PM0237
Programming manual

BlueNRG, BlueNRG-MS stacks 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) host applications using BlueNRG, BlueNRG-MS stacks APIs.

The document describes the BlueNRG, BlueNRG-MS stacks architecture, APIs interface and callbacks allowing to get access to the Bluetooth low energy functions provided by the BlueNRG, BlueNRG-MS network coprocessors.

This programming manual also provides some fundamental concepts about the Bluetooth low energy (BLE) technology in order to associate the BlueNRG, BlueNRG-MS APIs, parameters, and related events with the BLE protocol stack features. It is assumed that the user already has a basic knowledge about the BLE technology and its main features.

For more information related to the full set related to the BlueNRG, BlueNRG-MS devices and the Bluetooth specification v4.0 and v4.1, refer to *Section 4: References* at the end of this document.

The BlueNRG is a very low power Bluetooth low energy (BLE) single-mode network processor, compliant with Bluetooth specification v4.0 and supporting master or slave role.

The BlueNRG-MS is a very low power Bluetooth low energy (BLE) single-mode network processor, compliant with Bluetooth specification v4.1 and supporting both master and slave roles.

The manual is structured as follows:

- Fundamentals of Bluetooth low energy (BLE) technology
- BlueNRG, BlueNRG-MS stacks architecture and application command interface (ACI) overview
- How to design an application using the BlueNRG, BlueNRG-MS stacks ACI APIs

Note: *The document content is valid for both BlueNRG and BlueNRG-MS devices. Any specific difference is highlighted whenever it is needed.*

Contents

1 Bluetooth low energy technology 7

1.1 BLE stack architecture 8

1.2 Physical layer 9

1.3 Link Layer (LL) 11

1.3.1 BLE packets 12

1.3.2 Advertising state 15

1.3.3 Scanning state 16

1.3.4 Connection state 17

1.4 Host controller interface (HCI) 18

1.5 Logical link control and adaptation layer protocol (L2CAP) 18

1.6 Attribute Protocol (ATT) 18

1.7 Security Manager (SM) 20

1.8 Generic attribute profile (GATT) 22

1.8.1 Characteristic attribute type 22

1.8.2 Characteristic descriptors type 24

1.8.3 Service attribute type 24

1.8.4 GATT procedures 25

1.9 Generic access profile (GAP) 26

1.10 BLE profiles and applications 31

1.10.1 Proximity profile example 32

2 BlueNRG, BlueNRG-MS stacks architecture and ACI 34

2.1 ACI interface 35

2.2 ACI Interface resources 36

2.3 Other platforms resources files 38

2.3.1 Platforms configuration 38

2.4 How to port the ACI SPI interface framework to a selected microcontroller 40

3 Design an application using BlueNRG, BlueNRG-MS ACI APIs 41

3.1 Initialization phase and main application loop 42

3.1.1 BLE addresses 45

3.1.2 Set tx power level 47

3.2 BlueNRG, BlueNRG-MS events and events Callback 48

3.3 Services and characteristic configuration 52

3.4 Create a connection: discoverable and connectable APIs 54

3.4.1 Set discoverable mode & use direct connection establishment procedure 56

3.4.2 Set discoverable mode & use general discovery procedure (active scan) 58

3.5 Security (pairing and bonding) 63

3.6 Service and characteristic discovery 67

3.6.1 Service discovery procedures and related GATT events 69

3.6.2 Characteristics discovery procedures and related GATT events 73

3.7 Characteristic notification/indications, write, read 76

3.8 Basic/typical error conditions description 80

3.9 BlueNRG-MS simultaneously Master, Slave scenario 80

4 References 84

Appendix A List of acronyms and abbreviations 85

5 Revision history 87

List of figures

Figure 1. Bluetooth low energy technology enabled coin cell battery devices 7
Figure 2. Bluetooth low energy stack architecture 9
Figure 3. Link Layer state machine 11
Figure 4. Packet structure 12
Figure 5. Advertising packet with AD type flags 15
Figure 6. Example of characteristic definition 23
Figure 7. Client and server profiles 32
Figure 8. BlueNRG, BlueNRG-MS stacks architecture and interface to the external host. 34
Figure 9. MAC address storage 46
Figure 10. BlueNRG-MS simultaneous Master & Slave scenario 81

List of tables

Table 1. BLE RF channel types and frequencies . . . 10
Table 2. Advertising data header file content . . . 12
Table 3. Advertising packet types. . . . 13
Table 4. Advertising event type and allowable responses . . . 14
Table 5. Data packet header content . . . 14
Table 6. Packet length field and valid values . . . 14
Table 7. Connection request timings intervals . . . 17
Table 8. Attribute example . . . 19
Table 9. Attributes protocol messages . . . 19
Table 10. Combination of Input/Output capabilities on a BLE device. . . . 20
Table 11. Methods used for calculating the Temporary Key (TK). . . . 21
Table 12. Characteristic declaration . . . 23
Table 13. Characteristic value . . . 24
Table 14. Service declaration . . . 24
Table 15. Include declaration . . . 25
Table 16. Discovery procedures and related response events. . . . 25
Table 17. Client-initiated procedures and related response events . . . 25
Table 18. Server-initiated procedures and related response events . . . 26
Table 19. GAP roles . . . 26
Table 20. GAP broadcaster mode . . . 27
Table 21. GAP discoverable modes . . . 27
Table 24. GAP observer procedure . . . 28
Table 22. GAP connectable modes . . . 28
Table 23. GAP bondable modes. . . . 28
Table 26. GAP connection procedures. . . . 29
Table 25. GAP discovery procedures . . . 29
Table 27. GAP bonding procedures . . . 30
Table 28. ACI Interface. . . . 35
Table 29. ACI Interface resources files. . . . 37
Table 30. SW framework platforms drivers. . . . 38
Table 31. User application defines for BLE devices role . . . 41
Table 32. GAP_Init() role parameter values . . . 44
Table 33. ACI: main events, sub-events. . . . 48
Table 34. ACI: GAP modes APIs . . . 54
Table 35. ACI: discovery procedures APIs . . . 55
Table 36. ACI: connection procedures APIs. . . . 55
Table 37. EVT_BLUE_GAP_DEVICE_FOUND event: ADV_IND event. . . . 62
Table 38. ADV_IND advertising data . . . 62
Table 39. SCAN_RSP event. . . . 62
Table 40. Scan response data. . . . 62
Table 41. BlueNRG sensor profile demo services & characteristics handles. . . . 67
Table 42. BlueNRG-MS sensor profile demo services & characteristics handles . . . 67
Table 43. ACI: service discovery procedures APIs. . . . 69
Table 44. First evt_att_read_by_group_resp event data . . . 71
Table 45. Second evt_att_read_by_group_resp event data. . . . 71
Table 46. Third evt_att_read_by_group_resp event data. . . . 72
Table 47. BlueNRG ACI: characteristics discovery procedures APIs. . . . 73
Table 48. First evt_att_read_by_type_resp event data . . . 75

Table 49. Second evt_att_read_by_type_resp event data 76
Table 50. Characteristics update, read, write APIs 76
Table 51. References table. 84
Table 52. List of acronyms. 85
Table 53. Document revision history 87

1 Bluetooth low energy technology

Bluetooth low energy (BLE) wireless technology has been developed by the Bluetooth Special Interest Group (SIG) in order to achieve a very low power standard operating with a coin cell battery for several years.

Classic Bluetooth technology was developed as a wireless standard allowing to replace cables 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.

Bluetooth low energy devices consume only a fraction of the power of standard Bluetooth products and enable 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

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 Specification version 4.0 (on *Section 4: 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 for addressing two alternative implementations:

- Smart device
- Smart Ready device

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

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.

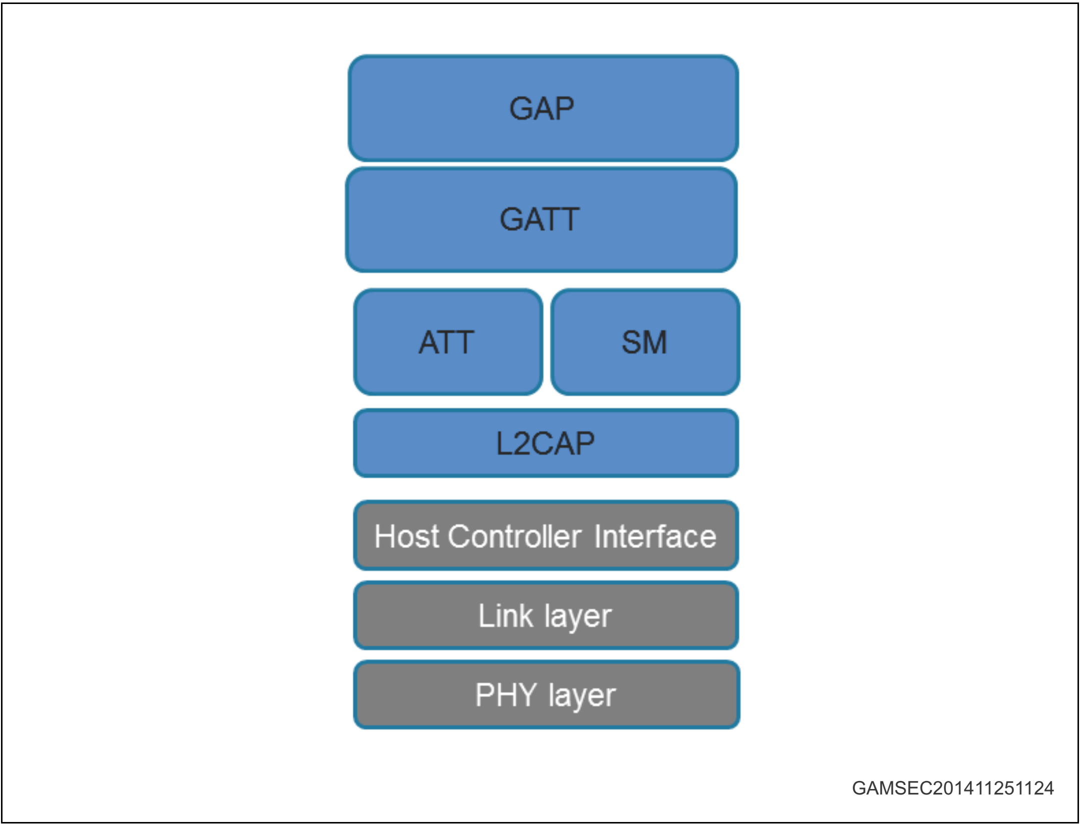
The Host includes the Logical Link Control and Adaptation Protocol (L2CAP), the Security Manager (SM), the Attribute protocol (ATT), Generic Attribute Profile (GATT) and the Generic Access Profile (GAP). The interface between the two components is called Host Controller Interface (HCI).

In addition, Bluetooth specification v4.1 have been released with new supported features:

- Multiple roles simultaneously support
- Support simultaneous advertising and scanning
- Support being Slave of up to two Masters simultaneously
- Privacy V1.1
- Low Duty Cycle Directed Advertising
- Connection parameters request procedure
- LE Ping
- 32 bits UUIDs
- L2CAP Connection Oriented Channels

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:

2402 + k * 2 MHz, where k = 0..39;

There are two channels 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 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

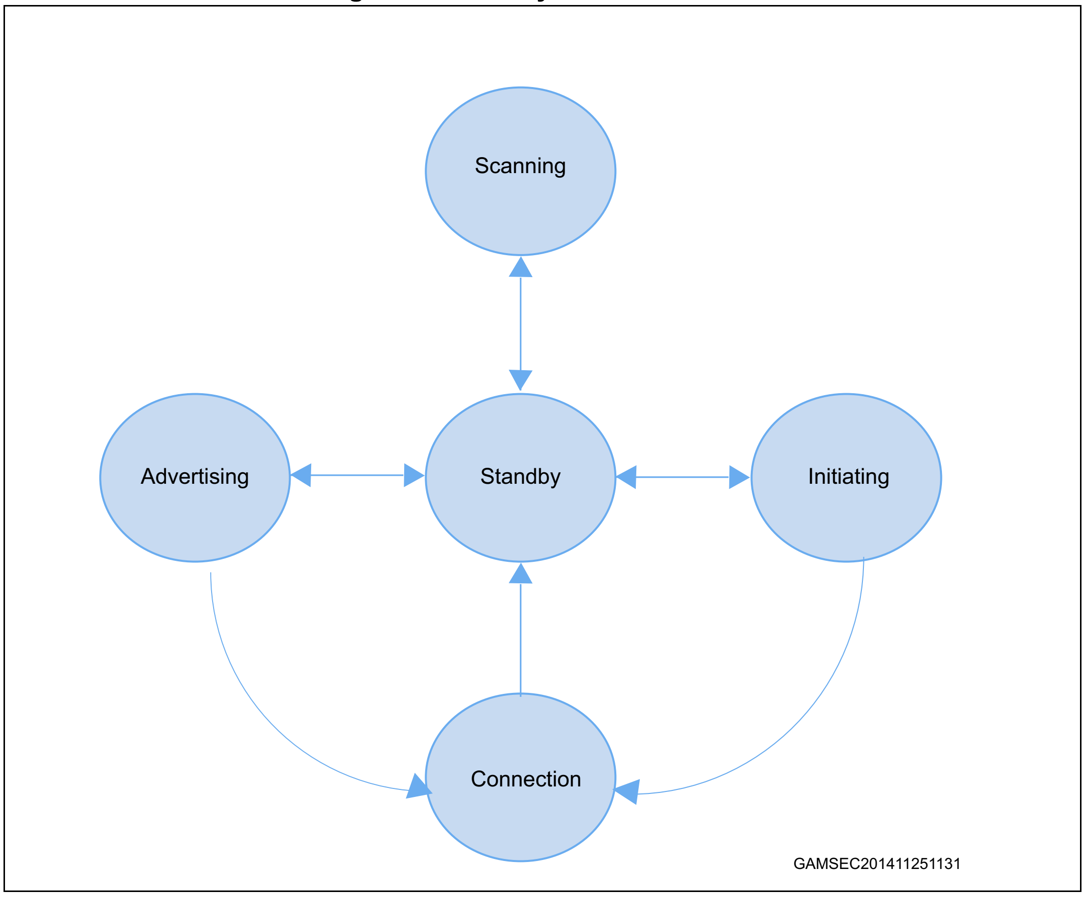
BLE is an Adaptive Frequency Hopping (AFH) technology that can use only a subset of all the available frequencies in order to avoid the frequencies used by other no-adaptive technologies. This allows to move from a bad channel to a known good channel by using a specific frequency hopping algorithm which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

1.3 Link Layer (LL)

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

The link layer defines a state machine with five states:

Figure 3. Link Layer state machine



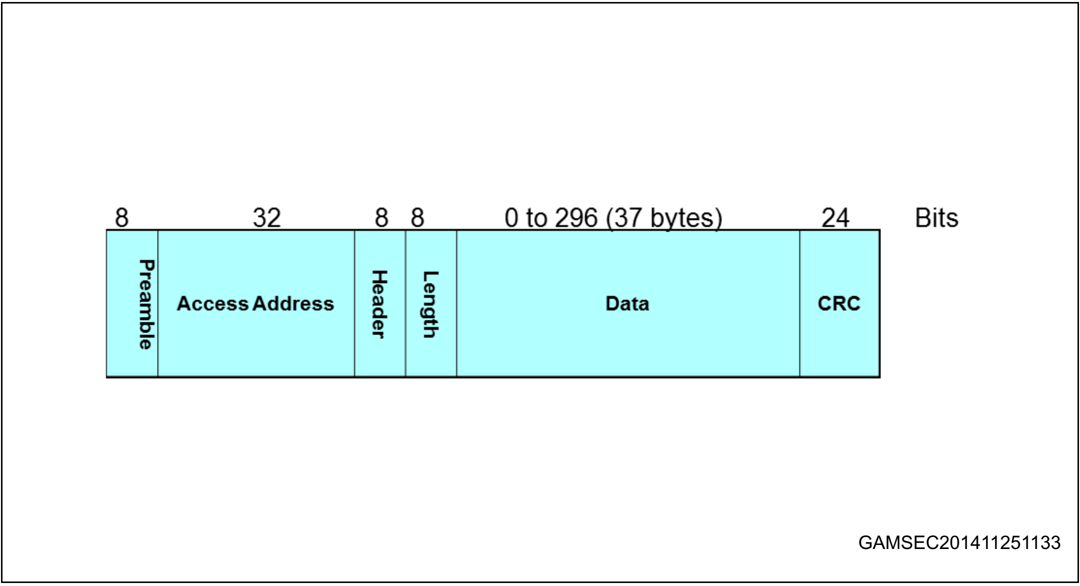
- Standby: the device does not transmit or receive packets
- Advertising: the device broadcasts advertisements in advertising channels (it is called an advertiser device)
- Scanning: device looks for advertisers devices (it is called a scanner device).
- Initiating: the device initiates connection to 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 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 physical layer channel)
- Header: its content depends on the packet type (advertising or data packet)
  a) Advertiser packet header:

Table 2. Advertising data header file 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. 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. 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 for performing packet acknowledgments. It informs the receiver device of the next sequence number that the transmitting device is expecting it to send. Packet is retransmitted until the NESN is different from the sequence number (SN) value in the sent packet.

More data bit is 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 & types refer to the Bluetooth specification v4.0 [Vol 2], on *Section 4: 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

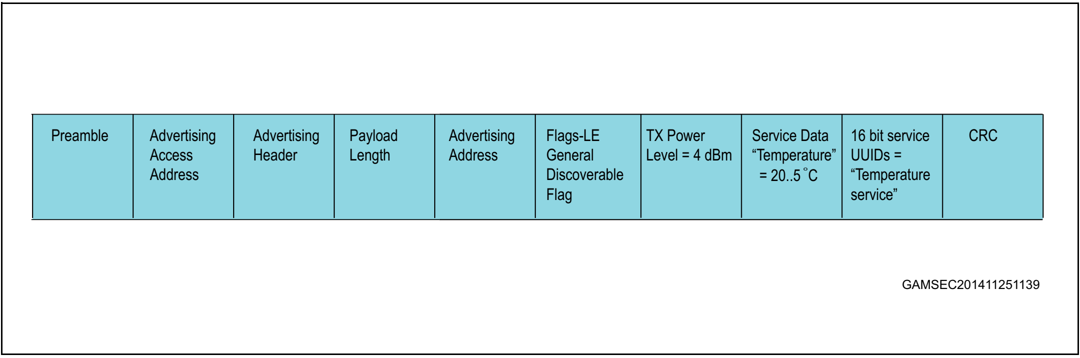
Advertising states allow Link Layer to transmit advertising packets and also to respond with scan responses to scan requests coming from devices which are actively scanning.

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

Each time a device advertises, it sends the same packet on each of the three advertising channels. This three packets sequence is called an 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.

Following is an example of advertising packet that lists the Service UUID that the device implements (General Discoverable flag, tx power = 4dbm, Service data = temperature service and 16 bits service UUIDs).

Figure 5. Advertising packet with AD type flags


Preamble
Advertising Access Address
Advertising Header
Payload Length
Advertising Address
Flags-LE General Discoverable Flag
TX Power Level = 4 dBm
Service Data “Temperature” = 20..5 °C
16 bit service UUIDs = “Temperature service”
CRC
GAMSEC201411251139

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 Same Device Capable (Controller) (bit 3);
- Simultaneous LE and BR/EDR to Same Device Capable (Host) (bit 4)

The Flags AD type shall be 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 devices in the white list
  - Process all scan/connection requests (default advertiser filter policy)
  - Process connection requests from all devices but only scan requests in the white list
  - Process scan requests from all devices but only connection requests in the white list

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

1.3.3 Scanning state

There are two types of scanning:

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

The following scan parameters can be set:

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

Once the scan parameters are set, it is possible to enable the device scanning. 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 the ones 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 for establishing and handling 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 (offset of transmit window start)
- Connection interval (time between two connection events)
- Slave latency (number of times slave can ignore connection events before it is forced to listen)
- Supervision timeout (max time between two correctly received packets before link is considered lost)
- Channel map: 37 bits (1= good; 0 = bad)
- Frequency-hop value (random number between 5 and 16).
- Sleep clock accuracy range (used to determine the uncertainty window of the slave device at connection event).

For a detailed description of the connection request packet refer to Bluetooth Specification V4.0 [Vol 6], Section 2.3.3.

The allowed timing ranges are summarised in *Table 7*:

Table 7. Connection request timings 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 wait 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 for allowing 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 specification, with new additional commands for low energy-specific functions.

1.5 Logical link control and adaptation layer protocol (L2CAP)

The Logical Link Control and Adaptation Layer Protocol (L2CAP), supports higher level protocol multiplexing, packet segmentation and reassembly operation, and the conveying of quality of service information.

1.6 Attribute Protocol (ATT)

The Attribute Protocol (ATT) allows a device to expose certain pieces of 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 bits 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 bits attribute UUIDs are defined by Bluetooth SIG;
- Attribute value: a (0 ~ 512) octets in length;
- Attribute permissions: they are defined by each higher 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 permissions 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).

Attribute example

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 methods protocol for discovering, reading and writing 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
  - Can indicate, notify an attribute value when data change
- Client role
  - Talk with server
  - Sends requests, wait for response (it can access (read), update (write) the data)
  - Can confirm 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*:

Table 9. Attributes protocol messages

Protocol Data Unit (PDU message)	Sent by	Description
Request	Client	Client requests something from 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 Cipher Block Chaining-Message Authentication Code (CCM) algorithm and a 128-bit AES block cipher. 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 to authenticate two devices and creates a common link key that can be used as a basis for a trusted relationship or a (single) secure connection.

Pairing procedure is a three-phase process.

Phase 1: pairing feature exchange

- The two connected devices communicates their input/output capabilities by using the Pairing request message. This message also contains a bit stating if out-of-band data is available and the authentication requirements.
- There are three input capabilities:
  a) no input;
  b) the ability to select yes/no;
  c) the 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

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

The information exchanged in Phase 1 is used to select which STK generation method is used 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 methods.
  a) The out-of-band (OOB) method which uses out of band communication (example: NFC) for the TK agreement (it is selected if the out-of-band bit is set);
  b) Passkey Entry method: user passes six numeric digits as the TK between the devices;
  c) Just Works: this method is not authenticated, and it does not provide any protection against man-in-the-middle (MITM) attacks.

The selection between PassKey and Just Works method is done based on the following table:

Table 11. Methods used for calculating 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

- 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): it is used to generate the 128-bit key used for Link Layer encryption and authentication;
  b) Connection signature resolving key (CSRK): it is used for the data signing performed at the ATT layer;
  c) Identity resolving key (IRK): it is used to generate a private address on the basis of a device public address.

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

Another security mechanism supported from BLE is the use of private addresses. A private address is generated by encrypting the public address of the device. This private address can be resolved by a trusted device that has been provided with the corresponding encryption key.This allows the device to use a private address for a more secure communication and to change it frequently (only devices with the related IRK are able to recognize it).

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 allows to provide protection 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 Generic attribute profile (GATT)

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

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

- GATT client: it is the device which accesses data on the remote GATT server via read, write, notify, or indicate operations.
- GATT server: it is the device which 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, it is not required   that a slave (peripheral) device has to be the GATT server and that a master (central) device has 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)

1.8.1 Characteristic attribute type

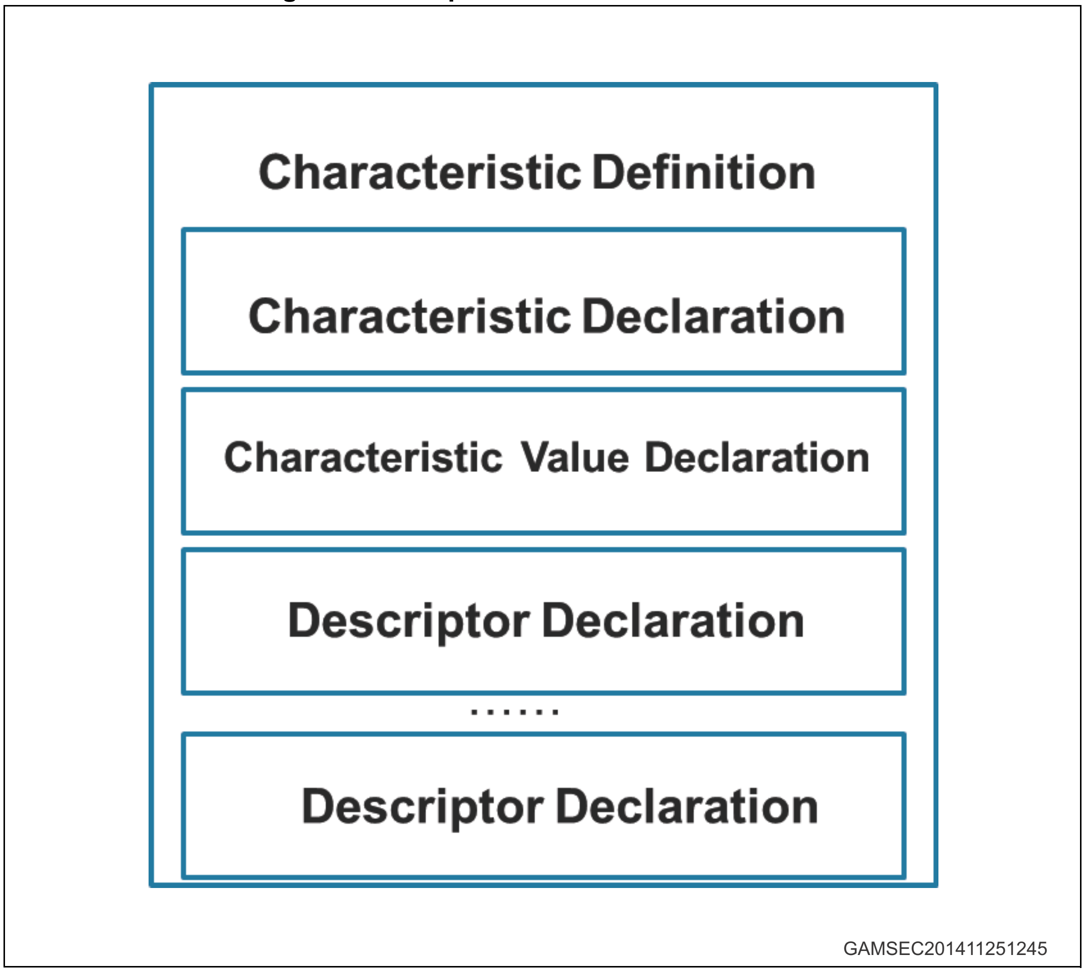
A characteristic is an attribute type which contains a single value and any number of descriptors describing the characteristic value that may make it understandable by the user.

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


Characteristic Definition
Characteristic Declaration
Characteristic Value Declaration
Descriptor Declaration
......
Descriptor Declaration
GAMSEC201411251245

A characteristic declaration is an attribute defined as follows:

Table 12. Characteristic declaration

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

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

Table 13. Characteristic value

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

1.8.2 Characteristic descriptors type

Characteristic descriptors are used to describe the characteristic value for adding 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 to add extended properties 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 for enabling 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 to define the characteristic value presentation format through some fields as format, exponent, unit namespace, description in order to correctly display the related value (example temperature measurement value in ^{o}C format);
6. Characteristic aggregation format: It allows to aggregate several characteristic presentation formats.

For a detailed description of the characteristic descriptors, refer to the Bluetooth specification v4.0.

1.8.3 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 between measurements. A service or primary service can refer other services that are called secondary services.

A service is defined as follows:

Table 14. Service declaration

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

A service shall contain a service declaration and may contain definitions and characteristic definitions. A service includes declaration follows the service declaration and any other attributes of the server.

Table 15. Include declaration

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

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

1.8.4 GATT procedures

The Generic Attribute Profile (GATT) defines a standard set of procedures allowing to discover services, characteristics, related descriptors and how to use them.

The following procedures are available:

- Discovery procedures (*Table 16*)
- Client-initiated procedures (*Table 17*)
- Server-initiated procedures (*Table 18*)

Table 16. Discovery procedures and related response events

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

Table 17. Client-initiated procedures and related response events

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

Table 17. Client-initiated procedures and related response events (continued)

Procedure	Response events
Write long characteristic value	Prepare write response Execute write response
Reliable write	Prepare write response Execute write response

Table 18. Server-initiated procedures and related response events

Procedure	Response events
Notifications	No event is generated
Indications	Confirmation event

For a detailed description about the GATT procedures and related responses events refer to the Bluetooth specification v4.0 on *Section 4: References*.

1.9 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 profiles roles are described in the table below:

Table 19. GAP roles[1]

Role	Description	Transmitter	Receiver	Typical example
Broadcaster	Sends advertising events	M	O	Temperature sensor which sends temperature values
Observer	Receives advertising events	O	M	Temperature display which just receives and display 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 modes: it configures a device to act in a specific way for a long time. There are four GAP modes types : broadcast, discoverable, connectable and bondable type.
- GAP procedures: it configures a device to perform a single action for a specific, limited time. There are four GAP procedures types: observer, discovery, connection, bonding procedures.

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

Table 20. GAP broadcaster mode

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

Table 21. GAP discoverable modes

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

Table 22. GAP connectable modes

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

Table 23. GAP bondable modes

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

The following GAP procedures are defined in *Table 24*:

Table 24. GAP observer procedure

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

Table 25. 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 Flags AD type information	Central
General discoverable procedure	It is used for discovery peripheral devices in general ad limited discovery mode	Device filtering is applied based on Flags AD type information	Central
Name discovery procedure	It is the procedure for retrieving the “Bluetooth Device Name” from connectable devices		Central

Table 26. GAP connection procedures

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

Table 27. GAP bonding procedures

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

For a detailed description of the GAP procedures, refer to the Bluetooth specification v4.0.

1.10 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

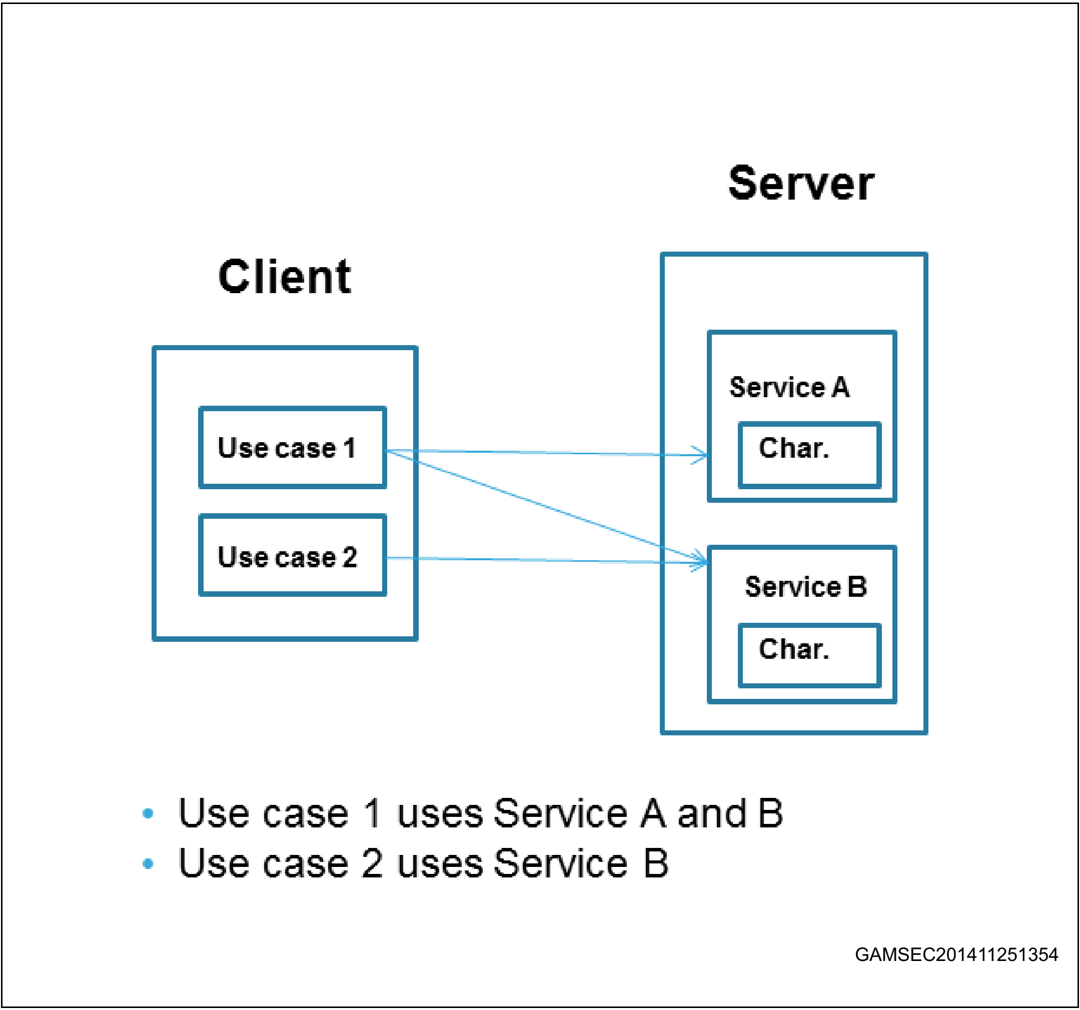
A profile may implement single or multiple services (available and specified at http://developer.bluetooth.org).

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

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

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

Figure 7. Client and server profiles



1.10.1 Proximity profile example

This section simply describes the Proximity Profile in terms its target, how it works and required services:

Target

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

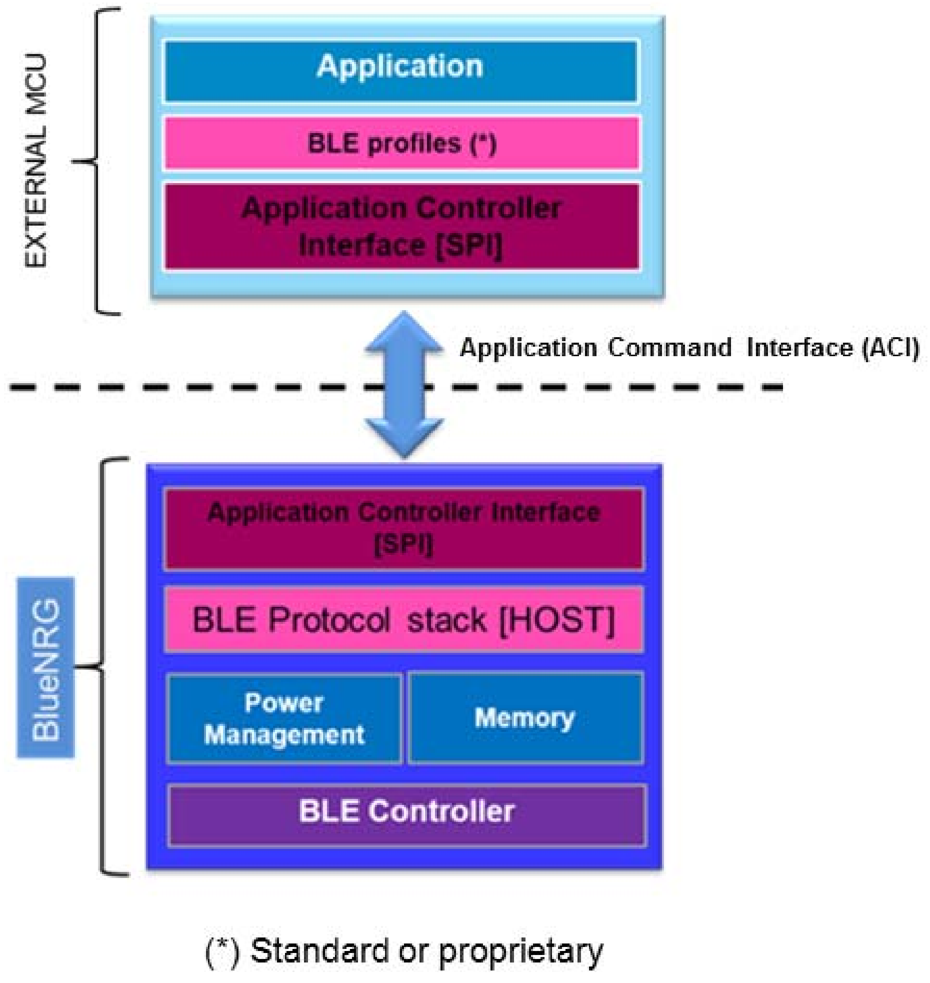
How it works

- if a device disconnects
- cause an alert
- alert on link loss: ≪Link Loss≫ service
  - if a device is too far away
  - cause an alert on path loss: ≪Immediate Alert≫ & ≪Tx Power≫ service
- ≪Link Loss≫ service
  - ≪Alert Level≫ characteristic
  - Behavior: on link loss, cause alert as enumerated
- ≪Immediate Alert≫ service
  - ≪Alert Level≫ characteristic
  - Behavior: when written, cause alert as enumerated
- ≪Tx Power≫ service
  - ≪Tx Power≫ characteristic
  - Behavior: when read, reports current Tx Power for connection

2 BlueNRG, BlueNRG-MS stacks architecture and ACI

The BlueNRG, BlueNRG-MS devices are network coprocessors which provide high-level interface to control its Bluetooth low energy functionalities. This interface is called ACI (application command interface).

Figure 8. BlueNRG, BlueNRG-MS stacks architecture and interface to the external host


EXTERNAL MCU
Application
BLE profiles (*)
Application Controller Interface [SPI]
Application Command Interface (ACI)
BlueNRG
Application Controller Interface [SPI]
BLE Protocol stack [HOST]
Power Management
Memory
BLE Controller
(*) Standard or proprietary
GAMSEC201411261359

BlueNRG and BlueNRG-MS devices embed, respectively, the Bluetooth Smart protocol stack v4.0 and v4.1 and, as a consequence, no BLE library is required on the external microcontroller, except for profiles and all the functions needed to communicate with the BlueNRG or BlueNRG-MS device SPI interface. The SPI interface communication protocol allows the external microcontroller to send ACI commands to control the BlueNRG or BlueNRG-MS device and to receive the ACI events generate from the BlueNRG or BlueNRG-MS device network coprocessor.

2.1 ACI interface

The ACI commands utilize and extend the standard HCI data format defined within the Bluetooth specification v4.0 and v4.1.

The ACI interface supports the following commands:

- Standard HCI commands for controller as defined by Bluetooth specification (v4.0 and v4.1)
- Vendor Specific (VS) HCI commands for controller
- Vendor Specific (VS) ACI commands for host (L2CAP,ATT, SM, GATT, GAP)

The reference ACI interface framework is provided within the BlueNRG, BlueNRG-MS kits software package targeting the BlueNRG, BlueNRG-MS kits based on STM32L1 external microcontroller (refer to *Section 4: References*).

The ACI interface framework contains the code that is used to send ACI commands to the BlueNRG and BlueNRG-MS network processors. It also provides definitions of device events. This framework allows to format each ACI command in the proper way and send the command inline with the defined ACI SPI communication protocol.

The ACI SPI communication protocol is described on the user manuals UM1755 “BlueNRG Bluetooth LE stack application command interface (ACI)” and UM1865 “BlueNRG-MS Bluetooth LE stack application command interface (ACI)", available on ST BlueNRG web pages. These user manuals also provide a complete description of all related devices ACI command formats, name parameters, return values and generated events.

The ACI framework interface is defined by the following header files:

Table 28. ACI Interface

File	Description	Location	Notes
hci.h	HCI library functions prototypes and error code definition.	Bluetooth LE\SimpleBlueNRG_HCI\includes	To be included on the user main application
hci_const.h	It contains constants and functions for HCI layer. See Bluetooth Core v 4.0, Vol. 2, Part E.	“”	
bluenrg_gatt_server.h	Header file for GATT server definition	“”	To be included on the user main application
sm.h	Header file for BlueNRG's security manager	“”	To be included on the user main application
bluenrg_gap.h	Header file for BlueNRG's GAP layer	“”	To be included on the user main application
bluenrg_aci.h	Header file that contains commands and events for BlueNRG FW stack	“”	To be included on the user main application
bluenrg_aci_const.h	Header file with ACI definitions for BlueNRG FW stack	“”	It is included by bluenrg_aci.h
bluenrg_hal_aci.h	Header file with HCI commands for BlueNRG FW stack	“”	It is included by bluenrg_aci.h

Table 28. ACI Interface (continued)

File	Description	Location	Notes
bluenrg_l2cap_aci.h	Header file with L2CAP commands for BlueNRG FW stack	“”	It is included by bluenrg_aci.h
bluenrg_gatt_aci.h	Header file with GATT commands for BlueNRG FW stack	“”	It is included by bluenrg_aci.h
bluenrg_gap_aci.h	Header file with GAP commands for BlueNRG FW stack	“”	It is included by bluenrg_aci.h
bluenrg_updater_aci.h	Header file with updater commands for BlueNRG FW stack	“”	It is included by bluenrg_aci.h

2.2 ACI Interface resources

In order to communicate with BlueNRG or BlueNRG-MS network processor through the ACI interface framework, the external microcontroller requires only the following main resources:

1. SPI interface
2. Platform-dependent code to write/read to/from SPI
3. A timer to handle SPI timeouts

The BlueNRG, BlueNRG-MS SPI interface is handled through the functions defined on files SDK_EVAL_SPI_Driver.[ch] and hal.[ch]. These APIs allows the external microcontroller to get access to BlueNRG or BlueNRG-MS device. The BlueNRG, BlueNRG-MS devices use the SPI IRQ pin to notify the external microcontroller (SPI master) when it has data to be read: this is handled through the HCI_Isr() placed within the SPI_IRQ_IRQHandler() handler on stm32l1xx_it.c. The SPI_IRQ_IRQHandler() is associated to the proper EXTI irq handler, based on the selected platform GPIO line for the BlueNRG, BlueNRG-MS SPI interrupt line.

The BlueNRG, BlueNRG-MS kits platforms are targeting the STM32L1xx microcontroller and the related libraries are used in order to get access to the device peripheral. The STM32L1xx microcontroller libraries are provided within the platform\STM32L1XX\Libraries\STM32L1xx_StdPeriph_Driver folder.

The files stm32l1xx_spi.[ch] are used for handling the low level platform-dependent code to write/read to/from SPI.

Table 29. ACI Interface resources files

File	Description	Location	Notes
SDK_EVAL_SPI_Driver.[ch]	Main APIs handling SPI communication with BlueNRG, BlueNRG-MS device	platform\STM32L1XX\Libraries\SDK_Eval_STM32L\src	These APIs are mapped to the specific microcontroller low level drivers handling the SPI peripheral (stm32l1xx_spi.c)
hal.[ch]	Other APIs handling communication with BlueNRG, BlueNRG-MS device	platform\STM32L1XX	
clock.[ch]	SPI timer APIs	platform\STM32L1XX	It provides the low level APIs handling the SPI timeouts
stm32l1xx_it.c	Main Interrupt Service Routines	It is defined within the specific user application folder	

When using another external microcontroller these files should be ported/adapted for addressing the ACI SPI communication.

In order to proper setup the ACI SPI interface, user is only requested to perform the following steps at initialization time, on main() function:

1. Init SPI interface by calling the following API:
   ```
   SdkEvalSpiInit(SPI_MODE_EXTI);
   ```
2. Reset the BlueNRG module by calling the following API:
   ```
   BlueNRG_RST();
   ```

The user is also requested to place HCI_Isr() within the SPI_IRQ_IRQHandler() handler on file stm32l1xx_it.c: this allows BlueNRG, BlueNRG-MS device to use the SPI IRQ pin to notify the external microcontroller (SPI master) when it has data to be read.

2.3 Other platforms resources files

The SW framework provides other files handling some platform-dependent resources as I/O communication channel (USB or UART), buttons, LEDs, EEPROM).

Table 30. SW framework platforms drivers

File	Description	Location	Notes
SDK_EVAL_Io.[ch]	Main APIs handling I/O communication (USB virtual COM or UART)	platform\STM32L1XX\Libraries\SDK_Eval_STM32L	These APIs are mapped to the specific microcontroller drivers handling USB virtual COM or UART.
SDK_EVAL_Buttons.[ch]	APIs handling platform buttons	platform\STM32L1XX\Libraries\SDK_Eval_STM32L	These APIs are mapped to the specific microcontroller drivers handling GPIOs
SDK_EVAL_Leds.[ch]	APIs handling platform LEDs	platform\STM32L1XX\Libraries\SDK_Eval_STM32L	These APIs are mapped to the specific microcontroller drivers handling GPIOs.
SDK_EVAL_Eeprom.[ch]	APIs handling EEPROM	platform\STM32L1XX\Libraries\SDK_Eval_STM32L	On BlueNRG, BlueNRG-MS kits, an external EEPROM is provided for storing platform manufacturing tests results.

These files should be ported/adapted to address another external microcontroller.

2.3.1 Platforms configuration

In order to easily support the BlueNRG, BlueNRG-MS kits platforms, the BlueNRG SW framework is designed for recognizing such platforms at runtime. User is only requested to call the SdkEvalIdentification() API at initialization time on main() function.

BlueNRG, BlueNRG-MS kits platforms can be also supported at compile time, by adding, respectively, only one of the following define on EWARM workspace preprocessor options:

`USER_DEFINED_PLATFORM=STEVAL_IDB002V1` (it is valid for both BlueNRG, BlueNRG development platforms).

`USER_DEFINED_PLATFORM=STEVAL_IDB003V1` (it is valid for both BlueNRG, BlueNRG USB dongles).

The following define values allow to select, at compile time, the specific platforms header files provided within the platform\STM32L1XX\Libraries\SDK_Eval_STM32L\inc folder:

```
#if USER_DEFINED_PLATFORM == STEVAL_IDB002V1
#include "USER_Platform_Configuration_STEVAL_IDB002V1.h"
#elif USER_DEFINED_PLATFORM == STEVAL_IDB003V1
#include "USER_Platform_Configuration_STEVAL_IDB003V1.h"
#endif
```

A user platform can be simply supported, at compile time, by following these steps:

1. Create a file "`USER_Platform_Configuration.h`" with specific user platform configuration: `USER_Platform_Configuration_STEVAL_IDB002V1.h` or `USER_Platform_Configuration_STEVAL_IDB003V1.h` can be used as reference (to be extended based on available user platform resources).
2. Place the "`USER_Platform_Configuration.h`" on the STM32L\platform\STM32L1XX\Libraries\SDK_Eval_STM32L\inc folder.
3. On the selected EWARM workspace preprocessor options, add this define: `USER_DEFINED_PLATFORM=USER_EVAL_PLATFORM`.

If no user platform is defined at compile time, through the related preprocessor option, USER_DEFINED_PLATFORM is automatically set to STEVAL_IDB00xV1. This allows to include the file USER_Platform_Configuration_auto.h which contains the BlueNRG, BlueNRG-MS kits platforms define values used during the runtime auto configuration procedure performed from SdkEvalIdentification() function. This header file must not be modified by user.

2.4 How to port the ACI SPI interface framework to a selected microcontroller

BlueNRG, BlueNRG-MS devices are network coprocessors providing the Bluetooth low energy features. In order to get access to its functionality, an external microcontroller can be used by implementing the ACI SPI interface framework previously described. BlueNRG, BlueNRG-MS development kits software package provides a reference framework targeting this ACI SPI interface. This framework can be ported to another external microcontroller by following these steps:

1. Define a specific "USER_Platform_Configuration.h" with specific user platform SPI configuration
2. On the selected user application preprocessor options, add this define: `USER_DEFINED_PLATFORM=USER_EVAL_PLATFORM`
3. Replace the STM32L1xx libraries on folder platform\STM32L1XX\Libraries\STM32L1xx_StdPeriph_Driver folder with the specific microcontroller low level drivers
4. Replace the CMSIS Cortex-M3 files and the startup file (file startup_stm32l1xx_md.s) accordingly to the selected microcontroller
5. Readapt/port accordingly to the selected microcontroller the file system_stm32l1xx.c handling the system clock configuration for STM32L1xx
6. Adapt/port the files described in the section in order to refer to the selected external microcontroller low level drivers.

Readapt/port accordingly to the selected microcontroller the stm32l1xx_it.c (make sure that HCI_Isr() is called within the SPI irq API handling the external IRQ interrupt on the IRQ line) .

Once the ACI SPI interface framework has been ported to the selected microcontroller, user can verify that SPI access from external microcontroller is working by performing the basic test described in the section “SPI Interface” of the application note AN4494 “Bringing up the BlueNRG, BlueNRG-MS”, available on ST BlueNRG and BlueNRG-MS web pages.

3 Design an application using BlueNRG, BlueNRG-MS ACI APIs

This section provides information and code examples about how to design and implement a Bluetooth low energy application on the selected microcontroller.

User implementing a BLE host application on the selected MCU has to go through some basic and common steps:

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

The STM32L1xx microcontroller is the reference external microcontroller used for the programming guidelines described on the following sections, since the available BlueNRG, BlueNRG-MS kits platforms are based on such microcontroller.

Note: On the following sections, some user application "Defines" are used to simply identify the devices Bluetooth low energy role (central, peripheral, client and server).

Further on each provided pseudo codes any reference to BlueNRG device is also valid for the BlueNRG-MS device. Any specific difference is highlighted whenever it is needed by using #ifdef BLUENRG_MS.

Table 31. User application defines for BLE devices role

Define	Description
GAP_CENTRAL	GAP central role
GAP_PERIPHERAL	GAP peripheral role
GATT_CLIENT	GATT client role
GATT_SERVER	GATT server role

3.1 Initialization phase and main application loop

The following main steps are required for properly configure the selected external microcontroller and the SPI communication with a BlueNRG or BlueNRG-MS device.

1. Configure selected BlueNRG platform
2. Configure clock, GPIOs and setup low power mode
3. Initialize the serial communication channel used for I/O communication (debug and utility information)
4. Initialize list heads of ready and free hci data packet queues
5. Init SPI interface for allowing external microcontroller to get access to the BlueNRG features properly
6. Reset the BlueNRG, BlueNRG-MS network coprocessor
7. Configure BlueNRG, BlueNRG-MS public address (if public address is used)
8. Init BLE NRG GATT layer
9. Init BLE NRG GAP layer depending on the selected device role
10. Set the proper security I/O capability and authentication requirement (if BLE NRG security is used)
11. Define the required Services & Characteristics if the device is a GATT server
12. Add a while(1) loop calling the HCI_Process() API and a specific user application function where user actions/events are processed (advertising, connections, services and characteristics discovery, notification and related events).

The following pseudocode example illustrates the required initialization steps:

```
int main(void)
{
  int ret;
  /* Setup the STM32L1xx NVIC vector table base address */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, VECTOR_TABLE_BASE_ADDRESS);
  /* Configure selected BlueNRG platform */
  SdkEvalIdentification();
  /* Configure clock */
  RCC_Configuration();
  /* Init I/O ports */
  Init_GPIOs ();
  PWR_PVDCmd(DISABLE);
  /* Disable FLASH during Sleep  */
  FLASH_SLEEPPowerDownCmd(ENABLE);
  /* Enable Ultra low power mode */
  PWR_UltraLowPowerCmd(ENABLE);
  PWR_FastWakeUpCmd(DISABLE);
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  /* Init timer used for SPI timeouts */
  Clock_Init();
  /* Configure I/O communication channel:
     It requires the void IO_Receive_Data(uint8_t * rx_data, uint16_t
data_size)function where user received data should be processed */
```

```
  SdkEval_IO_Config(processInputData);
  /* Initialize list heads of ready and free hci data packet queues */
  HCI_Init();
  /* Init SPI interface */
  SdkEvalSpiInit(SPI_MODE_EXTI);
  /* Reset the BlueNRG network coprocessor */
  BlueNRG_RST();
  /* Configure BlueNRG address as public (its public address is used) */
  {
     uint8_t bdaddr[] = {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
     ret = aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,
CONFIG_DATA_PUBADDR_LEN,bdaddr);
     if(ret)PRINTF("Setting BD_ADDR failed.\n");
  }
  /*  Init BlueNRG GATT layer */
  ret = aci_gatt_init();
  if(ret) PRINTF("GATT_Init failed.\n");
  /*  Init BlueNRG GAP layer as peripheral or central */
  {
    uint16_t service_handle, dev_name_char_handle, appearance_char_handle;
#if GAP_PERIPHERAL
     uint8_t role = GAP_PERIPHERAL_ROLE;
#else
     uint8_t role = GAP_CENTRAL_ROLE;
#endif
#if BLUENRG_MS
     ret = aci_gap_init(role, 0, 0x07, &service_handle,
&dev_name_char_handle, &appearance_char_handle);
#else
     ret = aci_gap_init(role, &service_handle, &dev_name_char_handle,
&appearance_char_handle);
#endif
if(ret) PRINTF("GAP_Init failed.\n");
  }
 /**** If security is used, set the I/O capability and authentication
requirement: refer to  Section*/
……
 #if  GATT_SERVER
  /* User application function where service and characteristics are
defined: refer to Section Services & Characteristics Configuration Section
*/
  ret = Add_Server_Services_Characteristics();
  if(ret == BLE_STATUS_SUCCESS)
      PRINTF("Services & Characteristics added successfully.\n");
  else
      PRINTF("Error while adding Services & Characteristics.\n");
```

```
#endif

  /* Main Application Loop */
  while(1)
  {
      /*  Process any pending HCI events read */
      HCI_Process()
    /* User specific application function where user actions and events are
processed (advertising, connections, services and  characteristics
discovery, notification)
     */
     User_Process();
  }
} /* end main() */
```

Note: *1. User_Process() is just an application dependent function. On the following sections, some reference specific actions/events are described based on the most common BLE functionalities. User developer can adapt/modify/replace them.*

2. When performing the GATT_Init() & GAP_Init() APIs, BlueNRG and BlueNRG-MS stacks always add two standard services: Attribute Profile Service (0x1801) with Service Changed Characteristic and GAP Service (0x1800) with Device Name and Appearance characteristics.

3. The last attribute handle reserved for the standard GAP service is 0x000F on BlueNRG stack and 0x000B on BlueNRG-MS stack.

4. The GAP_Init() role parameter values are as follow:

Table 32. GAP_Init() role parameter values

Device	Role parameter values	Note
BlueNRG	0x01:Peripheral 0x03: Central	Broadcaster, Observer are not supported on BlueNRG device
BlueNRG-MS	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)

Further, on BlueNRG-MS stack, two new parameters are available on GAP_Init() API:

– enable_Privacy: 0x00 for disabling privacy; 0x01 for enabling privacy;
– device_name_char_len: it allows to indicate the length of the device name characteristic.

For a complete description of this API and related parameters refer to the UM1755 and UM1865 User Manuals, on the *Section 4: References*.

3.1.1 BLE addresses

The following device addresses are supported from BlueNRG and BlueNRG-MS devices:

- Public address
- Random address
- Private address

Public MAC addresses (6 bytes- 48 bits address) uniquely identifies a BLE device, and they are defined by 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 to reserve a block of possible public addresses (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).

BlueNRG and BlueNRG-MS devices don't have a valid preassigned MAC address since the MAC address is specific to manufacturers. The public address must be set by the external processor.

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

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

The command ACI_HAL_WRITE_CONFIG_DATA should be sent to BlueNRG and BlueNRG-MS devices by the uC before starting BLE operations (after each power-up or reset of BlueNRG).

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")}
```

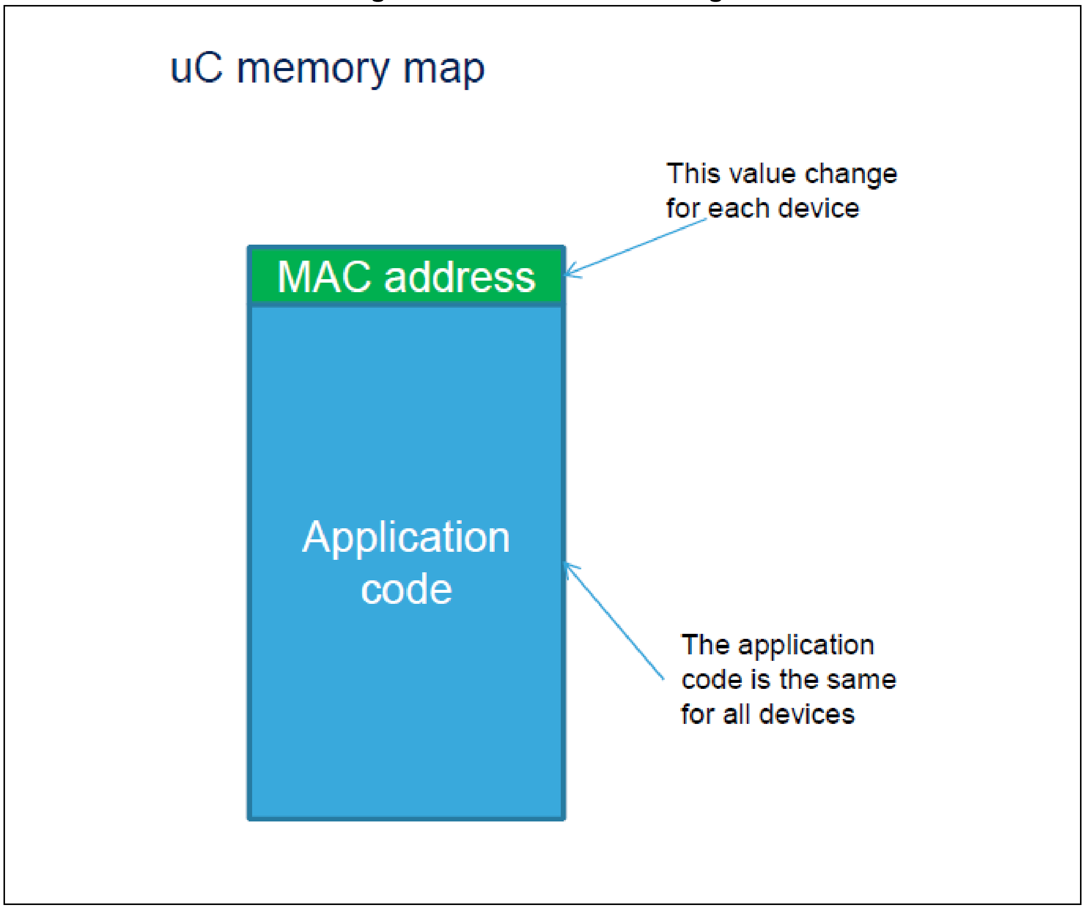
MAC address needs to be stored somewhere in the non-volatile memory associated to the product during product manufacturing.

A user can write its application assuming that the MAC address is placed at a known Flash location of the microcontroller. 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 location in a free 48 bits area of the Flash.

When the microcontroller's application needs to access the MAC address simply refers to the known Flash memory location.

Figure 9. MAC address storage


uC memory map
This value change for each device
MAC address
Application code
The application code is the same for all devices

Alternatively, the MAC address can be stored in a free area of the BlueNRG and BlueNRG-MS devices Information register (IFR) region, but this does not offer any advantage, since:

- Programming of the MAC in the IFR requires several SPI transaction as follows:
  - Make the device entering Updater mode (1 SPI transaction)
  - Program the device IFR with MAC address (1 SPI transaction)
  - Make the device leaving Updater mode (1 SPI transaction)
- Access to the MAC address during device initialization (at each power-up or reset) requires several SPI transaction as follows:
  - Make the device entering Updater mode (1 SPI transaction)
  - Read MAC address from the device IFR (1 SPI transaction)
  - Make the device leaving Updater mode (1 SPI transaction)

BLE standard can also use "random" addresses which are defined by users, and they do not follow the public addresses rules. The random addresses are handled autonomously by the device, are set at each reset but they can also be overwritten by the external processor using the `hci_le_set_random_address()` API.

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 UM1755 and UM1865 user manuals, on the *Section 4: References*.

3.2 BlueNRG, BlueNRG-MS events and events Callback

Whenever there is an ACI event to be processed, the ACI framework notifies this event to the user application through the `HCI_Event_CB()` callback. The `HCI_Event_CB()` callback is called within the `HCI_Process()` on file hci.c.

As a consequence, user application is requested to :

1. Define the `void HCI_Event_CB(void *pckt)` function within his main application (`pckt` is a pointer to the received ACI packet)
2. Based on its own application scenario, the user has to identify the required device events to be detected and handled and the application specific actions to be done as consequence of such events.

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

Table 33. ACI: main events, sub-events

Event/sub-event	Description	Main event	Where
`EVT_DISCONN_COMPLETE`	A connection is terminated	NA	GAP central/ peripheral
`EVT_LE_CONN_COMPLETE`	Indicates to both of the Hosts forming the connection that a new connection has been established	`EVT_LE_META_EVENT`	GAP central/ peripheral
`EVT_BLUE_GATT_ATTRIBUTE_MODIFIED`	Generated by the GATT server when a client modifies any attribute on the server, if event is enabled.	`EVT_VENDOR`	GATT server
`EVT_BLUE_GATT_NOTIFICATION`	Generated by the GATT client when a server notifies any attribute on the client	`EVT_VENDOR`	GATT client
`EVT_BLUE_GATT_INDICATION`	Generated by the GATT client when a server indicates any attribute on the client	`EVT_VENDOR`	GATT client
`EVT_BLUE_GAP_PASS_KEY_REQUEST`	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_response () API	`EVT_VENDOR`	GAP central/ peripheral
`EVT_BLUE_GAP_PAIRING_CMPLT`	Generated when the pairing process has completed successfully or a pairing procedure timeout has occurred or the pairing has failed	`EVT_VENDOR`	GAP central/ peripheral

Table 33. ACI: main events, sub-events (continued)

Event/sub-event	Description	Main event	Where
`EVT_BLUE_GAP_BOND_LOST`	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	`EVT_VENDOR`	GAP peripheral
`EVT_BLUE_ATT_READ_BY_GROUP_RESP`	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	`EVT_VENDOR`	GATT client
`EVT_BLUE_ATT_READ_BY_TYPE_RESP`	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.	`EVT_VENDOR`	GATT client

`EVT_BLUE_GAP_DEVICE_FOUND`
`(only for BlueNRG device)`
	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.	`EVT_VENDOR`	GAP central

`EVT_BLUE_GATT_PROCEDURE_COMPLETE`	A GATT procedure has been completed	`EVT_VENDOR`	GATT client
EVT_LE_ADVERTISING_REPORT (only for BlueNRG-MS device)	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.	`EVT_LE_META_EVENT`	GAP central

For a detailed description about the BLE events, and related formats refer to the user manual UM1755 and on *Table 34: ACI: GAP modes APIs* in the current document.

The following pseudocode provides an example of `HCI_Event_CB()` callback handling some of the described device events (`EVT_DISCONN_COMPLETE`, `EVT_LE_CONN_COMPLETE`, `EVT_BLUE_GATT_ATTRIBUTE_MODIFIED`, `EVT_BLUE_GATT_NOTIFICATION`):

```
void HCI_Event_CB(void *pckt)
{
   hci_uart_pckt *hci_pckt = pckt;
   hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;
   if(hci_pckt->type != HCI_EVENT_PKT return;
   switch(event_pckt->evt){
   case EVT_DISCONN_COMPLETE: /* BlueNRG disconnection event */
   {
      /* Add user code for handling BLE disconnect event based on 
application scenarios
      */
.........
   }
break;
case EVT_LE_META_EVENT:
{
   /* Get the meta event data */
   evt_le_meta_event *evt = (void *)event_pckt->data;
   /* Analyze the specific sub event */
   switch(evt->subevent){
   case EVT_LE_CONN_COMPLETE:/* BlueNRG connection event */
   {
      /* connection complete event: get the related data */
      evt_le_connection_complete *cc = (void *)evt->data;
      /* Connection parameters:
      cc->status: connection status (0x00: Connection successfully 
completed);
      cc->handle: connection handle to be used for the communication during 
the  connection;
      cc->role: BLE device role (0x01: master; 0x02: slave);
      cc->peer_bdaddr_type: connected device address type (0x00: public; 
0x01:  random);
      cc->peer_bdaddr: connected device address;
      cc->interval: connection interval;
      cc->latency: connection latency;
      cc->supervision_timeout: connection supervision timeout;
      cc->master_clock_accuracy: master clock accuracy;
   */
      /* Add user code for handling connection event based on application 
scenarios */
      conn_handle = cc->handle;
.....
      } /* EVT_LE_CONN_COMPLETE */
      break;
   } /* switch(evt->subevent) */
} /* EVT_LE_META_EVENT */
break;
case EVT_VENDOR:
{
   /* Get the vendor event data */
   evt_blue_aci *blue_evt = (void*)event_pckt->data;
   switch(blue_evt->ecode){
#if GATT_SERVER
   case EVT_BLUE_GATT_ATTRIBUTE_MODIFIED:
   {
   /* Get attribute modification event data */
      evt_gatt_attr_modified *evt = (evt_gatt_attr_modified*)blue_evt-
>data;
      evt->conn_handle: the connection handle which modified the attribute;
      evt->attr_handle: handle of the attribute that was modified;
      evt->data_length: the length of the data;
      evt->att_data: pointer to the new value (length is data_length).
      /* Add user code for handling attribute modification event based on
application scenarios */
....
   }/* EVT_BLUE_GATT_ATTRIBUTE_MODIFIED */
   break;
#endif /* GATT_SERVER */
#if  GATT_CLIENT
      case EVT_BLUE_GATT_NOTIFICATION:
{
   /* Get attribute notification event data */
      evt_gatt_attr_notification *evt =
(evt_gatt_attr_notification*)blue_evt->data;
      evt->conn_handle: the connection handle which notified the attribute;
      evt->event_data_length: length of attribute value + handle (2 bytes);
      evt->attr_handle: attribute handle;
      evt->attr_value: pointer to attribute value (length is
event_data_length - 2).
      /* Add user code for handling attribute notification event based on
application scenarios */
.....
   }/* EVT_BLUE_GATT_NOTIFICATION */
   break;
   break;
#endif /* GATT_CLIENT */
      }/* switch(blue_evt->ecode) */
   }/* EVT_VENDOR */
   break;
   }/* switch(evt->subevent)*/
}/* end HCI_Event_CB() */
```

3.3 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 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 & Characteristics 16 bits UUID (refer to Bluetooth SIG web page: https://www.bluetooth.org/en-us/specification/adopted-specifications).
2. Proprietary, non-standard profile. The user must define its own services and characteristics. In this case, 128-bits UIDS are required and must be generated by profile implementers (refer to UUID generator web page: http://www.famkruithof.net/uuid/uuidgen)

A service can be added using the following command:

```
- aci_gatt_add_serv (Service_UUID_Type, Service_UUID_16, Service_Type,
Max_Attributes_Records, &ServHandle);
```

This command returns the pointer to the Service Handle (ServHandle), which is used to identify the service within the user application. A characteristic can be added to this service using this command:

```
- aci_gatt_add_char (ServHandle, Char_UUID_Type, Char_UUID_16,
Char_Value_Length, Char_Properties, Security_Permissions, GATT_Evt_Mask,
Enc_Key_Size, Is_Variable, &CharHandle);
```

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_serv() and aci_gatt_add_char() functions parameters refer to the user manuals UM1755 and UM1865.

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

```
tBleStatus Add_Server_Services_Characteristics(void)
{
   tBleStatus ret;
   /*
   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 service_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};
/* Add the service with service_uuid  128bits UUID to the GATT server
database. The service handle ServHandle is returned
*/
ret = aci_gatt_add_serv(UUID_TYPE_128, service_uuid, PRIMARY_SERVICE, 7,
&ServHandle);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}

/* Add the characteristic with charUuid_1 128bits UUID to the service
ServHandle.
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(ServHandle, UUID_TYPE_128, charUuid_1, 20,
CHAR_PROP_NOTIFY, ATTR_PERMISSION_NONE, 0,16, 1, &CharHandle_1);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}

/* Add the characteristic with charUuid_2 128bits UUID to the service
ServHandle.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(ServHandle, UUID_TYPE_128, charUuid_2, 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) PRINTF("Failure.\n")}
   return ret ;
}/* end Add_Server_Services_Characteristics() */
```

3.4 Create a connection: discoverable and connectable APIs

In order to establish a connection between a BlueNRG GAP central (master) device and a BlueNRG GAP peripheral (slave) device, the GAP discoverable/connectable modes and procedures can be used as described in *Table 34: ACI: GAP modes APIs*, *Table 35: ACI: discovery procedures APIs*, *Table 36: ACI: connection procedures APIs* and by following the related ACI APIs described in the user manuals UM1755 and UM1865, *Section 4: References*.

GAP peripheral discoverable and connectable modes APIs

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

Table 34. ACI: GAP modes APIs

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

aci_gap_set_limited_discoverable( )	0x00: connectable undirected advertising (default);
	0x02: scannable undirected advertising;
	0x03: non-connectable undirected advertising.
		Sets the device in limited discoverable mode. The device is discoverable for a maximum period of TGAP (lim_adv_timeout) = 180 seconds. The advertising can be disabled at any time by calling `aci_gap_set_non_discoverable()` API

aci_gap_set_non_discoverable()	NA	Sets the device in non-discoverable mode. This command disables the LL advertising and sets the device in standby state.
aci_gap_set_direct_connectable()	NA	Sets the device in direct connectable mode. The device is directed connectable mode only for 1.28 seconds. If no connection is established within this duration, the device enters non-discoverable mode and advertising has to be enabled again explicitly.

Table 34. ACI: GAP modes 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 35. ACI: discovery procedures APIs

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

Table 36. ACI: connection procedures APIs

ACI API	Description
aci_gap_start_auto_conn_establishment()	Starts the auto connection establishment procedure. The devices specified are added to the white list of the controller and a LE_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 LE_Create_Connection call will be 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_conn_establishment()	Starts a general connection establishment procedure. The host 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 EVT_BLUE_GAP_DEVICE_FOUND on BlueNRG device and EVT_LE_ADVERTISING_REPORT on BlueNRG-MS device.

Table 36. ACI: connection procedures APIs (continued)

ACI API	Description
aci_gap_start_selective_conn_establishment()	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 whitelist”. All the devices found are sent to the upper layer by the event EVT_BLUE_GAP_DEVICE_FOUND on BlueNRG device and EVT_LE_ADVERTISING_REPORT on BlueNRG-MS device.
aci_gap_terminate_gap_procedure()	Terminate the specified GAP procedure.

3.4.1 Set discoverable mode & 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: *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 char local_name[]=
{AD_TYPE_COMPLETE_LOCAL_NAME,'B','l','u','e','N','R','G','_','T','e','s','
t'};

/* disable scan response: passive scan */

hci_le_set_scan_resp_data(0,NULL);

/* Put the GAP peripheral in general discoverable mode:

    Advertising_Event_Type: ADV_IND (undirected scannable and connectable);

    Adv_Interval_Min: 0;

    Adv_Interval_Max: 0;

    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: BlueNRG_Test;

      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, 0, 0, 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 )

{

       tBleStatus ret;

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

       /* Start the direct connection establishment procedure to the GAP
peripheral device in general discoverable mode using the following
connection parameters:

       Scan_Interval: 0x4000;

       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);

        Conn_Len_Min: 2000 (Minimum length of connection needed for the LE
connection);

        Conn_Len_Max: 2000 (Maximum length of connection needed for the LE
connection).

*/

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")}

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

Note: *1. If `ret = BLE_STATUS_SUCCESS` is returned, on termination of the GAP procedure, a `EVT_LE_CONN_COMPLETE` event is returned, on the `HCI_Event_CB()` event callback, 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 command `aci_gap_terminate_gap_procedure()`.

3. The last two parameters Conn_Len_Min and Conn_Len_Max 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 choosen.

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.

3.4.2 Set discoverable mode & 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.

```
/* GAP Peripheral:  general discoverable mode (scan responses are 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_PUBADDR_LEN,
                                    bdaddr);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
*/
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
0,0x0c,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_resp_data(18 , ServiceUUID_Scan);

/* Put the GAP peripheral in general discoverable mode:
   Advertising_Event_Type: ADV_IND (undirected scannable and connectable);
   Adv_Interval_Min: 0;
   Adv_Interval_Max: 0;
   Address_Type: PUBLIC_ADDR (public address: 0x00);
   Adv_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, 0, 0, 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:
   Scan_Interval: 0x4000;
   Scan_Window: 0x4000;
   Own_address_type: 0x00 (public device address);
   filterDuplicates: 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 `EVT_BLUE_GAP_DEVICE_FOUND` (BlueNRG) and `EVT_LE_ADVERTISING_REPORT` (BlueNRG-MS) events raised on `HCI_Event_CB()` callback (`EVT_VENDOR` as main event) . The end of the procedure is indicated by `EVT_BLUE_GAP_PROCEDURE_COMPLETE` event on the HCI_Event_CB() callback (`EVT_VENDOR` as main event):

```
void HCI_Event_CB(void *pckt)
{
   hci_uart_pckt *hci_pckt = pckt;
   hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;
   if(hci_pckt->type != HCI_EVENT_PKT return;
   switch(event_pckt->evt){
   case EVT_VENDOR:
{
/* Get the vendor event data */
evt_blue_aci *blue_evt = (void*)event_pckt->data;
switch(blue_evt->ecode){
case EVT_BLUE_GAP_DEVICE_FOUND:
{
evt_gap_device_found *pr = (void*)blue_evt->data;
/* evt_gap_device_found parameters:
   pr->evt_type: event type (advertising packets types);
   pr->bdaddr_type: type of the peer address (PUBLIC_ADDR,RANDOM_ADDR);
   pr->bdaddr: address of the peer device found during scanning;
   pr->length: length of advertising or scan response data;
   pr->data_RSSI[]:  length advertising or scan response data + RSSI.
RSSI is last octect (signed integer).
*/
/* Add user code for decoding the evt_gap_device_found event data based on
the specific pr->evt_type  (ADV_IND, SCAN_RSP, ..)*/
………
}/* EVT_BLUE_GAP_DEVICE_FOUND */
break;
case EVT_BLUE_GAP_PROCEDURE_COMPLETE:
{
   /* When the general discovery procedure is terminated
   EVT_BLUE_GAP_PROCEDURE_COMPLETE event is returned with the procedure
code set to GAP_GENERAL_DISCOVERY_PROC (0x02).
*/
   evt_gap_procedure_complete *pr = (void*)blue_evt->data;
   /* evt_gap_procedure_complete parameters:
   pr->procedure_code: terminated procedure code;
   pr->status: BLE_STATUS_SUCCESS, BLE_STATUS_FAILED or ERR_AUTH_FAILURE;
   pr->data[VARIABLE_SIZE]: procedure specific data, if applicable
   */
/* If needed, add user code for handling  the event data */
.....
         }/* EVT_BLUE_GAP_PROCEDURE_COMPLETE */
         break;
      }/* switch(blue_evt->ecode) */
   }/* EVT_VENDOR */
   break;

case EVT_LE_META_EVENT:
{
  evt_le_meta_event *evt = (void *)event_pckt->data;
  switch(evt->subevent)
  {
    case EVT_LE_ADVERTISING_REPORT: /* BlueNRG-MS stack */
    {
      le_advertising_info *pr = (void *)(evt->data+1); /* evt->data[0] is
number of reports (On BlueNRG-MS is always 1) */
    /* le_advertising_info parameters:
       pr->evt_type: event type (advertising packets types);
       pr->bdaddr_type: type of the peer address (PUBLIC_ADDR,RANDOM_ADDR);
       pr->bdaddr: address of the peer device found during scanning;
       pr->length: length of advertising or scan response data;
       pr->data_RSSI[]: length advertising or scan response data + RSSI.
       RSSI is last octect (signed integer).
     */
    /* Add user code for decoding the le_advertising_info event data based
on the specific pr->evt_type (ADV_IND, SCAN_RSP, ..)*/
       ...
    }/* EVT_LE_ADVERTISING_REPORT */
    break;
  }/* end switch() */
}/* EVT_LE_META_EVENT */
break;
  }/* switch(event_pckt->evt)*/
}/* end HCI_Event_CB() */
```

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

1. `EVT_BLUE_GAP_DEVICE_FOUND`(BlueNRG)/`EVT_LE_ADVERTISING_REPORT`(BlueNRG-MS) with advertising packet type (`evt_type = ADV_IND`)
2. `EVT_BLUE_GAP_DEVICE_FOUND`(BlueNRG device)/`EVT_LE_ADVERTISING_REPORT` (BlueNRG-MS) with scan response packet type (`evt_type = SCAN_RSP`)

Table 37. ADV_IND event

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

The advertising data can be interpreted as follows (refer to Bluetooth specification version 4.0 [Vol 3] and 4.1 [Vol 2] on *Section 4: References*):

Table 38. ADV_IND advertising data

Flags AD type field	Local name field	Tx power level
0x02: length of the field 0x01: AD type Flags 0x06: 0x110 (Bit 2: BR/EDR Not Supported; Bit 1: general discoverable mode)	0x08: length of the field 0x08: Shortened local name type 0x42,0x6C,0x75,0x65,0x4E0x52,0x47: BlueNRG	0x02: Length of the field 0x0A: Tx Power type 0x08: power value

Table 39. SCAN_RSP event

Event type	Address type	Address	Scan response data	RSSI
0x04 (SCAN_RSP)	0x00 (public 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 specification version 4.0 [Vol 3] and 4.1 [Vol 2]):

Table 40. Scan response data

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

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 coming next reconnections).

When using the security features, some low level parameters (root keys) must be set, before raising any other ACI commands:

- DIV root key used to derive CSRK
- Encryption root (ER) key used to derive LTK and CSRK
- Identity root (IR) key used to derive IRK and CSRK

The external microcontroller (MCU) has the responsibilities to provide these parameters as follows:

1. It has to randomly generate the three root key values (if not already generated) and store them in a non-volatile-memory. The three root keys have to be generated only one time, in order to univocally establish the specific BlueNRG, BlueNRG-MS device security settings.
2. Each time the MCU starts, it has to read the root keys from the non-volatile-memory and set them during the initialization phase of the BlueNRG, BlueNRG-MS device.

Following is a simple pseudo code showing how to set the read randomly generated security root keys on BlueNRG, BlueNRG-MS devices:

```
uint8_t DIV[2];
uint8_t ER[16];
uint8_t IR[16];

/* Reset BlueNRG, BlueNRG-MS device */
BlueNRG_RST();

/* Microcontroller specific implementation:
   1) MCU has to  randomly generate DIV, ER and IR and store them in
      a non volatile memory.
   2) When MCU starts it has to read DIV, ER and IR values from the
      non volatile memory.
*/
.........

/* Configure read root key DIV on BlueNRG, BlueNRG-MS device  */
ret = aci_hal_write_config_data(CONFIG_DATA_DIV_OFFSET,
CONFIG_DATA_DIV_LEN,(uint8_t *) DIV);

/* Configure read root key ER on BlueNRG, BlueNRG-MS device */
ret = aci_hal_write_config_data(CONFIG_DATA_ER_OFFSET,
CONFIG_DATA_ER_LEN,(uint8_t *) ER);

/* Configure read root key IR on BlueNRG, BlueNRG-MS device */
```

```
ret = aci_hal_write_config_data(CONFIG_DATA_IR_OFFSET,
CONFIG_DATA_IR_LEN,(uint8_t *) IR);
```

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` value:

```
0x00: Display Only
0x01: Display yes/no
0x02: Keyboard Only
0x03: No Input, no output
0x04: 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 *Section Table 11.: Methods used for calculating 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_auth_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_auth_requirement(MITM_PROTECTION_REQUIRED,
                                   OOB_AUTH_DATA_ABSENT, /* no OOB data is
                                  present */
                                   NULL, /* no OOB data */
                                   7,  /* Min. encryption key size */
                                   16, /* Max encryption key size */
                                DONOT_USE_FIXED_PIN_FOR_PAIRING,/* no fixed
                                pin */
                                0, /* fixed pin not used */
                                BONDING /* bonding is enabled */);
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 follow:

1. by using `aci_gap_slave_security_request()` on a GAP Peripheral (slave) device (it sends a slave security request to the master):

```
tBleStatus ret;
ret = aci_gap_slave_security_request(conn_handle,
                                     BONDING,
                                     MITM_PROTECTION_REQUIRED
);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
```

or

2. by using the `aci_gap_send_pairing_request()` on a GAP Central (master) device.

Since the `DONOT_USE_FIXED_PIN_FOR_PAIRING` (no fixed pin) has been set, once the paring procedure is initiated by one of the 2 devices, BlueNRG, BlueNRG-MS will generate the `EVT_BLUE_GAP_PASS_KEY_REQUEST` event (with related connection handle) for asking to the host application to provide the password to be used for establishing the encryption key. BlueNRG, BlueNRG-MS application has to provide the correct password by using the `aci_gap_pass_key_response(conn_handle,passkey)API.`

The following pseudocode example illustrates only the specific steps to be followed for providing the pass key (for example a random pin) to be used for the pairing process, when the `EVT_BLUE_GAP_PASS_KEY_REQUEST` event is generated on Device_1 ("Display Only" capability) :

```
tBleStatus ret;

/* Generate a random pin with an user specific function */

pin = generate_random_pin();

ret = aci_gap_slave_security_request(conn_handle,
                                     BONDING,
                                     MITM_PROTECTION_REQUIRED
                                     );

if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
```

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

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

```
tBleStatus ret;

ret = aci_gap_set_auth_requirement(MITM_PROTECTION_REQUIRED,
                                   OOB_AUTH_DATA_ABSENT,NULL,
                                   7,
                                   16,
                                   USE_FIXED_PIN_FOR_PAIRING,
                                   0x123456,/* Fixed pin */
                                   BONDING
                                   );
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
```

NOTEs:

1. When the pairing procedure is started by calling the described APIs (`aci_gap_slave_security_request() or aci_gap_send_pairing_request()`)and the value `ret = BLE_STATUS_SUCCESS` is returned, on termination of the procedure, a EVT_BLUE_GAP_PAIRING_CMPLT event is returned on the `HCI_Event_CB()` event callback to indicate the pairing status:

- 0x00: Pairing success;
- 0x01: Pairing Timeout;
- 0x02: Pairing Failed.

The pairing status is given from the status field of the `evt_gap_pairing_cmplt` data associated to the `EVT_BLUE_GAP_PAIRING_CMPLT` event.

2. When 2 devices get paired, the link is automatically encrypted during the first connection. If bonding is also enabled (keys are stored for a future time), when the 2 devices get connected again, the link can be simply encrypted (without no need to perform again the pairing procedure). Host applications can simply use the same APIs which will not perform the paring process but will just encrypt the link:

- `aci_gap_slave_security_request()` on the GAP Peripheral (slave) device

or

- `aci_gap_send_pairing_request()` 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 receives the `EVT_BLUE_GAP_BOND_LOST` event to inform the host 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.

4. Alternatively, the out-of-band method can be selected by using the aci_gap_set_auth_requirement() with OOB_Enable field enabled and the OOB data

specified in OOB_Data. This implies that both devices are using this method and they are setting the same OOB data defined through an out of band communication (example: NFC).

3.6 Service and characteristic discovery

This section describes the main functions allowing a BlueNRG, BlueNRG-MS GAP central device to discover the GAP peripheral services & 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 41. BlueNRG sensor profile demo services & characteristics handles

Service	Characteristic	Service/characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
Acceleration service	NA	0x0010	NA	NA	NA
	Free Fall characteristic	0x0011	0x0012	0x0013	NA
	Acceleration characteristic	0x0014	0x0015	0x0016	NA
Environmental service	NA	0x0017	NA	NA	NA
	Temperature characteristic	0x0018	0x0019	NA	0x001A

Table 42. BlueNRG-MS sensor profile demo services & characteristics 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

Table 42. BlueNRG-MS sensor profile demo services & characteristics handles (continued)

Service	Characteristic	Service / characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
	Acceleration characteristic	0x0010	0x0011	0x0012	NA
Environmenta l service	NA	0x0013	NA	NA	NA
	Temperature Characteristic	0x0014	0xx0015	NA	0x0016

Note: *The different attribute value handles are due to the last attribute handle reserved for the standard GAP service (0x000F on BlueNRG stack and 0x000B on BlueNRG-MS stack).*

For detailed information about the sensor profile demo, refer to the user manual UM1686 and the sensor demo source code available within the development kit software package (see *References*). On the following example, the BlueNRG GAP peripheral sensor profile demo environmental service is defining only the temperature characteristic (no expansion board with pressure and humidity sensors is used).

3.6.1 Service discovery procedures and related GATT events

Following is a list of the service discovery APIs with related description.

Table 43. ACI: service discovery procedures APIs

Discovery Service API	Description
aci_gatt_disc_all_prim_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_ prim_services_by_service_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_prim_services()` API:

```
/* GAP Central starts a discovery all services procedure: conn_handle is the
connection handle returned on HCI_Event_CB() event callback,
EVT_LE_CONN_COMPLETE event */

if (aci_gatt_disc_all_prim_services(conn_handle) != BLE_STATUS_SUCCESS)

{

  if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}

}
```

The responses of the procedure are given through the `EVT_BLUE_ATT_READ_BY_GROUP_RESP` event raised on `HCI_Event_CB()` callback (`EVT_VENDOR` as main event) . The end of the procedure is indicated by `EVT_BLUE_GATT_PROCEDURE_COMPLETE` event on the `HCI_Event_CB()` callback (`EVT_VENDOR` as main event):

```
void HCI_Event_CB(void *pckt)

{
  hci_uart_pckt *hci_pckt = pckt;
  hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;
  if(hci_pckt->type != HCI_EVENT_PKT return;
  switch(event_pckt->evt){
    case EVT_VENDOR:
    {
      /* Get the vendor event data */
      evt_blue_aci *blue_evt = (void*)event_pckt->data;
      switch(blue_evt->ecode){
      case EVT_BLUE_ATT_READ_BY_GROUP_RESP:
     {
    evt_att_read_by_group_resp *pr = (void*)blue_evt->data;
   /* evt_att_read_by_group_resp parameters:
      pr->conn_handle: connection handle;
      pr->event_data_length: total length of the event data;
      pr->attribute_data_length: length of each specific data within the
      attribute_data_list[];
      pr->attribute_data_list[]: event data.
 */
 /* Add user code for decoding the pr->attribute_data_list[] and getting
the services handle, end group handle and service uuid */
 ..........
 }/* EVT_BLUE_ATT_READ_BY_GROUP_RESP */
 break;
 case EVT_BLUE_GATT_PROCEDURE_COMPLETE:
 {
   evt_gatt_procedure_complete *pr = (void*)blue_evt->data;
    /* evt_gatt_procedure_complete parameters:
     pr->conn_handle: connection handle;
     pr->attribute_data_length: length of the event data;
     pr->data[]: event data.
 */
  /* If needed, add user code for using the event data */
   .......
   }/* EVT_BLUE_GATT_PROCEDURE_COMPLETE */
       break;
   }/* switch(blue_evt->ecode) */
   }/* EVT_VENDOR */
   break;
  }/* switch(evt->subevent)*/
}/* end HCI_Event_CB() */
```

In the context of the Sensor Profile Demo, the GAP Central application should get three `EVT_BLUE_ATT_READ_BY_GROUP_RESP` events, with following `evt_att_read_by_group_resp` data:

First `evt_att_read_by_group_resp` event data

```
      pr->conn_handle : 0x0801 (connection handle);

      pr->event_data_length: 0x0D (length of the event data);

      pr->handle_value_pair_length: 0x06 length of each discovered service
data: service handle, end group handle, service uuid);
```

```
pr-> attribute_data_list: 0x0C bytes as follows:
```

Table 44. First evt_att_read_by_group_resp event data

Service Handle	End Group Handle	Service UUID	Note
0x0001	0x0004	0x1801	Attribute profile service (GATT_Init() adds it). Standard 16 bits service UUID.
0x0005	0x000F (BlueNRG), 0x000B (BlueNRG-MS)	0x1800	GAP profile service (GAP_Init() adds it). Standard 16 bits service UUID.

Second `evt_att_read_by_group_resp` event data:

```
pr->conn_handle : 0x0801 (connection handle);

pr->event_data_length: 0x15 (length of the event data);

pr->attribute_data_length: 0x14 length of each discovered service
data: service handle, end group handle, service uuid);

pr-> attribute_data_list content: 0x14 bytes as follows
```

Table 45. Second evt_att_read_by_group_resp event data

Service Handle	End Group Handle	Service UUID	Note
0x0010 (BlueNRG), 0x000C (BlueNRG-MS)	0x0016 (BlueNRG), 0x0012 (BlueNRG-MS)	0x02366E80CF3A11E19AB40002A5D5 C51B	Acceleration service 128 bits service proprietary UUID

Third `evt_att_read_by_group_resp` event data:

```
pr->conn_handle : 0x0801 (connection handle);

pr->event_data_length: 0x15 (length of the event data);

pr->attribute_data_length: 0x14 length of each discovered service
data: service  handle, end group handle, service uuid);

pr-> attribute_data_list: 0x14 bytes as follows
```

Table 46. Third evt_att_read_by_group_resp event data

Service handle	End group handle	Service UUID	Note
0x0017 (BlueNRG), 0x0013 (BlueNRG-MS)	0x001A (BlueNRG), 0x0016 (BlueNRG-MS)	0x42821A40E47711E282D00002A5D5 C51B	Environmental service 128bits service proprietary UUID

In the context of the Sensor Profile Demo, when the discovery all primary service procedure completes, the `EVT_BLUE_GATT_PROCEDURE_COMPLETE` is generated on GAP Central application, with following `evt_gatt_procedure_complete` data:

```
pr->conn_handle : 0x0801 (connection handle);

pr-> data_length: 0x01 (length of the event data);

pr->data[]: 0x00 (event data).
```

3.6.2 Characteristics discovery procedures and related GATT events

Following is a list of the characteristic discovery APIs with associated description.

Table 47. BlueNRG ACI: characteristics discovery procedures APIs

Discovery service API	Description
aci_gatt_disc_all_charac_of_serv()	This API starts the GATT procedure to discover all the characteristics of a given service.
aci_gatt_discovery_characteristic_by_uuid()	This API starts the GATT the procedure to discover all the characteristics specified by a UUID.
aci_gatt_disc_all_charac_descriptors()	This API starts the procedure to discover all characteristic descriptors on the GATT server.

In the context of the BlueNRG sensor profile demo, follow a simple pseudocode illustrating how a GAP Central application can discover all the characteristics of the Acceleration service (refer to *Section Table 45.: Second evt_att_read_by_group_resp event data*):

```
uint16_t service_handle;
uint16_t end_group_handle;

#ifdef BLUENRG_MS

service_handle = 0x000C;
end_group_handle = 0x0012;

#else /*
service_handle_value  = 0x0010;
charac_handle_value = 0x0016;

#endif

/* BlueNRG GAP Central starts a discovery all the characteristics of a
service procedure: conn_handle is the connection handle returned on
HCI_Event_CB() event callback, EVT_LE_CONN_COMPLETE event */
if (aci_gatt_disc_all_charac_of_serv(conn_handle,
                                  service_handle,/*  Service handle */
                                  end_group_handle/*  End group handle */
                                  );) != BLE_STATUS_SUCCESS)
{
  if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
}
```

The responses of the procedure are given through the `EVT_BLUE_ATT_READ_BY_TYPE_RESP` event raised on `HCI_Event_CB()` callback (`EVT_VENDOR` as main event) . The end of the procedure is indicated by `EVT_BLUE_GATT_PROCEDURE_COMPLETE` event on the `HCI_Event_CB()` callback (`EVT_VENDOR` as main event):

```
void HCI_Event_CB(void *pckt)
```

```
{
  hci_uart_pckt *hci_pckt = pckt;
  hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;
  if(hci_pckt->type != HCI_EVENT_PKT return;
  switch(event_pckt->evt){
    case EVT_VENDOR:
    {
      /* Get the vendor event data */
      evt_blue_aci *blue_evt = (void*)event_pckt->data;
      switch(blue_evt->ecode){
      case EVT_BLUE_ATT_READ_BY_TYPE_RESP:
       {
         evt_att_read_by_type_resp *pr = (void*)blue_evt->data;
         /* evt_att_read_by_type_resp parameters:
            pr->conn_handle: connection handle;
            pr->event_data_length: total length of the event data;
            pr->handle_value_pair_length: length of each specific data
         within the handle_value_pair[];
            pr->handle_value_pair[]: event data.
        */
        /* Add user code for decoding the pr->handle_value_pair[] and get
       the characteristic handle, properties, characteristic value handle,
       characteristic UUID */
        ………
       }/* EVT_BLUE_ATT_READ_BY_TYPE_RESP */
       break;
       case EVT_BLUE_GATT_PROCEDURE_COMPLETE:
       {
         evt_gatt_procedure_complete *pr = (void*)blue_evt->data;
         /* evt_gatt_procedure_complete parameters:
            pr->conn_handle: connection handle;
            pr->data_length: length of the event data;
```

```
                pr->data[]: event data.

              */

              /* If needed, add user code for using the event data */

              ………

            }/* EVT_BLUE_GATT_PROCEDURE_COMPLETE */

            break;

          }/* switch(blue_evt->ecode) */

        }/* EVT_VENDOR */

        break;

      }/* switch(evt->subevent)*/

    }/* end HCI_Event_CB() */
```

In the context of the BlueNRG Senor Profile Demo, the GAP Central application should get two `EVT_BLUE_ATT_READ_BY_TYPE_RESP` events with following `evt_att_read_by_type_resp` data:

First `evt_att_read_by_type_resp` event data

```
pr->conn_handle : 0x0801 (connection handle);

pr->event_data_length: 0x16 (length of the event data);

pr->handle_value_pair_length: 0x15 length of each discovered
characteristic data: characteristic handle, properties,
characteristic value handle, characteristic UUID;

pr->handle_value_pair: 0x15 bytes as follows:
```

Table 48. First evt_att_read_by_type_resp event data

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x0011 (BlueNRG), 0x000D (BlueNRG-MS)	0x10 (notify)	0x0012 (BlueNRG), 0x000E (BlueNRG-MS)	0xE23E78A0CF4A11E18FFC0002A5D5 C51B	Free Fall characteristic 128 bits characteristic proprietary UUID

Second `evt_att_read_by_type_resp` **event data**

```
pr->conn_handle : 0x0801 (connection handle);

pr->event_data_length: 0x16 (length of the event data);
```

```
pr->handle_value_pair_length: 0x15 length of each discovered
characteristic data: characteristic handle, properties, characteristic
value handle, characteristic UUID;

pr->handle_value_pair: 0x15 bytes as follows:
```

Table 49. Second evt_att_read_by_type_resp event data

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x0014 (BlueNRG), 0x0010 (BlueNRG-MS)	0x12 (notify and read)	0x0015 (BlueNRG), 0x0011 (BlueNRG-MS)	0x340A1B80CF4B11E1AC360002A5D5C51B	Acceleration characteristic 128bits characteristic proprietary UUID

In the context of the Sensor Profile Demo, when the discovery all characteristics of a service procedure completes, the `EVT_BLUE_GATT_PROCEDURE_COMPLETE` is generated on GAP Central application, with following `evt_gatt_procedure_complete` data:

```
pr->conn_handle : 0x0801 (connection handle);
pr-> data_length: 0x01 (length of the event data);
pr->data[]: 0x00 (event data).
```

Similar steps can be followed in order to discover all the characteristics of the environment service (refer to *Table 41: BlueNRG sensor profile demo services & characteristics handles*) and *Table 42: BlueNRG-MS sensor profile demo services & characteristics handles*).

3.7 Characteristic notification/indications, write, read

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

Table 50. 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_charac_val()	It starts the procedure to read the attribute value.	GATT client
aci_gatt_write_charac_value()	It starts the procedure to write the attribute value (when the procedure is completed, a EVT_BLUE_GATT_PROCEDURE_COMPLETE event is generated).	GATT client
aci_gatt_write_without_response()	It starts the procedure to write a characteristic value without waiting for any response from the server.	GATT client

Table 50. Characteristics update, read, write APIs

Discovery service API	Description	Where
aci_gatt_write_charac_descriptor()	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 the event EVT_BLUE_GATT_INDICATION on the reception of a characteristic indication.	GATT client

In the context of the sensor profile demo, follow a simple pseudo code the GAP Central application should use in order to configure the free fall and the acceleration characteristics client descriptors configuration for notification:

```
tBleStatus ret;

uint16_t handle_value;

#ifdef BLUENRG_MS

handle_value = 0x000F;

#else /*

handle_value = 0x0013;

#endif

/* Enable the free fall characteristic client descriptor configuration for
ret = aci_gatt_write_charac_descriptor(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")}
#ifdef BLUENRG_MS

handle_value = 0x0012;

#else /*

handle_value = 0x0016;
```

```
#endif
/* Enable the acceleration characteristic client descriptor configuration
for notification */
ret = aci_gatt_write_charac_descriptor (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_value;
uint16_t charac_handle_value;

#ifdef BLUENRG_MS

service_handle_value = 0x000C;
charac_handle_value = 0x000D;

#else /*
service_handle_value  = 0x0010;
charac_handle_value = 0x0011;

#endif
/* GAP peripheral notifies free fall characteristic to GAP central*/
ret = aci_gatt_update_char_value (service_handle_value , /* acceleration
service handle */
                          charac_handle_value, /* free fall characteristic
                                           handle */
                                  0, /* characteristic
                                        value offset */
                                  0x01,/* characteristic value
                                          length*/
                                  &val, /* characteristic value */ );
ret = (accServHandle, freeFallCharHandle, 0, 1, &val);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}

tBleStatus ret;
```

```
uint8_t buff[6];
#ifdef BLUENRG_MS

charac_handle_value = 0x0010;

#else /*

charac_handle_value = 0x0014;

#endif
/* 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_value , /* acceleration
                                             service handle */
                                  charac_handle_value, /* acceleration
                                             characteristic
                                             handle */
                                  0 , /* characteristic
                                            value offset */
                                  0x06, /* characteristic
                                         value length */
                                  buff, /* characteristic
                                          value */
                               );
ret = (accServHandle, freeFallCharHandle, 0, 1, &val);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n")}
```

On GAP Central, `HCI_Event_CB()` callback (`EVT_VENDOR` as main event), the `EVT_BLUE_GATT_NOTIFICATION` is raised on reception of the characteristic notification (acceleration or free fall) from the GAP Peripheral device. Follow a pseudo code of the `HCI_Event_CB()` callback:

```
void HCI_Event_CB(void *pckt)
{
  hci_uart_pckt *hci_pckt = pckt;
  hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;
  if(hci_pckt->type != HCI_EVENT_PKT return;
  switch(event_pckt->evt){
    case EVT_VENDOR:
    {
      /* Get the vendor event data */
      evt_blue_aci *blue_evt = (void*)event_pckt->data;
      switch(blue_evt->ecode)
      {
        case EVT_BLUE_GATT_NOTIFICATION:
        {
```

```
  evt_gatt_attr_notification *evt = (evt_gatt_attr_notification*)blue_evt-
>data;
   /*
     evt_gatt_attr_notification data:
     evt->conn_handle: connection handle;
     evt->event_data_length: length of attribute value + handle (2 bytes);
     evt->attr_handle: handle of the notified characteristic;
     evt->attr_value[]: characteristic value.
    Add user code for handling the received notification based on the
    application scenario.
   */
     ........
        }
        break;
     }/* switch(blue_evt->ecode)*/
   }/* switch(evt->subevent)*/
}/* end HCI_Event_CB() */
```

3.8 Basic/typical error conditions description

On BlueNRG, BlueNRG-MS ACI framework, the `tBleStatus` type is defined in order to return the BlueNRG, BlueNRG-MS stack error conditions. The status and 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 detailed list of error conditions associated to each ACI API refer to the UM1755 and UM1865 user manuals, on *Section 4: References*.

3.9 BlueNRG-MS simultaneously Master, Slave scenario

BlueNRG-MS device 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 on Stack Mode 3), and to act as a Slave on another connection.

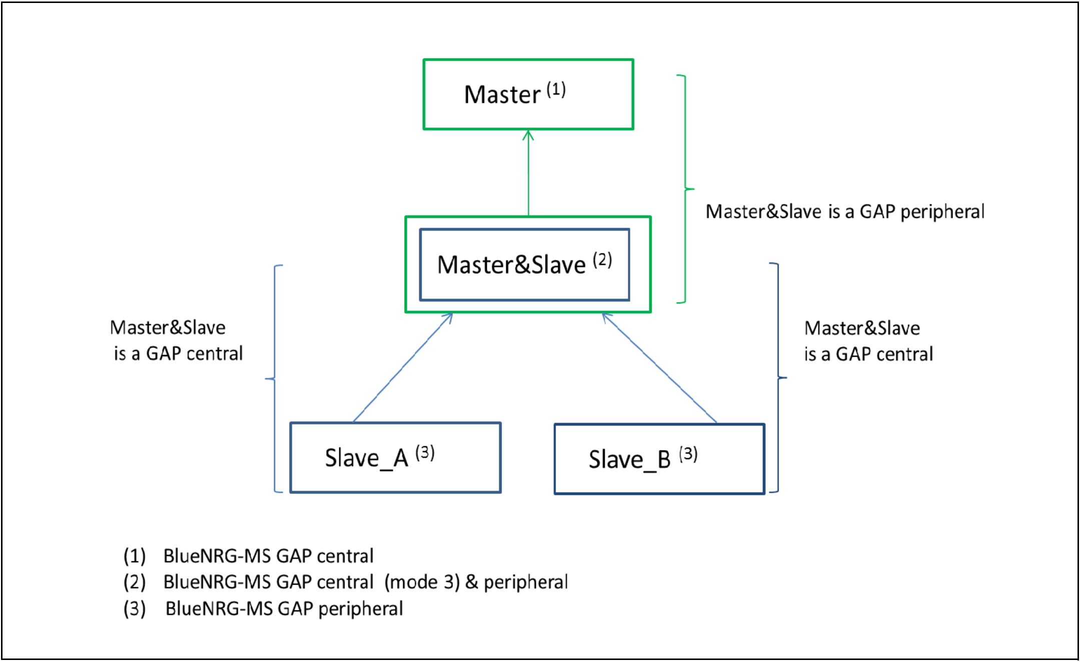
The following pseudo code describes how a BlueNRG-MS device can be initialized for supporting Central and Peripheral roles simultaneously:

```
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 easily targeted as follows:

Figure 10. BlueNRG-MS simultaneous Master & Slave scenario


Master (1)
Master&Slave is a GAP peripheral
Master&Slave (2)
Master&Slave is a GAP central
Master&Slave is a GAP central
Slave_A (3)
Slave_B (3)
(1) BlueNRG-MS GAP central
(2) BlueNRG-MS GAP central (mode 3) & peripheral
(3) BlueNRG-MS GAP peripheral

1. One BlueNRG-MS device (called Master&Slave) is configured as Central & Peripheral by setting role as `GAP_PERIPHERAL_ROLE | GAP_CENTRAL_ROLE` on `GAP_Init()` API. Further it is configured with stack mode 3 for being able to connect to more than one Peripheral device. Let's also assume that this device define a service with a characteristic.
2. Two BlueNRG-MS devices (called Slave_A, Slave_B) are configured as Peripheral by setting role as `GAP_PERIPHERAL_ROLE` on `GAP_Init()` API. Both Slave_A and Slave_B define the same service and characteristic as Master&Slave device.
3. One BlueNRG-MS device (called Master) is configured as Central by setting role as `GAP_CENTRAL_ROLE` on `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,
                              Local_Name_Length=0x05,
                              Local_Name=[0x08,0x74,0x65,0x73,0x74],
                              Slave_Conn_Interval_Min = 0x0006,
                              Slave_Conn_Interval_Max = 0x0008)
```

5. Master&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)
```

The two devices are discovered thorugh the EVT_LE_ADVERTISING_REPORT events.

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 addreess type and address have been found
during the discovery procedure within the EVT_LE_ADVERTISING_REPORT event
*/
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",
                               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 addreess type and address have been found
during the discovery procedure within the EVT_LE_ADVERTISING_REPORT event
*/
   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",
                               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_charac_descriptor()` API. Slave_A and Slave_B devices start the characterisitic 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,
                               Local_Name_Length=0x05,
                               Local_Name=[0x08,0x74,0x65,0x73,0x74],
                               Slave_Conn_Interval_Min = 0x0006,
                                Slave_Conn_Interval_Max = 0x0008)
```

Since Master&Slave device is also acting as a Central device, it receives the EVT_BLUE_GATT_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 BlueNRG-MS 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)
```

Master&Slave device is discovered thorugh the EVT_LE_ADVERTISING_REPORT event.

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 addreess
type and address have been found during the discovery procedure within the
EVT_LE_ADVERTISING_REPORT event */
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"
                                Conn_Interval_Min=0x6c,
                                Conn_Interval_Max=0x6c,
                                Conn_Latency=0x00,
                                Supervision_Timeout=0xc80,
                                Minimum_CE_Length=0x000c,
                                Maximum_CE_Length=0x000c)
```

11. Once connected, Master device enables the characteristic notification on Master&Slave device using the `aci_gatt_write_charac_descriptor()` 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 to exercise the described BlueNRG-MS simultaneously Master, Slave scenario are provided within the BlueNRG DK SW package (see Reference Section). These scripts can be run using the BlueNRG GUI and they can be taken as reference for implementing a firmware application using the BlueNRG-MS simultaneously master and slave feature.

4 References

Table 51. References table

Name	Title
AN4494	Bringing up the BlueNRG application note
BlueNRG datasheet	Bluetooth® low energy wireless network processor
BlueNRG-MS datasheet	Bluetooth® low energy wireless network processor
BlueNRG DK SW package	BlueNRG SW package for BlueNRG and BlueNRG-MS kits
Bluetooth specification V4.0	Specification of the Bluetooth system v4.0
Bluetooth specification V4.1	Specification of the Bluetooth system v4.1
UM1755	BlueNRG Bluetooth LE stack application command interface (ACI) user manual
UM1865	BlueNRG-MS Bluetooth LE stack application command interface (ACI) user manual
UM1686	BlueNRG development kits user manual

Appendix A List of acronyms and abbreviations

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

Table 52. 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
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 52. List of acronyms (continued)

Term	Meaning
UUID	Universally unique identifier
WPAN	Wireless personal area networks

5 Revision history

Table 53. Document revision history

Date	Revision	Changes
23-Jan-2015	1	Initial release.
21-Apr-2015	2	The document has been adapted to refer to both BlueNRG and BlueNRG-MS devices.