PM0257
Programming manual

BlueNRG-1, BlueNRG-2 BLE stack programming guidelines

Introduction

The main purpose of this document is to provide a developer with some reference programming guidelines about how to develop a Bluetooth low energy (BLE) application using BlueNRG-1, BlueNRG-2 BLE stacks APIs and related event callbacks.

The document describes the BlueNRG-1, BlueNRG-2 Bluetooth low energy stack library framework, API interfaces and event callbacks allowing the access to the Bluetooth low energy functions provided by the BlueNRG-1, BlueNRG-2 system-on-chip.

This programming manual also provides some fundamental concepts about the Bluetooth low energy (BLE) technology in order to associate the BlueNRG-1, BlueNRG-2 APIs, parameters, and related event callbacks with the BLE protocol stack features. The user must have a basic knowledge about the BLE technology and its main features.

For more information about the BlueNRG-1, BlueNRG-2 devices and the Bluetooth low energy specifications, refer to *Section 5: References* at the end of this document.

The BlueNRG-1 and BlueNRG-2 are very low power Bluetooth low energy (BLE) single-mode system-on-chips, compliant with Bluetooth low energy specifications and supporting master or slave role; BlueNRG-2 also supports the extended packet length feature.

The manual is structured as follows:

- Fundamentals of Bluetooth low energy (BLE) technology
- BlueNRG-1, BlueNRG-2 BLE stack library APIs and the event callback overview
- How to design an application using the BlueNRG-1, BlueNRG-2 stack library APIs and event callbacks

Note: The document content is valid for both BlueNRG-1 and BlueNRG-2 devices. Any reference to BlueNRG-1 device is also valid for the BlueNRG-2 device. Any specific difference is highlighted whenever it is needed.

Contents

List of tables

List of figures

1 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 has been developed as a wireless standard allowing cables to be replaced by 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 only consume a fraction of the power of standard Bluetooth products and enable the devices with coin cell batteries to be wirelessly connected to standard Bluetooth enabled devices.

Figure 1. Bluetooth low energy technology enabled coin cell battery devices

GAMSEC201411251047

The Bluetooth low energy technology is used on a broad range of sensor applications transmitting small amounts of data.

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

1.1 BLE stack architecture

Bluetooth low energy technology has been formally adopted by the Bluetooth core specifications (version 4.0, *Table 56: List of references*). This version of the Bluetooth standard supports two systems of wireless technology:

- Basic rate
- Bluetooth low energy

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

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

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

The Bluetooth low energy technology is designed to address two alternative implementations:

- Smart devices
- Smart ready devices

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

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

The Bluetooth low energy stack consists of two components:

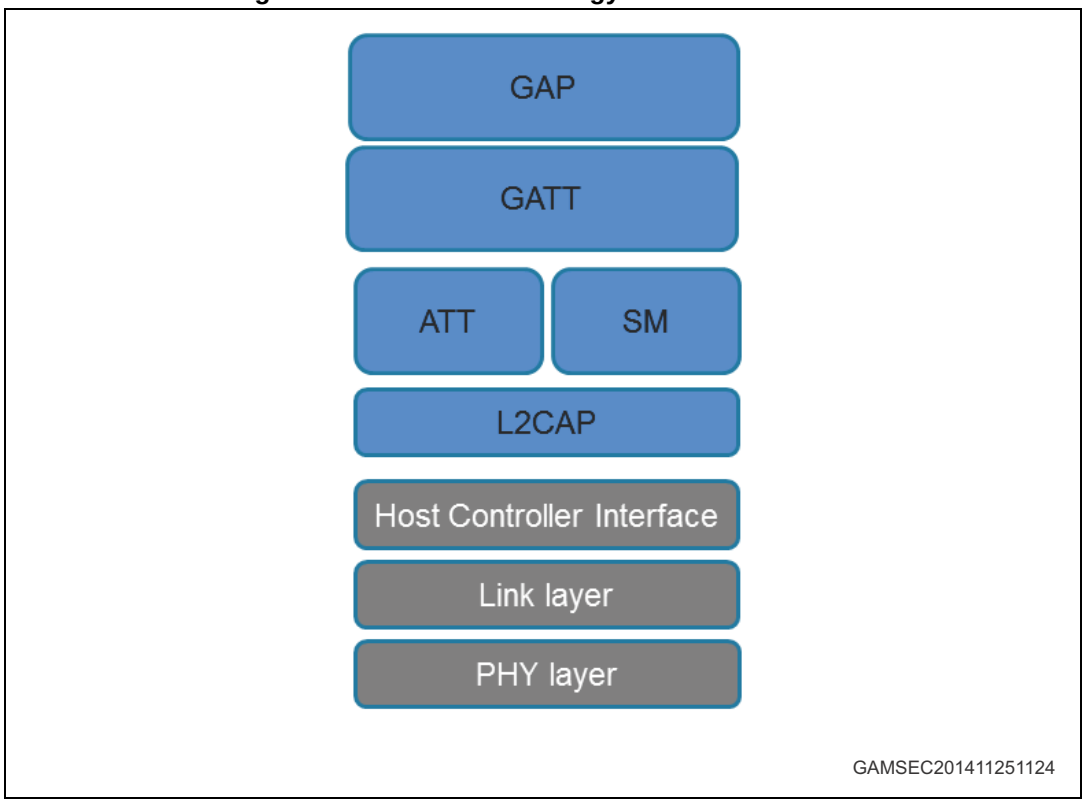
- Controller
- Host

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).

In addition, Bluetooth specifications v4.1, v4.2 and 5.0 have been released with new supported features.
For more information about these new features refer to the related specification document.

Figure 2. Bluetooth low energy stack architecture



1.2 Physical layer

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

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

Equation 1:

$$2042 + k \cdot 2MHz$$

where k = 0.39

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
37	2402 MHz	Advertising channel
0	2404 MHz	Data channel
1	2406 MHz	Data channel
….	….	Data channel
10	2424 MHz	Data channel
38	2426 MHz	Advertising channel
11	2428 MHz	Data channel
12	2430 MHz	Data channel
….	….	Data channel
36	2478 MHz	Data channel
39	2480 MHz	Advertising channel

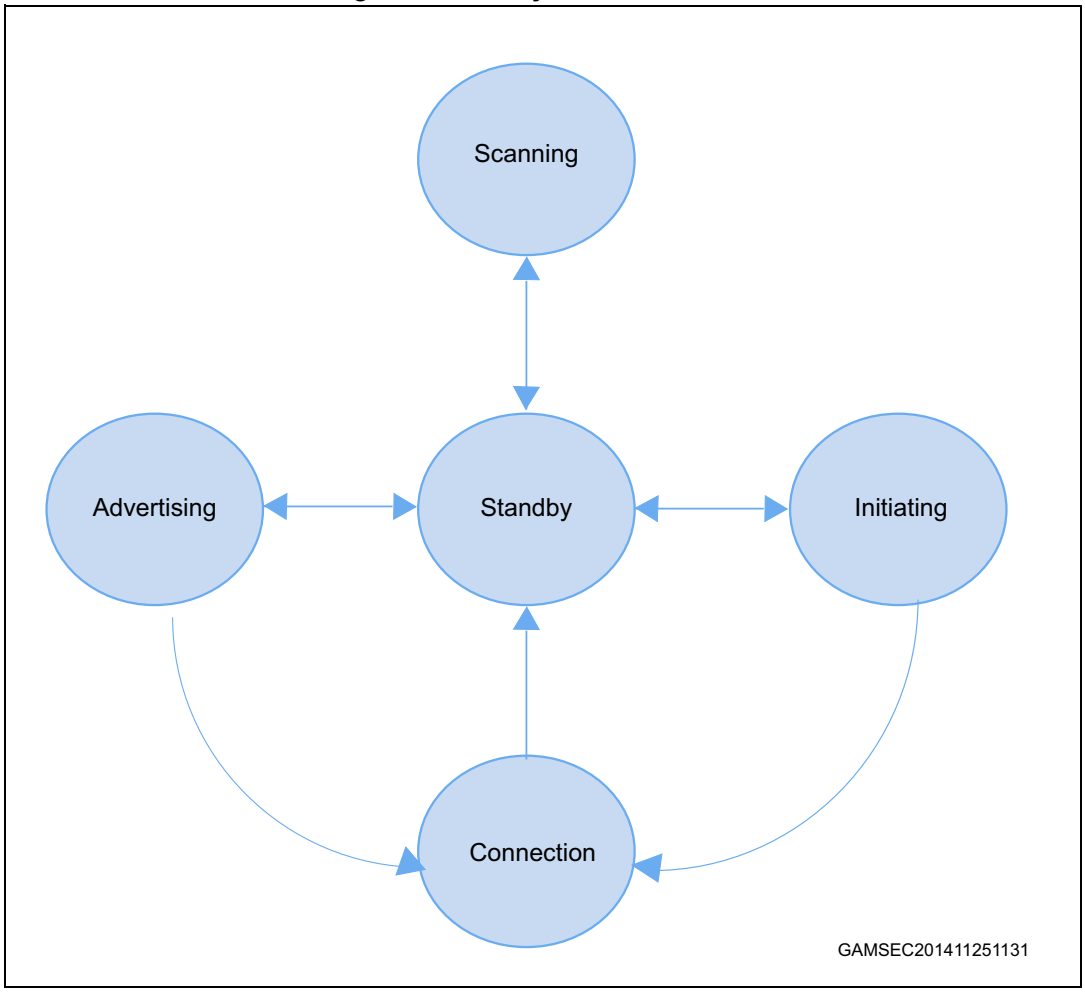
BLE is an adaptive frequency hopping (AFH) technology that can only use a subset of all the available frequencies in order to avoid all frequencies used by other no-adaptive technologies. This allows moving from a bad channel to a known good channel by using a specific frequency hopping algorithm, which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

1.3 Link layer (LL)

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

The link layer defines a state machine with five states:

Figure 3. Link layer state machine



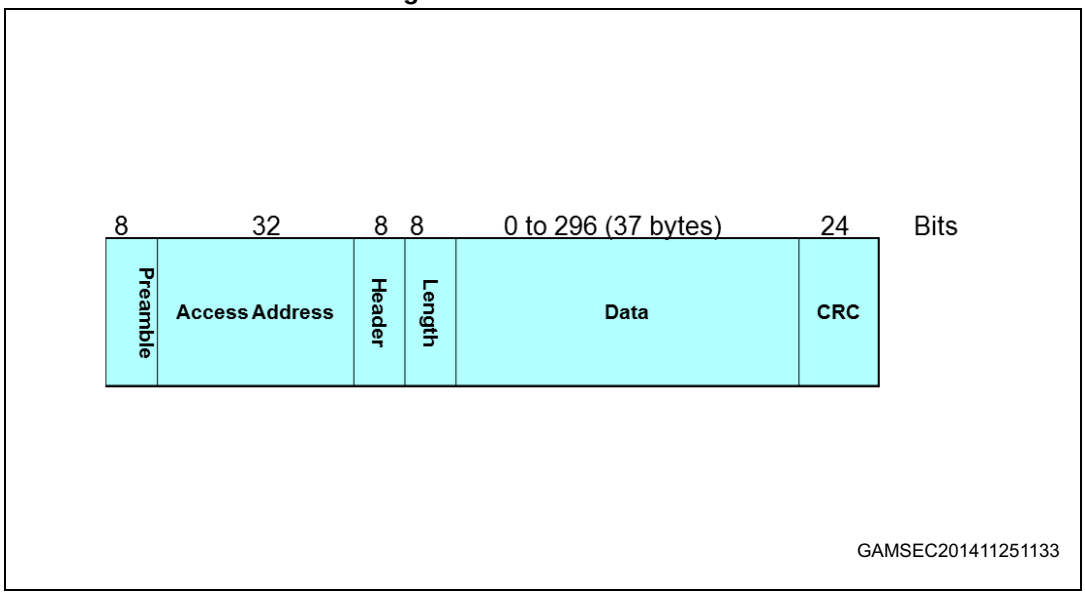
- Standby: the device does not transmit or receive packets
- Advertising: the device broadcast advertisements in advertising channels (it is called an advertiser device)
- Scanning: the device looks for the advertiser devices (it is called a scanner device).
- Initiating: the device initiates connection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

1.3.1 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



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

  a) Advertiser 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)

b) Advertising packet type:

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. The device can be scanned by a scanning device, or go into a connection as a slave 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. The device simply sends advertising packets on related channels, but it does not want to be connectable or scannable 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 to perform 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 specifications [Vol 2], in *Section 5: References*.

- Length: number of bytes on data field

Table 6. Packet length field and valid values

	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

- 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.

1.3.2 Advertising state

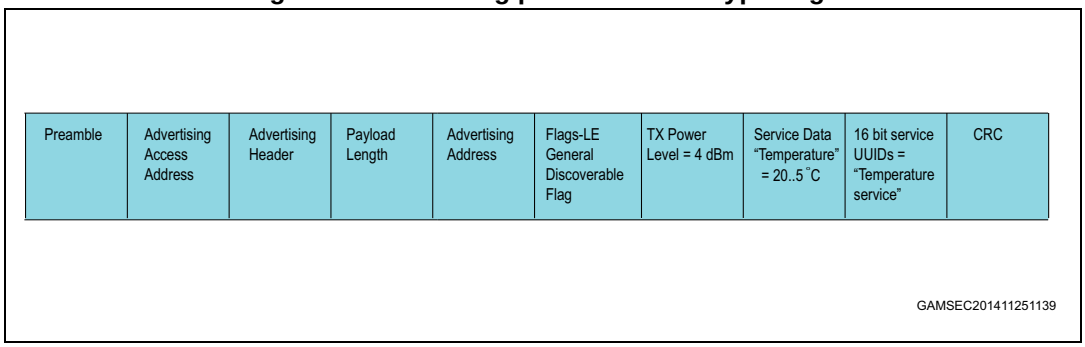
Advertising states allow link layer to transmit advertising packets and also to respond with scan responses to scan requests coming from those devices which are actively scanning.

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 packet 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 = 4 dBm, service data = temperature service and 16 bit service UUIDs).

Figure 5. Advertising packet with AD type flags



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

1.3.3 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, the device can send back a scan request packet, in order to get a scan response from the advertiser. This allows the scanner device to get additional information from the advertiser device.

The following scan parameters can be set:

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

Once 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

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

For a detailed description of the connection request packet refer to Bluetooth specifications [Vol 6].
The allowed timing ranges are summarized in *Table 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 slave device enters in receiver mode and waits for a packet from the master device. If no packet is received within this time, the slave leaves receiver mode, and it tries one connection interval again later. When a connection is established, a master has to transmit a packet to the slave on every connection event to allow slave to send packets to the master. Optionally, a slave device can skip a given number of connection events (slave latency).

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

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

1.4 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.

1.5 Logical link control and adaptation layer protocol (L2CAP)

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

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

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

Attributes exposed by a server can be discovered, read, and written by the client, and they can be indicated and notified by the server as described in *Table 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

1.7 Security manager (SM)

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

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

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

The 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:

a) The out-of-band (OOB) method, which uses out of band communication (e.g. 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).

b) Passkey Entry method: user passes six numeric digits as the TK between the devices. It provides authentication (MITM protection).

c) 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:

a) Long-term key (LTK): a 128-bit key used by the link layer for encryption and authentication

b) Connection signature resolving key (CSRK): a 128-bit key used for the data signing and verification performed at the ATT layer

c) 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:

a) 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).

b) Just Works: this method is not authenticated, and it does not provide any protection against man-in-the-middle (MITM) attacks.

c) Passkey entry method: this method is authenticated. User passes six numeric digits. This six-digit value is the base of the device authentication.

d) 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 master and slave:

- Connection signature resolving key (CSRK) for authentication of unencrypted data
- Identity resolving key (IRK) for device identity and privacy

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

Data signing

It is also possible to transmit authenticated data over an unencrypted link layer connection by using the CSRK key: a 12-byte signature is placed after the data payload at the ATT layer. The signature algorithm also uses a counter which protects against replay attacks (an external device 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).

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

Peripheral

A privacy-enabled peripheral in 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. The device does not send the device name to the advertising data.

Central

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

Broadcaster

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

Observer

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

1.8.1 The device filtering

Bluetooth LE 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 white list. When the white list is enabled, those devices, which are not in this list ,are ignored by the link layer.

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

1.9 Generic attribute profile (GATT)

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

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

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

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

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

As consequence, a slave (peripheral) device has not to be the GATT server and a master (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)

1.9.1 Characteristic attribute type

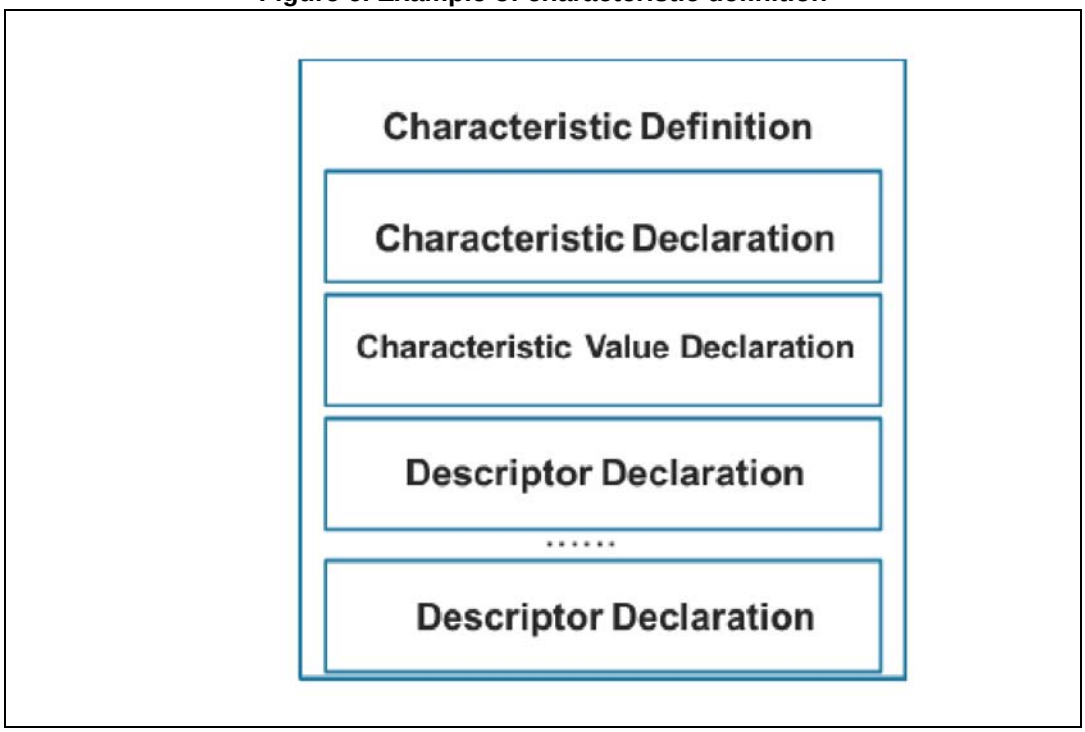
A characteristic is an attribute type which contains a single value and any number of descriptors describing the characteristic value that may make it understandable by the user.

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 6. Example of characteristic definition



A characteristic declaration is an attribute defined as follows:

Table 13. Characteristic declaration

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

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

Table 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

1.9.2 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 ^{o}C format)
6. Characteristic aggregation format: It allows several characteristic presentation formats to be aggregated

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

1.9.3 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 all 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.

1.9.4 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 characteristic 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 response events refer to the Bluetooth specifications in *Section 5: References*.

1.10 Generic access profile (GAP)

The Bluetooth system defines a base profile implemented by all Bluetooth devices called generic access profile (GAP). This generic profile defines the basic requirements of a Bluetooth device.

The four GAP profile roles are described in *Table 20: GAP roles*:

Table 20. GAP roles

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

1. M = Mandatory; O = Optional

On GAP context, two fundamental concepts are defined:

- GAP mode: 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 procedure: 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 flag AD type).	Broadcasts data that 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 flag 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 flag AD type	It is allowed for about 30 s. It is used by those 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 flag 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 device addresses	Peripheral
Undirected connectable mode	It uses the ADV_IND advertising packet	It is used from a device that wants to be connectable. Since ADV_IND advertising packet can include the flag AD type, a device can be in discoverable and undirected connectable mode at the same time. Connectable mode is terminated when the device moves to connection mode or when it moves to non-connectable mode.	Peripheral

Table 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 key is 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 procedures*:

Table 25. GAP observer procedures

Procedure	Description	Notes	Role
Observation procedure	It allows a device to look for broadcaster device 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 and limited discovery mode	Device filtering is applied based on flag AD type information	Central
Name discovery procedure	It is the procedure to retrieve the “Bluetooth device name” from connectable devices		Central

Table 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 following 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.

1.11 BLE profiles and applications

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

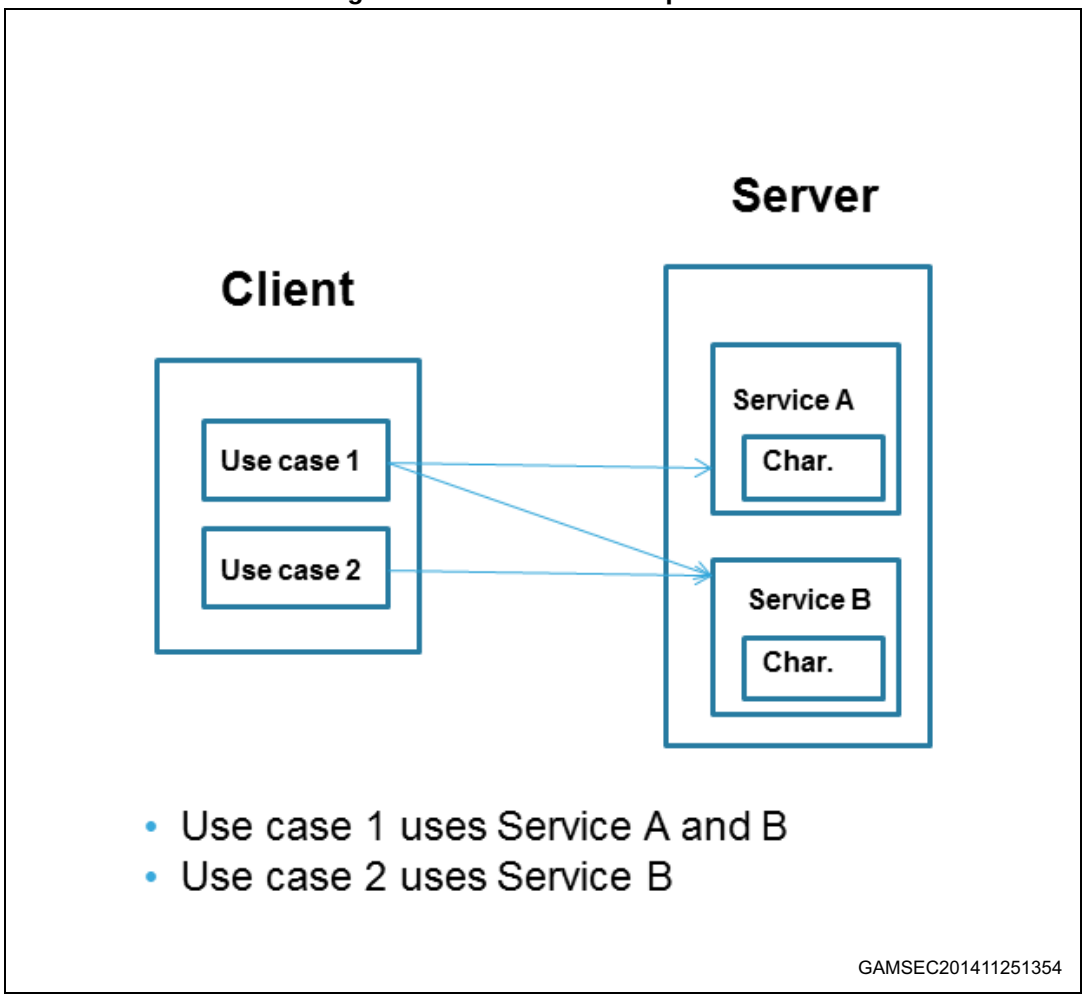
- Profile clients implement use cases
- Profile servers implement services

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, characteristic specifications and UUID assignations 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 7. Client and server profiles



1.11.1 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

2 BlueNRG-1, BlueNRG-2 Bluetooth low energy stack

The BlueNRG-1, BlueNRG-2 devices are system-on-chip with a Bluetooth low energy (BLE) radio. A Bluetooth low energy (BLE) stack standard C library, in binary format, provides a high level interface to control BlueNRG-1, BlueNRG-2 Bluetooth low energy functionalities.

The BLE binary library provides the following functionalities:

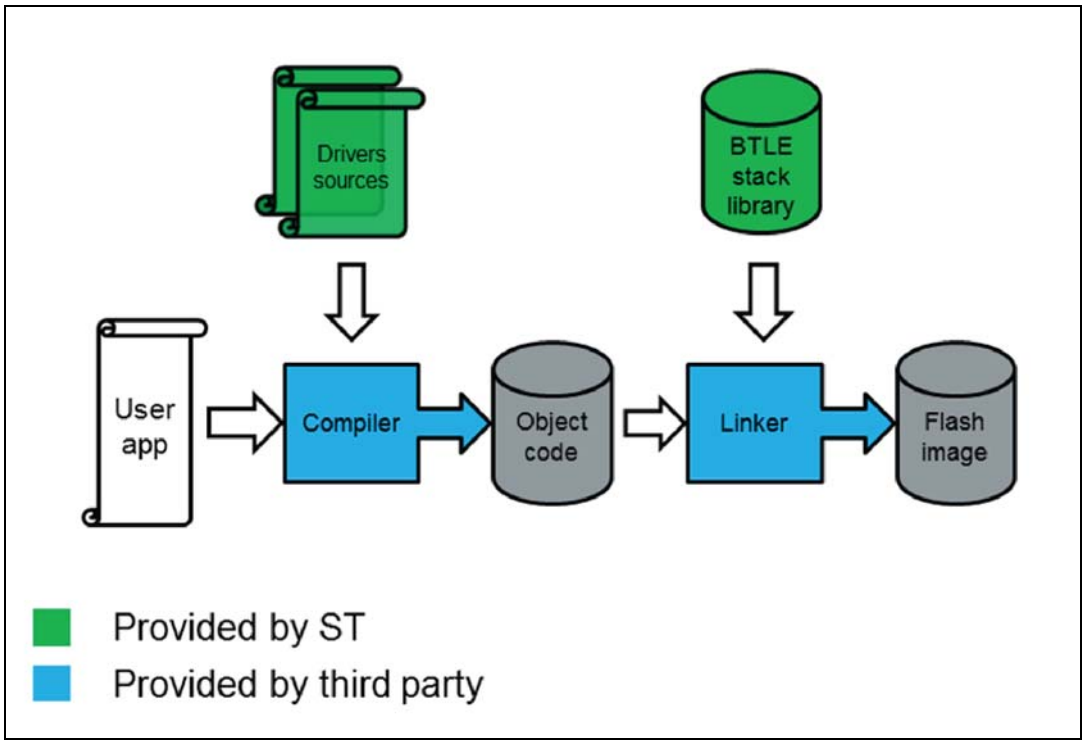
- Stack APIs for:
  - BLE stack initialization
  - BLE stack application command interface (HCI command prefixed with hci_, and vendor specific command prefixed with aci_)
  - Sleep timer access
  - BLE stack state machine handling
- Stack event callbacks
  - Inform user application about BLE stack events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

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

- Call the related stack APIs
- Handle the expected events through the provided stack callbacks

Linking the BLE stack binary library to the user application, as described *Figure 8: BLE stack reference application*.

Figure 8. BLE stack reference application



Note: 1. API is a C function defined by the BLE stack library and called by user application.

2. A callback is a C function called by the BLE stack library and defined by the user application.

3. Driver sources are a set of drivers (header and source files) which handles all the BlueNRG-1, BlueNRG-2 peripherals (ADC, I^2C, SPI, timers, watchdog, UART).

2.1 BLE stack library framework

The BLE stack library framework allows commands to be sent to the BlueNRG-1, BlueNRG-2 SoC BLE stack and it also provides definitions of BLE event callbacks.

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

The provided set of APIs supports the following commands:

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

The reference BLE API interface framework is provided within the BlueNRG-1_2 DK software package targeting the BlueNRG-1, BlueNRG-2 DK platforms (refer to *Section 5: References*).

The BLE stack library framework interface for both BlueNRG-1, BlueNRG-2 devices is defined by the following header files.

Table 29. BLE stack library framework interface

File(1)	Description	Notes
ble_const.h	It includes the required BLE stack header files	To be included on the user main application
ble_status.h	Header file for BLE stack error codes	It is included through ble_const.h header file
bluenrg1_api.h	Header file for BLE stack APIs	It is included through bluenrg1_stack.h header file
bluenrg1_events.h	Header file for BLE stack event callbacks	It is included through bluenrg1_stack.h header file
bluenrg1_gap.h	Header file for BLE GAP layer constants	It is included through ble_const.h header file
bluenrg1_gatt_server.h	Header file for GATT server constants	It is included through ble_const.h header file
bluenrg1_hal.h	Header file with HAL for BlueNRG-1, BlueNRG-2 devices	It is included through ble_const.h header file
bluenrg1_stack.h	Header file for BLE stack initialization, tick and sleep timer APIs	To be included on the user main application
hci_const.h	It contains constants for HCI layer	It is included through ble_const.h header file

Table 29. BLE stack library framework interface (continued)

File[1]	Description	Notes
link_layer.h	Header file for BLE link layer constants	It is included through ble_const.h header file
sm.h	Header file for BLE security manager constants	It is included through ble_const.h header file

1. File location: library\Bluetooth_LE\inc.

2.2 BLE stack event callbacks

The BLE stack library framework provides a set of events and related callbacks which are used to notify the user application of specific events to be processed.

The BLE event callback prototypes are defined on header file bluenrg1_events.h. All callbacks are defined by default through weak definitions (no check is done on event callback name defined from the user, so user should carefully check that each defined callbacks is in line with the expected function name).

The user application must define the used event callbacks with application code, in line with specific application scenario.

2.3 BLE stack init and tick APIs

The BLE stack must be initialized in order to proper configure some parameters in line with specific application scenario.

The following API must be called before using any other BLE stack functionality:
`BlueNRG_Stack_Initialization(&BlueNRG_Stack_Init_params);`
`BlueNRG_Stack_Init_params` is a variable which contains memory and low level hardware configuration data for the device, and it is defined using this structure:

```
typedef struct {
        uint8_t* bleStartFlashAddress;
        uint32_t secDbSize ;
        uint32_t serverDbSize ;
        uint8_t* stored_device_id_data_p;
        uint8_t* bleStartRamAddress;
        uint32_t total_buffer_size;
        uint16_t numAttrRecord;
        uint16_t numAttrServ ;
        uint16_t attrValueArrSize;
        uint8_t numOfLinks;
        uint8_t extended_packet_length_enable;
        uint8_t prWriteListSize;
        uint8_t mblockCount;
        uint16_t attMtu;
```

```
        hardware_config_table_t hardware_config;
} BlueNRG_Stack_Initialization_t;
```

The hardware_config_table_t structure is defined as follows:

```
typedef struct {
uint32_t *hot_ana_config_table;
uint32_t max_conn_event_length;
uint16_t slave_sca;
uint8_t master_sca;
uint8_t ls_source;
uint16_t hs_startup_time ;
} hardware_config_table_t;
```

Table 30. BLE Stack Initialization parameters

Name	Description	Value
bleStartFlashAddress	SDB base address: it is the start address for the non-volatile memory area allocated to the stack to store information for bonded devices	Aligned to 2048 bytes flash sector boundary(1)
secDbSize	Security DB size: it is the size of the security database used to store security information for bonded devices	1 kB(1)
serverDbSize	Server DB size: it is the size of the server database used to store service change notification for bonded devices	1 kB
stored_device_id_data_p	Storage area for stack internal parameters (security root keys, static random address, public address)	56 bytes, 32-bit aligned flash area, all elements must be initialized to 0xFF
bleStartRamAddress	Start address of the RAM buffer for stack GATT database	32-bit aligned RAM area
total_buffer_size	Total buffer size allocated for stack	TOTAL_BUFFER_SIZE(NUM_LINKS,NUM_GATT_ATTRIBUTES,NUM_GATT_SERVICES,ATT_VALUE_ARRAY_SIZE,PCKT_COUNT)

Table 30. BLE Stack Initialization parameters (continued)

Name	Description	Value
numAttrRecord	Maximum number of attributes records related to all the required characteristics (excluding the services) that can be stored in the GATT database, for the specific user BLE application	For each characteristic, the number of attributes goes from 2 to 5 depending on the characteristic properties: - minimum of 2 (one for declaration and one for the value) - add one more record for each additional property: notify or indicate, broadcast, extended property. Total calculated value must be increased of 9, due to the records related to the standard attribute profile and GAP services characteristics, automatically added when initializing GATT and GAP layers.
numAttrServ	Maximum number of services that can be stored in the GATT database, for the specific user BLE application	Total calculated value must be increased of 2 due to the standard attribute profile and GAP services, automatically added when initializing GATT and GAP layers.
attrValueArrSize	Size of the storage area for attribute values	Each characteristic contributes to the attrValueArrSize value as follows: - characteristic value length - characteristic UUID is 16 bits: adding 5 bytes - characteristic UUID is 128 bits: adding 19 bytes - characteristic has server configuration descriptor: adding 2 bytes - characteristic has client configuration descriptor: adding 2 bytes for each simultaneous connection - characteristic has extended properties: adding 2 bytes
numOfLinks	Maximum number of simultaneous connections that the device can support.	Valid values are from 1 to 8
extended_packet_length_enable	Unsupported feature (reserved for future use)	0
prWriteListSize(2)	Number of prepare write requests needed for a long write procedure for a characteristic with len > 20 bytes	The minimum required value is calculated using a specific macro provided on bluenrg1_stack.h file: PREP_WRITE_X_ATT().
mblockCount(2)	Number of allocated memory blocks for the BLE stack	The minimum required value is calculated using a specific macro provided on bluenrg1_stack.h file: PCKT_MBLOCKS_C().

Table 30. BLE Stack Initialization parameters (continued)

Name	Description	Value
attMtu(2)	Maximum supported ATT_MTU size	Supported values ranges is 23-158 bytes
hot_ana_config_table	Low level configuration parameters table for the radio subsystem.	To be set to NULL
max_conn_event_length	Maximum duration of the connection event when the device is in Slave mode in units of 625/256 µs (~2.44 µs)	<= 4000 (ms)
slave_sca	Sleep clock accuracy in Slave mode	ppm value
master_sca	Sleep clock accuracy in Master mode	0 to 7 corresponding to 500, 250, 150, 100, 75, 50, 30, 20 ppm
ls_source(3)	Source for the 32 kHz slow speed clock	1: internal RO 0: external crystal
hs_startup_time(3)	Start up time of the high speed (16 or 32 MHz) crystal oscillator in units of 625/256 µs (~2.44 us)	Positive integer(4)

1. These values cannot be changed. To be potentially optimized for making the BLE stack configuration more flexible.
2. New Radio initialization parameter supported on BLE stack v2.x.
3. High Speed and Low Speed crystal sources can be defined through these define values:
HS_SPEED_XTAL=HS_SPEED_XTAL_16MHZ (or HS_SPEED_XTAL_32MHZ)
LS_SOURCE=LS_SOURCE_EXTERNAL_32KHZ (or LS_SOURCE_INTERNAL_RO).
4. For information about how to define the proper hs_startup_time value refer to the Bringing up the BlueNRG-1, BlueNRG-2 devices application note (AN4818) on *Section 5: References* at the end of this document.

2.4 BlueNRG-1, BlueNRG-2 cold start configuration

During the device initialization phase, after BlueNRG-1, BlueNRG-2 device powers on, some specific parameters must be defined on BLE device controller registers, in order to define the following configurations:

- Application mode: user or test mode
- High speed crystal configuration: 32 or 16 MHz
- Low speed crystal source: external 32 kHz oscillator or internal RO
- SMPS: on or off (if on: 4.7 µH or 10 µH SMPS inductor)

The BlueNRG-1, BlueNRG-2 controller registers values are defined on file system_bluenrg1.c through the cold start configuration table:

```
/* Cold Start Configuration Table */
#define COLD_START_CONFIGURATION
{
  NUMBER_CONFIG_BYTE, ATB0_ANA_ENG_REG,    0x00,
  NUMBER_CONFIG_BYTE, ATB1_ANA_ENG_REG,    0x30,
  NUMBER_CONFIG_BYTE, RM1_DIG_ENG_REG,     SMPS_10uH_RM1,
  NUMBER_CONFIG_BYTE, CLOCK_LOW_ENG_REG,   SMPS_ON,
  NUMBER_CONFIG_BYTE, CLOCK_HIGH_ENG_REG, HIGH_FREQ_16M,
  NUMBER_CONFIG_BYTE, PMU_ANA_ENG_REG,     SMPS_10uH_PMU,
  NUMBER_CONFIG_BYTE, CLOCK_ANA_USER_REG, LOW_FREQ_XO,
```

```
  END_CONFIG
}
```

This table defines the default configuration as follows:

- User mode: ATB0_ANA_ENG_REG = 0x00, USER_MODE_ATB1 = 0x30
- SMPS ON, 10 µH inductor: CLOCK_LOW_ENG_REG = SMPS_ON, RM1_DIG_ENG_REG = SMPS_10uH_RM1
- 16 MHz high speed crystal: CLOCK_HIGH_ENG_REG = HIGH_FREQ_16M
- External 32 kHz oscillator: CLOCK_ANA_USER_REG = LOW_FREQ_XO
- BOR (brown-out threshold): disabled by default

When the device powers on, the function `SystemInit()` (system_bluenrg1.c file) sets the default cold start parameters defined on the COLD_START_CONFIGURATION table within the `cold_start_config[]` array. User application can define its specific cold start settings, based on its application scenario, by setting some preprocessor options which act on specific fields of the `cold_start_config[]` array, as described in the following table:

Table 31. Cold start configuration preprocessor options

Preprocessor option	Preprocessor option values	cold_start_config field	Description
HS_SPEED_XTAL	HS_SPEED_XTAL_32MHZ	cold_start_config[14]= HIGH_FREQ_32M;	High speed crystal: 32 MHz
HS_SPEED_XTAL	HS_SPEED_XTAL_16MHZ	cold_start_config[14]= HIGH_FREQ_16M;	High speed crystal configuration: 16 MHz (default configuration)
LS_SOURCE	LS_SOURCE_EXTERNAL_32kHZ	cold_start_config[20]= LOW_FREQ_XO;	Low speed crystal source: external 32 kHz oscillator (default configuration)
LS_SOURCE	LS_SOURCE_INTERNAL_RO	cold_start_config[20]= LOW_FREQ_RO;	Low speed crystal source: internal RO
SMPS_INDUCTOR	SMPS_INDUCTOR_10uH	cold_start_config[11] = SMPS_ON; cold_start_config[8]= SMPS_10uH_RM1; cold_start_config[17]= SMPS_10uH_PMU;	Enable SMPS with 10 µH (default configuration)
SMPS_INDUCTOR	SMPS_INDUCTOR_4_7uH	cold_start_config[11] = SMPS_ON; cold_start_config[8]= SMPS_4.7uH_RM1; cold_start_config[17]= SMPS_4.7uH_PMU;	Enable SMPS with 4.7 µH inductor
SMPS_INDUCTOR	SMPS_INDUCTOR_NONE	cold_start_config[11] = SMPS_OFF;	Disable SMPS

Table 31. Cold start configuration preprocessor options

Preprocessor option	Preprocessor option values	cold_start_config field	Description
BOR_CONFIG	BOR_ON	cold_start_config[11] &= ~(1<<2);	Enable BlueNRG-1, BlueNRG-2 BOR (brown-out threshold)
BOR_CONFIG	BOR_OFF	If BOR_OFF is selected no register set is required, since BOR is disabled by default	Disable BlueNRG-1, BlueNRG-2 BOR (brown-out threshold): default configuration

Regarding the ATB0_ANA_ENG_REG, ATB1_ANA_ENG_REG registers settings, some test modes are also available in order to address some test scenarios.

User should sets such registers as follows:

Table 32. Cold start test mode configurations

Test modes	cold_start_config field	Notes
Low speed crystal oscillator test mode	cold_start_config[2] = 0x37 cold_start_config[5] = 0x34	Refer to bringing up the BlueNRG-1, BlueNRG-2 devices AN4818 for more details about this specific test scenario
High speed start-up time test mode	cold_start_config[2] = 0x04 cold_start_config[5] = 0x34	Refer to bringing up the BlueNRG-1, BlueNRG-2 devices AN4818 for more details about this specific test scenario
TX/RX event alert enabling	cold_start_config[2] = 0x38 cold_start_config[5] = 0x34	Refer to BlueNRG-1, BlueNRG-2 datasheets for more details about the TX/RX event alert enabling

Please notice that the default user mode register setting must be restored for typical user application scenarios:

Table 33. Cold start user mode configuration

User mode	cold_start_config field	Notes
	cold_start_config[2] = 0x00 cold_start_config[5] = 0x30	User mode register settings for cold start configuration

The selected cold start configuration is defined within the BLE device controller through the following instructions executed on `DeviceConfiguration()` function called by `SystemInit()` API (system_bluenrg1.c file) at device initialization (power on):

```
/* Cold start configuration device */
BLUE_CTRL->RADIO_CONFIG = 0x10000U | (uint16_t)((uint32_t)cold_start_config
                          & 0x0000FFFFU);
while ((BLUE_CTRL->RADIO_CONFIG & 0x10000) != 0);
```

2.5 BLE stack tick function

BlueNRG-1, BlueNRG-2 BLE stack provides a special API BTLE_StackTick() which must be called in order to process the internal BLE stack state machines and when there are BLE stack activities ongoing (normally within the main application while loop).

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

If low speed ring oscillator is used instead of the LS crystal oscillator, this function also performs the LS RO calibration and hence must be called at least once at every system wake-up in order to keep the 500 ppm accuracy (at least 500 ppm accuracy is mandatory if acting as a master).

Note: No BLE stack function must be called while the BTLE_StackTick() is running. For example, if a BLE stack function may be called inside an interrupt routine, that interrupt must be disabled during the execution of BTLE_StackTick().

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

```
NVIC_DisableIRQ(UART_IRQn);
BTLE_StackTick();
NVIC_EnableIRQ(UART_IRQn);
```

3 Design an application using BlueNRG-1, BlueNRG-2 BLE stack

This section provides information and code examples about how to design and implement a Bluetooth low energy application on a BlueNRG-1, BlueNRG-2 device.

User implementing a BLE application on a BlueNRG-1, BlueNRG-2 device has to go through some basic and common steps:

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

Note: In the following sections, some user applications “defines” are used to simply identify the device Bluetooth low energy role (central, peripheral, client and server).

Table 34. User application defines the BLE device roles

Define	Description
GATT_CLIENT	GATT client role
GATT_SERVER	GATT server role

3.1 Initialization phase and main application loop

The following main steps are required to properly configure the BlueNRG-1, BlueNRG-2 devices.

1. Initialize the BLE device vector table, interrupt priorities, clock: `SystemInit()` API
2. Configure selected BLE platform: `SdkEvalIdentification()` API
3. Initialize the serial communication channel used for I/O communication as debug and utility information: `SdkEvalComUartInit(UART_BAUDRATE)` API
4. Initialize the BLE stack: `BlueNRG_Stack_Initialization(&BlueNRG_Stack_Init_params)` API
5. Configure BLE device public address (if public address is used): `aci_hal_write_config_data()` API
6. Init BLE GATT layer: `aci_gatt_init()` API
7. Init BLE GAP layer depending on the selected device role: `aci_gap_init("role")` API
8. Set the proper security I/O capability and authentication requirement (if BLE security is used): `aci_gap_set_io_capability()` and `aci_gap_set_authentication_requirement()` APIs
9. Define the required Services & Characteristics & Characteristics Descriptors if the device is a GATT server: `aci_gatt_add_service()`, `aci_gatt_add_char()`, `aci_gatt_add_char_desc()` APIs
10. Add a while(1) loop calling the BLE stack tick API `BTLE_StackTick()` and a specific user tick handler where user actions/events are processed. Further, a call to the `BlueNRG_Sleep()` API is added in order to enable BLE device sleep mode and preserve the BLE radio operating modes.

The following pseudocode example illustrates the required initialization steps:

```
int main(void)
{
  uint8_t ret;

     /* System Init */
    SystemInit();


    /* Identify BLE platform */
    SdkEvalIdentification();

    /* Configure I/O communication channel */
    SdkEvalComUartInit(UART_BAUDRATE);

    /* BLE stack init */
    ret = BlueNRG_Stack_Initialization(&BlueNRG_Stack_Init_params);

    if (ret != BLE_STATUS_SUCCESS) {
      printf("Error in BlueNRG_Stack_Initialization() 0x%02x\r\n", ret);
    while(1);
    }

    /* Device Initialization: BLE stack GATT and GAP Init APIs.
       It could add BLE services and characteristics (if it is a GATT
       server) and initialize its state machine and other specific drivers
       (i.e. leds, buttons, sensors, …) */
     ret = DeviceInit();
```

```
if (ret != BLE_STATUS_SUCCESS) {
  while(1);
}

 while(1)
 {
    /* BLE Stack Tick */
    BTLE_StackTick();

    /* Application Tick: user application where application state machine
       is handled */
    APP_Tick();

     /* Power Save management: enable sleep mode with wakeup on radio
        operating timings (adverting, connections intervals) */
     BlueNRG_Sleep(SLEEPMODE_WAKETIMER, 0, 0);

   }/* while (1) */ }

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

1. `BlueNRG_Stack_Init_params` *variable defines the BLE stack initialization parameters as described on Section 2.2: BLE stack event callbacks.*

2. `BTLE_StackTick()` *must be called in order to process BLE stack events.*

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

4. `BlueNRG_Sleep(SLEEPMODE_WAKETIMER, 0, 0)` *enables the BLE device HW Sleep low power mode: CPU is stopped and all the peripherals are disabled (only the low speed oscillator and the external wakeup source blocks are running). It's worth to notice that this API with the specified parameters (*`SLEEPMODE_WAKETIMER, 0, 0`*) must be called, on application main while loop, in order to allow the BlueNRG-1,* BlueNRG-2 *devices to enter sleep mode with wake-up source on BLE stack advertising and connection intervals. If not called, the BLE device always stays in running power save mode (BLE stack is not autonomously entering in sleep mode unless this specific API is called). User application can use the* `BlueNRG_Sleep()` *API for selecting one of the supported BLE device HW low power modes (CPU halt, sleep, standby) and set the related wakeup sources and sleep timeout, when applicable. The* `BlueNRG_Sleep()` *API combines the low power requests coming from the application with the radio operating mode, choosing the best low power mode applicable in the current scenario. The negotiation between the radio module and the application requests is done to avoid losing of data exchanged over the air.*

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

6. When performing the `aci_gatt_init()` *&* `aci_gap_init()` *APIs, BLE stack always adds two standard services: attribute profile service (0x1801) with service changed characteristic and GAP service (0x1800) with device name and appearance characteristics.*

7. 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 35. 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 36. 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						
	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 37. aci_gap_init() role parameter values

Parameter	Role parameter values	Notes
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 host-based privacy 0x02 for enabling controller-based privacy	Controller-based privacy is supported on BLE stack v2.x
device_name_char_len		It allows the length of the device name characteristic to be indicated

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and events documentation, on *Section 5: References*.

3.1.1 BLE addresses

The following device addresses are supported from BlueNRG-1, BlueNRG-2 devices:

- Public address
- Random address
- Private address

Public MAC addresses (6 bytes- 48-bit address) uniquely identifies a BLE device, and they are defined by 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).

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

The BLE API command to set the MAC address is `aci_hal_write_config_data()`.

The command `aci_hal_write_config_data()` should be sent to BlueNRG-1, BlueNRG-2 devices before starting any BLE operations (after BLE stack initialization API `BlueNRG_Stack_Initialization()`).

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

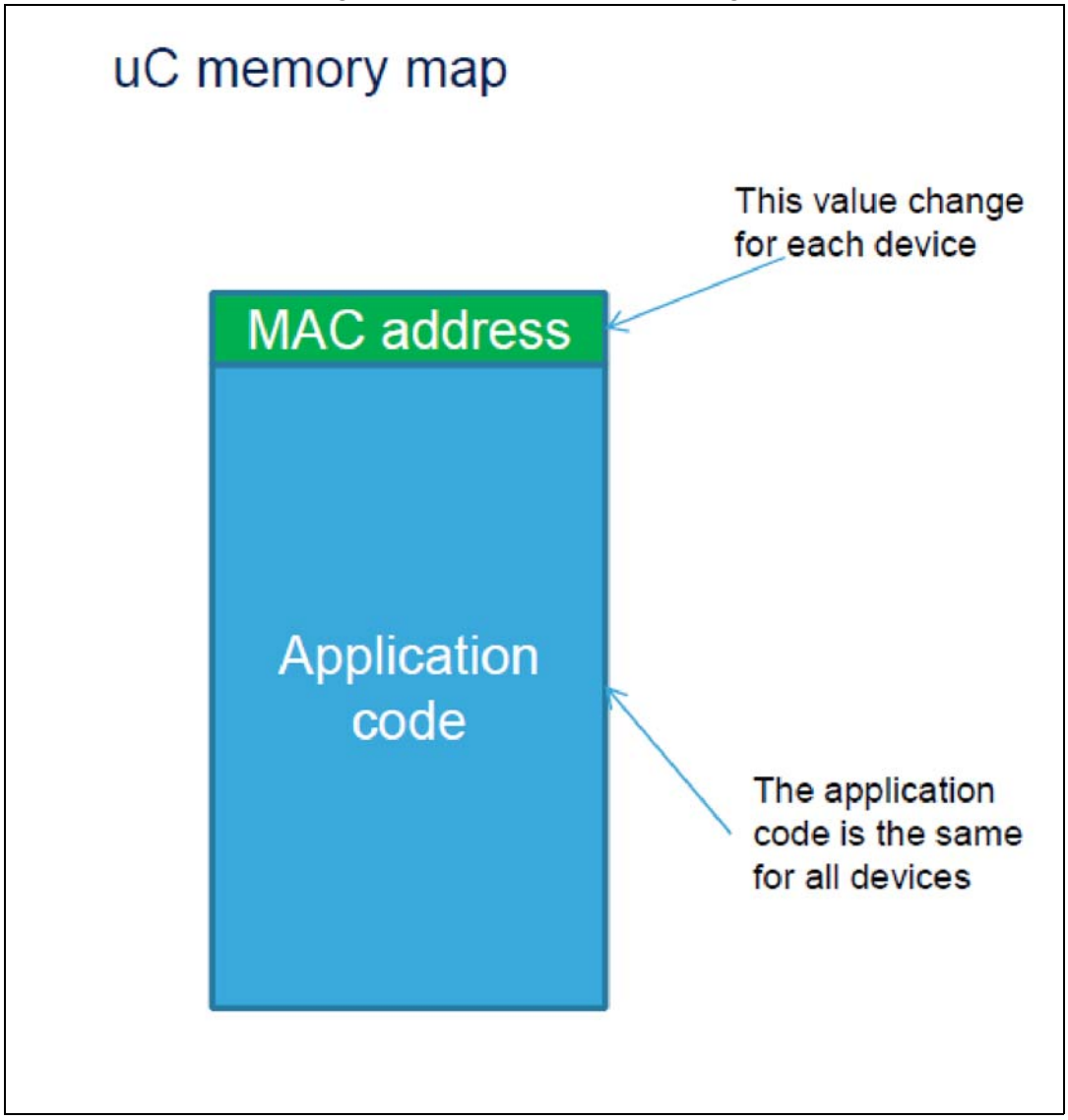
```
uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret=aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,CONFIG_DATA_PUBAD
DR_LEN, bdaddr);
if(ret)PRINTF("Setting address failed.\n")}
```

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

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

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

Figure 9. BLE MAC address storage



The BlueNRG-1, BlueNRG-2 devices do not have a valid preassigned MAC address, but a

unique serial number (read only for user).The unique serial number is a six bytes value stored at address 0x100007F4: it is stored as two words (8 bytes) at address 0x100007F4 and 0x100007F8 with unique serial number padded with 0xAA55.
The static random address is generated and programmed at very 1st boot of device on the dedicated Flash area. The value on Flash is the actual value the device uses: each time the user resets the device the stack checks if valid data are on the dedicated Flash area and it uses it (a special valid marker on FLASH is used to identify if valid data are present). If the user performs mass erase, the stored values (including marker) are removed so the stack generates a new random address and store it on the dedicated flash.
Private addresses are used when privacy is enabled and according to the Bluetooth low energy specification. For more information about private addresses, refer to *Section 1.7: Security manager (SM)*.

3.1.2 Set tx power level

During the initialization phase user can also select the transmitting power level using the following API: `aci_hal_set_tx_power_level(high or standard, power level)`

Follow a pseudocode example for setting the radio transmit power in high power and -2 dBm output power: `ret= aci_hal_set_tx_power_level (1,4);`

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and events documentation, on *Section 5: References*

3.2 Services and characteristic configuration

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*).

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.

For a detailed description of the `aci_gatt_add_service()` and `aci_gatt_add_char()` functions parameters refer to the header file Library\Bluetooth_LE\inc\bluenrg1_events.h.

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

```
/* Service and characteristic UUIDs variables. Refer to the header
file Library\Bluetooth_LE\inc\bluenrg1_api.h for a detailed description
 */
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,0xb1,0xe0,0xf2,0x7
3,0xd9};
    /*Characteristic_1 128bits UUID */
    const uint8_t charUuid_1[16] =

{0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe1,0xf2,0x7
3,0xd9};
    /*Characteristic_2 128bits UUID */
    const uint8_t charUuid_2[16] =

{0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe2,0xf2,0x7
3,0xd9};
    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() */
```

3.3 Create a connection: discoverable and connectable APIs

In order to establish a connection between a BLE GAP central (master) device and a BLE GAP peripheral (slave) device, the GAP discoverable/connectable modes and procedures can be used as described in *Table 38: GAP mode APIs*, *Table 39: GAP discovery procedure APIs* and *Table 40: Connection procedure APIs* and by using the related BLE stack APIs provided in header file: *Library\Bluetooth_LE\inc\bluenrg1_api.h*.

GAP peripheral discoverable and connectable modes APIs

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

Table 38. GAP mode APIs

API	Supported advertising event types	Description
`aci_gap_set_discoverable()`	0x00: connectable undirected advertising (default)	Sets the device in general discoverable mode. The device is discoverable until the device issues the `aci_gap_set_non_discoverable()` API.
	0x02: scannable undirected advertising	
	0x03: non-connectable undirected advertising	
`aci_gap_set_limited_discoverable()`	0x00: connectable undirected advertising (default);	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
	0x02: scannable undirected advertising;	
	0x03: non-connectable undirected advertising.	
`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.

Table 38. GAP mode APIs (continued)

API	Supported advertising event types	Description
aci_gap_set_non_connectable()	0x02: scannable undirected advertising	Puts the device into non- connectable mode.
	0x03: non-connectable undirected advertising	
aci_gap_set_undirect_connectable()	NA	Puts the device into undirected connectable mode.

Table 39. 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 40. 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.

3.3.1 Set discoverable mode and use direct connection establishment procedure

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

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

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");



void GAP_Peripheral_Make_Discoverable(void )
{
 tBleStatus ret;

 const charlocal_name[]=
{AD_TYPE_COMPLETE_LOCAL_NAME,'B','l','u','e','N','R','G','1','T','e','s','
t'};

 /* 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_Length:
    13
    Local_Name: BlueNRG1Test;
   Service_Uuid_Length: 0 (no service to be advertised); Service_Uuid_List:
NULL;
    Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
    Slave_Conn_Interval_Max:  0 (Slave 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 (Slave latency for the connection in a number of
   connection events);
   Supervision_Timeout: 60 (Supervision timeout for the LE Link);
   Minimum_CE_Length: 2000 (Minimum length of connection needed for the
   LE connection);
   Maximum_CE_Length: 2000 (Maximum length of connection needed for the LE
connection).

  */

   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()`.

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 master has to allocate for a single slave so they must be wisely chosen. In particular, when a master connects to more slaves, the connection interval for each slave must be equal or a multiple of the other connection intervals and user must not overdo the connection event length for each slave. Refer to Section 4: BLE multiple connection timing strategy connections timing strategy for detailed information about the timing allocation policy.*

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

The following pseudocode example illustrates only the specific steps to be followed for putting a GAP Peripheral device in general discoverable mode, and for a GAP central device to 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,'B','l','u','e','N','R','G' };
  /* As scan response data, a proprietary 128bits Service UUID is used.
     This 128bits data cannot be inserted within the advertising packet
     (ADV_IND) due its length constraints (31 bytes).
     AD Type description:
     0x11: length
     0x06: 128 bits Service UUID type
     0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x20,0x0c,
     0x9a,0x66: 128 bits Service UUID
   */
   uint8_t ServiceUUID_Scan[18]=
{0x11,0x06,0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x2
,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: BlueNRG;
   Service_Uuid_Length: 0 (no service to be advertised);
   Service_Uuid_List: NULL;
   Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
   Slave_Conn_Interval_Max: 0 (Slave 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 discoverable 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 ;

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

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

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

   /* data_length octets of advertising or scan response data formatted are
```

```
        on Advertising_Report[0].Data field: to be stored/filtered based on
        specific user application scenario*/

} /* hci_le_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 41. ADV_IND event type

Event type	Address type	Address	Advertising data	RSSI
0x00 (ADV_IND)	0x00 (public address)	0x0280E1003412	0x02,0x01,0x06,0x08,0x09,0x42,0x6C,0x75,0x65,0x4E,0x52,0x47,0x02,0x0A,0xFE	0xCE

The advertising data can be interpreted as follows (refer to Bluetooth specifications in *Section 5: References)*

Table 42. 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)	0x08: length of the field 0x09: complete local name type 0x42,0x6C,0x75,0x65,0x4E0x52,0x47: BlueNRG	0x02: length of the field 0x0A: Tx power type 0x08: power value

Table 43. 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 interpreted as follows (refer to Bluetooth specifications):

Table 44. 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 bits service UUID

3.4 BLE stack events and events callbacks

Whenever there is a BLE stack event to be processed, the BLE stack library notifies this event to the user application through a specific event callback. A event callback is a function defined by the user application and called by the BLE stack, while an API is a function defined by the stack and called by the user application. The BlueNRG-1, BlueNRG-2 BLE stack events callbacks prototypes are defined on file bluenrg1_events.h. Weak definitions are available for all the event callbacks in order to have a definition for each event callback. As consequence, based on its own application scenario, user has to identify the required device events callbacks to be called and the related application specific actions to be done.

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

Table 45. BLE stack: main event 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 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

Table 45. BLE stack: main event callbacks

Event callback	Description	Where
`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 slave security request from a master which has previously bonded with the slave. When this event is received, the upper layer has to issue the command `aci_gap_allow_rebond()` to allow the slave to continue the pairing process with the master	GAP peripheral
`aci_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 BlueNRG-1, BlueNRG-2 Bluetooth LE stack APIs and events documentation, on *Section 5: References.*

The following pseudocode provides an example of event 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.
*/
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 Master_Clock_Accuracy)

{
    /* Add user code for handling BLE connection complete event based on
       application scenario.
      NOTE: Refer to header file Library\Bluetooth_LE\inc\bluenrg1_events.h
       for a complete description of the event callback parameters.
    */

/* Store connection handle */
 connection_handle = Connection_Handle;
 …
}/* 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.
      NOTE: Refer to header file Library\Bluetooth_LE\inc\bluenrg1_events.h
       for a complete description of the event callback parameters.
    */
    ...
} /* 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.
    NOTE: Refer to header file Library\Bluetooth_LE\inc\bluenrg1_events.h
    for a complete description of the event callback parameters.
*/
```

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

3.5 Security (pairing and bonding)

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

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

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

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

Passkey entry example with 2 BlueNRG devices: Device_1, Device_2

The following pseudocode example illustrates only the specific steps to be followed for pairing 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 for setting 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 for setting 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_slave_security_req()` on a GAP Peripheral (slave) device (it sends a slave security request to the master):

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

  or

- By using the `aci_gap_send_pairing_req()` on a GAP central (master) device.

Since the no fixed pin has been set, once the paring procedure is initiated by one of the two devices, BLE device calls the `aci_gap_pass_key_req_event()` event callback (with related connection handle) for asking to the user application to provide the password to be used for establishing 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 through 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_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 */
                                               0x00,  /* fixed pin is used*/
                                                0x123456, /* fixed pin */
                                            0x00  /* Public Identity address
                                                                type */);
                  if (ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");
```

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

The `reason` parameter provides the pairing failed reason code in case of failure (0 if status parameter returns success or timeout)

2. When two devices get paired, the link is automatically encrypted during the first connection. If bonding is also enabled (keys are stored for a future time), when the 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 will not perform the paring process but will just encrypt the link:
- `aci_gap_slave_security_req )` on the GAP peripheral (slave) device or
- `aci_gap_send_pairing_req()` on the GAP central (master) device.

3. If a slave has already bonded with a master, it can send a slave security request to the master to encrypt the link. When receiving the slave security request, the master may encrypt the link, initiate the pairing procedure, or reject the request. Typically, the master only encrypts the link, without performing the pairing procedure. Instead, if the master starts the pairing procedure, it means that for some reasons, the master lost its bond information, so it has to start the pairing procedure again. As a consequence, the slave device calls the `aci_gap_bond_lost_event()` event callback to inform the user application that it is not bonded anymore with the master it was previously bonded. Then, the slave application can decide to allow the security manager to complete the pairing procedure and re-bond with the master by calling the command `aci_gap_allow_rebond()`, or just close the connection and inform the user about the security issue.

3.6 Service and characteristic discovery

This section describes the main functions allowing a BlueNRG-1, BlueNRG-2 GAP central device to discover the GAP peripheral services and characteristics, once the two devices are connected.

The sensor profile demo services & characteristics with related handles are used as reference services and characteristics on the following pseudocode examples. Further, it is assumed that a GAP central device is connected to a GAP peripheral device running the sensor demo profile application. The GAP central device use the service and discovery procedures to find the GAP peripheral sensor profile demo service and characteristics.

Table 46. BLE sensor profile demo services and characteristic handles

Service	Characteristic	Service /characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
Acceleration service	NA	0x000C	NA	NA	NA
	Free Fall characteristic	0x000D	0x000E	0x000F	NA
	Acceleration characteristic	0x0010	0x0011	0x0012	NA
Environmental service	NA	0x0013	NA	NA	NA
	Temperature characteristic	0x0014	0xx0015	NA	0x0016
	Pressure characteristic	0x0017	0xx0018	NA	0x0019

For detailed information about the sensor profile demo, refer to the BlueNRG-1_2 DK User Manual and the sensor demo source code available within the BlueNRG-1_2 DK software package, see *Section 5: References*.

Service discovery procedures and related GATT events

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

Table 47. 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 48. First read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Note
0x0001	0x0004	0x1801	Attribute profile service (GATT_Init() adds it). 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 49. Second read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Note
0x000C	0x0012	0x02366E80CF3A11E19AB40002A5D5C51B	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 50. Third read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Note
0x0013	0x0019	0x42821A40E47711E282D00002A5D5C51B	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
```

3.6.1 Characteristic discovery procedures and related GATT events

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

Table 51. 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 47: Service discovery procedures APIs*

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 by type response events (through related `aci_att_read_by_type_resp_event()` event callback), with the following callback parameters 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 52. 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 53. 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 46: BLE sensor profile demo services and characteristic handles*).

3.7 Characteristic notification/indications, write, read

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

Table 54. Characteristics update, read, write APIs

Discovery service API	Description	Where
aci_gatt_update_char_value()	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 start 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 sensor profile demo, follows a simple pseudo code the GAP central application should use in order to configure the free fall and the acceleration characteristics client descriptor configuration for notification:

```
tBleStatus ret;
uint16_t handle_value = 0x000F;
/*Enable the free fall characteristic client descriptor configuration for
ret = aci_gatt_write_charac_desc(conn_handle,
                                 handle_value  /* handle for free fall
                                                  client descriptor
                                                  configuration */
                                 0x02,         /* attribute value length */
                                 0x0001,       /* attribute value: 1 for
                                                  notification */
                                 );
if (ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");

handle_value = 0x0012;
```

```
/*Enable the acceleration characteristic client descriptor configuration
 for notification */
ret= aci_gatt_write_char_desc(conn_handle,
                              handle_value /* handle for acceleration
                                              client descriptor
                                                  configuration *
                              0x02, /*attribute value
                                        length */
                              0x0001, /* attribute value:
                                        1 for notification */
);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

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

```
tBleStatus ret;
uint8_t val = 0x01;
uint16_t service_handle = 0x000C;
uint16_t charac_handle = 0x000D;

/*GAP peripheral notifies free fall characteristic to GAP central*/
ret= aci_gatt_update_char_value(service_handle , /* acceleration
                                                    service handle */
                                charac_handle, /* free fall
                                                  characteristic handle
                                0, /* characteristic value offset */
                                0x01, /* characteristic value length */
                                &val /* characteristic value */
                                );
if(ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");

tBleStatus ret;
uint8_t buff[6];
uint16_t charac_handle = 0x0010;

/*Set the mems acceleration values on three axis x,y,z on buff array */
....
/*GAP peripheral notifies acceleration characteristic to GAP central*/
ret = aci_gatt_update_char_value(service_handle, /* acceleration
                                                    service handle */
                                 charac_handle, /* acceleration
                                                   characteristic handle */
                                 0 ,/* characteristic
                                       value offset */
                                 0x06, /* characteristic
                                        value length */
                                 buff, /* characteristic
                                        value */
                          );
if(ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

On GAP central, the `aci_gatt_notification_event()` event callback is called is raised on reception of the characteristic notification (acceleration or free fall) from the GAP peripheral device. Follow a pseudo code of the `aci_gatt_notification_event()` callback:

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

}/* aci_gatt_notification_event() */
```

3.8 Basic/typical error condition description

On BlueNRG-1, BlueNRG-2 BLE stack APIs framework, the `tBleStatus` type is defined in order to return the BlueNRG-1, BlueNRG-2 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 BlueNRG-1, BlueNRG-2 Bluetooth LE stack APIs and event documentation, on *Section 5: References*

3.9 BLE simultaneously master, slave scenario

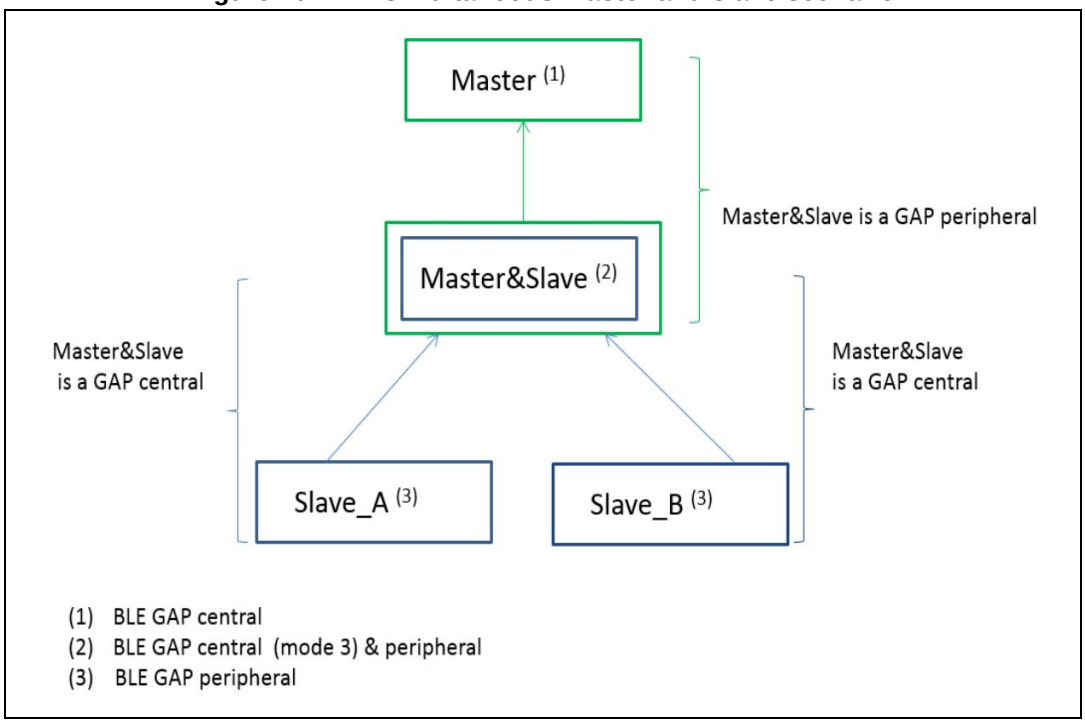
BlueNRG-1, BlueNRG-2 BLE stack supports multiple roles simultaneously. This allows the same device to act as master on one or more connections (up to eight connections are supported), and to act as a slave on another connection.

The following pseudo code describes how a BLE stack device can be initialized for supporting 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 master and slave test scenario can be targeted as follows:

Figure 10. BLE simultaneous master and slave scenario



Note: Same scenario is also valid when using BlueNRG-2 devices.

1. One BLE device (called master and slave) is configured as central and peripheral by setting role as GAP_PERIPHERAL_ROLE |GAP_CENTRAL_ROLE on `aci_gap_init()` API. Let's also assume that this device also defines a service with a characteristic.
2. Two BLE devices (called Slave_A, Slave_B) are configured as Peripheral by setting role as GAP_PERIPHERAL_ROLE on `aci_gap_init()` API. Both Slave_A and Slave_B define the same service and characteristic as Master&Slave device.
3. One BLE device (called Master) is configured as central by setting role as GAP_CENTRAL_ROLE on `aci_gap_init()` API.
4. Both Slave_A and Slave_B devices enter in 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;
                              Slave_Conn_Interval_Min = 0x0006,
                              Slave_Conn_Interval_Max = 0x0008);
```

5. Master and slave device performs a discovery procedure in order to discover the peripheral devices Slave_A and Slave_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, Master&Slave device starts two connections procedures (as Central) for connecting, respectively, to Slave_A and Slave_B devices:

```
/* Connect to Slave_A:Slave_A address type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010
                               Peer_Address_Type= ”Slave_A address type”
                               Peer_Address= ”Slave_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 Slave_B:Slave_B address type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010,
                               Peer_Address_Type= ”Slave_B address type”,
                               Peer_Address= ”Slave_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, Master&Slave device enables the characteristics notification, on both of them, using the `aci_gatt_write_char_desc()` API. Slave_A and Slave_B devices start the characteristic notification by using the `aci_gatt_upd_char_val()` API.

8. At this stage, Master&Slave device enters in discovery mode (acting as Peripheral):

```
/*Put Master&Slave 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;
                                    Slave_Conn_Interval_Min = 0x0006,
                                    Slave_Conn_Interval_Max = 0x0008);
```

Since Master&Slave device is also acting as a Central device, it receives the notification event related to the characteristics values notified from, respectively, Slave_A and Slave_B devices.

9. Once Master&Slave device enters in discovery mode, it also waits for connection request coming from the other BLE device (called Master) configured as GAP central. Master device starts discovery procedure for discovering the Master&Slave 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 Master&Slave device is discovered, Master device starts a connection procedure for connecting to it:

```
/* Master device connects to Master&Slave device: Master&Slave
   address type and address have been found during the discovery
   procedure through the Advertising Report events */
   ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                                  LE_Scan_Window=0x0010,
                                  Peer_Address_Type= ”Master&Slave
                                  address type,
                                  Peer_Address= “Master&Slave 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);
```

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

11. Once connected, Master device enables the characteristic notification on Master&Slave device using the `aci_gatt_write_char_desc()` API.
12. At this stage, Master&Slave device receives the characteristics notifications from both Slave_A, Slave_B devices , since it is a GAP Central and, as GAP Peripheral, it is also able to notify these characteristics values to the Master device.

Note: A set of test scripts allowing the described BLE to be executed simultaneously Master, Slave scenario are provided within the BlueNRG GUI SW package (see Section 5: References). These scripts can be run using the BlueNRG GUI and they can be taken as reference for implementing a firmware application using the BlueNRG-1, BlueNRG-2 simultaneously master and slave feature.

3.10 Privacy

BLE stack v2.x supports the Bluetooth low energy v4.2 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.

There are two variants of the privacy feature:

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

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

3.10.1 Controller-based privacy and the device filtering scenario

On BLE stack v2.x, the `aci_gap_init()` API supports the following options for the `privacy_enabled` parameter:

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

When a slave device wants to resolve a resolvable private address and be able to filter on private addresses for reconnection with bonded and trusted devices, it must perform the following steps:

1. Enable privacy controller on `aci_gap_init()`: use 0x02 as `privacy_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 for adding 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. Device enters in 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 white list only, allow connect request from white list only).
7. When a bonded master device performs a connection procedure for reconnection to the slave device, the slave device is able to resolve and filter the master address and connect with it.

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

3.10.2 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, BlueNRG-1, BlueNRG-2 stack automatically finds the correct LTK to encrypt the link.

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

Host-based privacy

If controller privacy is not enabled, a resolvable private address can be resolved by using `aci_gap_resolve_private_addr()`. The address is resolved if the corresponding IRK can be found among the stored IRKs of the bonded devices. A resolvable private address may be received when BlueNRG-1 and BlueNRG-2 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 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 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.

4 BLE multiple connection timing strategy

This section provides an overview of the connection timing management strategy of BlueNRG-1, BlueNRG-2 BLE stack when multiple master and slave connections are active.

4.1 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.

4.1.1 Advertising timing

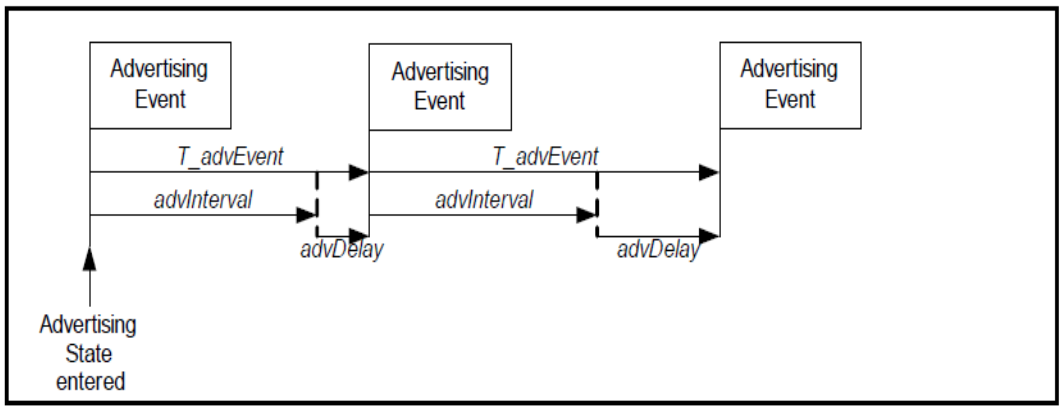
The timing of the advertising state is characterized by 3 timing parameters, linked by this formula:

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 is not 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 11. Advertising timings



4.1.2 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 an advertising channel index

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

The scanWindow is less than or equal to the scanInterval.

4.1.3 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 master starts transmitting a data channel PDU to the slave, which in turn listens to the packet sent by its master at the anchor point.

The master ensures that a connection event closes at least T_IFS=150 µs (Inter frame spacing time, i.e. time interval between consecutive packets on 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.

- *slave latency* (`connSlaveLatency`): allows a slave to use a reduced number of connection events. This parameter defines the number of consecutive connection events that the slave device is not required to listen to the master.

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 to choose the actual parameters in order to fulfill additional timing constraints e.g. in the case of multiple connections.

4.2 BLE stack timing and slot allocation concepts

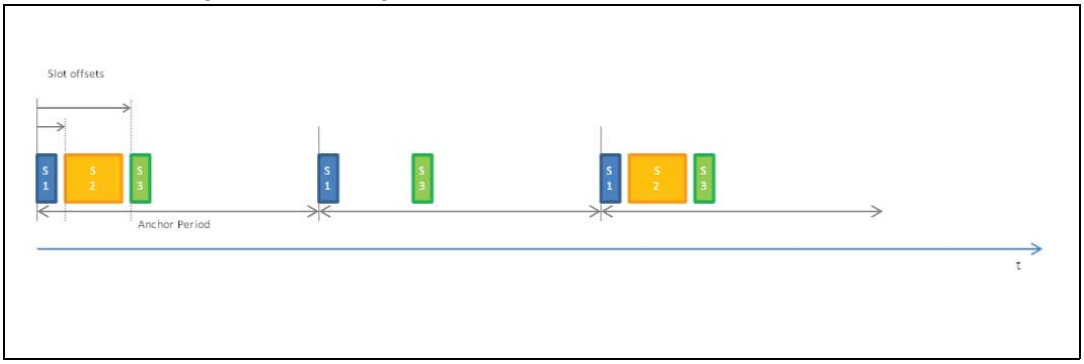
The BlueNRG-1, BlueNRG-2 BLE stack adopts a time slotting mechanism in order to allocate simultaneous master and slave connections. The basic parameters, controlling the slotting mechanism, are indicated in the below table:

Table 55. Timing parameters of the slotting algorithm

Parameter	Description
Anchor period	Recurring time interval inside which up to 8 connection slots can be allocated. Among these 8 slots, only 1 at a time may be a scanning or advertising slot (they are mutually exclusive)
Slot duration	Time interval inside which a full event (i.e. 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 *Figure 12: Example of allocation of three connection slots*.

Figure 12. 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 1.25 ms guard time.

4.2.1 Setting the timing for the first master connection

The time base mechanism above described, is actually started when the first master 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 master 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 master connections.

4.2.2 Setting the timing for further master 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:
   $Conn\_Interval\_Min \leq Anchor\_Period \times m \leq Conn\_Interval\_Max$
   If such value is found then the current anchor period is maintained and the connection interval for the new connection is set equal to $Anchor\_Period \cdot m$ 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:

$$\frac{Anchor\_Period}{k}$$

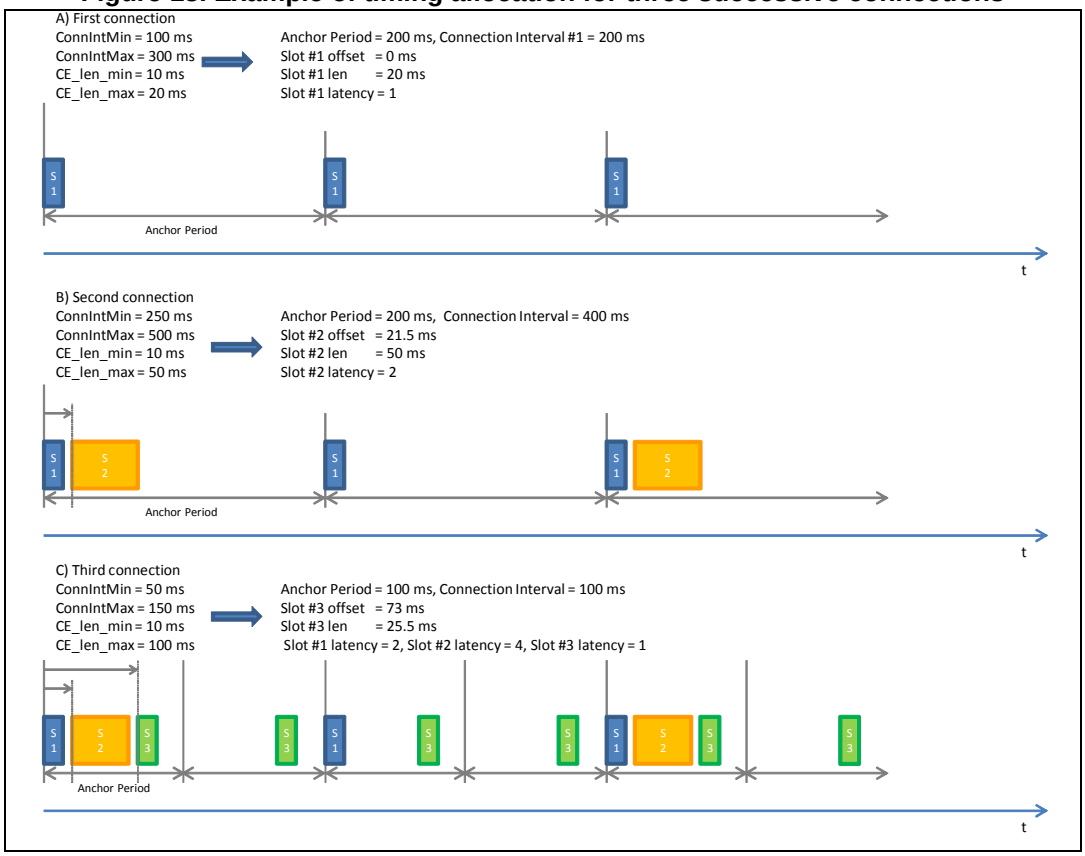
and the slot latency for the existing connections is multiplied by a factor k. Note that in this case the following conditions must also be satisfied:

- 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 13: 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 13. Example of timing allocation for three successive connections



4.2.3 Timing for advertising events

The periodicity of the advertising events, controlled by *advInterval*, is computed based on the following parameters specified by the slave 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 SW to be equal to 14.6 ms (i.e. the maximum allowed advertising event length) and it cannot be reduced.

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

4.2.4 Timing for scanning

Scanning timing is requested by the master 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 master time base

Scanning slots are allocated within the same time base of the other active master slots (i.e. 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 SW automatically tries to subsample the LE_Scan_Interval and to reduce the allocated scan slot length (up to ¼ of the LE_Scan_Window) in order 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).

4.2.5 Slave timing

The slave timing is defined by the master when the connection is created so the connection slots for slave links are managed asynchronously with respect to the time base mechanism described above. The slave assumes that the master may use a connection event length as long as the connection interval.

The scheduling algorithm adopts a round-robin arbitration strategy any time a collision condition is predicted between a slave and a master slot. In addition to this, the scheduler may also impose a dynamic limit to the slave connection slot duration to preserve both master and slave connections.

In particular:

- If the end of a master connection slot overlaps the beginning of a slave connection slot then master and slave connections are alternatively preserved/canceled.
- if the end of a slave connection slot overlaps the beginning of a master connection slot then the slave connection slot length is hard limited to avoid such overlap. If the resulting time interval is too small to allow at least two packets to be exchanged then round robin arbitration is used.

4.3 Multiple master and slave connection guidelines

The following guidelines should be followed to properly handle multiple master and slave connections using BlueNRG-1, BlueNRG-2 devices:

1. Avoid over-allocating connection event length: choose `Minimum_CE_Length` and `Maximum_CE_Length` as small as possible to strictly satisfy the application needs. In this manner 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 master connection:
   a) If possible, create the connection with the shortest connection interval as the first one so to allocate further connections with connection interval multiple of the initial anchor period.
   b) If possible, choose `Conn_Interval_Min = Conn_Interval_Max` as multiple of 10 *ms* to allocate further connections with connection interval sub multiple by a factor 2, 4 and 8 (or more) of the initial anchor period being still a multiple of 1.25 *ms*.
3. For additional master connections:
   a) Choose `ScanInterval` equal to the connection interval of one of the existing Master connections
   b) Choose `ScanWin` such that the sum of the allocated master 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 sub multiple 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 master slots (including advertising, if active) plus `Minimum_CE_Length` is lower than the shortest allocated connection interval
4. Every time you start advertising:
   a) If direct advertising, choose Advertising_Interval_Min = Advertising_Interval_Max = integer multiple of the shortest allocated connection interval
   b) If not Direct Advertising, choose Advertising_Interval_Min = Advertising_Interval_Max such that (Advertising_Interval_Min + 5ms) is an integer multiple of the shortest allocated connection interval
5. Every time you start scanning:
   a) Every time you start scanning: a) choose ScanInterval equal to the connection interval of one of the existing master connections
   b) Choose ScanWin such that the sum of the allocated master slots (including Advertising, if active) is lower than the shortest allocated connection interval
6. Keep in mind that the process of creating multiple connections, then closing some of them and creating new ones again, over time, tends to 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.

5 References

Table 56. List of references

Name	Title/description
AN4818	Bringing up the BlueNRG-1, BlueNRG-2 device application note
AN4820	BlueNRG-1, BlueNRG-2 device power mode application note
BlueNRG-1 datasheet	BlueNRG-1 SoC datasheet
BlueNRG-2 datasheet	BlueNRG-2 SoC datasheet
Bluetooth specifications	Specification of the Bluetooth system (v4.0, v4.1, v4.2, v5.0)
_	BlueNRG-1, BlueNRG-2, Bluetooth LE stack APIs and event oxygen documentation
STSW-BLUENRG1-DK	BlueNRG-1 SW package for BlueNRG-1, BlueNRG-2 devices
STSW-BNRGUI	BlueNRG GUI SW package
UM2071	BlueNRG-1, BlueNRG-2 DK user manual

Appendix A List of acronyms and abbreviations

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

Table 57. 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

Table 57. List of acronyms (continued)

Term	Meaning
UUID	Universally unique identifier
WPAN	Wireless personal area networks

6 Revision history

Table 58. Document revision history

Date	Revision	Changes
28-Nov-2016	1	Initial release.
29-Jun-2017	2	Added reference to BlueNRG-2 device throughout the document. Updated *Introduction*, *Section 1.7: Security manager (SM)*; *Section 2: BlueNRG-1, BlueNRG-2 Bluetooth low energy stack*, *Section 3.1: Initialization phase and main application loop*, *Section 3.5: Security (pairing and bonding)*, *Section 3.10: Privacy* and *Table 56: List of references*. Added *Section 1.8: Privacy*, *Section 2.4: BlueNRG-1, BlueNRG-2 cold start configuration*, *Section 2.5: BLE stack tick function*