus ret;
ret= aci_gap_peripheral_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 device.

Since the no fixed pin has been set,once the paring procedure is initiated by one of the two devices, BLE 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. BLE 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_peripheral_security_req()` or`aci_gap_send_pairing_req()` and the value `ret= BLE_STATUS_SUCCESS` is returned, on termination of the procedure, a `aci_gap_pairing_complete_event()` is returned to the event callback to indicate the pairing status:*
   - *0x00: Success*
   - *0x01: SMP timeout*
   - *0x02: Pairing failed*

   The pairing status is given from the status field of the `aci_gap_pairing_complete_event()`
   The reason parameter provides the pairing failed reason code in case of failure (0 if status parameter returns success or timeout).
2. *When 2 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 2 devices get connected again, the link can be simply encrypted (without 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_peripheral_security_req )` on the GAP peripheral device or*
   - *`aci_gap_send_pairing_req()` on the GAP central device.*
3. *If a peripheral has already bonded with a central, it can send a peripheral security request to the central to encrypt the link. When receiving the peripheral security request, the central may encrypt the link, initiate the pairing procedure, or reject the request. Typically, the central only encrypts the link, without performing the pairing procedure. Instead, if the central starts the pairing procedure, it means that for some reasons, the central lost its bond information, so it has to start the pairing procedure again. As a consequence, the peripheral device calls the `aci_gap_bond_lost_event()`event callback to inform the user application that it is not bonded anymore with the central it was previously bonded. Then, the peripheral application can decide to allow the security manager to complete the pairing procedure and re-bond with the central by calling the command `aci_gap_allow_rebond()`, or just close the connection and inform the user about the security issue.*
4. *Alternatively, the out-of-band method can be selected by calling the `aci_gap_set_oob_data()` API. This implies that both devices are using this method and they are setting the same OOB data defined through an out of band communication (example: NFC).*
5. *Moreover, the “secure connections” feature can be used by setting to 2 the SC_Support field of the `aci_gap_set_authentication_requirement()` API.*

4.6.1 Flow charts on pairing procedure: pairing request by central sequence (Legacy)

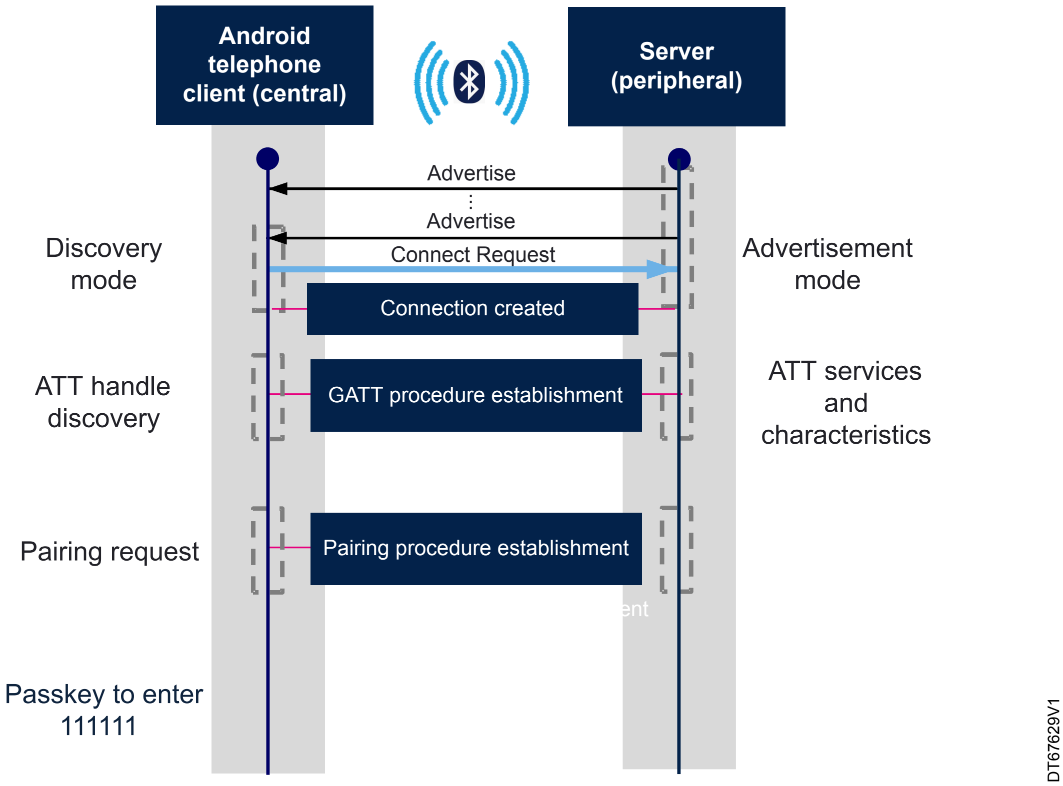
Flow charts on pairing procedure: Pairing request by central sequence (Legacy)

The following flow chart illustrates specific steps to be followed from central to create a security link in Legacy mode

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

```
Initialization:
Aci_gap_set_IO_capability(keyboard/display)
Aci_gap_set_auth_requirement(MITM,fixed pin,bonding=1,SC_Support=0x00)
```

Figure 10. Pairing request initiated by central sequence (Legacy) 1/3



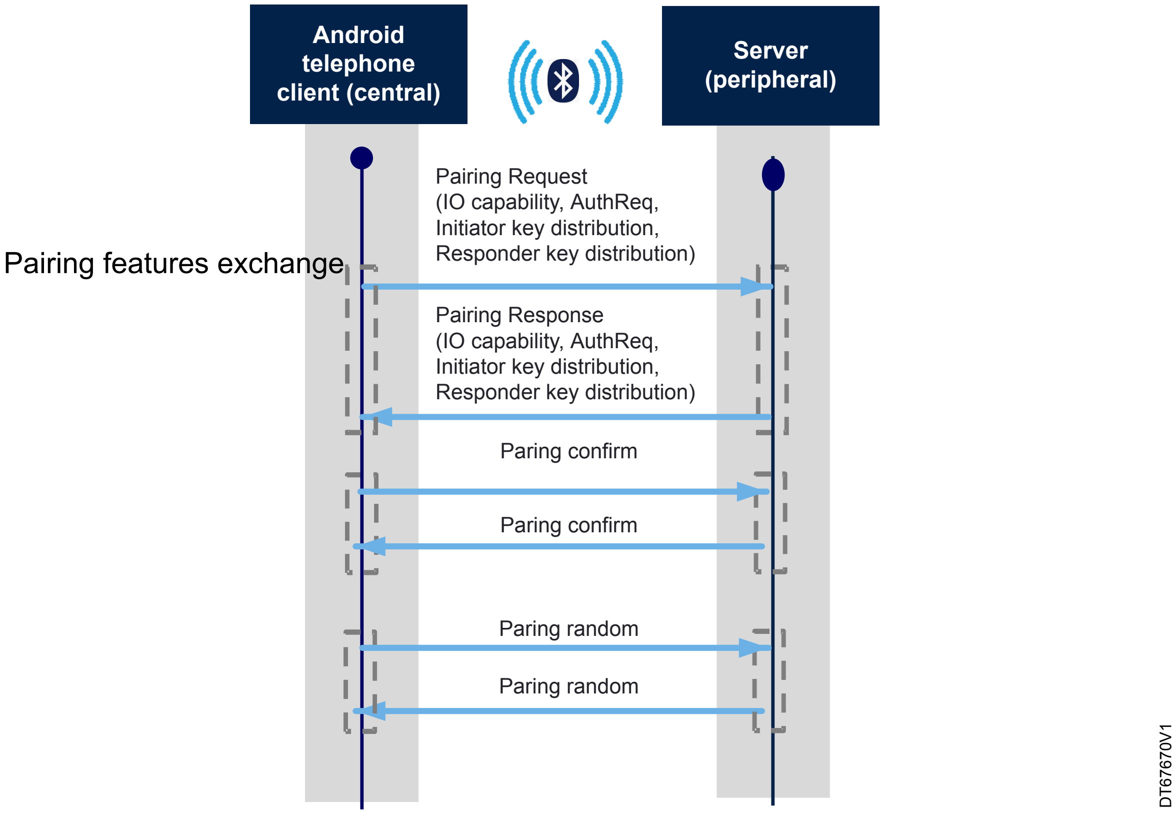


Figure 11. Pairing request initiated by central sequence (Legacy) 2/3

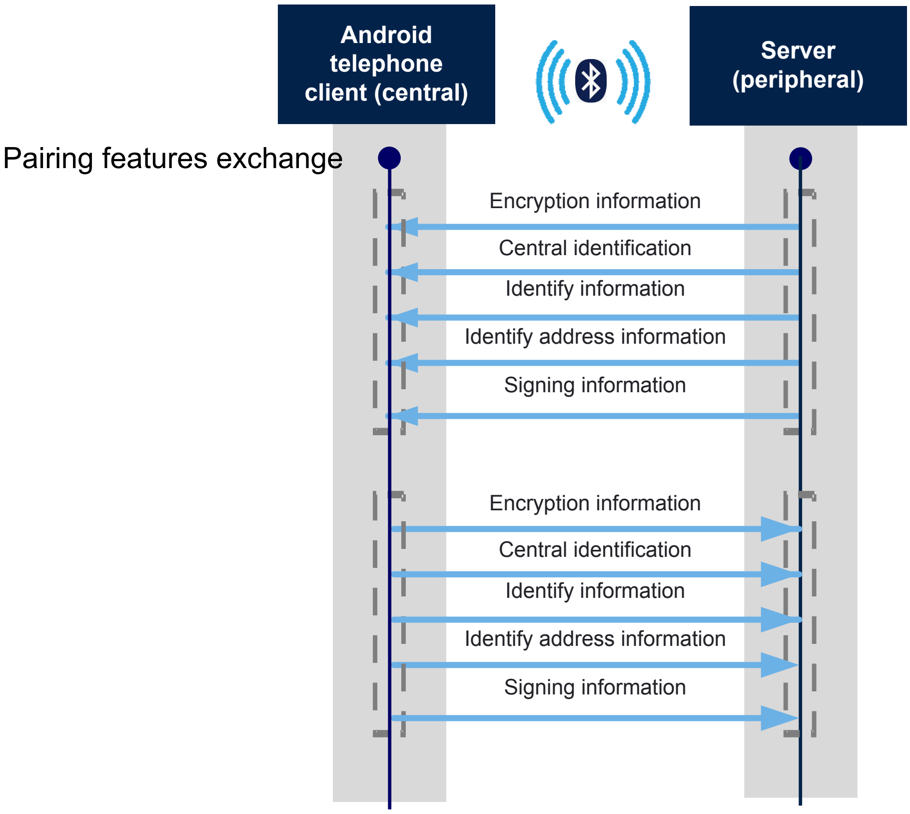


Figure 12. Pairing request initiated by central sequence (Legacy) 3/3

4.6.2 Flow charts on pairing procedure: pairing request by central sequence (secure)

Flow charts on pairing procedure: pairing request by central sequence (Secure)

The following flow chart illustrates specific steps to be followed from central to create a security link in secure mode.

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

```
Initialization:
Aci_gap_set_IO_capability(display_yes_no)
Aci_gap_set_auth_requirement(MITM,no
fixed pin,bonding=1,SC only mode)
```

Figure 13. Pairing request initiated by central sequence (secure connection) 1/3

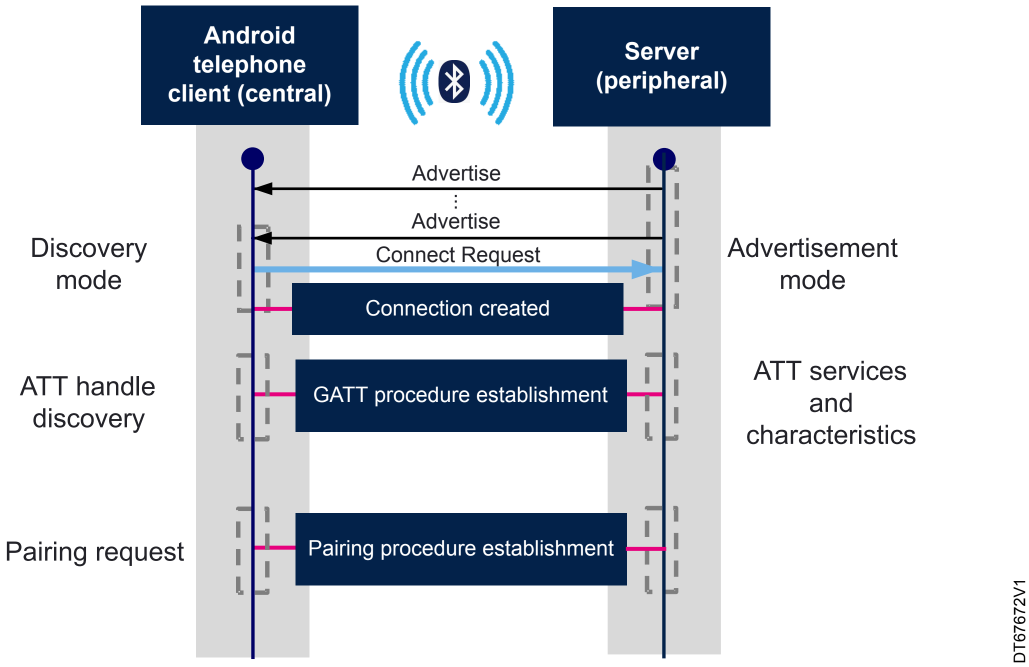


Figure 14. Pairing request initiated by central sequence (secure connection) 2/3

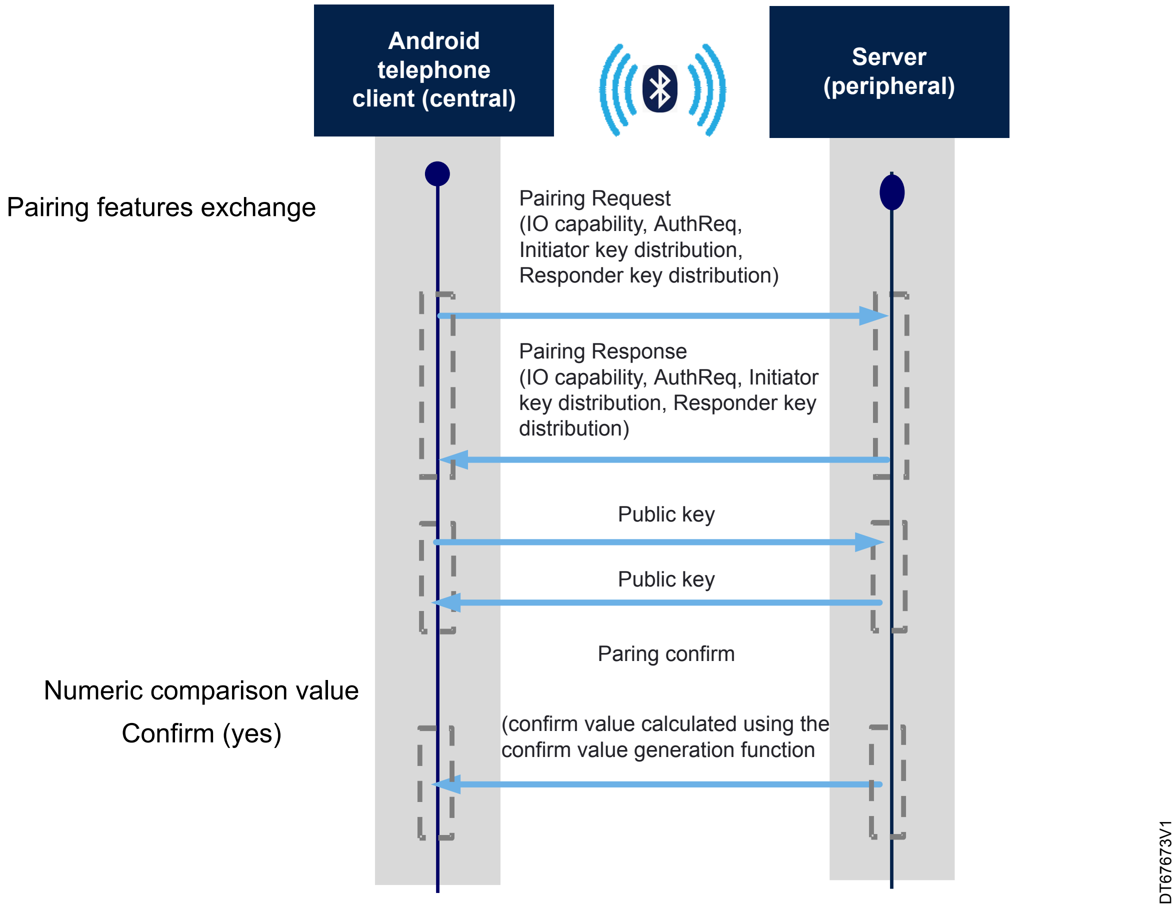
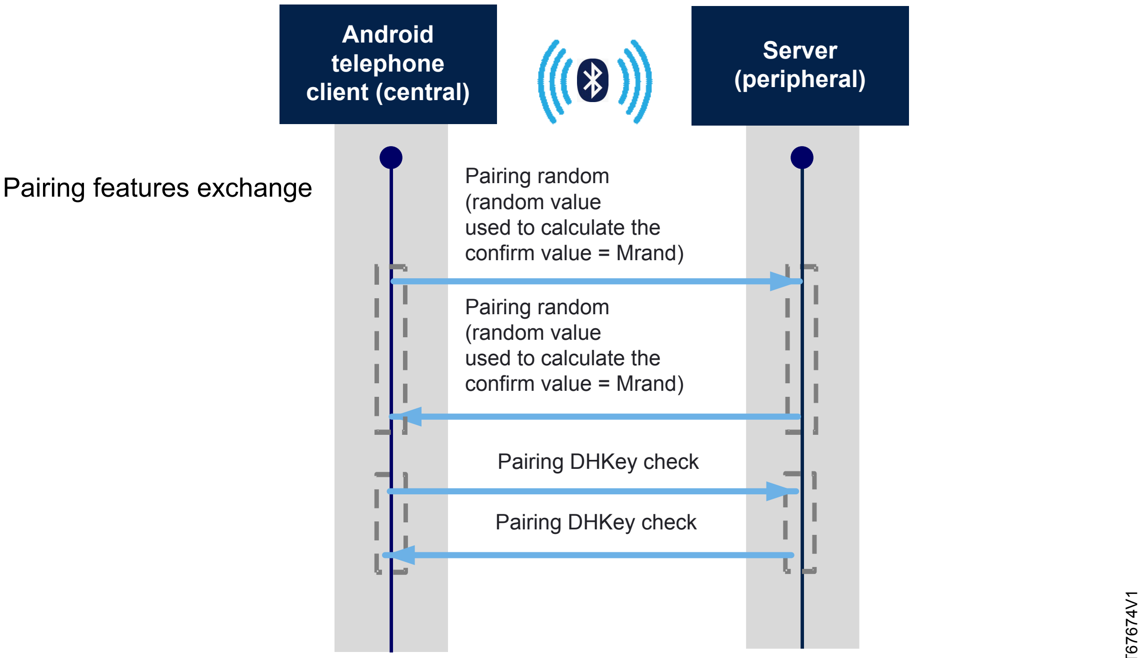


Figure 15. Pairing request initiated by central sequence (secure connection) 3/3



4.6.3 Flow charts on pairing procedure: pairing request by peripheral sequence (secure)

Flow charts on pairing procedure: pairing request by peripheral sequence (secure).

The following flow chart illustrates specific steps to be followed from central to create a security link in security mode

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

Initialization:

Aci_gap_set_IO_capability(display_yes_no)

Aci_gap_set_auth_requirement(MITM,no

fixed pin,bonding=1,SC only mode)

```
Initialization:
Aci_gap_set_IO_capability(display_yes_no)
Aci_gap_set_auth_requirement(MITM,no
fixed pin,bonding=1,SC only mode)
```

Figure 16. Pairing request initiated by peripheral sequence (secure connection) 1/2

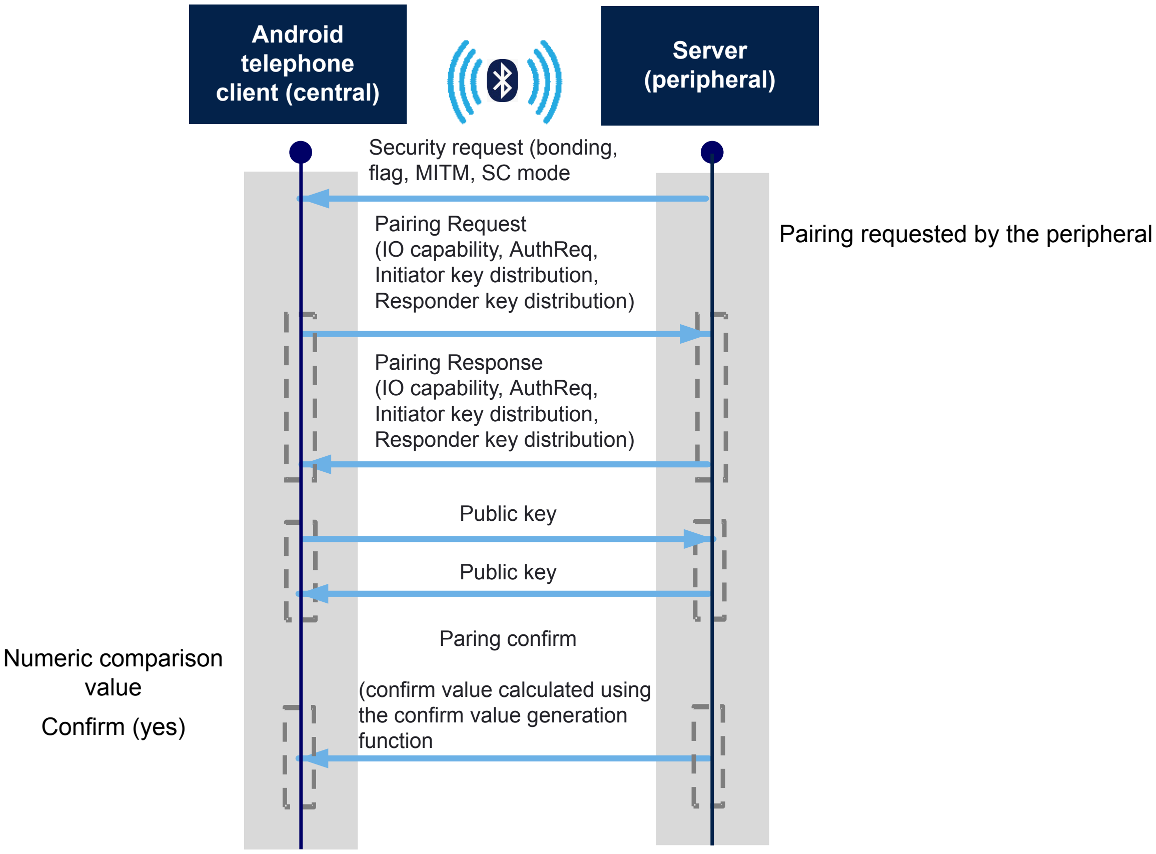
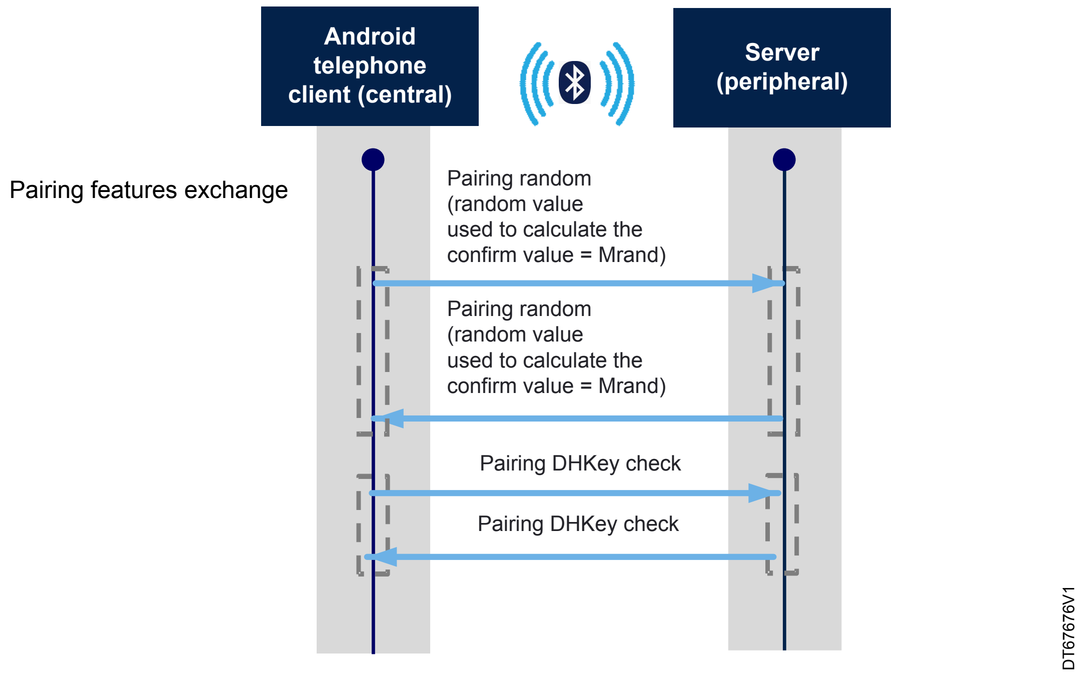


Figure 17. **Pairing request initiated by peripheral sequence (secure connection) 2/2**



4.7 Pairing failing and automatic pairing rejection guard time

To increase security and to prevent unauthorized devices from continuously retrying the pairing process, an automatic pairing rejection guard time is generated on both the client and the server.

In STM32WB and STM32WBA stacks, after LE secure connection pairing phase 2 starts, in case the pairing process fails, the next pairing attempt can only occur after a dedicated guard time and the remote device is put in blacklist mode by the stack. This guard time increases according to the number of failed attempts (5sec, 15 sec, 45 sec, etc.), while it goes back to normal after each secure pairing success. During blacklist mode, both devices cannot start the pairing process, and have to wait for the guard timer to end.

4.8 NVM information for GATT and security record

In STM32WB and STM32WBA, the BLE host stack needs nonvolatile memory (NVM) to support the bonding feature. Two types of records per bonded peer device are stored in the NVM:

- A security record containing SMP information, such as LTK, IRK, or CSRK keys.
- A GATT record containing a digest of the local GATT database. Depending on a BLE stack initialization parameter, there are two possible versions for this record: a full version and a reduced version. Refer to Section 5.1.1: Reduced GATT information in NVM for more details.

The BLE stack does not directly access the NVM but accesses an NVM cache in the RAM. It is up to the application to ensure that this NVM cache is stored correctly in the NVM before power-off, and restored correctly on power-up before the BLE stack is started (on STM32WB, these operations can be handled automatically by the Cortex®-M0 firmware).

The stack accesses the NVM cache to read information in the following circumstances:

- At the start of a connection (that is, at the reception of a connection complete event by the host) to search and read the security and GATT records corresponding to the connected peer device.
- At the execution of the following commands to read security records:
  - ACI_GAP_ADD_DEVICES_TO_LIST
  - ACI_GAP_ADD_DEVICES_TO_RESOLVING_LIST (deprecated command)
  - ACI_GAP_CONFIGURE_FILTER_ACCEPT_LIST
  - ACI_GAP_RESOLVE_PRIVATE_ADDR
  - ACI_GAP_GET_BONDED_DEVICES
  - ACI_GAP_IS_DEVICE_BONDED

The stack accesses the NVM cache to write information in the following circumstances:

- At successful pairing completion to write a security record (only if both devices request bonding).
- At disconnection or at GATT timeout (if the devices are bonded) to write a GATT record.
- At the execution of the following commands:
  - ACI_GATT_STORE_DB to write a GATT record for each connected and bonded device.
  - ACI_GAP_CLEAR_SECURITY_DB to clear all records.
  - ACI_GAP_REMOVE_BONDED_DEVICE to clear the records from a peer device.

Note: It is up to the application to manage the list of devices whose information is stored in NVM. For instance, if the NVM is full, the application must remove the oldest bonded device in the NVM before bonding with a new device. Indeed, if the BLE stack discovers that the NVM is full when it attempts to write a record into it, it first drains the NVM completely, leaving the information of only one peer device in the NVM.

4.9 Service and characteristic discovery

This section describes the main functions allowing an STM32WB or STM32WBA GAP central device to discover the GAP peripheral services and characteristics, once both devices are connected.

The P2PServer service & characteristics with related handles is used as reference service and characteristics on the following pseudo-code examples.

Further, it is assumed that a GAP central device (P2PClient application) is connected to a GAP peripheral device running the P2PServer application. The GAP central device uses the service and discovery procedures to find the GAP Peripheral P2PServer service and characteristics. The GAP central device is running the P2PClient application.

Table 42. BLE sensor profile demo services and characteristic handle

Service	Characteristic	Service / characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
Peer To Peer	NA	0x000C	NA	NA	NA
-	LED	0x000D	0x000E	NA	NA
-	Button	0x000F	0x0010	0x0011	NA

Note: The different attribute value handles are due to the last attribute handle reserved for the standard GAP service. In the following example, the STM32WB or STM32WBA GAP peripheral P2PServer service is defining only the LED characteristic and Button characteristic. For detailed information about tP2Pserver refer to Section 7: Reference documents.

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

Table 43. **Service discovery procedures APIs**

Discovery service API	Description
aci_gatt_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_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_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_disc_all_primary_services() API`:

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

}
```

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

```
/* This event is generated in response to a Read By Group Type
Request: refer to aci_gatt_disc_all_primary_services() */
void aci_att_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 Core
 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_att_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_att_read_by_group_type_resp_event()` event callback), with the following callback parameters 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 44. First read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x0001	0x0004	0x1801	Attribute profile service (GATT_Init() addsit). Standard 16-bit service UUID
0x0005	0x000B	0x1800	GAP profile service (GAP_Init() adds it). 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 45. Second read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x000C	0x0012	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 46. Third read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x0013	0x0019	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_proc_complete_event()` event callback is called on GAP central application, with the following parameters

```
Conn_Handle: 0x0801 (connection handle;
Error_Code: 0x00
```

4.9.1 Characteristic discovery procedures and related GATT events

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

Table 47. Characteristics discovery procedures APIs

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

In the context of the BLE sensor profile demo, follow a simple pseudocode illustrating how a GAP central application can discover all the characteristics of the acceleration service (refer to Table 45. Second read by group type response event callback parameters):

```
uint16_t service_handle= 0x000C;
uint16_t end_group_handle = 0x0012;
```

```
/*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,/* Servicehandle */
                                    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.

```
/* This event is generated in response to aci_att_read_by_type_req(). Refer to aci_gatt_disc_
all_char() API */
```

```
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 BLE 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 48. First read by type response event callback parameters

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x000D	0x10 (notify)	0x000E	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 49. **Second read by type response event callback parameters**

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x0010	0x12 (notify and read)	0x0011	0x340A1B80CF4B11E1AC360002A5D5C51B	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 42. BLE sensor profile demo services and characteristic handle).

4.10 Characteristic notification/indications, write, read

This section describes the main functions to get access to BLE device characteristics.

Table 50. **Characteristic update, read, write APIs**

Discovery service API	Description	Where
aci_gatt_update_char_value_ext()	If notifications (or indications) are enabled on the characteristic, this API sends a notification (or indication) to the client.	GATT server
aci_gatt_read_char_value()	It starts the procedure to read the attribute value.	GATT client
aci_gatt_write_char_value()	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_write_without_resp()	It starts the procedure to write a characteristic value without waiting for any response from the server.	GATT client
aci_gatt_write_char_desc()	It starts the procedure to write a characteristic descriptor.	GATT client
aci_gatt_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 P2PServer demo, follow a part of code the GAP Central application should use in order to configure the Button characteristics client descriptor configuration for notification:

```
/* Enable the Button characteristic client descriptor configuration for notification */aci_ga
tt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
uint8_t *)&enable);
```

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:

```
void P2PS_Send_Notification(void)
{
if(P2P_Server_App_Context.ButtonControl.ButtonStatus == 0x00){
P2P_Server_App_Context.ButtonControl.ButtonStatus=0x01;
} else {
P2P_Server_App_Context.ButtonControl.ButtonStatus=0x00;
}
if(P2P_Server_App_Context.Notification_Status){
APP_DBG_MSG("-- P2P APPLICATION SERVER : INFORM CLIENT BUTTON 1 USHED \n ");
APP_DBG_MSG(" \n\r");
P2PS_STM_App_Update_Char(P2P_NOTIFY_CHAR_UUID, (uint8_t*)&P2P_Server_App_Context.ButtonContro
l);
} else {
APP_DBG_MSG("-- P2P APPLICATION SERVER : CAN'T INFORM CLIENT - NOTIFICATION DISABLED\n ");
}
return;
}
tBleStatus P2PS_STM_App_Update_Char(uint16_t UUID, uint8_t *pPayload)
{
tBleStatus result = BLE_STATUS_INVALID_PARAMS;
switch(UUID)
{
case P2P_NOTIFY_CHAR_UUID:
result = aci_gatt_update_char_value(aPeerToPeerContext.PeerToPeerSvcHdle,
aPeerToPeerContext.P2PNotifyServerToClientCharHdle,
0, /* charValOffset */
2, /* charValueLen */
(uint8_t *) pPayload);
break;
default:
break;
}
return result;
}/* end P2PS_STM_Init() */
```

On GAP Central, Event_Handler (EVT_VENDOR as main event), the EVT_BLUE_GATT_NOTIFICATION is raised on reception of the characteristic notification (Button) from the GAP Peripheral device.

```
static SVCCTL_EvtAckStatus_t Event_Handler(void *Event)
{
SVCCTL_EvtAckStatus_t return_value;
hci_event_pckt *event_pckt;
evt_blue_aci *blue_evt;
P2P_Client_App_Notification_evt_t Notification;
return_value = SVCCTL_EvtNotAck;
event_pckt = (hci_event_pckt *)(((hci_uart_pckt*)Event)->data);
switch(event_pckt->evt) {
case EVT_VENDOR:
{
blue_evt = (evt_blue_aci*)event_pckt->data;
switch(blue_evt->ecode) {
….
case EVT_BLUE_GATT_NOTIFICATION:
{
aci_gatt_notification_event_rp0 *pr = (void*)blue_evt->data;
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) &&
(aP2PClientContext[index].connHandle != pr->Connection_Handle))
index++;
if(index < BLE_CFG_CLT_MAX_NBR_CB) {
if ( (pr->Attribute_Handle == aP2PClientContext[index].P2PNotificationCharHdle) &&
(pr->Attribute_Value_Length == (2)) )
{
Notification.P2P_Client_Evt_Opcode = P2P_NOTIFICATION_INFO_RECEIVED_EVT;
Notification.DataTransfered.Length = pr->Attribute_Value_Length;
Notification.DataTransfered.pPayload = &pr->Attribute_Value[0];
Gatt_Notification(&Notification);
/* INFORM APPLICATION BUTTON IS PUSHED BY END DEVICE */
}
```

```
}
}
break;/* end EVT_BLUE_GATT_NOTIFICATION */
….
void Gatt_Notification(P2P_Client_App_Notification_evt_t *pNotification) {
switch(pNotification->P2P_Client_Evt_Opcode) {
case P2P_NOTIFICATION_INFO_RECEIVED_EVT:
{
P2P_Client_App_Context.LedControl.Device_Led_Selection=pNotification->DataTransfered.pPayloa
d[0];
switch(P2P_Client_App_Context.LedControl.Device_Led_Selection) {
case 0x01 : {
P2P_Client_App_Context.LedControl.Led1=pNotification->DataTransfered.pPayload[1];
if(P2P_Client_App_Context.LedControl.Led1==0x00){
BSP_LED_Off(LED_BLUE);
APP_DBG_MSG(" -- P2P APPLICATION CLIENT : NOTIFICATION RECEIVED - LED OFF \n\r");
APP_DBG_MSG(" \n\r");
} else {
APP_DBG_MSG(" -- P2P APPLICATION CLIENT : NOTIFICATION RECEIVED - LED ON\n\r");
APP_DBG_MSG(" \n\r");
BSP_LED_On(LED_BLUE);
}
break;
}
default : break;
}
….
}
```

Note: While a server is connected with multi-clients and one characteristic has notification or indications enable for several clients, if one client changes the value, the server informs the upper layer that the characteristic has changed its value. It is up to the upper layer to send or not this update to the other(s) client(s) that are connected. Of course, if the server updates itself the value, all clients receive this updated value.

4.10.1 Getting access to BLE device long characteristics.

This section describes the main functions for getting access to BLE device long characteristics.

Table 51. **Characteristic update, read, write APIs for long Value**

Characteristic handling API	Description	API call side	Events to be used on client side
Aci_gatt_read_long_char_value()	Reads a long characteristic value.	GATT client	ACI_GATT_READ_EXT_EVENT (mask = 0x00100000)
Aci_gatt_write_long_char_value()	Writes a long characteristic value.	GATT client	ACI_ATT_EXEC_WRITE_RESP_EVENT (mask = 0x00001000) ACI_ATT_PREPARE_WRITE_RESP_EVENT (mask = 0x00000800)
Aci_gatt_update_char_value_ext()	Version of aci_gatt_update_char_value to support update of long attribute up to 512 bytes and indicate selectively the generation of indication/notification.	GATT server	ACI_GATT_NOTIFICATION_EXT_EVENT (mask = 0x00400000) or ACI_GATT_INDICATION_EXT_EVENT (mask = 0x00200000)
Aci_gatt_read_handle_value()	Reads the value of the attribute handle specified from the local GATT database.	GATT server	-

1. Characteristics are long when char_length > ATT_MTU – 4
2. Limitation due to the stack interface of events: event parameters length is an 8-bit value.

Read long distant data (client side)

To avoid limitation 2, new events have been added: ACI_GATT_READ_EXT_EVENT
(to be enabled with the following mask: 0x00100000 using aci_gatt_set_event_mask command)
It replaces three events:

```
ACI_ATT_READ_RESP_EVENT (1)
ACI_ATT_READ_BLOB_RESP_EVENT (2)
ACI_ATT_READ_MULTIPLE_RESP_EVENT (3)
```

Generated in response to:

```
Aci_gatt_read_char_value (1)
Aci_gatt_read_long_char_value (2)
Aci_gatt_read_multiple_char_value (3)
```

(condition ATT_MTU > sum of the multiple characteristics total length

Write long distant data (client side)

```
Aci_gatt_write_long_char_value()
```

The length of the data to be written is limited to 245 (with ATT_MTU = 251)
If the application requires writing data larger than 245 bytes (length > 245, ATT_MTU = 512), the `aci_att_prepare_write_req` and `aci_att_execute_write_req` commands are used, which can send up to 509 bytes to write. The size in aci_att_prepare_write_req command is set at 242 to avoid having several LL packets for each prepare write.

```
aci_att_prepare_write_req(conn_handle, char_handle , 0 , /* offset of the first octet to be w
ritten */ 242, /* length of attribute value */ (uint8_t *) pPayload);
aci_att_prepare_write_req(conn_handle, char_handle, 242, /* offset of the first octet to be w
ritten */ 242, /* length of attribute value */ (uint8_t *) pPayload+242);
aci_att_prepare_write_req(conn_handle, char_handle, 484, /* offset of the first octet to be w
ritten */ 25, /* length of attribute value */ (uint8_t *) pPayload+484);
aci_att_execute_write_req(conn_handle, 0x01); /* write all pending prepared values */
```

On the server side, after `aci_att_execute_write_req` is sent from central, `ACI_GATT_ATTRIBUTE_MODIFIED_VSEVT_CODE` is received. The process to recover long written data uses bit 15 of the offset field. This bit is used as a flag and when it is set, it indicates that more data are to come (fragmented event in case of long attribute data).

Read long local data (server side)

```
Aci_gatt_read_handle_value()
```

This command needs to be called several times.

Write long local data (server side)

```
ACI_GATT_NOTIFICATION_EXT_EVENT
```

(to be enabled with the following mask : 0x00400000 using aci_gatt_set_event_mask command)

In response to:

```
Aci_gatt_update_char_value_ext
```

command

How to use aci_gatt_update_char_value_ext:

When `ATT_MTU > (BLE_EVT_MAX_PARAM_LENGTH – 4) i.e ATT_MTU > 251`, two commands are necessary.

First command:

```
Aci_gatt_update_char_value_ext (conn_handle, Service_handle, TxCharHandle,
Update_Type = 0x00,
Total_length,
Value_offset,
Param_length,
&payload)
```

Second command

```
Aci_gatt_update_char_value_ext (conn_handle, Service_handle, TxCharHandle,
Update_Type = 0x01,
Total_length,
Value_offset = Param_length,
param_length2,
(&payload) + param_length)
```

After second command, a notification of total length is sent on the air and is received through ACI_GATT_NOTIFICATION_EXT_EVENT events.

The data can be re-assembled depending on the offset parameter of ACI_GATT_NOTIFICATION_EXT_EVENT event. Bit 15 is used as flag: when set to 1it indicates that more data are to come (fragmented event in case of long attribute data)

Idem for: ACI_GATT_INDICATION_EXT_EVENT (to be enabled with the following mask : 0x00200000 using aci_gatt_set_event_mask command)

In response to: Aci_gatt_update_char_value_ext() command.

In this case Update_Type = 0x00 for the first command, and Update_Type = 0x02 for the second command.

If we take an example of long data transfer:

Once the characteristics notification has been enabled from the GAP Central, the GAP peripheral can notify a new value:

```
static void SendData( void )
{
tBleStatus status = BLE_STATUS_INVALID_PARAMS;
uint8_t crc_result;
if( (DataTransferServerContext.ButtonTransferReq != DTS_APP_TRANSFER_REQ_OFF)
&& (DataTransferServerContext.NotificationTransferReq != DTS_APP_TRANSFER_REQ_OFF)
&& (DataTransferServerContext.DtFlowStatus != DTS_APP_FLOW_OFF) )
{
/*Data Packet to send to remote*/
Notification_Data_Buffer[0] += 1;
/* compute CRC */
crc_result = APP_BLE_ComputeCRC8((uint8_t*) Notification_Data_Buffer, (DATA_NOTIFICATION_MAX_
PACKET_SIZE - 1));
Notification_Data_Buffer[DATA_NOTIFICATION_MAX_PACKET_SIZE - 1] = crc_result;
DataTransferServerContext.TxData.pPayload = Notification_Data_Buffer;
//DataTransferServerContext.TxData.Length = DATA_NOTIFICATION_MAX_PACKET_SIZE; /* DATA_NOTIFI
CATION_MAX_PACKET_SIZE */
DataTransferServerContext.TxData.Length = Att_Mtu_Exchanged-10;
status = DTS_STM_UpdateChar(DATA_TRANSFER_TX_CHAR_UUID, (uint8_t *) &DataTransferServerContex
t.TxData);
if (status == BLE_STATUS_INSUFFICIENT_RESOURCES)
{
DataTransferServerContext.DtFlowStatus = DTS_APP_FLOW_OFF;
(Notification_Data_Buffer[0])-=1;
}
else
{
UTIL_SEQ_SetTask(1 << CFG_TASK_DATA_TRANSFER_UPDATE_ID, CFG_SCH_PRIO_0);
}
}
return;
}
tBleStatus DTS_STM_UpdateChar( uint16_t UUID , uint8_t *pPayload )
{
tBleStatus result = BLE_STATUS_INVALID_PARAMS;
switch (UUID)
{
case DATA_TRANSFER_TX_CHAR_UUID:
result = TX_Update_Char((DTS_STM_Payload_t*) pPayload);
break;
default:
break;
}
return result;
}/* end DTS_STM_UpdateChar() */
static tBleStatus TX_Update_Char( DTS_STM_Payload_t *pDataValue )
{
tBleStatus ret;
/**
* Notification Data Transfer Packet
*/
/* Total length corresponds to total length of data that will be sent through notification Va
lue offset corresponds to the offset of the value to modify Param length corresponds to the l
ength of the value to be modify at the offset defined previously */
```

On GAP Client, DTC_Event_Handler (EVT_VENDOR as main event), the EVT_BLUE_GATT_NOTIFICATION_EXT is raised on reception of the characteristic notification (Button) from the GAP Peripheral device.

```
static SVCCTL_EvtAckStatus_t DTC_Event_Handler(void *Event)
{
SVCCTL_EvtAckStatus_t return_value;
hci_event_pckt *event_pckt;
evt_blue_aci *blue_evt;
P2P_Client_App_Notification_evt_t Notification;
return_value = SVCCTL_EvtNotAck;
event_pckt = (hci_event_pckt *)(((hci_uart_pckt*)Event)->data);
switch(event_pckt->evt)
{
case EVT_VENDOR:
```

```
{
blue_evt = (evt_blue_aci*)event_pckt->data;
switch(blue_evt->ecode)
{
….
case EVT_BLUE_GATT_NOTIFICATION_EXT:
{
aci_gatt_notification_event_rp0 *pr = (void*)blue_evt->data;nnnn
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) &&
(aP2PClientContext[index].connHandle != pr->Connection_Handle))
index++;
if(index < BLE_CFG_CLT_MAX_NBR_CB)
{
if ( (pr->Attribute_Handle == aP2PClientContext[index].P2PNotificationCharHdle) &&
(pr->Attribute_Value_Length == (2)) )
{
Notification.P2P_Client_Evt_Opcode = P2P_NOTIFICATION_INFO_RECEIVED_EVT;
Notification.DataTransfered.Length = pr->Attribute_Value_Length;
Notification.DataTransfered.pPayload = &pr->Attribute_Value[0];
Gatt_Notification(&Notification);
/* INFORM APPLICATION BUTTON IS PUSHED BY END DEVICE */
}
}
}
break;/* end EVT_BLUE_GATT_NOTIFICATION */
```

4.11 End to end RX flow control using GATT

It is possible to benefit from an optimized RX flow control when using GATT to receive data from a peer.

Typically, the peer device uses several times the GATT write procedure to send the data by packets to a local device GATT characteristic. The user application of the local device then receives the packets through successive GATT events (ACI_GATT_ATTRIBUTE_MODIFIED_EVENT).

To get an RX flow control the user application needs to set the AUTHOR_WRITE flag when creating the characteristic using the ACI_GATT_ADD_CHAR primitive. The user application is then informed of each peer write tentative before it is executed by means of a dedicated event (ACI_GATT_WRITE_PERMIT_REQ_EVENT). The user application just needs to answer to that event with the ACI_GATT_WRITE_RESP primitive (Write_status = 0). If the user application takes time to answer to this event (for instance, it is still processing the previous data packet), this has the effect of blocking the local GATT and then blocking the peer when the local internal RX ACL data FIFO is full (the size of this FIFO depending on the BLE stack configuration).

4.12 Basic/typical error condition description

On the STM32WB and STM32WBA BLE stack APIs framework, the `tBleStatus` type is defined in order to return the STM32WB and STM32WBA 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 the STM32WB and STM32WBA Bluetooth® Low Energy stack APIs and event documentation, in Section 7: Reference documents.

5 STM32WB and STM32WBA BLE stack advanced features description

5.1 Generic attribute profile (GATT) advanced features

5.1.1 Reduced GATT information in NVM

STM32WB and STM32WBA BLE stacks offer the possibility to reduce the GATT information stored in NVM. This feature is not activated by default. It must be explicitly activated by the application during BLE stack initialization (refer to Initialization phase and main application loop).

When this mode is activated, the GATT does not store the whole GATT database in NVM for bonded devices. It only saves the “client dependent” data:

- The CCCD values
- The client supported features (only if “GATT caching” or “Enhanced ATT” is activated)
- The client “change aware/unaware” state (only if “GATT caching” is activated)
- The database hash (only if “no service changed” feature is not activated)

Note: Using the “Reduced GATT information in NVM” along with the “Service Changed” characteristic means: in case of any GATT database modification, the GATT server always indicates that the full range of attributes has been modified. Thus a remote device should rediscover the overall database. In that case, the “Service Changed” characteristic value does not indicate the beginning and ending attribute handles affected by the GATT database change. This is a limitation to be compared with the saved space in NVM.

5.1.2 GATT caching

The STM32WB and STM32WBA BLE stacks offer the BLE standard “GATT caching” feature. This feature is not activated by default. It must be explicitly activated by the application during BLE stack initialization (refer to Initialization phase and main application loop).

Once activated, the GATT caching feature operates automatically and does not need additional interaction with application (i.e. there is no dedicated command for GATT caching). Hence, if GATT caching is activated, the following operations are automatically performed by the GATT.

At GATT initialization (aci_gatt_init), the following characteristics are added to the GATT service:

- Client supported features
- Database hash

New errors are handled by the GATT:

- Database out-of-sync
- Value not allowed

New GATT information are stored in NVM:

- Client supported features
- Client “change aware/unaware” state

5.1.3 LE GATT Security Levels Characteristic (SLC)

STM32WB and STM32WBA stacks can indicate the security mode and level required for all their GATT functionality to be granted.

As defined by the BT5.4 standard, STM32WB and STM32WBA stacks use a new GATT characteristic called LE GATT Security Levels Characteristic (SLC), which allows any GATT client to determine which security condition must be required to satisfy access to all GATT functionalities.

From the GATT server, the attribute value and attribute permission must be indicated by the application itself, according to the highest security attributes available inside the entire database. These values should be automatically reviewed and updated by the application for any new added characteristics with higher security levels.

From the GATT client, the SLC attribute is read and compared by the application itself. If the current connection has a lower security mode and level, the upper layer application can decide to request upgraded security to satisfy SLC GATT server access to the host stack.

In the case that the GATT client has subscribed to notification/indication with a dedicated security mode and level to a GATT server, once reconnected, it is up to the application not to consider notification (or indication) until the required level of security is put in place. However, it is necessary for the application to confirm the received indications to avoid any GATT timeout.

5.1.4 GATT operation timers

Regarding the GATT timeout procedure, in STM32WB and STM32WBA BLE stacks, there are only four possible timeouts:

1. Server request/response procedure is:
   - Started on server side when a client request is received.
   - Stopped once the response has been sent to the client.
     Depending on the attribute configuration, the server application may cause the timeout if it does not answer to events such as:
     - ACI_GATT_READ_PERMIT_REQ_EVENT
     - ACI_GATT_WRITE_PERMIT_REQ_EVENT
     - ACI_GATT_READ_MULTI_PERMIT_REQ_EVENT
     - ACI_GATT_PREPARE_WRITE_PERMIT_REQ_EVENT
2. Server indication/confirmation procedure is:
   - Started on server side when an indication is sent to the client.
   - Stopped when the confirmation is received from the client.
     - Server application action is not needed.
3. Client request/response procedure is:
   - Started on client side when a request is sent to the server.
   - Stopped when the response is received from the server.
     - Client application action is not needed.
4. Client indication/confirmation procedure is:
   - Started on client side when an indication is received from the server.
   - Stopped once the confirmation has been sent to the server.
     Client application can cause the timeout if it does not answer to:
     - ACI_GATT_INDICATION_EVENT (or ACI_GATT_INDICATION_EXT_EVENT) by issuing ACI_GATT_CONFIRM_INDICATION.

Note: These timeouts are cleared if the corresponding link is closed.

5.1.5 Enhanced ATT

The STM32WB and STM32WBA BLE stacks offer the BLE standard “Enhanced ATT” feature (EATT). This feature is not activated by default. It must be explicitly activated by the application during BLE stack initialization (refer to Initialization phase and main application loop).

When EATT is activated, two new characteristics to the GATT service are added:

- Client supported features (for “Enhanced ATT bearer” and “Multiple Handle Value Notifications”)
- Server supported features (for “EATT Supported”)

5.1.5.1 EATT connection

To create an EATT bearer between a client and a server, it is first necessary to:

- Create a GAP connection
  (this can be done by any GAP mean: there is no specific restriction)
- Perform a pairing
  (this is mandatory as EATT requires the link to be encrypted)
  It is then needed to open a Connection-Oriented channel dedicated to EATT:
- On initiator side, to open the channel, one must use aci_l2cap_coc_connect with SPSM = 0x0027, requesting the creation of an enhanced credit based connection-oriented channel (Channel_Number > 0).
- On responder side, once the aci_l2cap_coc_connect event with SPSM = 0x0027 is received, one must use aci_l2cap_coc_connect_confirm to open the channel.

- On both sides, once the channel is opened, the aci_gatt_eatt_bearer_event is received to confirm the creation of the EATT bearer (EAB_State = 0). This same event is received when the bearer is terminated (EAB_State = 1).

It is recommended to set Initial_Credits to 1 (the number of Initial_Credits given to the EATT bearer depends on EATT MTU and the number of data blocks allocated to the BLE stack in RAM).

5.1.5.2 GATT commands over EATT

Once the EATT bearer is created, it is possible to use GATT commands upon this bearer. For that purpose, the Connection_Handle parameter of GATT commands must be set to a value equal to 0xEA00 | channel_index. Channel_index is the channel index of the connection-oriented channel being used as EATT bearer (this index is given by aci_l2cap_coc_connect_confirm_event on initiator side and by response of aci_l2cap_coc_connect_confirm command on responder side).

Note: *The BLE stack can itself retrieve the connection handle, as the connection-oriented channel index is unique among all connections.*

The commands that can be used over EATT on client side are:

- aci_gatt_disc_all_primary_services
- aci_gatt_disc_primary_service_by_uuid
- aci_gatt_find_included_services
- aci_gatt_disc_all_char_of_service
- aci_gatt_disc_char_by_uuid
- aci_gatt_disc_all_char_desc
- aci_gatt_read_char_value
- aci_gatt_read_using_char_uuid
- aci_gatt_read_long_char_value
- aci_gatt_read_multiple_char_value
- aci_gatt_read_long_char_desc
- aci_gatt_read_char_desc
- aci_gatt_write_char_value
- aci_gatt_write_long_char_value
- aci_gatt_write_char_reliable
- aci_gatt_write_long_char_desc
- aci_gatt_write_char_desc
- aci_gatt_write_without_resp
- aci_gatt_confirm_indication

The commands that can be used over EATT on server side are:

- aci_gatt_update_char_value_ext
- aci_gatt_write_resp
- aci_gatt_allow_read
- aci_gatt_deny_read

5.1.5.3 GATT events over EATT

In same principle as GATT commands, the GATT events referring to a specific EATT bearer return the value 0xEA00 | channel_index for the Connection_Handle parameter.

At disconnection, all remaining EATT notifications or indications packets are delivered by STM32WB and STM32WBA.

Events that can process EATT bearer on client side are:

- aci_gatt_proc_complete_event
- aci_att_find_info_resp_event
- aci_att_find_by_type_value_resp_event
- aci_att_read_by_type_resp_event
- aci_att_read_resp_event
- aci_att_read_blob_resp_event

- aci_att_read_multiple_resp_event
- aci_att_read_by_group_type_resp_event
- aci_att_prepare_write_resp_event
- aci_att_exec_write_resp_event
- aci_gatt_error_resp_event
- aci_gatt_disc_read_char_by_uuid_resp_event
- aci_gatt_indication_event
- aci_gatt_indication_ext_event
- aci_gatt_notification_event
- aci_gatt_notification_ext_event

Events that can process EATT bearer on server side are:

- aci_gatt_server_confirmation_event
- aci_gatt_read_permit_req_event
- aci_gatt_read_multi_permit_req_event
- aci_gatt_write_permit_req_event
- aci_gatt_prepare_write_permit_req_event

5.1.5.4 *EATT limitations*

With the current STM32WB and STM32WBA BLE stacks, the following limitations apply to EATT:

- EATT MTU is limited to 246 bytes.
- The number of EATT bearers plus the number of active GATT connections are limited by the maximum number of links plus 4 (e.g. there can be no more than 11 EATT bearers if the maximum link number is set to 8 at BLE stack initialization and if only one connection is active).

5.2 BLE simultaneously central, peripheral scenario

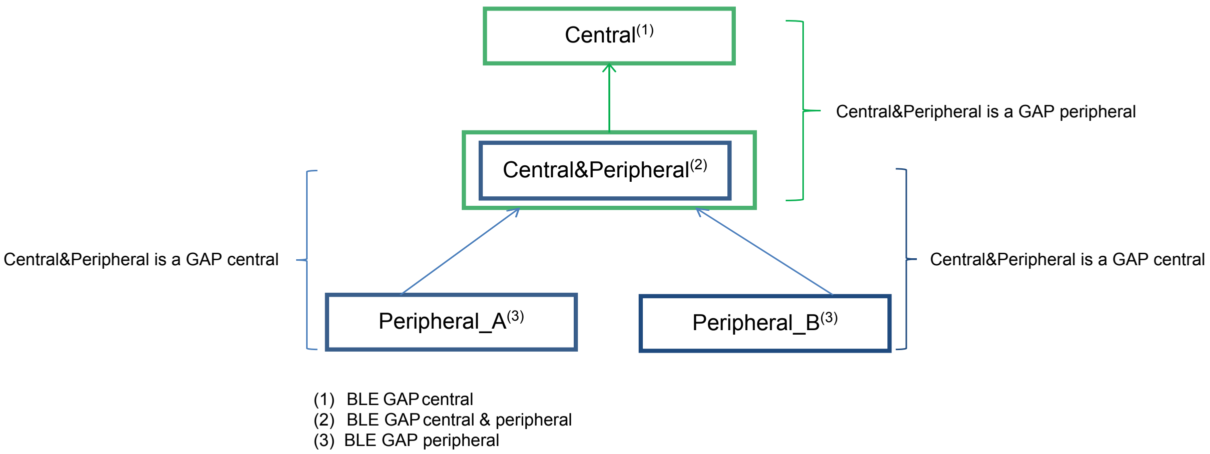
The STM32WB and STM32WBA BLE stacks support multiple roles simultaneously (for more details see Section 6: BLE multiple connection timing strategy). This allows the same device to act as central on one or more connections (up to eight connections are supported), and to act as a peripheral on another connection.

The following pseudo code describes how a BLE 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, &service_handle,
&dev_name_char_handle, &appearance_char_handle);
```

A simultaneous central and peripheral test scenario can be targeted as follows:

Figure 18. BLE simultaneous central and peripheral scenario



1. One BLE device (called central&peripheral) 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.
2. Two BLE devices (called peripheral_A, peripheral_B) are configured as peripheral by setting role as `GAP_PERIPHERAL_ROLE` on `GAP_Init()` API. Both peripheral_A and peripheral_B define the same service and characteristic as central&peripheral device.
3. One BLE device (called central) is configured as central by setting role as `GAP_CENTRAL_ROLE` on `GAP_Init()API`.
4. Both peripheral_A and peripheral_B devices enter discovery mode as follows:

```
ret =aci_gap_set_discoverable(Advertising_Type=0x00,
                              Advertising_Interval_Min=0x20,
                              Advertising_Interval_Max=0x100,
                              Own_Address_Type= 0x0;
                              Advertising_Filter_Policy= 0x00;
                              Local_Name_Length=0x05,
                              Local_Name=[0x08,0x74,0x65,0x73,0x74],
                              Service_Uuid_length = 0;
                              Service_Uuid_length = NULL;
                              peripheral_Conn_Interval_Min = 0x0006,
                              peripheral_Conn_Interval_Max = 0x0008);
```

5. central&peripheral device performs a discovery procedure in order to discover the peripheral devices peripheral_A and peripheral_B:

```
ret = aci_gap_start_gen_disc_proc (LE_Scan_Interval=0x10,
                                   LE_Scan_Window=0x10,
                                   Own_Address_Type = 0x0,
                                   Filter_Duplicates = 0x0);
```

   The two devices are discovered through the advertising report events notified with the `hci_le_advertising_report_event()` event callback.
6. Once the two devices are discovered, central&peripheral device starts two connection procedures (as central) to connect, respectively, to peripheral_A and peripheral_B devices:

```
/* Connect to peripheral_A:peripheral_Aaddress type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010
                               Peer_Address_Type= "peripheral_A address type"
                               Peer_Address= "peripheral_A address,
                               Own_Address_Type = 0x0;
                               Conn_Interval_Min=0x6c,
                               Conn_Interval_Max=0x6c,
                               Conn_Latency=0x00,
                               Supervision_Timeout=0xc80,
                               Minimum_CE_Length=0x000c,
                               Maximum_CE_Length=0x000c);
```

```
/* Connect to peripheral_B:peripheral_Baddress type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010,
                               Peer_Address_Type= "peripheral_B address type",
                               Peer_Address= "peripheral_B address",
                               Own_Address_Type = 0x0;
                               Conn_Interval_Min=0x6c,
                               Conn_Interval_Max=0x6c,
                               Conn_Latency=0x00,
                               Supervision_Timeout=0xc80,
                               Minimum_CE_Length=0x000c,
                               Maximum_CE_Length=0x000c);
```

7. Once connected, central&peripheral device enables the characteristics notification, on both of them, using the `aci_gatt_write_char_desc()` API. peripheral_A and peripheral_B devices start the characteristic notification by using the `aci_gatt_upd_char_val()` API.

8. At this stage, central&peripheral device enters discovery mode (acting as peripheral):

```
/*Put central&peripheral device in Discoverable Mode with Name = 'Test' = [0x08,0x74,0x65,
0x73,0x74*/
ret =aci_gap_set_discoverable(Advertising_Type=0x00,
                             Advertising_Interval_Min=0x20,
                             Advertising_Interval_Max=0x100,
                             Own_Address_Type= 0x0;
                             Advertising_Filter_Policy= 0x00;
                             Local_Name_Length=0x05,
                             Local_Name=[0x08,0x74,0x65,0x73,0x74],
                             Service_Uuid_length = 0;
                             Service_Uuid_List = NULL;
                             peripheral_Conn_Interval_Min = 0x0006,
                             peripheral_Conn_Interval_Max = 0x0008);
```

   Since central&peripheral device also acts as a central device, it receives the notification event related to the characteristic values notified from, respectively, peripheral_A and peripheral_B devices.

9. Once central&peripheral device enters discovery mode, it also waits for the connection request coming from the other BLE device (called central) configured as GAP central. central device starts discovery procedure to discover the central&peripheral device:

```
    ret = aci_gap_start_gen_disc_proc(LE_Scan_Interval=0x10,
                                      LE_Scan_Window=0x10,
                                      Own_Address_Type = 0x0,
                                      Filter_Duplicates = 0x0);
```

10. Once the central&peripheral device is discovered, central device starts a connection procedure to connect to it:

```
/* central device connects to central&peripheral device: central&peripheral
   address type and address have been found during the discovery
   procedure through the Advertising Report events */
   ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                                  LE_Scan_Window=0x0010,
                            Peer_Address_Type= "central&peripheral address type",
                            Peer_Address= " central&peripheral address",
                                  Own_Address_Type = 0x0;
                                  Conn_Interval_Min=0x6c,
                                  Conn_Interval_Max=0x6c,
                                  Conn_Latency=0x00,
                                  Supervision_Timeout=0xc80,
                                  Minimum _CE_Lenght=0x000c
                                  Maximum_CE_Length=0x000c);
```

   central&peripheral device is discovered through the advertising report events notified with the `hci_le_advertising_report_event()` event callback.

11. Once connected, central device enables the characteristic notification on central&peripheral device using the `aci_gatt_write_char_desc()` API.

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

5.2.1 STM32WB background scan aspect

As explain in previous chapter, in multi-role scenario, STM32WB device has opportunity to advertise, to scan and being connected at same time with different remote devices.

The STM32WB BLE stack offers, in this multi-role scenario, possibility to prioritize advertising or connection packet transmission over scanning process, while scanning procedure could be continued in background activity.

This feature is not activated by default: application could activate or deactivate at any time this new scanning policy, by using the following aci_hal_write_config_data() command, with offset parameter 0xC1.

Time slot scheduling in STM32WB is schedule as follow: advertising events are scheduled asynchronously, while scanning process operation is scheduled synchronously. Meaning time slot collision could happen while staying predictable.

On current STM32WB behavior, to avoid such predicted conflict, slot dedicated for advertising or scanning is postponed, to leave place to either connection, scanning or advertising slot, while the overall slot allocation is fair distributed on next expected transmission.

By introducing background scan on STM32WB, in case of conflict, slot priority always is allocated for advertising packet or connection slot against scanning slot.

This means that if application activate this new background scan policy, its scan windows are stopped automatically before predicted advertising or/and connection event occurs, in order to guarantee advertising or connection event transmission. And scanning process restarts automatically on next scanning interval, giving scan bandwidth process in-between advertising or connection event.

5.3 Bluetooth® Low Energy privacy

Both STM32WB and STM32WBA BLE stacks support the Bluetooth® Low Energy privacy 1.2.

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

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

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).

5.3.1 Controller-based privacy and the device filtering scenario

On STM32WB and STM32WBA with `aci_gap_init()` API, support the following options for the `privacy_enabled` parameter:

- 0x00: privacy disabled
- 0x02: controller privacy enabled

When a peripheral 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_enabled`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. Call the `aci_gap_configure_whitelist()` API to add the address of bonded device into the BLE device controller's whitelist.
4. Get the bonded device identity address and type using the `aci_gap_get_bonded_devices()` API.
5. Add the bonded device identity address and type to the list of address translations used to resolve resolvable private addresses in the controller, by using the `aci_gap_add_devices_to_resolving_list()` API.
6. The device enters the undirected connectable mode by calling the `aci_gap_set_undirected_connectable()` API with `Own_Address_Type = 0x02` (resolvable private address) and `Adv_Filter_Policy` = 0x03 (allow scan request from whitelist only, allow connect request from whitelist only).
7. When a bonded central device performs a connection procedure for reconnection to the peripheral device, the peripheral device is able to resolve and filter the central address and connect with it.

5.3.2 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, STM32WB and STM32WBA stacks automatically find 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 (for instance, 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 STM32WB or STM32WBA are in scanning, through `hci_le_advertising_report_event()`, or when a connection is established, through `hci_le_connection_complete_event()`.

Controller-based privacy

If the resolution of addresses is enabled at the 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_add_devices_to_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 (if the `hci_le_enhanced_connection_complete_event()` is masked, only the `hci_le_connection_complete_event()` is returned).

When scanning, the `hci_le_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 a directed advertisement, the resolved private address is reported through the `hci_le_advertising_report_event()` or through the `hci_le_direct_advertising_report_event()` if it has been unmasked and the scanner filer policy is set to 0x02 or 0x03.

How to retrieve a resolvable private address (RPA) when advertising with GAP privacy

Once advertising is started, it is possible to get the RPA currently used by issuing the command `hci_le_read_local_resolvable_address`. The peer address given in the parameter is the last one added in resolving the list using `aci_gap_add_devices_to_list` (or `aci_gap_add_devices_to_resolving_list`).

Attention, however, that this address changes regularly (see also `hci_le_set_resolvable_private_address_timeout`).

5.4 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_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.

5.5 LE data packet length extension APIs and events

On BLE specification v 4.2, the 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 STM32WB and STM32WBA stacks support LE data packet length extension features and related APIs, events:

- HCI LE APIs (API prototypes)
  – `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)
  – `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 on an STM32WB or STM32WBA device, 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.

In the case that the connection update parameter procedure is in progress, `hci_le_set_data_length` is not possible, but available once the connection update parameter is functional at link layer side. While the connection update parameter is in progress, `hci_le_set_data_length` returns 0x3A (controller busy).

5.6 STM32WB and STM32WBA LE 2M PHY

Introduced in the Bluetooth® core specification version 5.0, LE 2M PHY allows the physical layer to operate at higher data rate up to 2Mbit/s. LE 2M PHY double data rate versus standard LE 1M PHY, this reduces power consumption using the same transmit power. The transmit distance is lower than the LE 1M PHY, due to the increased symbol rate. Within the STM32WB and STM32WBA stacks, both LE 1M PHY and LE 2M PHY are supported, and it is up to the application to select default PHY requirement. Application can initiate a change of PHY parameters at any point of time and as often as required, with different PHY parameters on each connection channel selected (via connection handle). And since STM32WB and STM32WBA handle asymmetric connection, application can also use different PHYs in each direction of connection RX and TX (via connection handle). PHY negotiation is transparent at application side and depends on remote feature capabilities. STM32WB and STM32WBA stacks support the following commands:

- HCI_LE_SET_DEFAULT_PHY: to allow the host to specify its preferred for TX and RX PHY parameters.
- HCI_LE_SET_PHY: to allow the host to set PHY preferences for current connection (identified by the connection handle) for TX and RX PHY parameters.
- HCI_LE_READ_PHY: to hallow the host to read TX and RX PHY parameters on current connection(identify by connection handle).

5.7 STM32WBA LE Coded PHY

This LE Coded PHY feature is available for STM32WBA devices only.

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.

STM32WBA devices use 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) value and, as a consequence, it increases the transmitter distance without the need to increase the transmitter power level.

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:

- FEC encoding, which generates two further bits for each bit
- Pattern mapper, which converts each bit from 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).

STM32WBA main characteristics regarding Coded PHY:

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

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 52. STM32WBA 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® Low Energy specification 5.x requirement type	Mandatory	Optional	Optional	Optional

5.8 STM32WB LE additional beacon

Introduced as a proprietary solution for STM32WB devices only, this feature allows the end user to get an additional advertising beacon, behaving as an extra beacon (not connectable) in addition or not to the basic advertising feature.

STM32WB extra beacon solution is proposed with undirected nonconnectable mode, without privacy feature, and filter accept list ignored. The extra beacon includes a selection set of three fixed 1 Mbit/s PHY channels (channels 37, 38 and 39), with dedicated Tx power level for this set, and its advertising data refers to the raw 0...31 bytes long payload that is available for application use.

The address can be a random address or public address (if it is not currently used by the standard advertising) and it is up to the end-user to write this new BD address (both addresses: standard advertising and extra beacon are different addresses).

The application can initialize extra beacon feature via GAP command level (no HCI commands related), such that STM32WB LE additional beacon supports:

- GAP_Additional_Beacon_Mode_Start to allow the host to start an additional beacon with the following parameters:
  - Advertising type: ADV_NONCONN_IND (nonconnectable undirected advertising) and parameter for further enhancements
  - Advertising interval Min/Max: range from 20 ms up to 10.24 s
  - Own beacon address type and value: type of address (Public or Random) and end-user write new BD address Value
  - AdDataLen and AdvData: length of the data and its data value
  - PA level: the transmission output level in dBm ranges from - 40dBm up to 5 dBm
- GAP_Additional_Beacon_Mode_Stop: to allow the host to stop the additional beacon
- GAP_Additional_Beacon_Update_Data: to allow the host to change additional beacon data with the following parameters:
  - AdvDataLen and AdvData: length of the data and its data value

Note: The advertising always reserves 14.6 ms (10 ms for random advertising delay and remaining for three channels advertising, scan req/rsp and guard time).

The advertising interval is selected to allow room for additional advertising.

The first advertising enabled selects the HostBaseTime. All additional Min/Max intervals are allowed to fit with it as a modulo (a margin of timings computation is required when selecting Min/Max advertising).

The old advertising slot reservation remains impacting the new advertising Min interval. If the Min interval is increased with one slot length the reservation is accepted, and if it is below it is not accepted.

5.9 STM32WB and STM32WBA LE extended advertising

Introduced in the Bluetooth® core specification version 5.0, LE extended advertising allows the end user to advertise and discover more data than previous "legacy advertising".

This advertising extension capability allows:

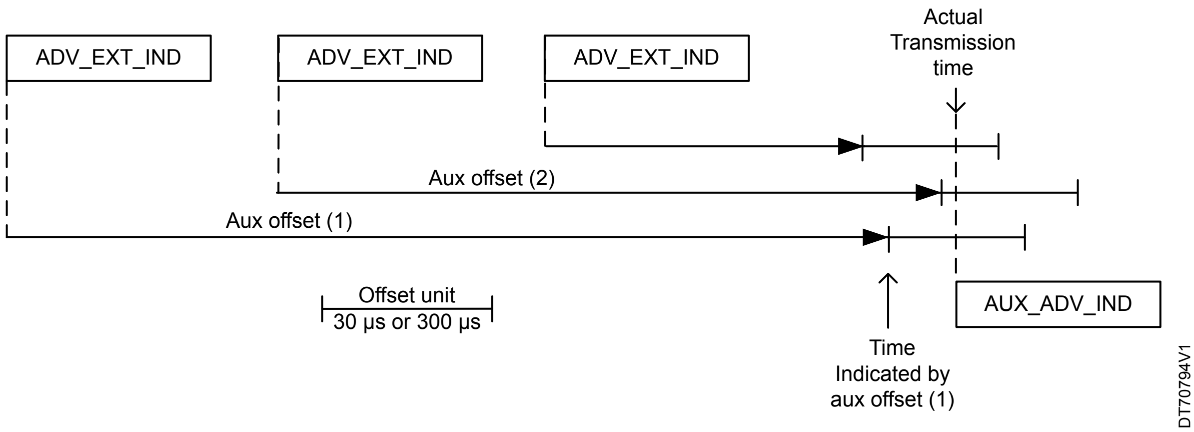
- to extend the data length in connectionless scenarios
- to have multiple sets of advertising data to be sent
- to have advertising sent in a deterministic way

5.9.1 Extended advertising set

Initial advertising and legacy PDUs are transmitted on 3 RF channels (37.38, 39), known as “primary advertising physical channel”. New extended advertising packets can use the Bluetooth® Low Energy 4.x connection channels (0-36) for extending the advertising payload, known as “secondary advertising physical channel”.

ADV_EXT_IND new packet can be sent on the primary advertising PHY channel. The header field includes a new data AuxPtr, which contains the channel number (0-36), and a pointer to an auxiliary packet on the secondary adv phy channel: most of the info is on the auxiliary packet called AUX_ADV_IND (see Figure 19).

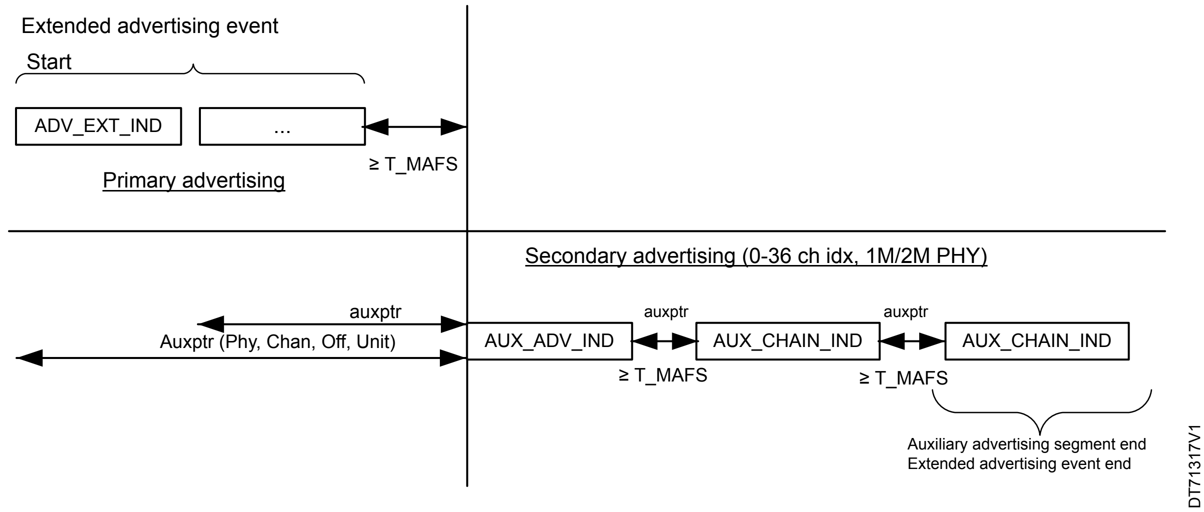
Figure 19. Example of advertising set



This AUX_ADV_IND packet (up to 207 bytes) could be sent on either 1Mbit PHY or 2 Mbit PHY as defined previously in the ADV_EXT_IND packet.

It is also possible to create a chain of advertising packets on secondary channels in order to transmit more advertising payload data: AUX_CHAIN_IND data packet (up to 1650 bytes) as described in Figure 20. Each advertising packet on a secondary channel includes on its AuxPtr the number of the next secondary channel for the next adverting packet on the chain. As a consequence, each chain in the advertising packet chain can be sent on a different secondary channel.

Figure 20. Example of chained advertising set



5.9.2 Extended scannable set

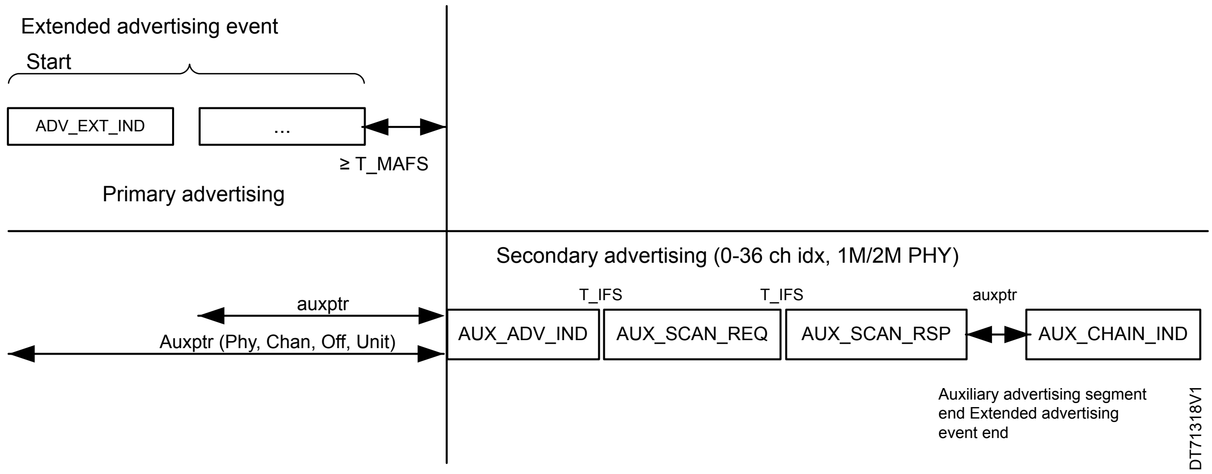
Extended scannable set allows the advertiser to send data only if a scan request is received, and only responds with data on the secondary advertising channel index.

Advertising in the undressed event type using the ADV_EXT_IND in the primary channel indicates the coming AUX_ADV_IND on the secondary advertising channel. The scanner requests more information via AUX_SCAN_REQ, and the advertiser responds with AUX_SCAN_RSP on the same secondary advertising channel index (as describe in Figure 21).

As indicated for the advertising set, it is also possible in the extended scannable set to create a chain of advertising packets on secondary channels in order to transmit more advertising payload data:

AUX_CHAIN_IND data packet (up to 1650 bytes).

Figure 21. Example of scannable set

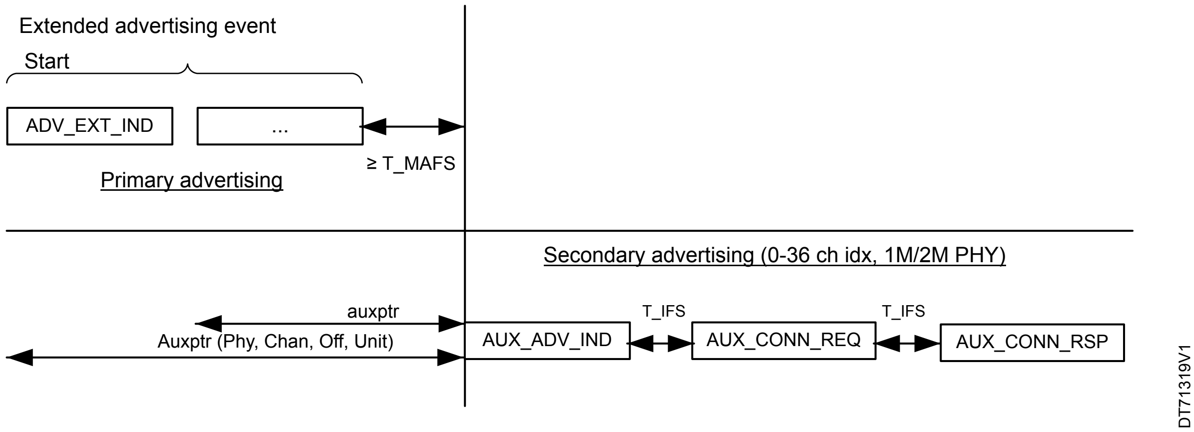


5.9.3 Extended connectable set

The connectable directed advertising event type using ADV_EXT_IND allows also an initiator to respond with a connect request on the secondary advertising physical channel to establish an ACL connection.

After every AUX_ADV_IND related to this event, the scanner sends AUX_CONNECT_REQ on the same secondary advertising channel index, and the advertiser responds with AUX_CONNECT_RSP, as shown in Figure 22.

Figure 22. Example of connectable set



5.9.4 Extended multiple sets

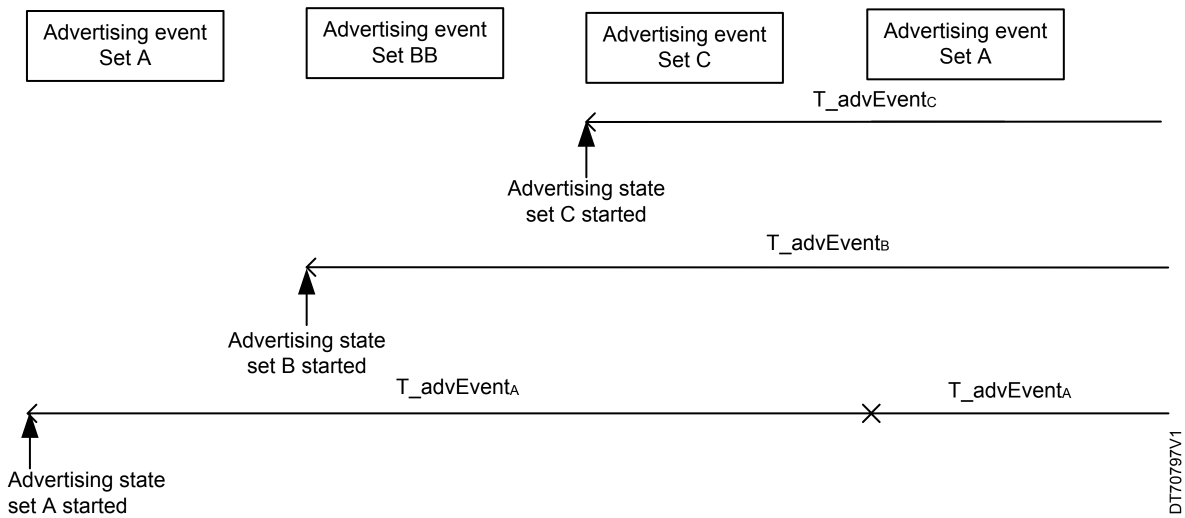
In the STM32WB and STM32WBA stacks, concerning advertising event type, the end user can define up to eight sets being a combination of either extended advertising sets, extended scannable sets, or extended connectable sets. For the STM32WBx stack only, among these possible combinations sets, only two sets can be legacy advertising with only one set to connectable mode.

Data transmission depends on the number of sets defined by the end user. With eight different sets, the maximum amount of data fairly distributed among all sets is up to 463 bytes for STM32WBA and 207 bytes for STM32WB. Transmitting a maximum of 1650 bytes of data is possible up to three sets.

For advertising parameters, such as advertising PDU type, advertising interval, and PHY, the end user can define each set with different parameters.

Note: When advertising with the ADV_EXT_IND or AUX_ADV_IND PDUs, the advertising set is identified by the advertising SID subfield of the ADI field.

Figure 23. Example of extended multiple sets



5.9.5 LE extended scanning

On the same basis, the STM32WB and STM32WBA stacks allow the extended scanning feature.

As the 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:

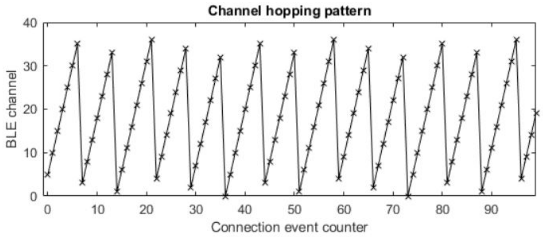
- Legacy events

- Extended adverting events, possibly switching to other PHYs on secondary advertising physical channel.
  The extended scanning is also available for multiple scan sets (up to 8). While currently the privacy feature is not supported when using the extended scanning interface.

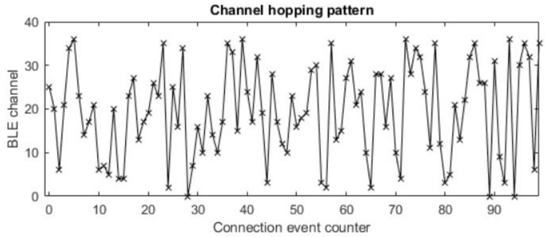
Note: The extended scan feature on STM32WB shows the following behavior when scanning with the maximum duty cycle (that is, when the scan window equals the scan interval): If at the end of the scan window, a long advertising sequence containing many AUX_ CHAIN_IND (non-scannable advertisements, or scan response to a scannable advertisement), the STM32WB continues to catch all AUX_ CHAIN_IND packets, and therefore, can overlap with the moment when the next scan window should start. In this case, the next scan window is skipped completely (not executed) to privilege the full reception of the long advertisement sequence.

In both extended advertising and extended scanning, the new channel selection algorithm (CSA) #2 is mandatory to be used. This new algorithm is more complex and harder to track for obtaining the channel index for the next connection event. And it is more effective at avoiding interference and multipath fading effects than CSA #1. Figure 24 shows these two different algorithms.

Figure 24. Two different channel hopping systems



CSA#1



CSA#2

5.9.6 Legacy and extended advertising/scanning commands and event impact

As defined by Bluetooth® specifications, commands and events are impacted by the mode selected by the application, either *legacy* or *extended*.

The following section lists all legacy and extended advertising/scanning commands and events impacted according to the application's choice.

On the STM32WBA application, the dedicated mode can be selected by:

- Setting the extended option bit (BLE_OPTIONS_EXTENDED_ADV) to 0 or 1 in the BLE stack initialization (BleStack_Init)

If the option bit is set to 1, the extended mode is selected, otherwise the legacy mode is selected.

Once the mode is selected, the user can call the commands dedicated to the selected mode. The periodic commands are only supported in the extended mode. These commands are listed in Table 53 highlighted in green.

Refer to the following tables to find the allowed commands for each mode (legacy or extended).

5.9.6.1 Full mode commands (legacy and extended)

Table 53. Full mode commands for advertising/scanning/connection legacy or extended requirements

Mode	Legacy command	Extended command
Advertising	ACI_GAP_SET_BROADCAST_MODE	-
	ACI_GAP_SET_NON_DISCOVERABLE	-
	HCI_LE_SET_SCAN_RESPONSE_DATA	ACI_GAP_ADV_SET_SCAN_RESP_DATA
	ACI_GAP_SET_LIMITED_DISCOVERABLE	-
	ACI_GAP_SET_DISCOVERABLE	-
	ACI_GAP_UPDATE_ADV_DATA	-
	-	ACI_GAP_ADV_SET_ENABLE
	ACI_GAP_SET_NON_CONNECTABLE	-

Mode	Legacy command	Extended command
Advertising	ACI_GAP_SET_UNDIRECTED_CONNECTABLE	-
	ACI_GAP_SET_DIRECT_CONNECTABLE	-
	-	HCI_LE_READ_MAXIMUM_ADVERTISING_DATA_LENGTH
	-	ACI_GAP_ADV_SET_CONFIGURATION
	-	ACI_GAP_ADV_SET_ADV_DATA
	-	HCI_LE_READ_NUMBER_OF_SUPPORTED_ADVERTISING _SETS
	-	ACI_GAP_ADV_REMOVE_SET
	-	ACI_GAP_ADV_CLEAR_SETS
	-	ACI_GAP_ADV_SET_PERIODIC_PARAMETERS
	-	ACI_GAP_ADV_SET_PERIODIC_DATA
	-	ACI_GAP_ADV_SET_PERIODIC_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_RECEIVE_ENABLE
	-	HCI_LE_PERIODIC_ADVERTISING_SET_INFO_TRANSFER
	-	HCI_LE_SET_PERIODIC_SYNC_TRANSFER_PARAMETERS
	-	HCI_LE_SET_DEFAULT_PERIODIC_ADVERTISING_SYNC _TRANSFER_PARAMETERS
Scanning	ACI_GAP_START_OBSERVATION_PROC	ACI_GAP_START_OBSERVATION_PROC(1)
	ACI_GAP_START_GENERAL_DISCOVERY_PROC	ACI_GAP_START_GENERAL_DISCOVERY_PROC(1)
	ACI_GAP_START_GENERAL_CONNECTION _ESTABLISH_PROC	ACI_GAP_START_GENERAL_CONNECTION_ESTABLISH_PROC (1)
	ACI_GAP_START_SELECTIVE_CONNECTION _ESTABLISH_PROC	ACI_GAP_START_SELECTIVE_CONNECTION_ESTABLISH_PROC(1)
	ACI_GAP_START_LIMITED_DISCOVERY_PROC	ACI_GAP_START_LIMITED_DISCOVERY_PROC (1)
	-	ACI_GAP_EXT_START_SCAN
	ACI_GAP_TERMINATE_GAP_PROC	ACI_GAP_TERMINATE_GAP_PROC
	-	HCI_LE_PERIODIC_ADVERTISING_SYNC_TRANSFER
	-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC
	-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC_CANCEL
	-	HCI_LE_PERIODIC_ADVERTISING_TERMINATE_SYNC
Connection	ACI_GAP_CREATE_CONNECTION	ACI_GAP_CREATE_CONNECTION(2)
	ACI_GAP_START_AUTO_CONNECTION _ESTABLISH_PROC	ACI_GAP_START_AUTO_CONNECTION_ESTABLISH _PROC(2)
	-	ACI_GAP_EXT_CREATE_CONNECTION
	ACI_GAP_START_CONNECTION_UPDATE	ACI_GAP_START_CONNECTION_UPDATE
	ACI_GAP_TERMINATE_GAP_PROC	ACI_GAP_TERMINATE_GAP_PROC
Disconnection	ACI_GAP_TERMINATE	ACI_GAP_TERMINATE

1. *Those ACI GAP commands are also supported on Extended mode, but the most appropriate ACI command to use in Extended mode is the ACI_GAP_EXT_START_SCAN command.*
2. *Those ACI GAP commands are also supported on Extended mode, but the most appropriate ACI command to use in Extended mode is the ACI_GAP_EXT_CREATE_CONNECTION command.*

Table 54. Full mode with dedicated legacy and extended events

Modes	Legacy command	Extended command
Advertising	ACI_GAP_LIMITED_DISCOVERABLE_EVENT	-
	-	HCI_LE_SCAN_REQUEST_RECEIVED_EVENT
Scanning	HCI_LE_ADVERTISING_REPORT_EVENT	HCI_LE_EXTENDED_ADVERTISING_REPORT _EVENT
	HCI_LE_DIRECTED_ADVERTISING_REPORT _EVENT	HCI_LE_EXTENDED_ADVERTISING_REPORT _EVENT
Periodic	-	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
Connection	HCI_LE_CONNECTION_COMPLETE_EVENT	HCI_LE_ENHANCED_CONNECTION_COMPLETE _EVENT
Disconnection	HCI_DISCONNECTION_COMPLETE_EVENT	HCI_DISCONNECTION_COMPLETE _EVENT
General	ACI_GAP_PROC_COMPLETE_EVENT	ACI_GAP_PROC_COMPLETE_EVENT

5.9.6.2 *HCI mode only (legacy and extended)*

Table 55. HCI mode commands for advertising/scanning/connection legacy or extended requirements

Modes	Legacy command	Extended command
Advertising	HCI_LE_SET_ADVERTISING_DATA	HCI_LE_SET_EXTENDED_ADVERTISING_DATA
	HCI_LE_SET_ADVERTISING_PARAMETERS	HCI_LE_SET_EXTENDED_ADVERTISING_PARAMETERS
	HCI_LE_SET_ADVERTISING_ENABLE	HCI_LE_SET_EXTENDED_ADVERTISING_ENABLE
	HCI_LE_SET_SCAN_RESPONSE_DATA	HCI_LE_EXTENDED_SCAN_RESPONSE_DATA
	-	HCI_LE_READ_MAXIMUM_ADVERTISING_DATA_LENGTH
	-	HCI_LE_READ_NUMBER_OF_SUPPORTED_ADVERTISING_SETS
	-	HCI_LE_REMOVE_ADVERTISING_SET
	-	HCI_LE_CLEAR_ADVERTISING_SETS
Scanning	HCI_LE_SET_SCAN_PARAMETERS	HCI_LE_SET_EXTENDED_SCAN_PARAMETERS
	HCI_LE_SET_SCAN_ENABLE	HCI_LE_SET_EXTENDED_SCAN_ENABLE
Periodic advertising	-	HCI_LE_SET_PERIODIC_ADVERTISING_PARAMETERS
	-	HCI_LE_SET_PERIODIC_ADVERTISING_DATA
	-	HCI_LE_SET_PERIODIC_ADVERTISING_ENABLE
	-	HCI_LE_ADD_DEVICE_TO_PERIODIC_ADVERTISER_LIST
	-	HCI_LE_REMOVE_DEVICE_FROM_PERIODIC_ADVERTISER_LIST
	-	HCI_LE_CLEAR_PERIODIC_ADVRTISER_LIST

Modes	Legacy command	Extended command
Periodic advertising	-	HCI_LE_READ_PERIODIC_ADVERTISER_LIST_SIZE
	-	HCI_LE_SET_PERIODIC_ADVERTISING_RECIEVE_ENABLE
	-	HCI_LE_PERIODIC_ADVERTISING_SET_INFO_TRANSFER
	-	HCI_LE_SET_PERIODIC_SYNC_TRANSFER_PARAMETERS
	-	HCI_LE_SET_DEFAULT_PERIODIC_ADVERTISING_SYNC _TRANSFER_PARAMETERS
Periodic scanning	-	HCI_LE_PERIODIC_ADVERTISING_SYNC_TRANSFER
	-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC
	-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC_CANCEL
	-	HCI_LE_PERIODIC_ADVERTISING_TERMINATE_SYNC
Connection	HCI_LE_CREATE_CONNECTION	HCI_LE_EXTENDED_CREATE_CONNECTION
	HCI_LE_CONNECTION_UPDATE	HCI_LE_CONNECTION_UPDATE
	HCI_LE_CREATE_CONNECTION_CANCEL	HCI_LE_CREATE_CONNECTION_CANCEL
Disconnection	HCI_DISCONNECT	HCI_DISCONNECT

Table 56. **HCI mode with dedicated legacy and extended events**

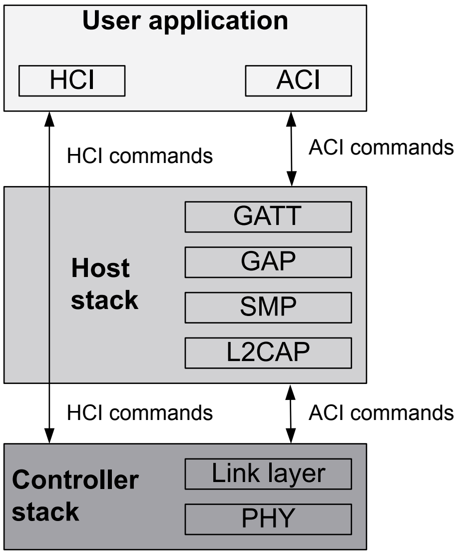
Modes	Legacy event	Extended event
Advertising	HCI_LE_ADVERTISING_REPORT_EVENT	HCI_LE_EXTENDED_ADVERTISING_REPORT_EVENT
	HCI_LE_DIRECTED_ADVERTISING_REPORT_EVENT	
Connection	HCI_LE_CONNECTION_COMPLETE_EVENT	HCI_LE_CONNECTION_COMPLETE_EVENT
	-	HCI_LE_ENHANCED_CONNECTION _COMPLETE_EVENT
	HCI_LE_CONNECTION_UPDATE_COMPLETE_EVENT	HCI_LE_CONNECTION_UPDATE_COMPLETE_EVENT
Disconnection	HCI_DISCONNECTION_COMPLETE_EVENT	HCI_DISCONNECTION_COMPLETE_EVENT
Scanning	-	HCI_LE_SCAN_REQUEST_RECEIVED_EVENT
Periodic	-	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

5.9.7 ACI command guidelines for user applications

The STM32WB and STM32WBA stacks are built with the host and controller stacks combined in the same solution. This way, the user application has direct access to these stacks, through a dedicated application command interface (ACI).

Moreover, this ACI also supports HCI commands (command, ACL data, and event packets), so all ACI/HCI commands received from the application are checked and directed either to the host stack layer or the controller layer (bypassing the host). Figure 25 shows an ACI/HCI command diagram.

Figure 25. User application diagram for ACI/HCI commands



5.9.8 HCI command guidelines for user applications

HCI commands can be forwarded to the controller without informing the host stack. Several parameters that have already defined by the host (or not) can be (over)written by the controller and are thus not considered by the host.

To avoid such invalid operations, Table 57 lists all HCI commands that a user application can execute without interfering with the host for STM32WB and STM32WBA host stacks.

Table 57. List of HCI commands available to user applications with host stack

STM32WB devices	STM32WBA devices
HCI_READ_REMOTE_VERSION_INFORMATION	
HCI_SET_EVENT_MASK	
HCI_RESET	
HCI_READ_RSSI	
HCI_READ_RSSI HCI_LE_SET_EVENT_MASK	
HCI_LE_SET_SCAN_RESPONSE_DATA	
HCI_LE_SET_HOST_CHANNEL_CLASSIFICATION	
HCI_LE_READ_CHANNEL_MAP	
HCI_LE_READ_REMOTE_FEATURES	
HCI_LE_ENCRYPT	
HCI_LE_RAND	
HCI_LE_SET_DATA_LENGTH	
HCI_LE_READ_SUGGESTED_DEFAULT_DATA_LENGTH	
HCI_LE_WRITE_SUGGESTED_DEFAULT_DATA_LENGTH	
HCI_LE_READ_LOCAL_P256_PUBLIC_KEY	
HCI_LE_SET_RESOLVABLE_PRIVATE_ADDRESS_TIMEOUT	

STM32WB devices	STM32WBA devices
HCI_LE_READ_MAXIMUM_DATA_LENGTH	
-	HCI_READ_AFH_CHANNEL_ASSESSMENT_MODE
-	HCI_WRITE_AFH_CHANNEL_ASSESSMENT_MODE
-	HCI_READ_CONNECTION_ACCEPT_TIMEOUT
-	HCI_WRITE_CONNECTION_ACCEPT_TIMEOUT
-	HCI_SET_EVENT_MASK_PAGE_2
-	HCI_READ_AUTHENTICATED_PAYLOAD_TIMEOUT
-	HCI_WRITE_AUTHENTICATED_PAYLOAD_TIMEOUT
-	HCI_SET_ECOSYSTEM_BASE_INTERVAL
-	HCI_CONFIGURE_DATA_PATH
-	HCI_READ_LOCAL_SUPPORTED_CODECS_V2
-	HCI_READ_LOCAL_SUPPORTED_CODEC_CAPABILITIES
-	HCI_READ_LOCAL_SUPPORTED_CONTROLLER_DELAY
-	HCI_LE_SET_DATA_RELATED_ADDRESS_CHANGES
-	HCI_LE_REMOTE_CONNECTION_PARAMETER_REQUEST_REPLY
-	HCI_LE_REMOTE_CONNECTION_PARAMETER_REQUEST_NEGATIVE_REPLY
-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC
-	HCI_LE_PERIODIC_ADVERTISING_CREATE_SYNC_CANCEL
-	HCI_LE_PERIODIC_ADVERTISING_TERMINATE_SYNC
-	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_RECEIVE_ENABLE
-	HCI_LE_PERIODIC_ADVERTISING_SYNC_TRANSFER
-	HCI_LE_PERIODIC_ADVERTISING_SET_INFO_TRANSFER
-	HCI_LE_SET_PERIODIC_ADVERTISING_SYNC_TRANSFER_PARAMETERS
-	HCI_LE_SET_DEFAULT_PERIODIC_ADVERTISING_SYNC_TRANSFER_PARAMETERS
-	HCI_LE_SET_CONNECTIONLESS_CTE_TRANSMIT_PARAMETERS
-	HCI_LE_SET_CONNECTIONLESS_CTE_TRANSMIT_ENABLE
-	HCI_LE_SET_CONNECTIONLESS_IQ_SAMPLING_ENABLE
-	HCI_LE_SET_CONNECTION_CTE_RECEIVE_PARAMETERS
-	HCI_LE_SET_CONNECTION_CTE_TRANSMIT_PARAMETERS
-	HCI_LE_CONNECTION_CTE_REQUEST_ENABLE
-	HCI_LE_CONNECTION_CTE_RESPONSE_ENAB
-	HCI_LE_READ_ISO_TX_SYNC
-	HCI_LE_SET_CIG_PARAMETERS
-	HCI_LE_CREATE_CIS
-	HCI_LE_REMOVE_CIG
-	HCI_LE_ACCEPT_CIS_REQUEST

STM32WB devices	STM32WBA devices
-	HCI_LE_REJECT_CIS_REQUEST
-	HCI_LE_CREATE_BIG
-	HCI_LE_TERMINATE_BIG
-	HCI_LE_BIG_CREATE_SYNC
-	HCI_LE_BIG_TERMINATE_SYNC
-	HCI_LE_SETUP_ISO_DATA_PATH
-	HCI_LE_REMOVE_ISO_DATA_PATH
-	HCI_LE_READ_ISO_LINK_QUALITY
-	HCI_LE_SET_HOST_FEATURE
-	HCI_LE_MODIFY_SLEEP_CLOCK_ACCURACY
-	HCI_LE_REQUEST_PEER_SCA
-	HCI_LE_ENHANCED_READ_TRANSMIT_POWER_LEVEL
-	HCI_LE_READ_REMOTE_TRANSMIT_POWER_LEVEL
-	HCI_LE_SET_TRANSMIT_POWER_REPORTING_ENABLE
-	HCI_LE_SET_PATH_LOSS_REPORTING_PARAMETERS
-	HCI_LE_SET_PATH_LOSS_REPORTING_ENABLE
-	HCI_LE_SET_DEFAULT_SUBRATE
-	HCI_LE_SUBRATE_REQUEST

User applications that require some dedicated information from the controller or use HCI commands for test purposes or for specific proprietary (nonstandard) development, Table 58 provides a list of HCI commands they can use.

Attention: *These commands are available only for testing purposes or specific proprietary (nonstandard) development.*

Table 58. List of HCI commands available to user applications for tests or information purposes

STM32WB devices	STM32WBA devices
HCI_READ_LOCAL_VERSION_INFORMATION	
HCI_READ_LOCAL_SUPPORTED_COMMANDS	
HCI_READ_LOCAL_SUPPORTED_FEATURES	
HCI_READ_BD_ADDR HCI_LE_READ_BUFFER_SIZE	
HCI_LE_READ_LOCAL_SUPPORTED_FEATURES	
HCI_LE_READ_ADVERTISING_PHYSICAL_CHANNEL_TX_POWER	
HCI_LE_SET_ADVERTISING_DATA	
HCI_LE_READ_FILTER_ACCEPT_LIST_SIZE	
HCI_LE_LONG_TERM_KEY_REQUEST_REPLY	
HCI_LE_LONG_TERM_KEY_REQUEST_NEGATIVE_REPLY	
HCI_LE_READ_SUPPORTED_STATES	
HCI_LE_READ_RESOLVING_LIST_SIZE	
HCI_LE_READ_PEER_RESOLVABLE_ADDRESS	
HCI_LE_READ_LOCAL_RESOLVABLE_ADDRESS	
HCI_LE_READ_MAXIMUM_ADVERTISING_DATA_LENGTH	
HCI_LE_READ_NUMBER_OF_SUPPORTED_ADVERTISING_SETS	

STM32WB devices	STM32WBA devices
HCI_LE_READ_TRANSMIT_POWER	
HCI_LE_READ_RF_PATH_COMPENSATION	
HCI_LE_WRITE_RF_PATH_COMPENSATION	
HCI_LE_RECEIVER_TEST	
HCI_LE_TRANSMITTER_TEST	
HCI_LE_TEST_END	
HCI_LE_RECEIVER_TEST_V2	
HCI_LE_TRANSMITTER_TEST_V2	
HCI_LE_GENERATE_DHKEY(1)	
-	HCI_LE_RECEIVER_TEST_V3
-	HCI_LE_TRANSMITTER_TEST_V3
-	HCI_LE_READ_ANTENNA_INFORMATION
-	HCI_LE_READ_BUFFER_SIZE_V2
-	HCI_LE_SET_CIG_PARAMETERS_TEST
-	HCI_LE_CREATE_BIG_TEST
-	HCI_LE_ISO_TRANSMIT_TEST
-	HCI_LE_ISO_RECEIVE_TEST
-	HCI_LE_ISO_READ_TEST_COUNTERS
-	HCI_LE_ISO_TEST_END
-	HCI_LE_TRANSMITTER_TEST_V4

1. *This command must not be called when a pairing operation is in progress.*

To avoid interference or interaction issues with the host stack, the user application must not use the HCI commands listed in Table 59 when the host stack is present.

Warning: *These commands must **not** be used when the host stack is present.*

Table 59. List of HCI commands user applications must not use with the host stack

STM32WB devices	STM32WBA devices
HCI_SET_CONTROLLER_TO_HOST_FLOW_CONTROL	
HCI_HOST_BUFFER_SIZE	
HCI_DISCONNECT	
HCI_HOST_NUMBER_OF_COMPLETED_PACKETS	
HCI_LE_SET_RANDOM_ADDRESS	
HCI_LE_SET_ADVERTISING_PARAMETERS	
HCI_LE_SET_ADVERTISING_ENABLE	
HCI_LE_SET_SCAN_PARAMETERS	
HCI_LE_SET_SCAN_ENABLE	
HCI_LE_CREATE_CONNECTION	
HCI_LE_CREATE_CONNECTION_CANCEL	
HCI_LE_CLEAR_FILTER_ACCEPT_LIST	
HCI_LE_ADD_DEVICE_TO_FILTER_ACCEPT_LIST	

STM32WB devices	STM32WBA devices
HCI_LE_REMOVE_DEVICE_FROM_FILTER_ACCEPT_LIST	
HCI_LE_CONNECTION_UPDATE	
HCI_LE_ENABLE_ENCRYPTION	
HCI_LE_ADD_DEVICE_TO_RESOLVING_LIST	
HCI_LE_REMOVE_DEVICE_FROM_RESOLVING_LIST	
HCI_LE_CLEAR_RESOLVING_LIST	
HCI_LE_SET_ADDRESS_RESOLUTION_ENABLE	
HCI_LE_SET_ADVERTISING_SET_RANDOM_ADDRESS	
HCI_LE_SET_EXTENDED_ADVERTISING_PARAMETERS	
HCI_LE_SET_EXTENDED_ADVERTISING_DATA	
HCI_LE_SET_EXTENDED_SCAN_RESPONSE_DATA	
HCI_LE_SET_EXTENDED_ADVERTISING_ENABLE	
HCI_LE_REMOVE_ADVERTISING_SET	
HCI_LE_CLEAR_ADVERTISING_SETS	
HCI_LE_SET_EXTENDED_SCAN_PARAMETERS	
HCI_LE_SET_EXTENDED_SCAN_ENABLE	
HCI_LE_EXTENDED_CREATE_CONNECTION	
HCI_LE_GENERATE_DHKEY_V2	
-	HCI_LE_SET_PERIODIC_ADVERTISING_PARAMETERS
-	HCI_LE_SET_PERIODIC_ADVERTISING_DATA
-	HCI_LE_SET_PERIODIC_ADVERTISING_ENABLE
-	HCI_LE_SET_EXTENDED_ADVERTISING_PARAMETERS_V2

5.10 Encrypted advertising data

STM32WB and STM32WBA BLE stacks support the *encrypted advertising data* feature. Bluetooth® 5.4 allows the encryption of advertising data in a standardized way.

5.10.1 Encrypted data key material sharing (advertiser side)

Bluetooth® 5.4 standard defines a new GAP characteristic called *encrypted data key material* to share encryption key material (Key and IV), that is subsequently used by data encryption or decryption.

On STM32WB and STM32WBA devices, this characteristic is not created within the BLE stack at GAP initialization. The application must create this characteristic, as follows:

1. Before calling the ACI_GAP_INIT() command, the application must reserve the attribute handles for this characteristic within the GAP service using the command ACI_HAL_WRITE_CONFIG_DATA with the following parameters:
   – Offset = CONFIG_DATA_GAP_ADD_REC_NBR_OFFSET
   – Length = 1
   – Value = 3
2. After calling the ACI_GAP_INIT(), the application must add the *encrypted data key material* characteristic to this new GAP service using the command ACI_GATT_ADD_CHAR with the following parameters:
   – Service_Handle = GAP Service Handle
   – Char_UUID_Type = UUID_TYPE_16
   – Char_UUID_16 = ENCRYPTED_DATA_KEY_MATERIAL_UUID
   – Char_Value_Length = 24
   – Char_Properties = CHAR_PROP_READ | CHAR_PROP_INDICATE
   – Security_Permissions = ATTR_PERMISSION_AUTHEN_READ | ATTR_PERMISSION_AUTHOR_READ
   – GATT_Evt_Mask = GATT_NOTIFY_ATTRIBUTE_WRITE
   – Enc_Key_Size = 16
   – Is_Variable = 0
3. Each time the application computes (or changes) key material (Key and IV), the application must update the characteristic value using ACI_GATT_UPDATE_CHAR_VALUE (or ACI_GATT_UPDATE_CHAR_VALUE_EXT).
4. On pairing completion with a central device, the application grants (or not) the authorization to read the characteristic using the command ACI_GAP_AUTHORIZATION_RESP.

5.10.2 Encryption of advertising data (advertiser side)

Advertising data can be encrypted using the command ACI_HAL_EAD_ENCRYPT_DECRYPT with the following parameters:

- Mode = 0
- Key = pointer to the session key used for EAD operation (in little endian format)
- IV = pointer to the initialization vector used for EAD operation (in little endian format)
- In_Data_Length = advertising data length in bytes
- In_Data = pointer to the advertising data

The application must then encapsulate the encrypted advertising data in an AD structure using AD type *encrypted advertising data* (0x31). This AD structure can then be set as advertising data using a GAP command such as ACI_GAP_UPDATE_ADV_DATA or ACI_GAP_ADV_SET_ADV_DATA.

Note: *If privacy is used, it is recommended to re-encrypt the advertising data and update it after each advertising packet (even if the advertising data has not changed). This can be achieved by using ACI_HAL_END_OF_RADIO_ACTIVITY_EVENT.*

The BLE stack automatically regenerates the randomizer value at each call of ACI_HAL_EAD_ENCRYPT_DECRYPT.

The advertising data to be encrypted must be a sequence of one or more AD structures.

On STM32WB, the advertising data length that can be encrypted is limited to 228 bytes.

5.10.3 Decryption of advertising data (scanner side)

On STM32WB and STM32WBA devices, the BLE stack does not automatically decrypt the received encrypted advertising data (encrypted data in advertising reports are left as is). The application must decrypt the data with AD type *encrypted advertising data* (0x31). To perform this operation, the application can use the command ACI_HAL_EAD_ENCRYPT_DECRYPT with the following parameters:

- Mode = 1
- Key = pointer to the session key used for EAD operation (in little endian format)
- IV = pointer to the initialization vector used for EAD operation (in little endian format)
- In_Data_Length = encrypted data length in bytes
- In_Data = pointer to the encrypted data

Note: The Key and IV parameters must be retrieved first by securely connecting to the server and reading the encrypted data key material characteristic.

On STM32WB, the encrypted data length is limited to 228 bytes, which corresponds to 219 bytes of plain data.

5.11 L2CAP connection oriented channels

STM32WB and STM32WBA BLE stacks support the *(enhanced) Connection Oriented Channels* feature (noted *COC* in the document). This L2CAP feature enables the transfer of bidirectional data between two Bluetooth® Low Energy devices.

5.11.1 L2CAP COC configuration

The COC feature is activated by default. It does not need to be explicitly activated by the application during BLE stack initialization (refer to initialization phase and main application loop). However, some parameters must be passed at BLE stack initialization to configure the COC feature:

- `max_coc_mps`
  Maximum value of MPS for COC. MPS is the maximum payload size (in octets) that the L2CAP layer entity can receive on a channel (range: 23 to (BLE_EVT_MAX_PARAM_LEN - 7) that is, 23 to 248 for BLE_EVT_MAX_PARAM_LEN default value).

  Note: `max_coc_mps` defines the maximum amount of data per L2CAP frame used for COC. It is recommended to set it to 248.

- `max_coc_nbr`
  Maximum number of COC channels (range: 0 to 64)

  Note: `max_coc_nbr` defines the maximum number of supported channels, while `max_coc_initiator_nbr` specifies, among these channels, the number dedicated to initiator mode, the other channels being in acceptor mode.

- `max_coc_initiator_nbr`
  Maximum number of COC channels in initiator mode (range: 0 to `max_coc_nbr`)

5.11.2 L2CAP COC channel creation

A COC channel can be created either in acceptor mode or initiator mode.

- **Acceptor mode:**
  Once a BLE connection is established, the BLE stack automatically waits for a request from the remote device to create a COC channel. When a request occurs, the BLE stack sends an event ACI_L2CAP_COC_CONNECT_EVENT to the application. The application must then respond to this event with the command ACI_L2CAP_COC_CONNECT_CONFIRM with a result equal to 0 to confirm the creation (or a different value to reject the creation, for example, if the SPSM is not recognized). If the creation is accepted, the channel is created, and the index of the created channel returned by ACI_L2CAP_COC_CONNECT_CONFIRM must be stored in the application memory.
- **Initiator mode:**
  Once a BLE connection is established, the application can trigger the creation of a COC channel by calling the command ACI_L2CAP_COC_CONNECT. The application must then wait for the event ACI_L2CAP_COC_CONNECT_CONFIRM_EVENT. If the result parameter in this event is equal to 0, it means that the channel is created. Its index, present in the event parameters, must be stored in the application memory.

Channel indexes (range: 0 to 63) returned by the event ACI_L2CAP_COC_CONNECT_CONFIRM_EVENT and the command ACI_L2CAP_COC_CONNECT_CONFIRM are unique among all the L2CAP connections and enable referencing the COC channels for subsequent commands.

5.11.3 L2CAP COC data transmission

Once a COC channel is created, the application can send data packets with size up to N bytes, where N is the MTU value specified by the remote device (between 23 and 65535). The application must handle segmentation and transmission flow control, as these tasks are not automatically handled by the STM32WB and STM32WBA BLE stack. To send data packets, the application must perform the following process:

- At channel creation, set a variable TX_CREDITS to the value Initial_Credits specified by the remote device.
- At each data packet transmission:
  1. Append the 2 bytes coding the packet length in front of the packet data (in little endian format) to build the full data to send.
  2. Wait for TX_CREDITS > 0.
  3. Build a K-frame by cutting the first K bytes of the data to send, where K is the local MPS value (23 to 248).
  4. Send the K-frame using the command ACI_L2CAP_COC_TX_DATA.
  5. Decrement TX_CREDITS by 1.
  6. Go to 2 if there is still data to send for the packet.
- At reception of the event ACI_L2CAP_COC_FLOW_CONTROL_EVENT, increment TX_CREDITS by the received credits parameter.

5.11.4 L2CAP COC data reception

Once a COC channel is created, the application can receive data packets with size up to N bytes, where N is the MTU value specified at channel creation (between 23 and 65535). The application must handle re-assembly and reception flow control, as these tasks are not automatically handled by the BLE stack.

To receive data packets, the application must perform the following process:

- At channel creation, specify a value of Initial_Credits in the command ACI_L2CAP_COC_CONNECT (or ACI_L2CAP_COC_CONNECT_CONFIRM) that corresponds to the number of K-frames that can be stored locally in a reception buffer.
- At reception of the event ACI_L2CAP_COC_RX_DATA_EVENT, append the event data (that is, the K-frame) to the reception buffer. If the total received data length corresponds to the data packet length (the first 2 bytes of the first K-frame in the reception buffer):
  1. Report the full data to the upper layer.
  2. Free the reception buffer.
  3. Call the command ACI_L2CAP_COC_FLOW_CONTROL with the credits parameter equal to the number of received K-frames.

5.11.5 L2CAP COC channel termination

A COC channel can be terminated in different ways:

- **Termination triggered by the application:**
  To terminate a channel, the application must call the command ACI_L2CAP_COC_DISCONNECT. It then waits for the event ACI_L2CAP_COC_DISCONNECT_EVENT that confirms the termination.

Note: The event ACI_L2CAP_PROC_TIMEOUT_EVENT can be received instead in case of an issue in the link with the remote device.

- **Termination triggered by the remote device:**
  In case the remote device selects to terminate the channel, the application receives the ACI_L2CAP_COC_DISCONNECT_EVENT event. The application then considers the channel as terminated. There is no need to confirm the termination. It is handled automatically by the BLE stack.
- **Automatic termination:**
  The BLE stack automatically (and silently) disconnects a COC channel in case the BLE connection that it belongs to is terminated.

6 BLE multiple connection timing strategy

This section provides an overview of the connection timing management strategy of the STM32WB stack when multiple central and multiple peripheral connections are active.

6.1 Basic concepts about Bluetooth® Low Energy timing

This section describes the basic concepts related to the Bluetooth® Low Energy timing management related to the advertising, scanning and connection operations.

6.1.1 Advertising timing

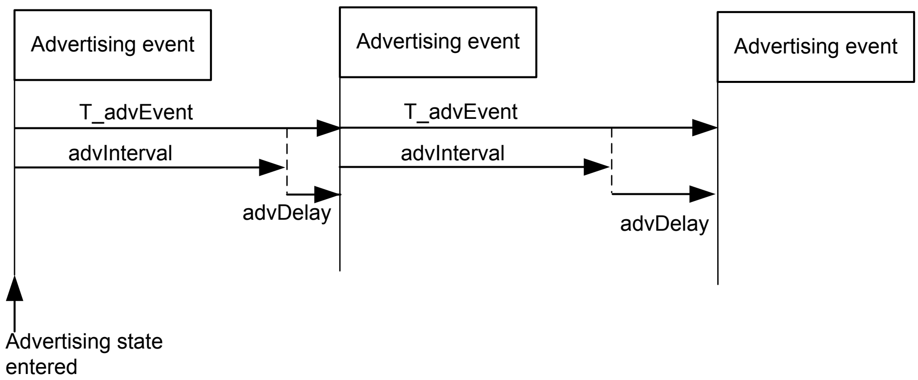
The timing of the advertising state is characterized by 3 timing parameters, linked by this formula:

T_advEvent = advInterval + advDelay

where:

- T_advEvent: time between the start of two consecutive advertising events; if the advertising event type is either a scannable undirected event type or a non-connectable undirected type, the advInterval shall not be less than 100 ms; if the advertising event type is a connectable undirected event type or connectable directed event type used in a low duty cycle mode, the advInterval can be 20 ms or greater.
- advDelay: pseudo-random value with a range of 0 ms to 10 ms generated by the link layer for each advertising event.

Figure 26. Advertising timings



DT66324V1

6.1.2 Scanning timing

The timing of the scanning state is characterized by 2 timing parameters:

- scanInterval: defined as the interval between the start of two consecutive scan windows
- scanWindow: time during which link layer listens to on an advertising channel index (channel index (37/38/39) is changed every new scanWindow time frame)

The scanWindow and scanInterval parameters are less than or equal to 10.24 s.

The scanWindow is less than or equal to the scanInterval.

6.1.3 Connection timing

The timing of connection events is determined by 2 parameters:

- connection event interval (*connInterval*): time interval between the start of two consecutive connection events, which never overlap; the point in time where a connection event starts is named an *anchor point*.

At the anchor point, a central starts transmitting a data channel PDU to the peripheral, which in turn listens to the packet sent by its central at the anchor point.

The central ensures that a connection event closes at least T_IFS=150 µs (inter frame spacing time, for instance time interval between consecutive packets on the same channel index) before the anchor point of next connection event.

The connInterval is a multiple of 1.25 ms in the range of 7.5 ms to 4.0 s.

- *peripheral latency* (*connperipheralLatency*): allows a peripheral to use a reduced number of connection events. This parameter defines the number of consecutive connection events that the peripheral device is not required to listen to the central.

When the host wants to create a connection, it provides the controller with the maximum and minimum values of the connection interval (*Conn_Interval_Min*, *Conn_Interval_Max*) and connection length (*Minimum_CE_Length*, *Maximum_CE_Length*) thus giving the controller some flexibility in choosing the current parameters in order to fulfill additional timing constraints for instance, in the case of multiple connections.

6.2 STM32WB BLE stack timing and slot allocation concepts

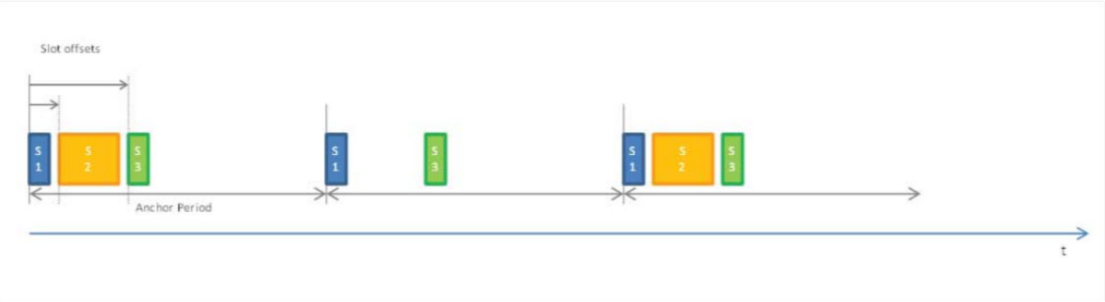
The STM32WB BLE stack adopts a time slotting mechanism in order to allocate simultaneous central and peripheral connections. The basic parameters, controlling the slotting mechanism, are indicated in the table below:

Table 60. **Timing parameters of the slotting algorithm**

Parameter	Description
Anchor period	Recurring time interval for slot allocation. Anchor period defines the overall period allocation in the slot during which the device listens to a transmission from another device, or is in an advertising or scanning slot. (All slots are mutually exclusive.)
Slot duration	Time interval inside which a full event (for instance, advertising or scanning, and connection) takes place; the slot duration is the time duration assigned to the connection slot and is linked to the maximum duration of a connection event.
Slot offset	Time value corresponding to the delay between the beginning of an anchor period and the beginning of the connection slot.
Slot latency	Number representing the actual utilization rate of a certain connection slot in successive anchor periods. (For instance, a slot latency equal to "1" means that a certain connection slot is actually used in each anchor period; a slot latency equal to n means that a certain connection slot is actually used only once every n anchor periods).

Timing allocation concept allows a clean time to handle multiple connections but at the same time imposes some constraints to the actual connection parameters that the controller can accept. An example of the time base parameters and connection slot allocation is shown in the figure below

Figure 27. **Example of allocation of three connection slots**



Slot #1 has offset 0 with respect to the anchor period, slot #2 has slot latency = 2, all slots are spaced by 1.5 ms guard time.

6.2.1 Setting the timing for the first central connection

The time base mechanism above described, is actually started when the first central connection is created. The parameters of such first connection determine the initial value for the anchor period and influence the timing settings that can be accepted for any further central connection simultaneous with the first one.

In particular:

- The initial anchor period is chosen equal to the mean value between the maximum and minimum connection period requested by the host
- The first connection slot is placed at the beginning of the anchor period
- The duration of the first connection slot is set equal to the maximum of the requested connection length

Clearly, the relative duration of such first connection slot compared to the anchor period limits the possibility to allocate further connection slots for further central connections.

6.2.2 STM32WB time setting for further central connections

Once that the time base has been configured and started as described above, then the slot allocation algorithm tries, within certain limits, to dynamically reconfigure the time base to allocate further host requests.

In particular, the following three cases are considered:

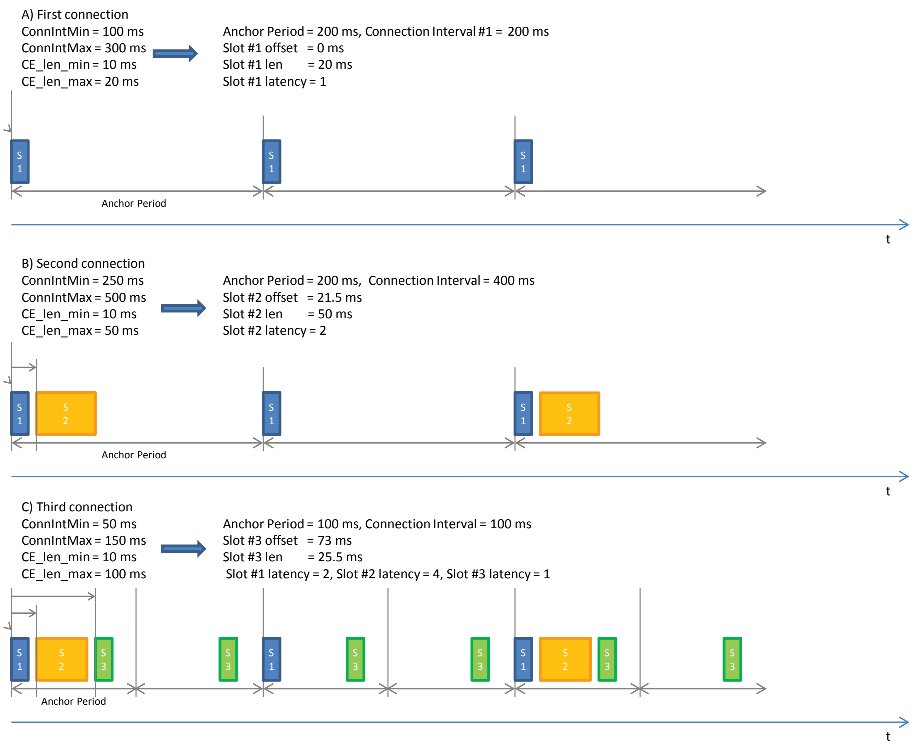
1. The current anchor period falls within the *Conn_Interval_Min* and *Conn_Interval_Max* range specified for the new connection. In this case no change is applied to the time base and the connection interval for the new connection is set equal to the current anchor period.
2. The current anchor period in smaller than the *Conn_Interval_Min* required for the new connection. In this case the algorithm searches for an integer number *m* such that: *Conn_Interval_Min* ≤ *Anchor_Period* × *m* ≤ *Conn_Interval_Max*
   If such value is found then the current anchor period is maintained and the connection interval for the new connection is set equal to *Anchor_Period·m* with slot latency equal to *m*.
3. The current anchor period in larger than the *Conn_Interval_Max* required for the new connection. In this case the algorithm searches for an integer number *k* such that:
   $$Conn\_Interval\_Min \leq \frac{Anchor\_Period}{k} \leq Conn\_Interval\_Max$$
   If such value is found then the current anchor period is reduced to:
   $$\frac{Anchor\_Period}{k}$$
   The connection interval for the new connection is set equal to:
   $$\frac{Anchor\_Period}{k}$$
   and the slot latency for the existing connections is multiplied by a factor *k*. Note that in this case the following conditions must also be satisfied:
   – *Anchor_Period/k* must be a multiple of 1.25 ms
   – *Anchor_Period/k* must be large enough to contain all the connection slots already allocated to the previous connections

Once that a suitable anchor period has been found according to the criteria listed above, then a time interval for the actual connection slot is allocated therein. In general, if enough space can be found in the anchor period, the algorithm allocates the maximum requested connection event length otherwise reduces it to the actual free space.

When several successive connections are created, the relative connection slots are normally placed in sequence with a small guard interval between (1.5 ms); when a connection is closed this generally results in an unused gap between two connection slots. If a new connection is created afterwards, then the algorithm first tries to fit the new connection slot inside one of the existing gaps; if no gap is wide enough, then the connection slot is placed after the last one.

Figure 28. Example of timing allocation for three successive connections shows an example of how the time base parameters are managed when successive connections are created.

Figure 28. Example of timing allocation for three successive connections



6.2.3 STM32WB dedicated timing for advertising events

The periodicity of the advertising events, controlled by *advInterval*, is computed based on the following parameters specified by the peripheral through the host in the `HCI_LE_Set_Advertising_parameters` command:

- *Advertising_Interval_Min, Advertising_Interval_Max;*
- *Advertising_Type*;

if *Advertising_Type* is set to high duty cycle-directed advertising, then advertising interval is set to 3.75 ms regardless of the values of *Advertising_Interval_Min* and *Advertising_Interval_Max*; in this case, a timeout is also set to 1.28 s, that is the maximum duration of the advertising event for this case.

In all other cases the advertising interval is chosen equal to the mean value between (*Advertising_Interval_Min* + 5 ms) and (Advertising_Interval_Max + 5 ms). The advertising has not a maximum duration as in the previous case, but it is stopped only if a connection is established, or upon explicit request by host.

The length of each advertising event is set by default by the software to be equal to 14.6 ms (for instance, the maximum allowed advertising event length) and it cannot be reduced.

Advertising slots are allocated within the same time base of the central slots (for instance, scanning and connection slots). For this reason, the advertising enable command to be accepted by the software when at least one central slot is active, the advertising interval has to be an integer multiple of the actual anchor period.

6.2.4 STM32WB dedicated timing for scanning

Scanning timing is requested by the central through the following parameters specified by the host in the HCI_LE_Set_Scan_parameters command:

- LE_Scan_Interval: used to compute the periodicity of the scan slots
- LE_Scan_Window: used to compute the length of the scan slots to be allocated into the central time base

Scanning slots are allocated within the same time base of the other active central slots (for instance, connection slots) and of the advertising slot (if there is one active).

If there is already an active slot, the scan interval is always adapted to the anchor period.

Every time the LE_Scan_Interval is greater than the actual anchor period, the software automatically tries to subsample the LE_Scan_Interval and to reduce the allocated scan slot length (up to ¼ of the LE_Scan_Window) to keep the same duty cycle required by the host, given that scanning parameters are just recommendations as stated by BT official specifications (v.4.1, vol.2, part E, §7.8.10).

6.2.5 STM32WB dedicated peripheral timing

The peripheral timing is defined by the central when the connection is created, this means in that case that the connection slots for peripheral links are managed asynchronously. The peripheral assumes that the central may use a connection event length as long as the connection interval. The scheduling algorithm dynamically adopts an estimation on peripheral slot length based on a continuous computation of the connect events duration, with priority given on less connection slot provided in case of collision.

The scheduler may also impose a dynamic limit to the peripheral connection slot duration to preserve both central and peripheral connections. As explained in the next section.

6.3 Multiple central and peripheral piconets topologies connection guidelines

STM32WB and STM32WBA devices can be used in different piconet topologies. For the STM32WB and STM32WBA BLE stacks, the multiple central/peripheral features offer the capability for one device (called central_peripheral in this context), to handle several connections at the same time, as detailed in the following sections, Section 6.3.1: STM32WB piconets topology guidelines and Section 6.3.2: STM32WBA piconets topology guidelines.

6.3.1 STM32WB piconets topology guidelines

STM32WB can handle several connection combinations simultaneously depending on the firmware variant and hardware used, as Table 61 illustrates.

Table 61. **STM32WB connection combinations**

STM32WB Stack Variant	STM32WB5x/3x	STM32WB1x
Full or full extended	Maximum of 8 central/peripheral combinations	Maximum of 5 central/peripheral combinations
Light	Maximum of 4 peripheral links	Maximum of 4 peripheral links

Possible combinations for STM32WB5x using the full or full extended variant:

1. Central of multiple peripherals:
   – Central_peripheral can connect up to eight peripheral devices.
2. Peripheral of multiple centrals:
   – Central_peripheral can be connected to up to eight central devices.
3. Simultaneously multicentrals and multiperipherals:
   a. Central_peripheral, acting as a central, can connect up to x peripheral devices (x ≤ 8) and the same central_peripheral device, acting as a peripheral, can be connected to up to 8 -x central devices.
   b. The device can scan, advertise, and connect as central while in multiple peripheral mode.
   c. The device can scan, advertise, and be connected as a peripheral while in multiple central mode.

The following guidelines must be followed to properly handle multiple central and peripheral connections using the full or full extended variant for STM32WB devices:

1. Avoid overallocating connection event length: choose *Minimum_CE_Length* and *Maximum_CE_Length* as small as possible to satisfy the application needs. This way, the allocation algorithm allocates several connections within the anchor period and reduces the anchor period, if needed, to allocate connections with a small connection interval.
2. For the first central connection:
   a. If possible, create the connection with the shortest connection interval as the first one so to allocate further connections with connection interval multiples of the initial anchor period.
   b. If possible, choose *Conn_Interval_Min* = *Conn_Interval_Max* as a multiple of 10 ms to allocate further connections with a connection interval submultiple by a factor of 2, 4, and 8 (or more) of the initial anchor period (having a multiple of 1.25 ms).
3. For additional central connections:
   a. Choose *ScanInterval* equal to the connection interval of one of the existing central connections.
   b. Choose *ScanWin* such that the sum of the allocated central slots (including advertising, if active) is lower than the shortest allocated connection interval.
   c. Choose *Conn_Interval_Min* and *Conn_Interval_Max* such that the interval contains either:
      - A multiple of the shortest allocated connection interval
      - A submultiple of the shortest allocated connection interval being also a multiple of 1.25 ms
   d. Choose *Maximum_CE_Length* = *Minimum_CE_Length* such that the sum of the allocated central slots (including advertising, if active) plus *Minimum_CE_Length* is lower than the shortest allocated connection interval.
4. Every time you start advertising for further peripheral connections:
   a. Choose *Advertising_Interval_Min* = *Advertising_Interval_Max* = integer multiple of the shortest allocated connection interval.
   b. Once connected with the central device, for additional peripheral connections with other centrals, it is recommended to allocate as a minimum central connection interval for:
      - Two links peripheral: 18.75 ms
      - Three links peripheral: 25 ms
      - Four links peripheral: 31.25 ms
      - Five links peripheral: 37 ms
      - Six links peripheral: 50 ms
      - Seven links peripheral: 55 ms
      - Eight links peripheral: 62 ms
5. Every time you start scanning:
   a. Choose *ScanInterval* equal to the connection interval of one of the existing central connections.
   b. Choose *ScanInterval* equal to the connection interval of one of the existing central connections.
   c. Choose *ScanWin* such that the sum of the allocated central slots (including advertising, if active) is lower than the shortest allocated connection interval.
6. The process of creating multiple connections, then closing some of them and creating new ones again, over time, can decrease the overall efficiency of the slot allocation algorithm. In case of difficulties in allocating new connections, the time base can be reset to the original state closing all existing connections.

6.3.2 STM32WBA piconets topology guidelines

STM32WBA can handle up to 8 or 20 connections depending on the Link Layer library used. In the following example, the library used 20 connections.

1. Central of multiple peripherals:
   – Central_peripheral can connect up to twenty peripheral devices.
2. Peripheral of multiple centrals:
   – Central_peripheral can be connected to up to twenty central devices.
3. Simultaneously multicentrals and multiperipherals:
   a. Central_peripheral, acting as a central, can connect up to x peripheral devices ($x \leq 20$) and the same central_peripheral device, acting as a peripheral, can be connected to up to 20 -x central devices.
   b. The device can scan, advertise, and connect as central while in multiple peripheral mode.
   c. The device can scan, advertise, and be connected as a peripheral while in multiple central mode.

The following guidelines must be followed to properly handle multiple central and peripheral connections using the STM32WBA device:

- Once connected with the central device, for additional peripheral connections with other centrals, it is recommended to allocate as a lower limit the connection interval:
  – Up to 5 links peripheral: 7.5 ms
  – Up to 10 links peripherals: 13.5 ms
  – Up to 20 links peripherals: 21.25 ms

7 Reference documents

Table 62. **Reference documents**

Name	Title/description
AN5289	Building wireless applications with STM32WB series microcontrollers
AN5379	Examples of AT commands on STM32WB series microcontrollers
AN5270	STM32WB Bluetooth® Low Energy (BLE) wireless interface
AN5155	STM32Cube MCU package examples for STM32WB series
Bluetooth® specifications	Specification of the Bluetooth® system (v4.0, v4.1, v4.2, v5.0, v5.1, 5.2, v5.3, v5.4, v6.0)
AN5378	STM32WB series microcontrollers bring-up procedure
AN5071	STM32WB series microcontrollers ultra-low-power features overview

8 List of acronyms and abbreviations

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

Table 63. List of acronyms

Term	Meaning
ACI	Application command interface
ATT	Attribute protocol
BLE	Bluetooth® Low Energy
BR	Basic rate
CRC	Cyclic redundancy check
CSRK	Connection signature resolving key
EDR	Enhanced data rate
DK	Development kits
EXTI	External interrupt
GAP	Generic access profile
GATT	Generic attribute profile
GFSK	Gaussian frequency shift keying
HCI	Host controller interface
IFR	Information register
IRK	Identity resolving key
ISM	Industrial, scientific and medical
LE	Low energy
L2CAP	Logical link control adaptation layer protocol
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
USB	Universal serial bus
UUID	Universally unique identifier
WPAN	Wireless personal area networks

Revision history

Table 64. Document revision history

Date	Revision	Changes
02-Jul-2020	1	Initial release
11-Dec-2020	2	Added: • Section 4.6.1: Flow charts on pairing procedure: pairing request by central sequence (Legacy) • Section 4.6.2: Flow charts on pairing procedure: pairing request by central sequence (secure) • Section 4.6.3: Flow charts on pairing procedure: pairing request by peripheral sequence (secure) • Section 4.11: End to end RX flow control using GATT • STM32WB formula for converting RSSI raw value in dBm Updated: • Section 2.8.1: Device filtering
11-Feb-2021	3	Updated: • Section Introduction
01-Dec-2021	4	Updated: • Section 2.10: Generic access profile (GAP) • Section 4.1: BLE addresses • Section 4.3: Services and characteristics configuration • Section 5.8: STM32WB LE additional beacon • Section 6: BLE multiple connection timing strategy • Section 6.1.2: Scanning timing • Section 6.2.2: STM32WB time setting for further central connections • Section 6.2.5: STM32WB dedicated peripheral timing • Section 6.3: Multiple central and peripheral piconets topologies connection guidelines • Piconet topologies connection formula Deleted: • Figure 10 BLE MAC address storage • Section 5.4 Piconet topologies connection formula
06-Jul-2022	5	Updated: • Introduction • Section 4: Design an application using the STM32WB and STM32WBA BLE stacks Added: • Section 5.9: STM32WB and STM32WBA LE extended advertising • Section 5.9.1: Extended advertising set • Section 5.9.2: Extended scannable set • Section 5.9.3: Extended connectable set • Section 5.9.4: Extended multiple sets • Section 5.9.5: LE extended scanning • STM32WB and STM32WBA LE connected oriented channel
10-Feb-2023	6	Updated: • Section 4: Design an application using the STM32WB and STM32WBA BLE stacks Added: • Section 5: STM32WB and STM32WBA BLE stack advanced features description, Section 5.2.1: STM32WB background scan aspect
03-Apr-2023	7	Updated: • STM32WBA included (where applicable) • Section 2.1: BLE stack architecture • Section 5.8: STM32WB LE additional beacon • Section 3.3: BLE stack library framework • Section 5.3: Bluetooth® Low Energy privacy • Section 5.3.1: Controller-based privacy and the device filtering scenario • Section 7: Reference documents

Date	Revision	Changes
		Added: • Section 3.1: STM32WB BLE stack architecture and interface • Section 3.2: STM32WBA BLE stack architecture and interface • Section 5.7: STM32WBA LE Coded PHY Deleted: • Section 4.1 Initialization phase and main application loop • Section 5.9 STM32WB formula for converting RSSI raw value in dBm
07-Jul-2023	8	Updated: • The terms slave changed to peripheral and master to central. • Section 2.1: BLE stack architecture • Section 2.2: Physical layer • Section 4.10: Characteristic notification/indications, write, read Added: • Section 4.7: Pairing failing and automatic pairing rejection guard time • Section 5.1.3: LE GATT Security Levels Characteristic (SLC) • Section 5.1.4: GATT operation timers • Section 5.10: Encrypted advertising data
15-Jan-2024	9	Updated: • Section 4.1: BLE addresses • Section 5.3.2: Resolving addresses • Section 5.5: LE data packet length extension APIs and events • Section 5.9.5: LE extended scanning, Figure 24
09-Apr-2024	10	Updated: • Section 2.2: Physical layer • Section 5.9.5: LE extended scanning • STM32WB and STM32WBA LE connected oriented channel Added: • Section 6.3: Multiple central and peripheral piconets topologies connection guidelines • Section 6.3.1: STM32WB piconets topology guidelines • Section 6.3.2: STM32WBA piconets topology guidelines
25-Jun-2024	11	Updated: • Section 5.1.3: LE GATT Security Levels Characteristic (SLC) • Section 5.9.6.1: Full mode commands (legacy and extended) • Section 5.9.6.2: HCI mode only (legacy and extended) • Section 5.10.1: Encrypted data key material sharing (advertiser side) Added: • Section 5.10: Encrypted advertising data • Section 5.11: L2CAP connection oriented channels Deleted Section 5.9.8 STM32WB and STM32WBA LE connected oriented channel
10-Dec-2024	12	Updated: • Introduction • Section 2.1: BLE stack architecture • Section 2.8: Privacy • Section 2.8.1: Device filtering • Section 4: Design an application using the STM32WB and STM32WBA BLE stacks • Section 4.2: Set tx power level • Section 4.4.2: Set discoverable mode and use general discovery procedure (active scan) • Section 4.10.1: Getting access to BLE device long characteristics. • Section 5.3: Bluetooth® Low Energy privacy • Section 5.8: STM32WB LE additional beacon • Section 5.9.4: Extended multiple sets • Table 60. Timing parameters of the slotting algorithm • Section 6.3.1: STM32WB piconets topology guidelines • Section 6.3.2: STM32WBA piconets topology guidelines • Section 7: Reference documents

Date	Revision	Changes
		Added: • Section 5.1.1: Reduced GATT information in NVM • Section 5.9.7: ACI command guidelines for user applications • Section 5.9.8: HCI command guidelines for user applications
16-Dec-2024	13	Updated: • Section 5.9.6.1: Full mode commands (legacy and extended) • Section 5.9.6.2: HCI mode only (legacy and extended) • Table 59. List of HCI commands user applications must not use with the host stack

Contents

List of tables

List of figures