BLUEGIGA BLUETOOTH SMART SOFTWARE

V.1.3 API DOCUMENTATION

Wednesday, 10 September 2014

Version 3.5

Table of Contents

1 Version History

,

Version	
1.3	API documentation for SW version v.1.0.3 (Build 43)
2.0	API documentation for v.1.1.0 beta (Build 46)
2.1	API documentation for v.1.1.0 beta (Build 55) Note: API changes history is now included here (not separate) Changed APIs: * Attribute Database – User Read Response (function implemented for Beta 2) * Connection – Connection Status Flags (fixed) Doc improved for following APIs: * Attribute Client – Attribute Value, Indicated, Procedure Completed, Group Found * Attribute Database – User Read Request * Generic Access Profile – Discover, Set Adv Parameters * Hardware – I2c Read, I2c Write, Set Soft Timer, Set Txpower * Security Manager – Delete Bonding, Get Bonds * System – Whitelist Append Other sections (outside API reference) has also been updated to improve the document
2.2	Added documentation how to use BGAPI protocol without UART flow control. Section updated: BGAPI protocol definition
2.3	API documentation for v1.1.0 (Build 71+) * Various typos and wording corrected.
3.0	Documentation updates for SW v1.2 compatibility Changed APIs: • Channel quality testing commands added: Get Channel Map and Channel mode • Out of Bonds and Command Too Long error code added • Protocol error event added for indicating the invalid command or wrong length • GAP Discoverable Mode is updated to support the Enhanced Broadcasting. Doc improved for following APIs/referenses: • Updated ADC internal reference to 1.24V (was 1.15V), • GAP - Set Scan Paremeters, Connect Selective, Connect Direct
3.1	Documentation updates for SW v1.2.2 compatibility Added APIs: • • Added API's for reading (Read Data), writing (Write Data), and erasing (Erase Page) the user area data on the internal flash memory • Added API's for handling I/O port interrupts (Io Port Irq Enable) and setting the directions (Io Port Irq Direction) • Added testing API's for sending and receiving data (Phy Tx, Phy Rx, Phy End) • Added API's for handling the comparator functionality under HW commands and events.

<table>
<tr><th>Version</th><th></th></tr>
<tr><td>3.2</td><td>Documentation updates for SW v1.3.0 compatibility

Added APIs:
Added Set RXGain API for controlling RX Gain for lowering the sensitivity (Hardware commands)
Added Usb Enable API for controlling whether USB interface is on or off (Hardware commands)
Added AES API’s for using AES engine for de-/encryptions (System commands)</td></tr>
<tr><td>3.3</td><td>Documentation updates for SW v1.3.1 compatibility

Added APIs:
Added Send Attributes (attributes_send) command for controlling sending of notifications and indications (Attributes commands)
Added Whitelist Bonds (sm_whitelist_bonds) command for adding all the bonded devices to the whitelist (Security Manager commands).</td></tr>
<tr><td>3.4</td><td>Editorial changes and improvements and enhancements to command, response and event descriptions.</td></tr>
<tr><td>3.5</td><td>Editorial changes and improvements and enhancements to command, response and event descriptions.</td></tr>
</table>

2 Introduction to Bluegiga Bluetooth Smart Software

The Bluegiga *Bluetooth* Smart Software enables developers to quickly and easily develop *Bluetooth* Smart applications without in-depth knowledge of the *Bluetooth* Smart technology. The *Bluetooth* Smart Software consists of two main parts:

- The *Bluetooth* Smart Stack
- The *Bluetooth* Smart Software Development Kit (SDK)

2.1 The Bluegiga Bluetooth Smart Stack

The *Bluetooth* Smart stack is a fully *Bluetooth* 4.0 single mode compatible software stack implementing slave and master modes, all the protocol layers such as L2CAP, Attribute Protocol (ATT), Generic Attribute Profile (GATT), Generic Access Profile (GAP) and security manager (SM). The Bluetooth Smart stack also implements various other features such as interface APIs to SPI, UART, GPIO, ADC, flash etc. and other features like the Device Firmware Update (DFU) API.

The *Bluetooth* Smart is meant for the Bluegiga *Bluetooth* Smart products such as BLE112, BLE113 BLE121LR and BLED112.

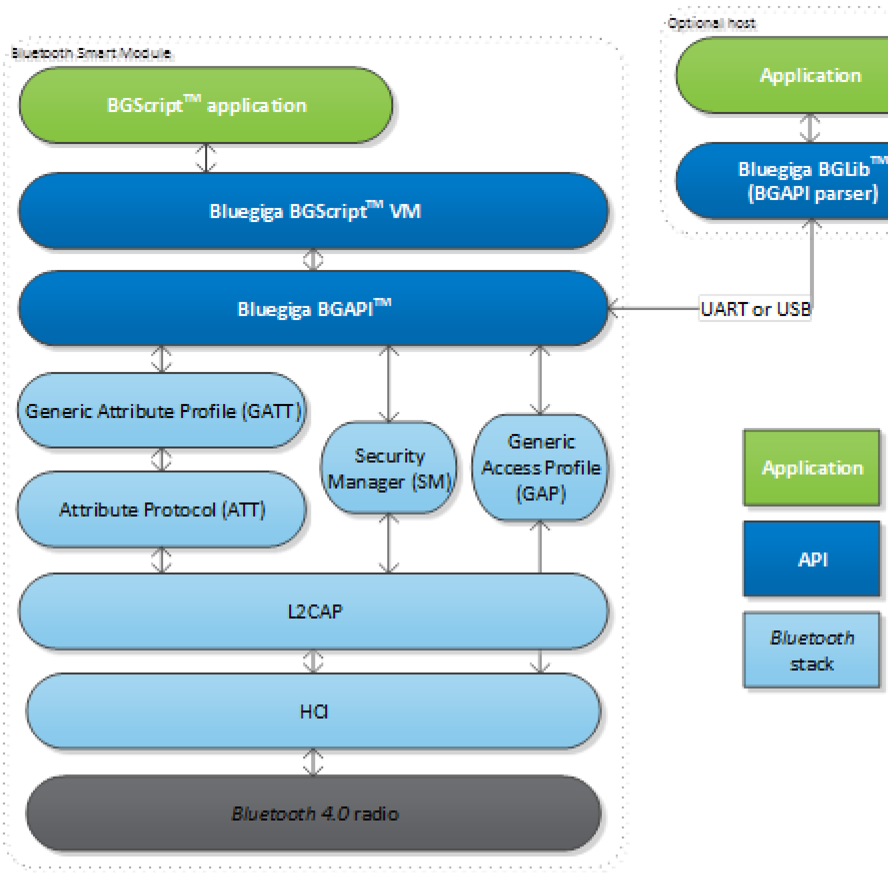


Figure: The Bluegiga Bluetooth Smart Stack

2.2 The Bluegiga Bluetooth Smart SDK

The Bluegiga *Bluetooth* Smart SDK is a software development kit, which enables the device and software vendors to develop products on top of the Bluegiga's *Bluetooth* Smart hardware and software.

The *Bluetooth* Smart SDK supports multiple development models and the software developers can decide whether the device's application software runs on a separate host (for example a MCU) or whether they want to make fully standalone devices and execute their application on-board the Bluegiga *Bluetooth* Smart modules.

The SDK also contains documentation, tools for compiling the firmware, installing it into the hardware and lot of example application speeding up the development process.

The *Bluetooth Smart* SDK contains the following components:

- **The BGAPI™ protocol** is a binary based commend and response protocol that allows the Bluetooth Smart stack to be controller form an external host and an application over for example UART or USB interface.
- **The BGScript™ scripting language** is a simple BASIC like scripting language that allows the software developers to embed applications on-board the Bluegiga *Bluetooth* Smart modules. The BGScript applications are executed in the BGScript Virtual Machine (VM) and the benefit of this is that no external host MCU is required.
- **The BGLIB™ host library** is a lightweight parser for the BGAPI host protocol and it implements C functions and callback handlers for all the BGAPI commands, responses and events. The benefit of the BGLIB library is that speeds up the application development for the external host processors.
- **The Profile Toolkit™** is a simple XML based description language that enables quick and easy development of GATT Bluetooth Smart services and characteristics on a device.

Each of these components are described in more detail in the following chapters.

2.3 The BGAPI TM Protocol

For applications where a separate host is used to implement the end user application, a transport protocol is needed between the host and the *Bluetooth* Smart stack. The transport protocol is used to communicate with the *Bluetooth* stack as well to transmit and receive data packets. This protocol is called BGAPI and it's a lightweight binary based communication protocol designed specifically for ease of implementation within host devices with limited resources.

The BGAPI protocol is a simple command, response and event based protocol and it can be used over UART or USB physical interfaces.

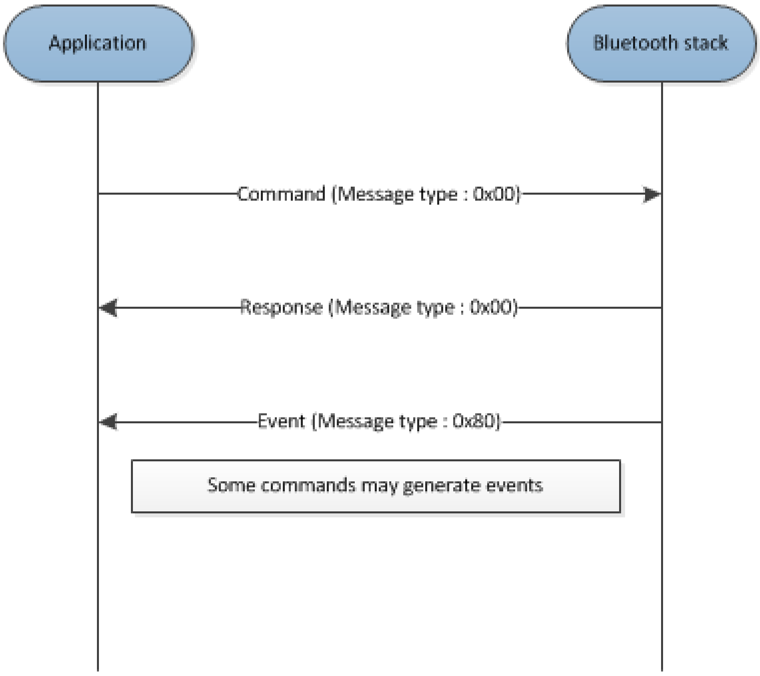


Figure: BGAPI message exchange

The BGAPI provides access for example to the following layers in the *Bluetooth* Smart Stack:

- **Generic Access Profile** - GAP allows the management of discoverability and connetability modes and open connections
- **Security manager** - Provides access the *Bluetooth* low energy security functions
- **Attribute database** - An class to access the local attribute database
- **Attribute client** - Provides an interface to discover, read and write remote attributes
- **Connection** - Provides an interface to manage *Bluetooth* low energy connections
- **Hardware -** An interface to access the various hardware layers such as timers, ADC and other hardware interfaces
- **Persistent Store** - User to access the parameters of the radio hardware and read/write data to non-volatile memory
- **System** - Various system functions, such as querying the hardware status or reset it

2.4 The BGLIB TM Host Library

For easy implementation of BGAPI protocol an ANSI C host library is available. The library is easily portable ANSI C code delivered within the *Bluetooth* Smart SDK. The purpose is to simplify the application development to various host environments.

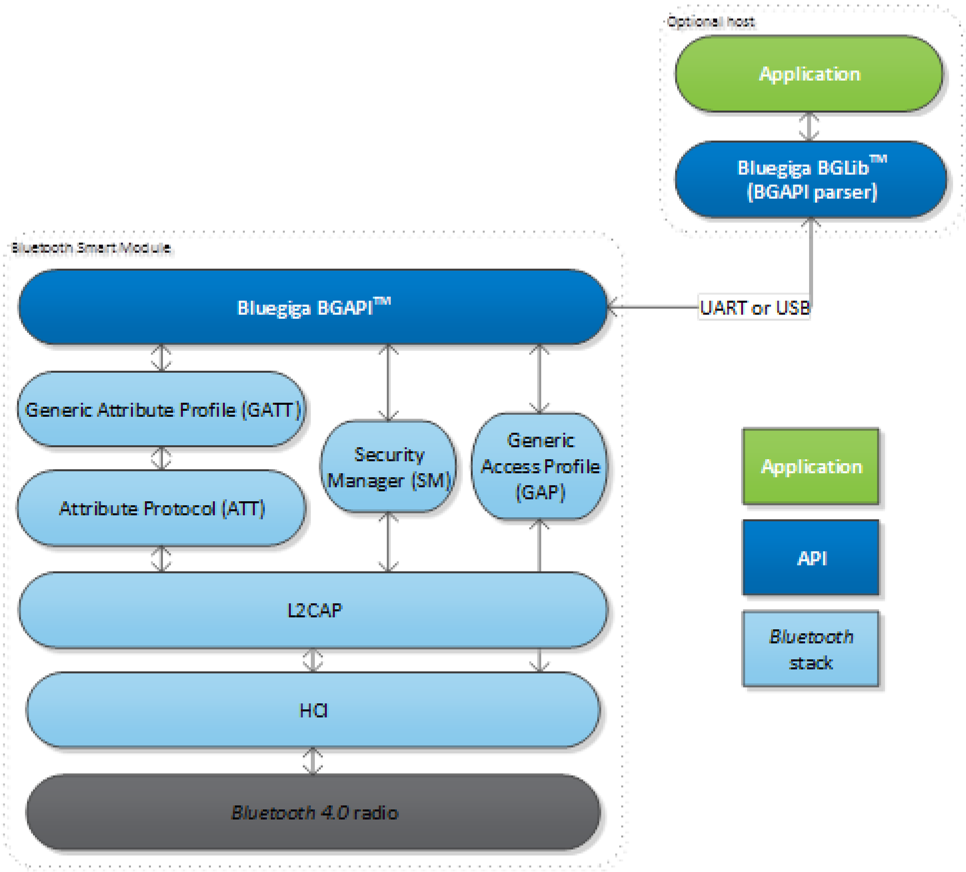


Figure: The BGLIB host library

2.5 The BGScript TM Scripting Language

The *Bluetooth* Smart SDK Also allows the application developers to create fully standalone devices without a separate host MCU and run all the application code on the Bluegiga *Bluetooth* Smart modules. The *Bluetooth* Smart modules can run simple applications along the *Bluetooth* Smart stack and this provides a benefit when one needs to minimize the end product's size, cost and current consumption. For developing standalone *Bluetooth* Smart applications the SDK includes a BGScript VM, compiler and other BGScript development tools. BGScript provides access to the same software and hardware interfaces as the BGAPI protocol and the BGScript code can be developed and compiled with free-of-charge tools provided by Bluegiga.

Typical BGScript applications are only few tens to hundreds lines of code, so they are really quick and easy to develop and lots of readymade examples are provides with the SDK.

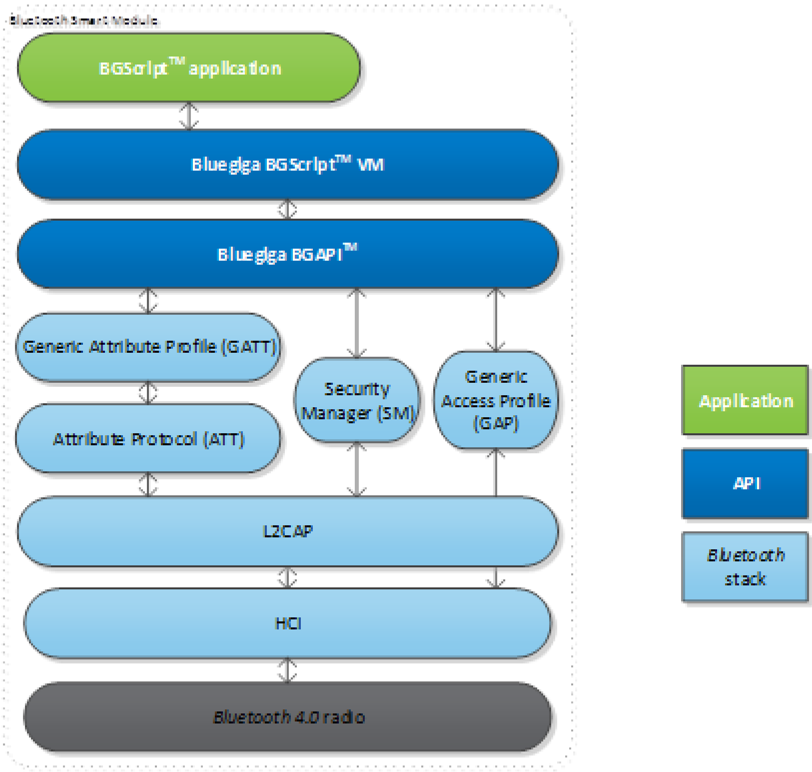


Figure: BGScript application model

```
# System Started
event system_boot(major, minor, patch, build, ll_version, protocol_version,hw)
          #Enable advertising mode
          call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
          #Enable bondable mode
          call sm_set_bondable_mode(1)
          #Start timer at 1 second interval (32768 = crystal frequency)
          call hardware_set_soft_timer(32768)
end
```

Figure: BGScript code example

2.6 The Profile Toolkit TM

The *Bluetooth* Smart profile toolkit is a simple set of tools, which can used to describe GATT based *Bluetooth* Smart services and characteristics. The profile toolkit consists of a simple XML based description language and templates, which can be used to describe the devices GATT database. The profile toolkit also contains a compiler, which converts the XML to binary format and generates API to access the characteristic values.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <service uuid="1800">
      <description>Generic Access Profile</description>

      <characteristic uuid="2a00">
        <properties read="true" const="true" />
        <value>BGDemo sensor</value>
      </characteristic>

      <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">4142</value>
      </characteristic>
    </service>

</configuration>
```

Figure: A profile toolkit example of GAP service

3 Introduction to Bluetooth Smart Technology

This section gives a quick introduction to the *Bluetooth* Smart technology and its most important features. The chapter does not contain complete detailed technology walkthrough but gives developers more insight into the technology and to help them develop *Bluetooth* Smart applications.

3.1 Physical layer

The features of physical the layer in *Bluetooth* low energy are:

Feature	Value
Frequency band	2.4GHz (2402Mhz - 2480MHz
Modulation	GFSK, 1 Mbps
Modulation index	0.5
Channel spacing	2 MHz
Advertising channels	3
Data channels	37
Frequency hopping	Adaptive FHSS

The requirements for the *Bluetooth* low energy radio are:

Feature	Value
Minimum TX power	0.01mW (-20 dBm)
Maximum TX power	10 mW (10 dBm)
Minimum RX sensitivity	-70 dBm (BER 0.1%)

The typical range for *Bluetooth* low energy radios is:

TX power	RX sensitivity	Range
0 dBm	-70 dBm	~30 meters
10 dBm	-90 dBm	100+ meters

The figure below illustrates the link layer channels. There are 37 data channels and 3 advertisement channels.

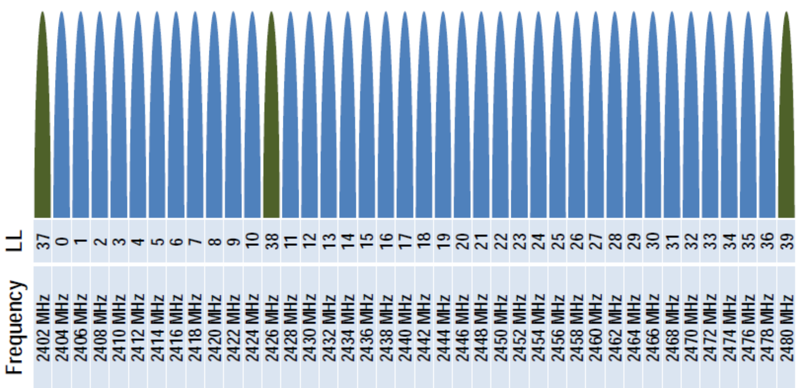


Figure: Link layer channels

3.2 Packet format

3.2.1 Generic packet format

Bluetooth Smart technology uses one generic packet format used for both advertisement and data packets.

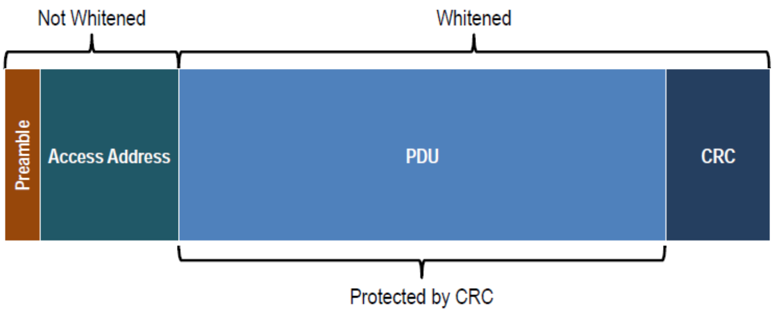


Figure: Generic packet format

- **Preamble:** either 010101010 or 101010101
- **Access address:** advertisement packets use a fixed access address of 0x8E89BED6. Data packets use a random access address depending on the connection.
- **PDU:** protocol data unit depends on the packet type.
- **CRC:** a 24-bit CRC checksum is used to protect the PDU.

3.2.2 Advertisement packet format

The advertisement packets use the following structure and can contain 0 to 31 bytes of advertisement data.

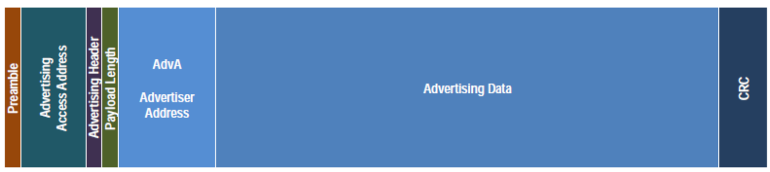


Figure: Advertisement packet structure

3.2.3 Data packet format

The data packets on the other hand use the following structure. An unencrypted data packet can have 0 to 27 bytes of payload.

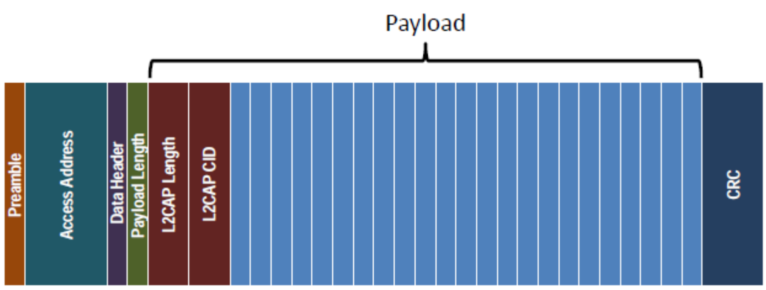


Figure: Unencrypted data packet

An encrypted data packet can have 0 to 31 bytes of payload length, but MIC (Message Integrity Check) is part of it.

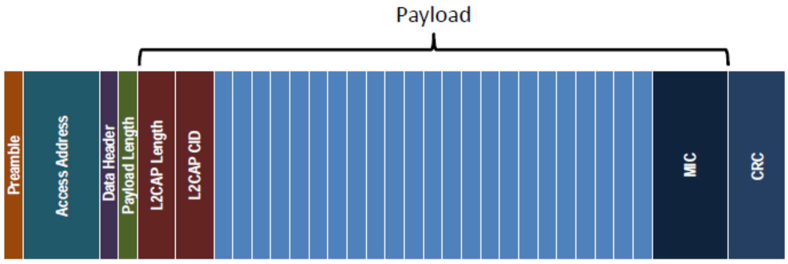


Figure: Encrypted data packet

3.3 Link layer state machine

The *Bluetooth* low energy link layer state machine and state transitions are illustrated in the figure below.

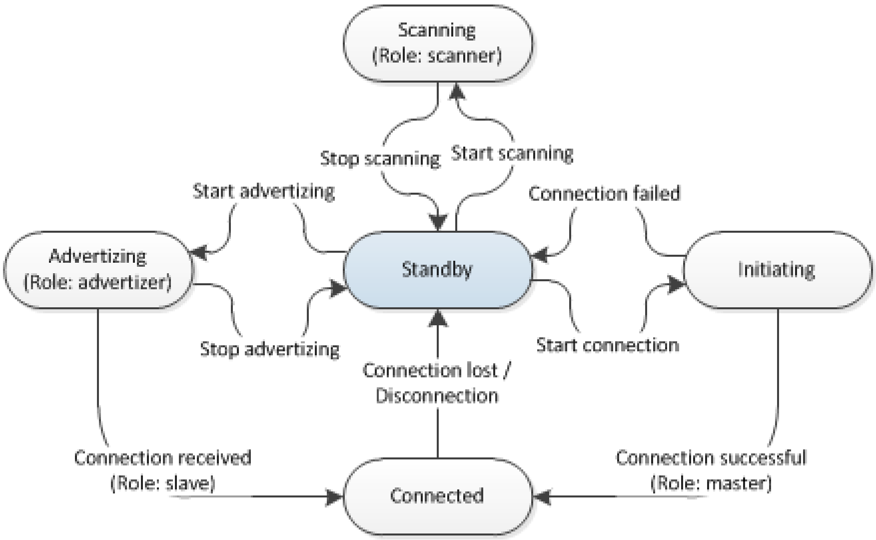


Figure: Link layer state machine and state transitions

3.4 Link layer operations

This section describes the *Bluetooth* low energy link layer operations.

3.4.1 Passive scanning

In passive scanning mode the advertiser simply broadcasts advertisement packets on the advertising channels and a scanner simply listens to incoming advertisements.

Typically in passive scanning scenario:

- Advertiser sends three advertisement packet one on each advertisement channel separated by 150us.
- Scanner only listens to one advertisement channel at a time, but keeps switching between the three advertisement channels.

The advertisement events are separated by a time called advertisement interval, which can vary from 20ms to 10240ms. In addition a random delay is added to the advertisement interval to avoid interference with other devices.

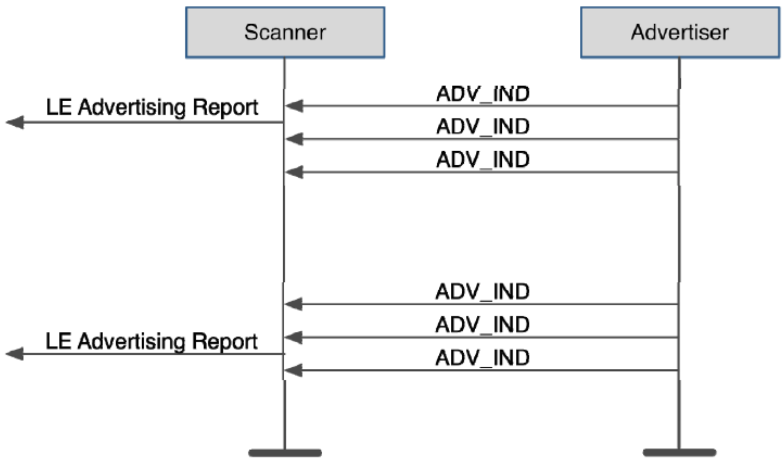


Figure: Passive scanning

The advertisement packets typically contains information like:

- Discoverability and connectability modes
- The address of advertiser
- TX power level
- Supported services
- Application data

3.4.2 Active scanning

In active scanning mode the scanner will request more information from the Advertiser after it has received an advertisement packet. The scanner will send a scan request packet to the advertiser and, which the advertiser can respond by sending back scan response packet and scan response data.

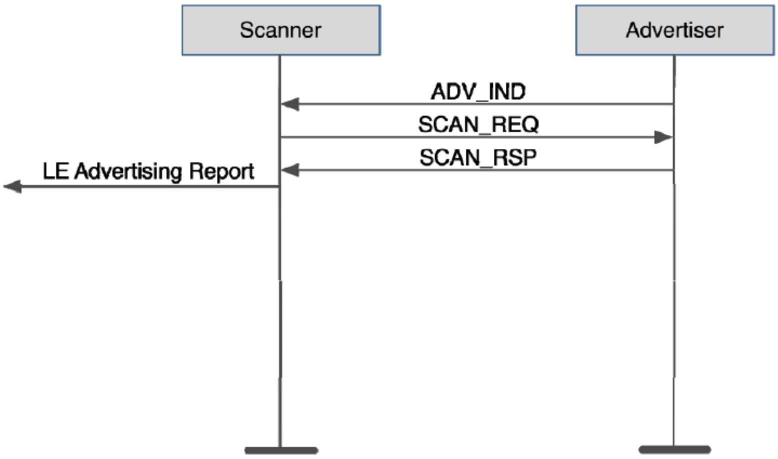


Figure: Active scanning

The scan response packets typically contains information like:

- Device friendly name
- Supported services (profiles)
- Application data

3.4.3 Connection establishment

The figure below illustrates how the connection establishment happens at the link layer level.

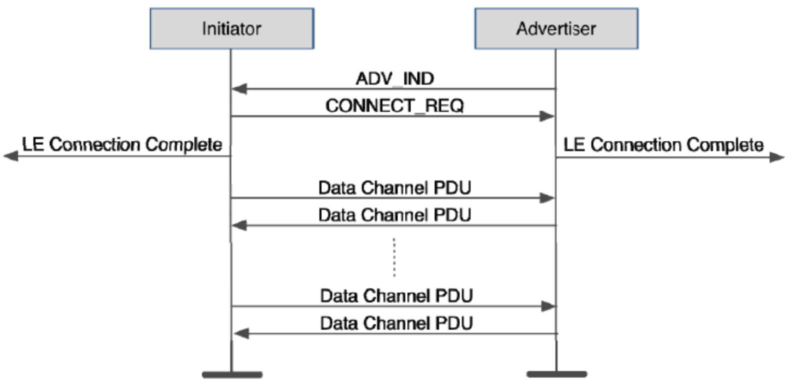


Figure: Bluetooth low energy connection establisment

3.5 Topologies

Bluetooth 4.0 specification defines four device roles: advertiser, scanner, master and slave. The 4.0 version of the specification supports point-to-point and start topologies. The figure below illustrates the device roles, and topologies.

- **Advetiser** : Broadcasts advertisement packets, but is not able to receive them
- **Scanner :** Listens for advertisement packets sent out by advertisers. Can try to connect an advertiser.
- **Master :** A device that is connected to one or several slaves
- **Slave :** A deivce that is connected to a master. Can only be connected to one master at a time

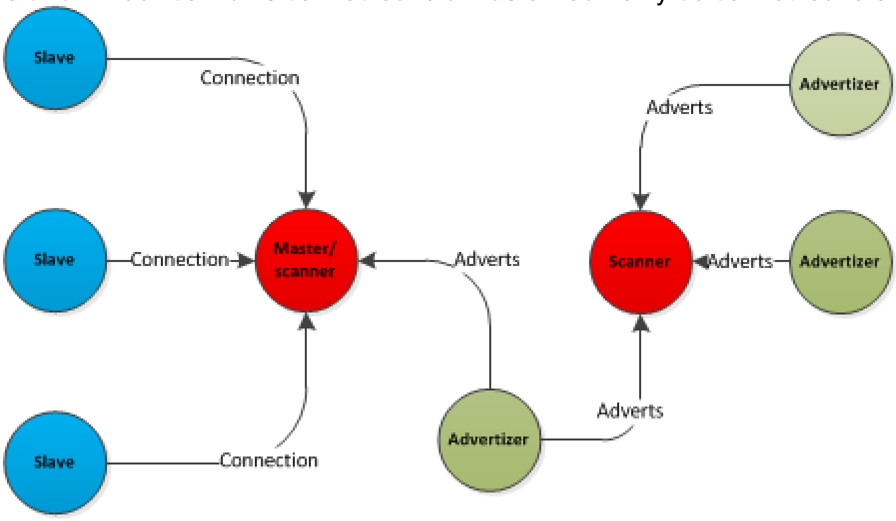


Figure: Bluetooth low energy topologies

Devices can change roles and topologies as illustrated below.

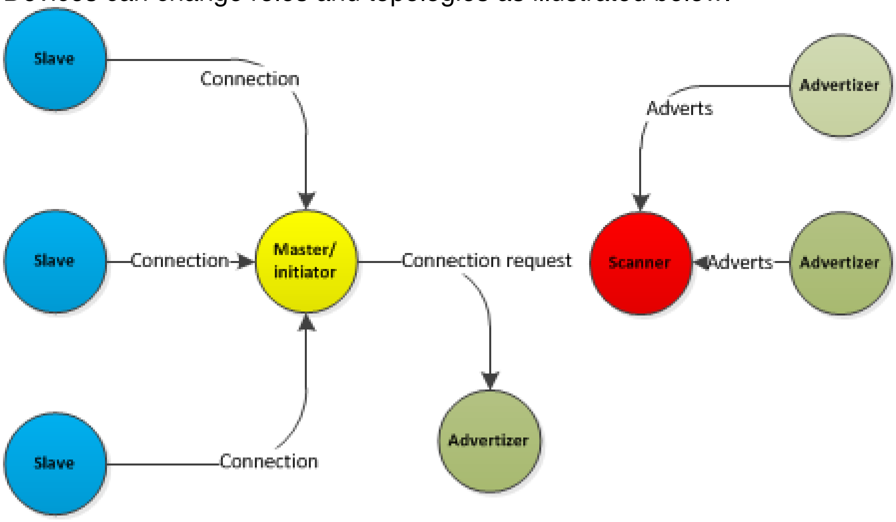


Figure: Topology and role change

3.6 Connections and packet timings

Connections allow application data to be transmitted reliably and robustly. The data sent in a connection can be acknowledged, integrity is protected by CRC and to protect privacy the data can also be encrypted. On addition the Adaptive Frequency Hopping (AFH) guarantees reliable data transmission even in noisy environments.

In *Bluetooth* Smart technology the connection procedures are very simple and connections are always starts when master sends a connection request packet to the slave. The connection request packet can only be sent right after a successful reception of an advertisement packet. The connection request packet contains the following information:

Parameter	Description
Conn_Interval_Min	Minimum value for the connection event interval **Range: 7.5 ms to 4000ms**
Conn_Interval_Max	Maximum value for the connection event interval **Range: 7.5 ms to 4000ms** Shall be greater then Conn_Interval_Min
Conn_Latency	Slave latency for the connection in number of connection events. Slave latency allows the slave devices to skip a number of connection events in case it does not have any data to send. **Range: 0 to 500**
Supervision_Timeout	Supervision timeout **Range: 100ms to 32 seconds** Shall be greater than Connection Interval

The connection parameters can be updated during the connection.

The connection timeline and events are illustrated below.

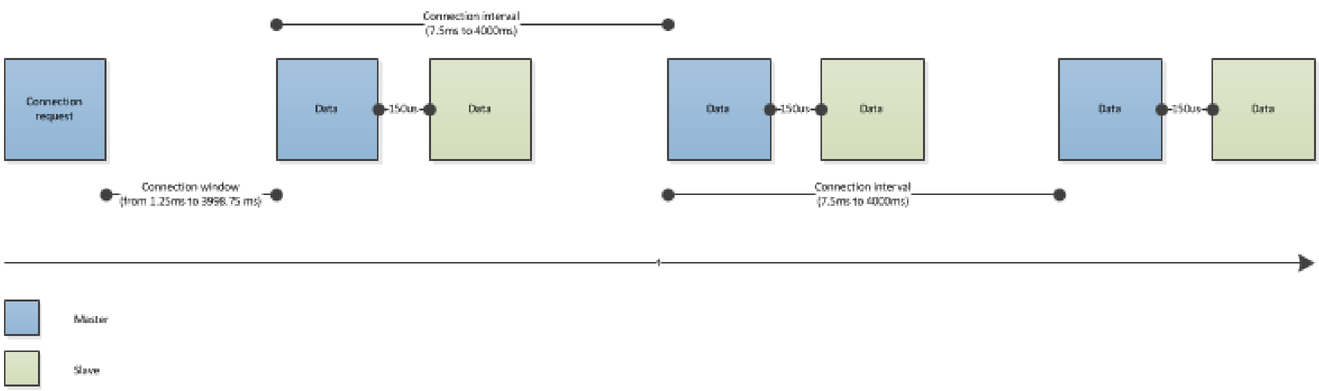


Figure: Bluetooth LE connection

The connection event starts, when master sends a packet to the slave at the defined connection interval. The slave can respond 150us after it has received a packet from the master. However if the slave has no data to send it can skip a certain number of connection events defined by the slave latency parameter. If no packets are received by the master or slave within the time defined by the supervision timeout, the connection is terminated.

If the slave has more data to send than what can be fitted into a single packet, the connection event will automatically extend and the slave can send as many packets as there is time until the beginning of next connection interval. This however can only be used with attribute protocol operations, that do not require an acknowledgement.

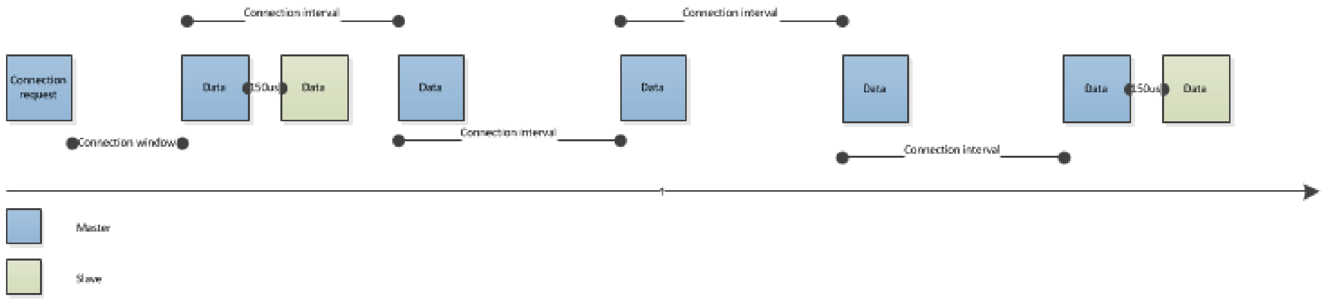


Figure: Slave latency in function (latency=3)

3.7 Encryption

Bluetooth low energy uses AES-128 link layer encryption block with Counter Mode CBC MAC (defined in RFC 3610).

The data packets are encrypted as show below.

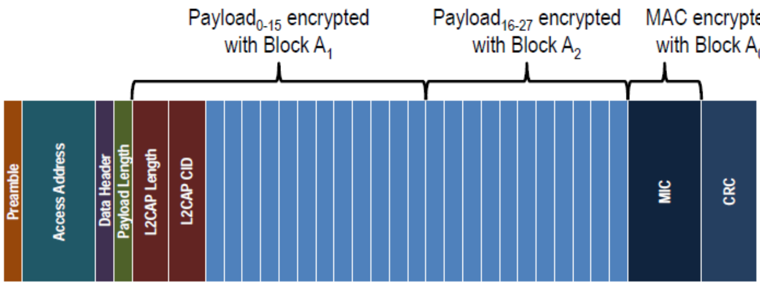


Figure: Encrypted data packet

The full AES encryption procedure is illustrated below.

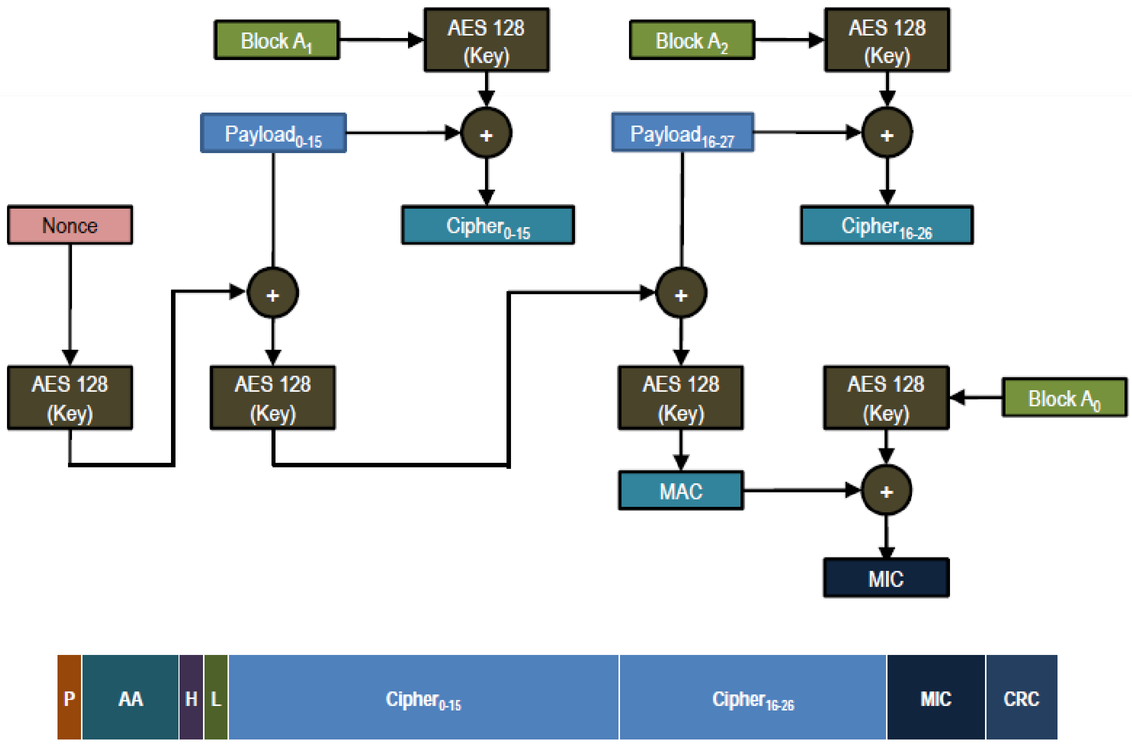


Figure: AES encryption procedure

Limitations of link layer encryption

- Maximum 2^39 packets per Long Term Key (LTK)
    - 13.7 TB of data / connection
    - ~12 years at maximum data rate

3.8 L2CAP

L2CAP stands for Logical Link Control and Adaptation Protocol and it is acts as a protocol multiplexer and handles segmentation and reassembly of packets. It also provides logical channels, which are multiplexed over a or more logical links.

All application data is sent over L2CAP packets and the L2CAP structure is illustrated below.

Figure: L2CAP packet format

The following CIDs are defined:

CID	Description	Notes
0x0000	Null identifier	Not used
0x0001	L2CAP Signaling Channel	BR/EDR only
0x0002	Connectionless Channel	BR/EDR only
0x0003	AMP Manager Protocol	BR/EDR only
0x0004	Attribute Protocol	LE only
0x0005	LE L2CAP Signaling Channel	LE only
0x0006	Security Manager Protocol	LE only

3.9 Security Manager

The security manager protocol is responsible of:

- Pairing
- Key distribution
- Generating hashes and short term keys

The security manager uses asymmetric model and more responsibility is given to the master device, so the memory and processing requirements on the slaves can be kept to minimum.
The basic security manager concepts include:

- **Distributing key model**
  Slave generates and distributes key information to master
  Master can use this key information when reconnecting

- **Pairing**
  Authentication of devices based on their capabilities and security requirements

- **Signing Data**
  Signing allows authentication of sender without encryption

- **Bonding**
  GAP concept – device save keys for bonded devices

Three pairing methods are supported:

- Just works pairing, similar to *Bluetooth* 2.1 + EDR
- Man-in-the-Middle pairing using a passkey entry or comparison, similar to *Bluetooth* 2.1 + EDR
- Out-of-band pairing, where security keys are exchanged over an other medium like NFC

3.9.1 I/O capabilities and Man-in-the-Middle (MITM) protection

Same I/O capabilities and MITM features are supported as in *Bluetooth* 2.1 + EDR.

	No Input	Yes / No	Keyboard
No Output	No Input No Output	No Input No Output	Keyboard Only
Numeric Output	Display Only	Display Yes No	Keyboard Display

Figure: I/O capabilities

3.10 Attribute Protocol (ATT)

Bluetooth low energy profiles expose a state of a device. The state is exposed as one or several values called attributes and the protocol to access these attributes is called the Attribute protocol (ATT).

The attribute protocol uses a client server architecture and has two roles:

- **Server**
  Service is the device that exposes the information as one or several attributes

- **Client**
  Client device that collects the information for one or more servers

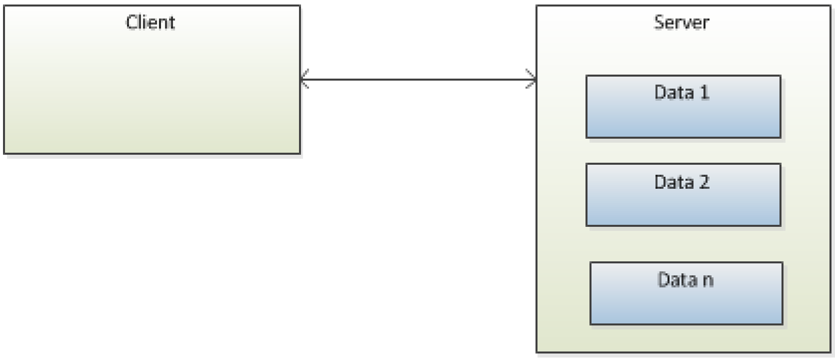


Figure: Device roles

Attribute types:

Attributes are values:

- Arrays of octets
- From 0 to 512 octets
- Can be fixed or variable length

Example:

Value
0x0000
0x426c75656769676120546563686e6f6c6f67696573

Attribute have handles, which are used to address an individual attribute. The client accesses the server's attributes using this handle.

Example:

Handle	Value
0x0001	0x0000
0x0002	0x426c7565676967612054656363686e6f6c6f6769657

Attributes also have a type, described by a UUID. UUID determines what the attribute value means.

Two types of UUIDs are used:

- Globally unique 16-bit UUID defined in the characteristics specifications (http://developer.bluetooth.org/)
- Manufacturer specific 128-bit UUIDs, which can for example be generated online. (http://www.uuidgenerator.com/)

Example:

Handle	UUID	Value	Description
0x0001	0x1804	0x0000	TX power as dBm
0x0002	0x2a00	0x426c75656769676120546563686e6f6c6f6769657	Device name, UTF-8

Attribute permissions:

Attributes also have permissions, which can be:

- Readable / Not readable
- Writable / Not writable
- Readable and writable / Not readable and not writable

The attributes may also require:

- Authentication to read or write
- Authorization to read or write
- Encryption and pairing to read or write

The attribute types and handles are public information, but the permissions are not. Therefore and read or write request may result an error *Read/Write Not Permitted* or *Insufficient authentication.*

Attribute protocol methods:

The attribute protocol is a stateless sequential protocol, meaning that no state is stored in the protocol and only one operation can be performed at a time.

The available Attribute Protocol methods are described in the table below:

Method	Description	Direction
Find Information (starting handle, ending handle)	Used to discover attribute handles and their types (UUIDs)	Client -> Server
Find By Type Value (starting handle, ending handle, type, value)	Returns the handles of all attributes matching the type and value	Client -> Server
Read By Group Type (starting handle, ending handle, type)	Reads the value of each attribute of a given type in a range	Client -> Server
Read By Type (starting handle, ending handle, type)	Reads the value of each attribute of a given type in a range	Client -> Server
Read (handle)	Reads the value of given handle **Maximum payload : 22 bytes**	Client -> Server
Read Blob (handle, offset)	Can be used to read long attributes larger than 22 bytes. **Maximum payload: 64 kBytes**	Client -> Server
Read Multiple ([Handle]*)	Used to read multiple values at the same time	Client -> Server
Write (handle, value)	Writes the value to the given handle, with no response **Maximum payload: 20 bytes**	Client -> Server
Prepare Write (handle, offset, value) and Execute (exec/cancel)	Prepares a write procedure, which is queued in server until the write is executed.	Client -> Server
Handle Value Notification (handle, value)	Server notifies client of an attribute with a new value **Maximum payload: 20 bytes**	Server -> Client
Handle Value Indication (handle, value)	Server indicates to client an attribute with a new value. Client must confirm reception. **Maximum payload: 20 bytes**	Server -> Client
Error response	Any request can cause an error and error response contains information about the error	Server -> Client

3.11 Generic Attribute Profile (GATT)

The Generic ATTribute profile (GATT) has similar client server structure as Attribute Protocol. However the GATT encapsulates data (attributes) into *services* and the data is exposed as *characteristics*.

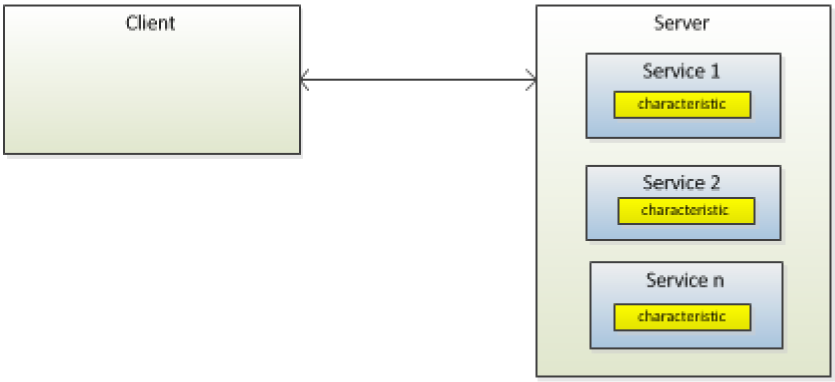


Figure: GATT architecture

GATT defines concepts of:

- Service Group
- Characteristic Group
- Declarations
- Descriptors

It's important also to understand that GATT does not does not define rules for their use.

Characteristics

Characteristic is a value, with a known type, and a known format. They characteristics are defined in "Characteristic Specification" available at http://developer.bluetooth.org.

Characteristics consist of:

- Characteristic Declaration
  Describes the properties of *characteristic value* (read, write, indicate etc.), *characteristic value* handle and *characteristic value* type (UUID)

- Characteristic Value
  Contains the value of the characteristic.

- Characteristic Descriptor(s)
  Provide additional information about the characteristic (characteristic user description, characteristic client configuration, vendor specific information etc.)

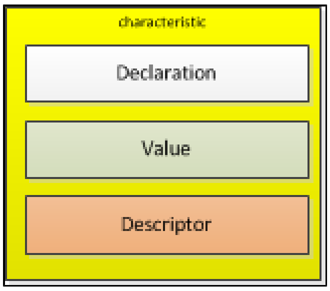


Figure: Characteristic format

Service

A service is:

- defined in a service specification (http://developer.bluetooth.org)
- collection of characteristics
- references to other services

There are two types of service:

- Primary services
  A primary service exposes primary functionality of a device. It can be included by an other service.

- Secondary services
  Secondary service is a subservient of another primary or a secondary service. It's only relevant in the context of an other service.

Attributes alone are just flat:

Handle	Type	Value	Permissions
0x0001	«Primary Service»	«GAP»	R
0x0002	«Characteristic»	{r, 0x0003, «Device Name»}	R
0x0003	«Device Name»	"Temperature Sensor"	R
0x0004	«Characteristic»	{r, 0x0006, «Appearance»}	R
0x0006	«Appearance»	«Thermometer»	R
0x000F	«Primary Service»	«GATT»	R
0x0010	«Characteristic»	{r, 0x0012, «Attribute Opcodes Supported»}	R
0x0012	«Attribute Opcodes Supported»	0x00003FDF	R
0x0020	«Primary Service»	«Temperature»	R
0x0021	«Characteristic»	{r, 0x0022, «Temperature Celsius»}	R
0x0022	«Temperature Celsius»	0x0802	R*

Figure: List of attributes

Grouping attributes into services gives structure:

Handle	Type	Value	Permissions
0x0001	«Primary Service»	«GAP»	R
0x0002	«Characteristic»	{r, 0x0003, «Device Name»}	R
0x0003	«Device Name»	"Temperature Sensor"	R
0x0004	«Characteristic»	{r, 0x0006, «Appearance»}	R
0x0006	«Appearance»	«Thermometer»	R
0x000F	«Primary Service»	«GATT»	R
0x0010	«Characteristic»	{r, 0x0012, «Attribute Opcodes Supported»}	R
0x0012	«Attribute Opcodes Supported»	0x00003FDF	R
0x0020	«Primary Service»	«Temperature»	R
0x0021	«Characteristic»	{r, 0x0022, «Temperature Celsius»}	R
0x0022	«Temperature Celsius»	0x0802	R*

Figure: Attributes grouped into services

GATT procedures

The available Attribute Protocol methods are described in the table below:

Procedure	Sub-Procedures
Server Configuration	Exchange MTU
Primary Service Discovery	Discovery All Primary Service Discover Primary Service by Service UUID.
Relationship Discovery	Find Included Services
Characteristic Discovery	Discover All Characteristics of a Service Discover Characteristics by UUID
Characteristic Descriptor Discovery	Discover All Characteristic Descriptors
Characteristic Value Read	Characteristic Value Read Read Characteristic Value Read Using Characteristic UUID Read Long Characteristic Values Read Multiple Characteristic Values
Characteristic Value Write	Write Without Response Write Without Response With Authentication Write Characteristic Value Write Long Characteristic Values Reliable Writes
Characteristic Value Notifications	Notifications
Characteristic Value Indications	Indications
Characteristic Descriptors	Read Characteristic Descriptors Read Long Characteristic Descriptors Write Characteristic Descriptors Write Long Characteristic Descriptors

3.12 Generic Access Profile (GAP)

GAP defines device roles:

- **Broadcaster** : Sends advertising events, including characteristics, including service data (does not need RX)
- **Observer** : Receives advertising events, listens for characteristics, listens for service data (does not need TX)
- **Peripheral** : Has RX and TX, is always slave, is connectable and advertising
- **Central** : Has RX and TX, is always master, never advertises

GAP also defines modes and procedures for

- Discovery
- Connections
- Bonding

Privacy

- Non-Resolvable and Resolvable Private Addresses

4 API definition

This section of the document contains the generic *Bluetooth* Smart Stack API definition. The definition consist of three parts:

- The BGAPI^{TM} host protocol API definition
- The BGLIB^{TM} host library API description
- The BGScript^{TM} scripting language API description

This section of the document only provides the generic definition and description of the API. The actual commands, responses and events are described in detail in the API reference section.

4.1 The BGAPI protocol definition

The BGAPI protocol is a command, response and event protocol that can be used to communicate with the *Bluetooth* Smart stack over one of the physical interfaces like UART or USB. The BGAPI protocol can be used to instruct the *Bluetooth* Smart stack to do something like advertise, discover and connect other *Bluetooth* devices or access the physical interfaces like SPI or I2C of the *Bluetooth* Smart module.

The BGAPI commands, responses and events use a binary format and the generic protocol format is described in this section.

BGAPI Packet format

The generic BGAPI protocol format is described in the table below. The BGAPI protocol uses a four (4) byte header and data payload.

Packets in either direction use the following format.

Table: BGAPI packet format

Octet	Octet bits	Length	Description	Notes
Octet 0	7	1 bit	**Message Type (MT)**	**0:** Command/Response **1:** Event
...	6:3	4 bits	**Technology Type (TT)**	**0000**: *Bluetooth* Smart **0001**: Wi-Fi
...	2:0	3 bits	**Length High (LH)**	Payload length (high bits)
Octet 1	7:0	8 bits	**Length Low (LL)**	Payload length (low bits)
Octet 2	7:0	8 bits	**Class ID (CID)**	Command class ID
Octet 3	7:0	8 bits	**Command ID (CMD)**	Command ID
Octet 4-n	-	0 - 2048 Bytes	**Payload (PL)**	Up to 2048 bytes of payload

With the Bluegiga *Bluetooth* Smart products the maximum allowed BGAPI packet size is 64 bytes and longer packet sizes cannot be used. Four (4) bytes will be used for the BGAPI protocol header so the maximum payload size is 60 bytes.

4.1.1 Message types

The following message types exist in the BGAPI protocol.

Table: BGAPI message types

Message type	Message Type (MT) Value	Description
Command	0x00	Command from host to the stack
Response	0x00	Response from stack to the host
Event	0x80	Event from stack to the host

4.1.2 Command Class IDs

The following command classes exist.

Table: BGAPI command classes

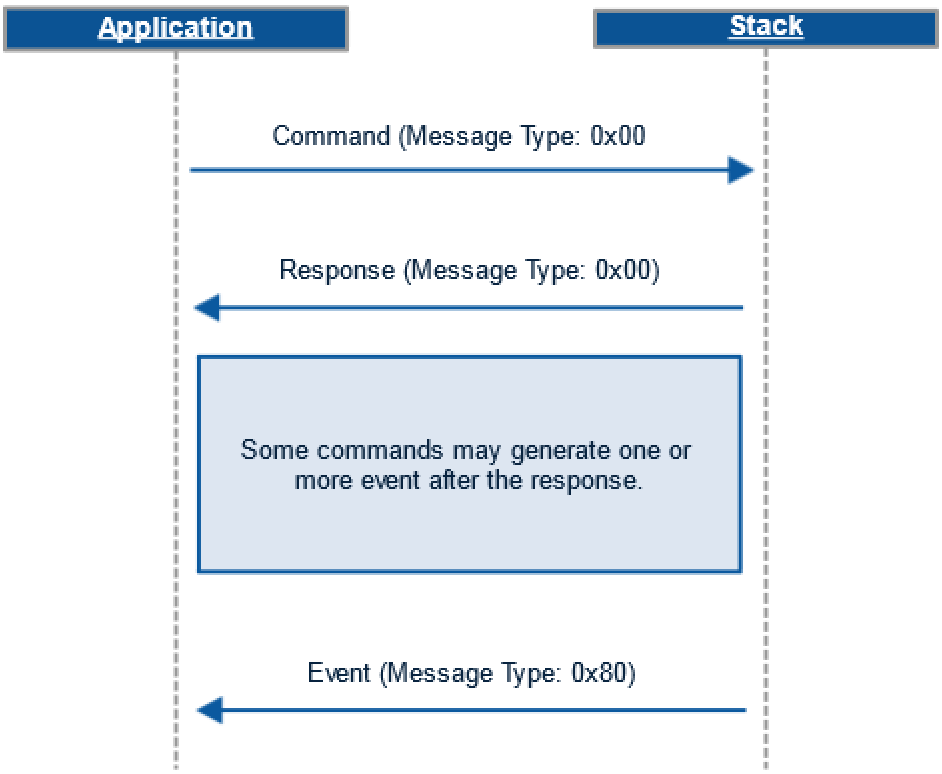
Class ID	Description	Explanation
0x00	System	Provides access to system functions
0x01	Persistent Store	Provides access the persistence store (parameters)
0x02	Attribute database	Provides access to local GATT database
0x03	Connection	Provides access to connection management functions
0x04	Attribute client	Functions to access remote devices GATT database
0x05	Security Manager	Bluetooth low energy security functions
0x06	Generic Access Profile	GAP functions
0x07	Hardware	Provides access to hardware such as timers and ADC

4.1.3 Packet Exchange

The BGAPI protocol is a simple command / response protocol and the BGAPI messages are exchanged as show in the picture below.

The command messages are transmitted from the Application to the Stack and the Stack provides a response to every successfully received command.

Some commands may generate events, which are transmitted from the Stack to the Application.



> ⚠ The Application should always wait for the response to a command before issuing another command.

Using BGAPI protocol without UART flow control

By default the BGAPI protocol assumes that UART flow control (RTS/CTS) is used to ensure reliable data transmission and to prevent lost data because of buffer overflows. It is however possible to use the BGAPI protocol without UART flow control.

When using the BGAPI protocol without UART flow control over a simple 2-wire (TX and RX) UART interface and additional ***length byte*** needs to be added to the BGAPI packets, which tells the total length of the BGAPI packet excluding the ***length byte*** itself. This is used by the BGAPI protocol parser to identify the length of incoming commands and data and make sure they are fully received.

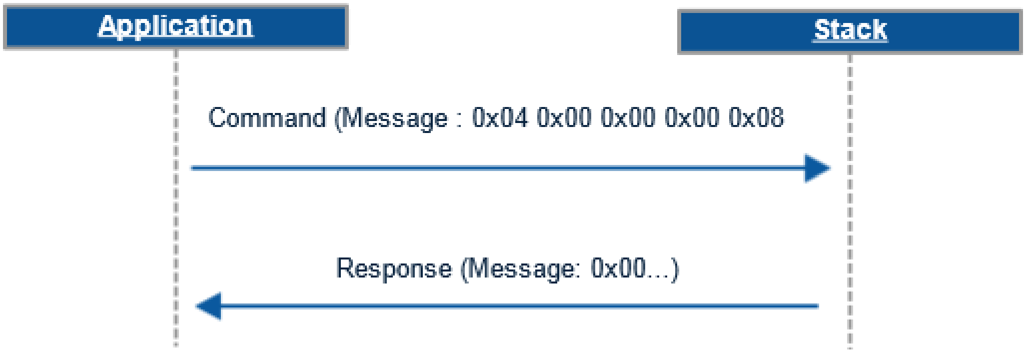
In this case the BGAPI protocol uses the following format:

Table: BGAPI packet format

Octet	Octet bits	Length	Description	Notes
Octet 0	7:0	8 bits	**BGAPI command length**	Tells the length of the BGAPI command excluding the length byte itself Range of this octet is 4 - 62
Octet 1	7	1 bit	**Message Type (MT)**	**0:** Command/Response **1:** Event
...	6:3	4 bits	**Technology Type (TT)**	**0000:** Bluetooth Smart **0001:** Wi-Fi
...	2:0	3 bits	**Length High (LH)**	Payload length (high bits)
Octet 2	7:0	8 bits	**Length Low (LL)**	Payload length (low bits)
Octet 3	7:0	8 bits	**Class ID (CID)**	Command class ID
Octet 4	7:0	8 bits	**Command ID (CMD)**	Command ID
Octet 5-n	-	0 - 2048 Bytes	**Payload (PL)**	Up to 64 bytes of payload

This operational mode needs to be especially enabled in devices hardware configuration file (typically **hardware.xml**) and is not used by default. The default operational mode assumes a UART with flow control is used.

Below is a simple example which shows how a **System Get Info** command (Raw: 0x00 0x00 0x00 0x08) is sent using the BGAPI packet format.



Application
Stack
Command (Message : 0x04 0x00 0x00 0x00 0x08
Response (Message: 0x00...)

4.2 The BGLIB functions definition

Bluegiga provides a reference parser for the BGAPI protocol called the BGLIB. The BGLIB is an ANSI C implementation of BGAPI packet parser and it's provided in source code format with the Bluegiga Bluetooth Smart SDK. The purpose of the BGLIB is to simplify and speed up the development process and also to provide higher level, easier to use C functions and callbacks so the developers do not need to fully learn the raw BGAPI protocol.

In BGLIB all of the BGAPI commands are available as C functions and for the BGAPI responses and events there are callback handlers.

The BGLIB functions and callbacks are documented as show below:

C Functions

```
/* Function */
void ble_cmd_gap_connect_direct(
    bd_addr address ,
    uint8 addr_type ,
    uint16 conn_interval_min ,
    uint16 conn_interval_max ,
    uint16 timeout
);

/* Callback */
void ble_rsp_gap_connect_direct(
    uint16 result ,
    uint8 conn
);
```

The command parameters and return values are the same as used in the BGAPI protocol and they are not documented separately in the API reference section.

Callback programming

Callback programming is a style of computer programming, which allows lower layer of software to call functions defined on a higher layer. Callback is piece of code or a reference to a piece of code that is passed as an argument. The figure below illustrates the callback architecture used with BGLIB.

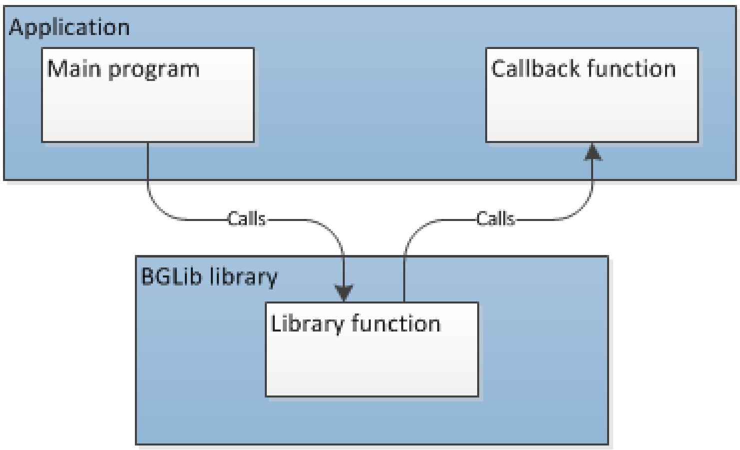


Figure: Callback arhitecture

If you are not familiar with callback programming a basic tutorial can for example be found from here:

http://www.codeguru.com/cpp/cpp/cpp_mfc/callbacks/article.php/c10557

4.3 The BGScript API definition

The BGScript functions are also documented in the API reference section. The format of the commands varies slightly from the BGLIB functions and instead of using callbacks the BGScript functions take the return values as parameters.

BGScript commands are documented as follows:

BGScript Functions

```
CALL gap_connect_direct(address ,addr_type ,conn_interval_min ,conn_interval_max ,timeout )(result
,conn )
```

The BGScript command parameters and return values are the same as used in the BGAPI binary protocol and they are not documented separately.

4.4 Data Types

The following data types are used in this documentation.

Table: Used data types

Type	Description	Example: Human readable	Example Packet data in hex
int8	signed integer stored in 1 byte twos complement form	-42	0xd6
uint8	unsigned integer stored in 1 byte	42	0x2a
uint16	unsigned integer stored in 2 bytes little endian format	1701	0xa5 0x06
uint32	unsigned integer stored in 4 bytes little endian format	1000000	0x40 0x42 0x0f 0x00
uint8array	byte array, first byte is array size	"Hello"	0x05 0x68 0x65 0x6c 0x6c 0x6f
bd_addr	Bluetooth address in little endian format	00:07:80:c0:ff:ee	0xee 0xff 0xc0 0x80 0x07 0x00

5 API Reference

This section of the document contains the actual API description, so the description of commands, responses, events and enumerations and their possible parameters and values. The high level categorization is made based on the command classes, which are:

Description	Explanation
Attribute database	Provides access to local GATT database and allows data to be written there for remote devices to access it.
Attribute client	Provides access to ATT protocol operationsa and allows a remote devices data to be accessed.
Connection	Provides access to *Bluetooth* connection and status management
Generic Access Profile	Provides access to *Bluetooth* GAP functions which allows one to control the local devices discoverability and connectability
Hardware	Provides access to hardware interfaces such as SPI, I2C, timers and ADC
Persistent Store	Provides access to the local persistence store, which allows data to be written and read to the devices flash.
Security Manager	*Provides* to *Bluetooth* security functions
System	Provides access to various system functions
Testing	Functions needed for *Bluetooth* conformance testing
Device Firmware Upgrade	Provides access to functions required for field firmware upgrades

Final section of the API reference contains description of the error codes categorized as follows:

Description
BGAPI errors
Bluetooth errors
Security manager errors
Attribute protocols errors

5.1 Attribute Client

The Attribute Client class implements the *Bluetooth* Smart Attribute Protocol (ATT) and provides access to the ATT protocol methods. The Attribute Client class can be used to discover services and characteristics from the ATT server, read and write values and manage indications and notifications.

5.1.1 Commands

Attribute Client commands

Attribute Write

This command can be used to write an attributes value on a remote device. In order to write the value of an attribute a *Bluetooth* connection must exists and you need to know the handle of the attribute you want to write.

A successful attribute write will be acknowledged by the remote device and this will generate an event attclient_procedure_completed. The acknowledgement should happen within a 30 second window or otherwise the Bluetooth connection will be dropped.

The data payload for the Attribute Write command can be up to 20 bytes.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x05	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**atthandle**	Attribute handle to write to
7	**uint8array**	**data**	Attribute value

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x05	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0** : write was successful Otherwise error occurred

Table: EVENTS

Event	Description
attclient procedure_completed	This event is generated when the write operation has been acknowledged by remote device.

C Functions

```
/* Function */
void ble_cmd_attclient_attribute_write(
    uint8 connection,
    uint16 atthandle,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_attclient_attribute_write_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_attribute_write(
    const struct ble_msg_attclient_attribute_write_rsp_t * msg
)
```

BGScript Functions

```
call attclient_attribute_write(connection, atthandle, data_len, data_data)(connection, result)
```

Execute Write

This command can be used to execute or cancel a previously queued prepare_write command on a remote device.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x0A	method	Message ID
4	**uint8**	**connection**	Connection Handle
5	**uint8**	**commit**	**1:** commits queued writes **0**: cancels queued writes

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x0A	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
attclient procedure_completed	Write operation has been acknowledged by remote end

C Functions

```
/* Function */
void ble_cmd_attclient_execute_write(
    uint8 connection,
    uint8 commit
);

/* Callback */
struct ble_msg_attclient_execute_write_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_execute_write(
    const struct ble_msg_attclient_execute_write_rsp_t * msg
)
```

BGScript Functions

```
call attclient_execute_write(connection, commit)(connection, result)
```

Find By Type Value

This command can be used to find specific attributes on a remote device based on their 16-bit UUID value and value. The search can be limited by a starting and ending handle values.

The command returns the handles of all attributes matching the type (UUID) and value.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x08	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**start**	First requested handle number
7 - 8	**uint16**	**end**	Last requested handle number
9 - 10	**uint16**	**uuid**	2 octet UUID to find
11	**uint8array**	**value**	Attribute value to find

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0** : the operation was successful Otherwise error occurred

Table: EVENTS

Event	Description
attclient group_found	Attributes found
attclient procedure_completed	Procedure has completed and new procedure can be started on GATT server

C Functions

```
/* Function */
void ble_cmd_attclient_find_by_type_value(
    uint8 connection,
    uint16 start,
    uint16 end,
    uint16 uuid,
    uint8 value_len,
    const uint8* value_data
);

/* Callback */
struct ble_msg_attclient_find_by_type_value_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_find_by_type_value(
    const struct ble_msg_attclient_find_by_type_value_rsp_t * msg
)
```

BGScript Functions

```
call attclient_find_by_type_value(connection, start, end, uuid, value_len, value_data)(connection,
result)
```

Find Information

This command is used to discover attribute handles and their types (UUIDs) in a given handle range.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x03	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**start**	First attribute handle
7 - 8	**uint16**	**end**	Last attribute handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x03	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0:** if the command was successful Otherwise error occurred

Table: EVENTS

Event	Description
attclient find_information_found	Handle, type - mapping found
attclient procedure_completed	Find information procedure has completed

C Functions

```
/* Function */
void ble_cmd_attclient_find_information(
    uint8 connection,
    uint16 start,
    uint16 end
);

/* Callback */
struct ble_msg_attclient_find_information_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_find_information(
    const struct ble_msg_attclient_find_information_rsp_t * msg
)
```

BGScript Functions

```
call attclient_find_information(connection, start, end)(connection, result)
```

Indicate Confirm

This command can be used to send a acknowledge a received indication from a remote device. This function allows the application to manually confirm the indicated values instead of the *Bluetooth* smart stack automatically doing it. The benefit of this is extra reliability since the application can for example store the received value on the flash memory before confirming the indication to the remote device.

> ⚠ In order to use this feature the manual indication acknowledgements must be enabled to the application configuration file (config.xml).

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x07	method	Message ID
4	**uint8**	**connection**	Connection Handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x07	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_attclient_indicate_confirm(
    uint8 connection
);

/* Callback */
struct ble_msg_attclient_indicate_confirm_rsp_t{
    uint16 result
}
void ble_rsp_attclient_indicate_confirm(
    const struct ble_msg_attclient_indicate_confirm_rsp_t * msg
)
```

BGScript Functions

```
call attclient_indicate_confirm(connection)(result)
```

Prepare Write

This command will send a prepare write request to a remote device for queued writes. Queued writes can for example be used to write large attribute values by transmitting the data in chunks using prepare write command.

Once the data has been transmitted with multiple prepare write commands the write must then be executed or canceled with Execute Write command, which if acknowledged by the remote device triggers a Procedure Completed event.

The example below shows how this approach can be used to write a 30-byte characteristic value:

1. **attclient_prepare_write(...., partial data)**
2. wait for rsp_attclient_prepare_write
3. wait for evt_attclient_procedure_completed
4. **attclient_prepare_write(...., partial data)**
5. wait for rsp_attclient_prepare_write
6. wait for evt_attclient_procedure_completed
7. **attclient_execute_write(1)**
8. wait for rsp_attclient_execute_write
9. wait for evt_attclient_procedure_completed

> It is not mandatory for an ATT server to support this command. It is only recommended to use this command to write long-attributes which do not fit in single ATT packet.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x06	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x09	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**atthandle**	Attribute handle
7 - 8	**uint16**	**offset**	Offset to write to
9	**uint8array**	**data**	Data to write Maximum amount of data that can be sent in single command is 18 bytes.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x09	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
attclient procedure_completed	Write operation has been acknowledged by remote end

C Functions

```
/* Function */
void ble_cmd_attclient_prepare_write(
    uint8 connection,
    uint16 atthandle,
    uint16 offset,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_attclient_prepare_write_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_prepare_write(
    const struct ble_msg_attclient_prepare_write_rsp_t * msg
)
```

BGScript Functions

```
call attclient_prepare_write(connection, atthandle, offset, data_len, data_data)(connection,
result)
```

Read By Group Type

This command reads the value of each attribute of a given type and in a given handle range.

The command is typically used for primary (UUID: 0x2800) and secondary (UUID: 0x2801) service discovery.

Discovered services are reported by Group Found event.

Finally when the procedure is completed a Procedure Completed event is generated.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x06	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**start**	First requested handle number
7 - 8	**uint16**	**end**	Last requested handle number
9	**uint8array**	**uuid**	Group UUID to find

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
attclient group_found	Attributes found
attclient procedure_completed	Procedure has completed and new procedure can be started on GATT server

C Functions

```
/* Function */
void ble_cmd_attclient_read_by_group_type(
    uint8 connection,
    uint16 start,
    uint16 end,
    uint8 uuid_len,
    const uint8* uuid_data
);

/* Callback */
struct ble_msg_attclient_read_by_group_type_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_read_by_group_type(
    const struct ble_msg_attclient_read_by_group_type_rsp_t * msg
)
```

BGScript Functions

```
call attclient_read_by_group_type(connection, start, end, uuid_len, uuid_data)(connection, result)
```

Read By Handle

This command reads a remote attribute's value with the given handle. Read by handle can be used to read attributes up to 22 bytes long.

For longer attributes Read Long command must be used.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x04	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**chrhandle**	Attribute handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x04	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0** : the command was successful Otherwise an error occurred

Table: EVENTS

Event	Description
attclient attribute_value	Attribute value received
attclient procedure_completed	ATT command failed

C Functions

```
/* Function */
void ble_cmd_attclient_read_by_handle(
    uint8 connection,
    uint16 chrhandle
);

/* Callback */
struct ble_msg_attclient_read_by_handle_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_read_by_handle(
    const struct ble_msg_attclient_read_by_handle_rsp_t * msg
)
```

BGScript Functions

```
call attclient_read_by_handle(connection, chrhandle)(connection, result)
```

Read By Type

The command reads the value of each attribute of a given type (UUID) and in a given attribute handle range.

The command can for example be used to discover the characteristic declarations (UUID: 0x2803) within a service.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x06	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**start**	First attribute handle
7 - 8	**uint16**	**end**	Last attribute handle
9	**uint8array**	**uuid**	Attribute type (UUID)

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	**0**: the command was successful Otherwise an error occurred

Table: EVENTS

Event	Description
attclient attribute_value	Attribute value read from GATT server
attclient procedure_completed	Returned if error occurred

C Functions

```
/* Function */
void ble_cmd_attclient_read_by_type(
    uint8 connection,
    uint16 start,
    uint16 end,
    uint8 uuid_len,
    const uint8* uuid_data
);

/* Callback */
struct ble_msg_attclient_read_by_type_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_read_by_type(
    const struct ble_msg_attclient_read_by_type_rsp_t * msg
)
```

BGScript Functions

```
call attclient_read_by_type(connection, start, end, uuid_len, uuid_data)(connection, result)
```

Read Long

This command can be used to read long attribute values, which are longer than 22 bytes and cannot be read with a simple Read by Handle command.

The command starts a procedure, where the client first sends a normal read command to the server and if the returned attribute value length is equal to MTU, the client will send further read long read requests until rest of the attribute is read.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x08	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**chrhandle**	Attribute handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x08	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	**0**: Command executed successfully **Non-zero** : An error occurred

Table: EVENTS

Event	Description
attclient attribute_value	Data received from remote end
attclient procedure_completed	Full attribute has read, or error occurred

C Functions

```
/* Function */
void ble_cmd_attclient_read_long(
    uint8 connection,
    uint16 chrhandle
);

/* Callback */
struct ble_msg_attclient_read_long_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_read_long(
    const struct ble_msg_attclient_read_long_rsp_t * msg
)
```

BGScript Functions

```
call attclient_read_long(connection, chrhandle)(connection, result)
```

Read Multiple

This command can be used to read multiple attributes from a server.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x0B	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8array**	**handles**	List of attribute handles to read from the remote device

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x0B	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
attclient read_multiple_response	Attribute data if command was succesful
attclient procedure_completed	Operation has failed

C Functions

```
/* Function */
void ble_cmd_attclient_read_multiple(
    uint8 connection,
    uint8 handles_len,
    const uint8* handles_data
);

/* Callback */
struct ble_msg_attclient_read_multiple_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_read_multiple(
    const struct ble_msg_attclient_read_multiple_rsp_t * msg
)
```

BGScript Functions

```
call attclient_read_multiple(connection, handles_len, handles_data)(connection, result)
```

Write Command

Writes the value of a remote devices attribute. The handle and the new value of the attribute are gives as parameters.

Write command will not be acknowledged by the remote device unlike Attribute Write.

The maximum data payload for Write Command is 20 bytes.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x06	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**atthandle**	Attribute handle to write
7	**uint8array**	**data**	Value for the attribute

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x06	method	Message ID
4	**uint8**	**connection**	Connection Handle
5 - 6	**uint16**	**result**	**0**: Command executed successfully **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_attclient_write_command(
    uint8 connection,
    uint16 atthandle,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_attclient_write_command_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_attclient_write_command(
    const struct ble_msg_attclient_write_command_rsp_t * msg
)
```

BGScript Functions

```
call attclient_write_command(connection, atthandle, data_len, data_data)(connection, result)
```

5.1.2 Enumerations

Attribute Client enumerations

Attribute Value Types

These enumerations are in the Attribute Client class

Table: VALUES

Value	Name	Description
0	attclient_attribute_value_type_read	Value was read
1	attclient_attribute_value_type_notify	Value was notified
2	attclient_attribute_value_type_indicate	Value was indicated
3	attclient_attribute_value_type_read_by_type	Value was read
4	attclient_attribute_value_type_read_blob	Value was part of a long attribute
5	attclient_attribute_value_type_indicate_rsp_req	Value was indicated and the remote device is waiting for a confirmation. Indicate Confirm command can be used to send a confirmation.

5.1.3 Events

Attribute Client events

Attribute Value

This event is produced at the GATT client side when an attribute value is passed from the GATT server to the GATT client. This event is for example produced after a successful Read by Handle operation or when an attribute is indicated or notified by the remote device.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x05	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x05	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**atthandle**	Attribute handle
7	**uint8**	**type**	Attribute type
8	**uint8array**	**value**	Attribute value (data)

C Functions

```
/* Callback */
struct ble_msg_attclient_attribute_value_evt_t{
    uint8 connection,
    uint16 atthandle,
    uint8 type,
    uint8 value_len,
    const uint8* value_data
}
void ble_evt_attclient_attribute_value(
    const struct ble_msg_attclient_attribute_value_evt_t * msg
)
```

BGScript Functions

```
event attclient_attribute_value(connection, atthandle, type, value_len, value_data)
```

Find Information Found

This event is generated when characteristics type mappings are found. This happens yypically after Find Information command has been issued to discover all attributes of a service.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x04	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x04	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**chrhandle**	Characteristics handle
7	**uint8array**	**uuid**	Characteristics type (UUID)

C Functions

```
/* Callback */
struct ble_msg_attclient_find_information_found_evt_t{
    uint8 connection,
    uint16 chrhandle,
    uint8 uuid_len,
    const uint8* uuid_data
}
void ble_evt_attclient_find_information_found(
    const struct ble_msg_attclient_find_information_found_evt_t * msg
)
```

BGScript Functions

```
event attclient_find_information_found(connection, chrhandle, uuid_len, uuid_data)
```

Group Found

This event is produced when an attribute group (a service) is found. Typically this event is produced after Read by Group Type command.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x06	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**start**	Starting handle
7 - 8	**uint16**	**end**	Ending handle Note: "end" is a reserved word and in BGScript so "end" cannot be used as such.
9	**uint8array**	**uuid**	UUID of a service Length is 0 if no services are found.

C Functions

```
/* Callback */
struct ble_msg_attclient_group_found_evt_t{
    uint8 connection,
    uint16 start,
    uint16 end,
    uint8 uuid_len,
    const uint8* uuid_data
}
void ble_evt_attclient_group_found(
    const struct ble_msg_attclient_group_found_evt_t * msg
)
```

BGScript Functions

```
event attclient_group_found(connection, start, end, uuid_len, uuid_data)
```

Indicated

This event is produced at the GATT server side when an attribute is successfully indicated to the GATT client.

This means the event is only produced at the GATT server if the indication is acknowledged by the GATT client (the remote device).

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**attrhandle**	Attribute handle

C Functions

```
/* Callback */
struct ble_msg_attclient_indicated_evt_t{
    uint8 connection,
    uint16 attrhandle
}
void ble_evt_attclient_indicated(
    const struct ble_msg_attclient_indicated_evt_t * msg
)
```

BGScript Functions

```
event attclient_indicated(connection, attrhandle)
```

Procedure Completed

This event is produced at the GATT client when an attribute protocol event is completed a and new operation can be issued.

This event is for example produced after an Attribute Write command is successfully used to write a value to a remote device.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x05	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x01	method	Message ID
4	**uint8**	**connection**	Object Handle
5 - 6	**uint16**	**result**	**0**: The operation was successful **Otherwise**: attribute protocol error code returned by remote device
7 - 8	**uint16**	**chrhandle**	Characteristic handle at which the event ended

C Functions

```
/* Callback */
struct ble_msg_attclient_procedure_completed_evt_t{
    uint8 connection,
    uint16 result,
    uint16 chrhandle
}
void ble_evt_attclient_procedure_completed(
    const struct ble_msg_attclient_procedure_completed_evt_t * msg
)
```

BGScript Functions

```
event attclient_procedure_completed(connection, result, chrhandle)
```

Read Multiple Response

This event is a response to a Read Multiple request.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x02	lolen	Minimum payload length
2	0x04	class	Message class: Attribute Client
3	0x06	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8array**	**handles**	List of attribute handles read from a remote device.

C Functions

```
/* Callback */
struct ble_msg_attclient_read_multiple_response_evt_t{
    uint8 connection,
    uint8 handles_len,
    const uint8* handles_data
}
void ble_evt_attclient_read_multiple_response(
    const struct ble_msg_attclient_read_multiple_response_evt_t * msg
)
```

BGScript Functions

```
event attclient_read_multiple_response(connection, handles_len, handles_data)
```

5.2 Attribute Database

The Attribute Database class provides methods to read and write attributes to the local devices attribute database. This class is usually only needed on sensor devices (Attribute server) for example to update attribute values to the local database based on the sensor readings. A remote device then can access the GATT database and these values over a *Bluetooth* connection.

5.2.1 Commands

Attribute database commands

Read

The command reads the given attribute's value from the local database. There is a 32-byte limit in the amount of data that can be read at a time. In order to read larger values multiple read commands must be used with the offset properly used.

For example to read a 64 bytes attribute:

1. Read first 32 bytes using offset 0
2. Read second 32 bytes using offset 32

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x01	method	Message ID
4 - 5	**uint16**	**handle**	Handle of the attribute to read
6 - 7	**uint16**	**offset**	Offset to read from. Maximum of 32 bytes can be read at a time.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x07	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x01	method	Message ID
4 - 5	**uint16**	**handle**	Handle of the attribute which was read
6 - 7	**uint16**	**offset**	Offset read from
8 - 9	**uint16**	**result**	**0** : the read was successful **Non-zero**: An error occurred
10	**uint8array**	**value**	Value of the attribute

C Functions

```
/* Function */
void ble_cmd_attributes_read(
    uint16 handle,
    uint16 offset
);

/* Callback */
struct ble_msg_attributes_read_rsp_t{
    uint16 handle,
    uint16 offset,
    uint16 result,
    uint8 value_len,
    const uint8* value_data
}
void ble_rsp_attributes_read(
    const struct ble_msg_attributes_read_rsp_t * msg
)
```

BGScript Functions

```
call attributes_read(handle, offset)(handle, offset, result, value_len, value_data)
```

Read Type

This command reads the given attribute's type (UUID) from the local database.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x02	method	Message ID
4 - 5	**uint16**	**handle**	Handle of the attribute to read

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x05	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x02	method	Message ID
4 - 5	**uint16**	**handle**	Handle of the attribute which was read
6 - 7	**uint16**	**result**	**0:** if the read was successful **Non-zero**: An error occurred
8	**uint8array**	**value**	Value of the attribute type (UUID)

C Functions

```
/* Function */
void ble_cmd_attributes_read_type(
    uint16 handle
);

/* Callback */
struct ble_msg_attributes_read_type_rsp_t{
    uint16 handle,
    uint16 result,
    uint8 value_len,
    const uint8* value_data
}
void ble_rsp_attributes_read_type(
    const struct ble_msg_attributes_read_type_rsp_t * msg
)
```

BGScript Functions

```
call attributes_read_type(handle)(handle, result, value_len, value_data)
```

Send--attributes

This command will send an attribute, identified by handle, notification or an indication to remote device(s), but does not modify the value in the local GATT database.

If this the attribute, identified by handle, does not have notification or indication property or no remote device listens for notifications or indications of this attribute an error will be returned.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x05	method	Message ID
4	**uint8**	**connection**	Connection handle to send to. Use 0xFF to send to all connected clients which have subscribed to receive the notifications or indications. An error is returned as soon as the first failed transmission occurs.
5 - 6	**uint16**	**handle**	Attribute handle to send.
7	**uint8array**	**value**	Data to send.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x05	method	Message ID
4 - 5	**uint16**	**result**	Command result **0:** notification or indication was successfully sent **Otherwise:** An error occurred

C Functions

```
/* Function */
void ble_cmd_attributes_send(
    uint8 connection,
    uint16 handle,
    uint8 value_len,
    const uint8* value_data
);

/* Callback */
struct ble_msg_attributes_send_rsp_t{
    uint16 result
}
void ble_rsp_attributes_send(
    const struct ble_msg_attributes_send_rsp_t * msg
)
```

BGScript Functions

```
call attributes_send(connection, handle, value_len, value_data)(result)
```

User Read Response

This command is used to respond to an attribute Read request by a remote device, but only for attributes which have been configured with the user property. Attributes which have the user property enabled allow the attribute value to be requested from the application instead of the *Bluetooth* Smart stack automatically responding with the data in it's local GATT database.

This command is normally used in response to a User Read Request event, which is generated when a remote device tries to read an attribute with a user property enabled.

The response to User Read Request events must happen within 30 seconds or otherwise a timeout will occur.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x03	method	Message ID
4	**uint8**	**connection**	Connection handle to respond to
5	**uint8**	**att_error**	**0:** User Read Request is responded with data. In case of an error an application specific error code can be sent.
6	**uint8array**	**value**	Data to send

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x03	method	Message ID

C Functions

```
/* Function */
void ble_cmd_attributes_user_read_response(
    uint8 connection,
    uint8 att_error,
    uint8 value_len,
    const uint8* value_data
);

/* Callback *
void ble_rsp_attributes_user_read_response(
    const void *nul
)
```


BGScript Functions

```
call attributes_user_read_response(connection, att_error, value_len, value_data)
```

User Write Response

This command is used by the GATT server to acknowledge to the remote device that the attribute's value was written. This feature again allows the user application to acknowledged the attribute write operations instead of the *Bluetooth* Smart stack doing it automatically.

The command should be used when a Value event is received where the reason why value has changed corresponds to attributes_attribute_change_reason_write_request_user.

This response must be sent within 30 seconds or otherwise a timeout will occur.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x04	method	Message ID
4	**uint8**	**connection**	Connection handle to respond to
5	**uint8**	**att_error**	Attribute error code to send if an error occurs. **0x0:** Write was accepted **0x80-0x9F**: Reserved for user defined error codes

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x04	method	Message ID

C Functions

```
/* Function */
void ble_cmd_attributes_user_write_response(
    uint8 connection,
    uint8 att_error
);

/* Callback *
void ble_rsp_attributes_user_write_response(
    const void *nul
)
```

BGScript Functions

```
call attributes_user_write_response(connection, att_error)
```

Write

This command writes an attribute's value to the local database.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x00	method	Message ID
4 - 5	**uint16**	**handle**	Handle of the attribute to write
6	**uint8**	**offset**	Attribute offset to write data
7	**uint8array**	**value**	Value of the attribute to write

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x00	method	Message ID
4 - 5	**uint16**	**result**	**0**: the write was successful **Non-zero**: An error occured

C Functions

```
/* Function */
void ble_cmd_attributes_write(
    uint16 handle,
    uint8 offset,
    uint8 value_len,
    const uint8* value_data
);

/* Callback */
struct ble_msg_attributes_write_rsp_t{
    uint16 result
}
void ble_rsp_attributes_write(
    const struct ble_msg_attributes_write_rsp_t * msg
)
```

BGScript Functions

```
call attributes_write(handle, offset, value_len, value_data)(result)
```

5.2.2 Enumerations

Attribute Database enumerations

Attribute Change Reason

This enumeration contains the reason for an attribute value change.

Table: VALUES

Value	Name	Description
0	attributes_attribute_change_reason_write_request	Value was written by remote device using write request
1	attributes_attribute_change_reason_write_command	Value was written by remote device using write command
2	attributes_attribute_change_reason_write_request_user	Local attribute value was written by the remote device, but the *Bluetooth* Smart stack is waiting for the write to be confirmed by the application. User Write Response command should be used to send the confirmation. For this reason to appear the attribute in the GATT database must have the user property enabled. See *Profile Toolkit Developer Guide* for more information how to enable the user property for an attribute.

Attribute Status Flags

Attribute status flags

Table: VALUES

Value	Name	Description
1	attributes_attribute_status_flag_notify	Notifications are enabled
2	attributes_attribute_status_flag_indicate	Indications are enabled

5.2.3 Events

Attribute Database events

Status

This event indicates attribute status flags have changed

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x02	method	Message ID
4 - 5	**uint16**	**handle**	Attribute handle
6	**uint8**	**flags**	Attribute status flags See: Attribute Status Flags

C Functions

```
/* Callback */
struct ble_msg_attributes_status_evt_t{
    uint16 handle,
    uint8 flags
}
void ble_evt_attributes_status(
    const struct ble_msg_attributes_status_evt_t * msg
)
```

BGScript Functions

```
event attributes_status(handle, flags)
```


User Read Request

This event is generated when a remote device tries to read an attribute which has the user property enabled. This event should be responded within 30 seconds with User Read Response command either containing the data or an error code.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x06	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection ID which requested attribute
5 - 6	**uint16**	**handle**	Attribute handle requested
7 - 8	**uint16**	**offset**	Attribute offset to send data from
9	**uint8**	**maxsize**	Maximum data size to respond with If more data is sent than indicated by this parameter, the extra bytes will be ignored.

C Functions

```
/* Callback */
struct ble_msg_attributes_user_read_request_evt_t{
    uint8 connection,
    uint16 handle,
    uint16 offset,
    uint8 maxsize
}
void ble_evt_attributes_user_read_request(
    const struct ble_msg_attributes_user_read_request_evt_t * msg
)
```

BGScript Functions

```
event attributes_user_read_request(connection, handle, offset, maxsize)
```

Value

This event is produced at the GATT server when a local attribute value was written by a remote device.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x07	lolen	Minimum payload length
2	0x02	class	Message class: Attribute Database
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8**	**reason**	Reason why value has changed see: enum Attribute Change Reason
6 - 7	**uint16**	**handle**	Attribute handle, which was changed
8 - 9	**uint16**	**offset**	Offset into attribute value where data starts
10	**uint8array**	**value**	Attribute value

C Functions

```
/* Callback */
struct ble_msg_attributes_value_evt_t{
    uint8 connection,
    uint8 reason,
    uint16 handle,
    uint16 offset,
    uint8 value_len,
    const uint8* value_data
}
void ble_evt_attributes_value(
    const struct ble_msg_attributes_value_evt_t * msg
)
```

BGScript Functions

```
event attributes_value(connection, reason, handle, offset, value_len, value_data)
```

5.3 Connection

The Connection class provides methods to manage *Bluetooth* connections and query their statuses.

5.3.1 Commands

Connection class commands

Disconnect

This command disconnects an active *Bluetooth* connection.

When link is disconnected a Disconnected event is produced.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle to close

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0**: disconnection procedure successfully started **Non-zero**: An error occurred

Table: EVENTS

Event	Description
connection disconnected	Sent after connection has disconnected

C Functions

```
/* Function */
void ble_cmd_connection_disconnect(
    uint8 connection
);

/* Callback */
struct ble_msg_connection_disconnect_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_connection_disconnect(
    const struct ble_msg_connection_disconnect_rsp_t * msg
)
```

BGScript Functions

```
call connection_disconnect(connection)(connection, result)
```

Get Rssi

This commands returns the Receiver Signal Strength Indication (RSSI) of an active connection.

> ⚠ At -38 dBm the BLE112 receiver is saturated. The measurement value may depend on the used hardware and design.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**int8**	**rssi**	RSSI value of the connection in dBm. **Range**: -103 to -38

C Functions

```
/* Function */
void ble_cmd_connection_get_rssi(
    uint8 connection
);

/* Callback */
struct ble_msg_connection_get_rssi_rsp_t{
    uint8 connection,
    int8 rssi
}
void ble_rsp_connection_get_rssi(
    const struct ble_msg_connection_get_rssi_rsp_t * msg
)
```

BGScript Functions

```
call connection_get_rssi(connection)(connection, rssi)
```

Get Status

This command returns the status of the given connection.

Status is returned in a Status event.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x07	method	Message ID
4	**uint8**	**connection**	Connection handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x01	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x07	method	Message ID
4	**uint8**	**connection**	Connection handle

Table: EVENTS

Event	Description
connection status	Reports the status of a connection

C Functions

```
/* Function */
void ble_cmd_connection_get_status(
    uint8 connection
);

/* Callback */
struct ble_msg_connection_get_status_rsp_t{
    uint8 connection
}
void ble_rsp_connection_get_status(
    const struct ble_msg_connection_get_status_rsp_t * msg
)
```

BGScript Functions

```
call connection_get_status(connection)(connection)
```

Update

This command updates the connection parameters of a given connection. The parameters have the same meaning and follow the same rules as for the GAP class command: Connect Direct.

If this command is issued at a master device, it will send parameter update request to the *Bluetooth* link layer.

On the other hand if this command is issued at a slave device, it will send L2CAP connection parameter update request to the master, which may either accept or reject it.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x09	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**interval_min**	Minimum connection interval (units of 1.25ms)
7 - 8	**uint16**	**interval_max**	Maximum connection interval (units of 1.25ms)
9 - 10	**uint16**	**latency**	Slave latency which defines how many connections intervals a slave may skip.
11 - 12	**uint16**	**timeout**	Supervision timeout (units of 10ms)

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0** : the update was successful **Non-zero**: An error occurred.

C Functions

```
/* Function */
void ble_cmd_connection_update(
    uint8 connection,
    uint16 interval_min,
    uint16 interval_max,
    uint16 latency,
    uint16 timeout
);

/* Callback */
struct ble_msg_connection_update_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_connection_update(
    const struct ble_msg_connection_update_rsp_t * msg
)
```

BGScript Functions

```
call connection_update(connection, interval_min, interval_max, latency, timeout)(connection,
result)
```

Version Update

This command requests a version exchange of a given connection.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x03	method	Message ID
4	**uint8**	**connection**	Connection handle

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x03	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**result**	**0** : the request was successful **Non-zero**: An error occurred

Table: EVENTS

Event	Description
connection version_ind	Sent after receiving version indication from other end

C Functions

```
/* Function */
void ble_cmd_connection_version_update(
    uint8 connection
);

/* Callback */
struct ble_msg_connection_version_update_rsp_t{
    uint8 connection,
    uint16 result
}
void ble_rsp_connection_version_update(
    const struct ble_msg_connection_version_update_rsp_t * msg
)
```

BGScript Functions

```
call connection_version_update(connection)(connection, result)
```

5.3.2 Enumerations

Connection class enumerations

Connection Status Flags

The possible connection status flags are described in the table below. The flags field is a bit mask, so multiple flags can be set at a time. If the bit is 1 the flag is active and if the bit is 0 the flag is inactive.

Table: VALUES

Value	Name	Description
bit 0	connection_connected	This status flag tells the connection exists to a remote device.
bit 1	connection_encrypted	This flag tells the connection is encrypted.
bit 2	connection_completed	Connection completed flag, which is used to tell a new connection has been created.
bit 3	connection_parameters_change	This flag tells that connection parameters have changed and. It is set when connection parameters have changed due to a link layer operation.

5.3.3 Events

Connection class events

Disconnected

This event is produced when a *Bluetooth* connection is disconnected.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x04	method	Message ID
4	**uint8**	**connection**	Connection handle
5 - 6	**uint16**	**reason**	Disconnection reason code **0** : disconnected by local user

C Functions

```
/* Callback */
struct ble_msg_connection_disconnected_evt_t{
    uint8 connection,
    uint16 reason
}
void ble_evt_connection_disconnected(
    const struct ble_msg_connection_disconnected_evt_t * msg
)
```

BGScript Functions

```
event connection_disconnected(connection, reason)
```

Feature Ind

This event indicates the remote devices features.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x02	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x02	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8array**	**features**	CtrData field from LL_FEATURE_RSP - packet

C Functions

```
/* Callback */
struct ble_msg_connection_feature_ind_evt_t{
    uint8 connection,
    uint8 features_len,
    const uint8* features_data
}
void ble_evt_connection_feature_ind(
    const struct ble_msg_connection_feature_ind_evt_t * msg
)
```

BGScript Functions

```
event connection_feature_ind(connection, features_len, features_data)
```

Status

This event indicates the connection status and parameters.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x10	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x00	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8**	**flags**	Connection status flags use connstatus-enumerator
6 - 11	**bd_addr**	**address**	Remote devices Bluetooth address
12	**uint8**	**address_type**	Remote address type see: Bluetooth Address Types--gap
13 - 14	**uint16**	**conn_interval**	Current connection interval (units of 1.25ms)
15 - 16	**uint16**	**timeout**	Current supervision timeout (units of 10ms)
17 - 18	**uint16**	**latency**	Slave latency which tells how many connection intervals the slave may skip.
19	**uint8**	**bonding**	Bonding handle if the device has been bonded with. **Otherwise**: 0xFF

C Functions

```
/* Callback */
struct ble_msg_connection_status_evt_t{
    uint8 connection,
    uint8 flags,
    bd_addr address,
    uint8 address_type,
    uint16 conn_interval,
    uint16 timeout,
    uint16 latency,
    uint8 bonding
}
void ble_evt_connection_status(
    const struct ble_msg_connection_status_evt_t * msg
)
```

BGScript Functions

```
event connection_status(connection, flags, address, address_type, conn_interval, timeout, latency,
bonding)
```

Version Ind

This event indicates the remote devices version.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x06	lolen	Minimum payload length
2	0x03	class	Message class: Connection
3	0x01	method	Message ID
4	**uint8**	**connection**	Connection handle
5	**uint8**	**vers_nr**	*Bluetooth* controller specification version
6 - 7	**uint16**	**comp_id**	Manufacturer of the *Bluetooth* controller
8 - 9	**uint16**	**sub_vers_nr**	*Bluetooth* controller version

C Functions

```
/* Callback */
struct ble_msg_connection_version_ind_evt_t{
    uint8 connection,
    uint8 vers_nr,
    uint16 comp_id,
    uint16 sub_vers_nr
}
void ble_evt_connection_version_ind(
    const struct ble_msg_connection_version_ind_evt_t * msg
)
```

BGScript Functions

```
event connection_version_ind(connection, vers_nr, comp_id, sub_vers_nr)
```

5.4 Generic Access Profile

The Generic Access Profile (GAP) class provides methods to control the *Bluetooth* GAP level functionality of the local device. The GAP call for example allows remote device discovery, connection establishment and local devices connection and discovery modes. The GAP class also allows the control of local devices privacy modes.

5.4.1 Commands

Generic Access Profile class commands

Connect Direct

This command will start the GAP direct connection establishment procedure to a dedicated *Bluetooth* Smart device.

The *Bluetooth* module will enter a state where it continuously scans for the connectable advertisement packets from the remote device which matches the Bluetooth address gives as a parameter. Upon receiving the advertisement packet, the module will send a connection request packet to the target device to imitate a *Bluetooth* connection. A successful connection will bi indicated by a Status event.

The connection establishment procedure can be cancelled with End Procedure command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x0F	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x03	method	Message ID
4 - 9	**bd_addr**	**address**	Bluetooth address of the target device
10	**uint8**	**addr_type**	see: Bluetooth Address Types
11 - 12	**uint16**	**conn_interval_min**	Minimum connection interval (in units of 1.25ms). **Range: 6 - 3200** The lowest possible connection interval is 7.50ms and the largest is 4000ms. When more then one connection is supported the connection interval values (minimum and maximum) used in **all** connection commands must be divisible by ***connection count**** ***2.5ms***
13 - 14	**uint16**	**conn_interval_max**	Maximum connection interval (in units of 1.25ms). **Range: 6 - 3200** Must be equal or bigger than minimum connection interval.
15 - 16	**uint16**	**timeout**	Supervision timeout (in units of 10ms). The supervision timeout defines how long the devices can be out of range before the connection is closed. **Range: 10 - 3200** Minimum time for the supervision timeout is 100ms and maximum value: 32000ms. Supervision timeout must also be equal or grater than maximum connection interval.

Byte	Type	Name	Description
17 - 18	**uint16**	**latency**	This parameter configures the slave latency. Slave latency defines how many connection intervals a slave device can skip. Increasing slave latency will decrease the energy consumption of the slave in scenarios where slave does not have data to send at every connection interval. **Range: 0 - 500** **0** : Slave latency is disabled. **Example:** Connection interval is 10ms and slave latency is 9: this means that the slave is allowed to communicate every 100ms, but it can communicate every 10ms if needed. **Note:** ***Slave Latency*** x ***Connection interval*** can NOT be higher than supervision timeout.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x03	method	Message ID
4 - 5	**uint16**	**result**	**0** : procedure was successfully started **Non-zero**: An error occurred
6	**uint8**	**connection_handle**	Connection handle that is reserved for new connection

Table: EVENTS

Event	Description
connection status	Sent after connection is established

C Functions

```
/* Function */
void ble_cmd_gap_connect_direct(
    bd_addr address,
    uint8 addr_type,
    uint16 conn_interval_min,
    uint16 conn_interval_max,
    uint16 timeout,
    uint16 latency
);

/* Callback */
struct ble_msg_gap_connect_direct_rsp_t{
    uint16 result,
    uint8 connection_handle
}
void ble_rsp_gap_connect_direct(
    const struct ble_msg_gap_connect_direct_rsp_t * msg
)
```

BGScript Functions

```
call gap_connect_direct(address, addr_type, conn_interval_min, conn_interval_max, timeout,
latency)(result, connection_handle)
```

Connect Selective

This command will start the GAP direct connection establishment procedure to a set of dedicated Bluetooth Smart devices.

When this command is issued the the *Bluetooth* module will enter a state where it scans connectable advertisement packets from the remote devices which are registered in the local white list. Upon receiving an advertisement packet from one of the registered devices, the module will send a connection request to this device, and a successful connection will produce a connection status event.

The connect selective command can be cancelled with End Procedure command.

When in *Initiating State* there are no scan response events*.*

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x08	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x05	method	Message ID
4 - 5	**uint16**	**conn_interval_min**	Minimum connection interval (in units of 1.25ms). **Range: 6 - 3200** The lowest possible connection interval is 7.50ms and the largest is 4000ms. When more then one connection is supported the connection interval values (minimum and maximum) used in **all** connection commands must be divisible by ***connection count**** ***2.5ms***
6 - 7	**uint16**	**conn_interval_max**	Maximum connection interval (in units of 1.25ms). **Range: 6 - 3200** Must be equal or bigger than minimum connection interval.
8 - 9	**uint16**	**timeout**	Supervision timeout (in units of 10ms). The supervision timeout defines how long the devices can be out of range before the connection is closed. **Range: 10 - 3200** Minimum time for the supervision timeout is 100ms and maximum value: 32000ms. Supervision timeout must also be equal or grater than maximum connection interval.

Byte	Type	Name	Description
10 - 11	**uint16**	**latency**	This parameter configures the slave latency. Slave latency defines how many connection intervals a slave device can skip. Increasing slave latency will decrease the energy consumption of the slave in scenarios where slave does not have data to send at every connection interval. **Range: 0 - 500** **0** : Slave latency is disabled. **Example:** Connection interval is 10ms and slave latency is 9: this means that the slave is allowed to communicate every 100ms, but it can communicate every 10ms if needed. **Note:** ***Slave Latency*** x ***Connection interval*** can NOT be higher than supervision timeout.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x05	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was executed successfully **Non-zero**: An error occurred
6	**uint8**	**connection_handle**	Connection handle reserved for connection

Table: EVENTS

Event	Description
connection status	Sent after connected to any whitelisted device

C Functions

```
/* Function */
void ble_cmd_gap_connect_selective(
    uint16 conn_interval_min,
    uint16 conn_interval_max,
    uint16 timeout,
    uint16 latency
);

/* Callback */
struct ble_msg_gap_connect_selective_rsp_t{
    uint16 result,
    uint8 connection_handle
}
void ble_rsp_gap_connect_selective(
    const struct ble_msg_gap_connect_selective_rsp_t * msg
)
```

BGScript Functions

```
call gap_connect_selective(conn_interval_min, conn_interval_max, timeout, latency)(result,
connection_handle)
```

Discover

This command starts the GAP discovery procedure to scan for advertising devices i.e. to perform a device discovery.

Scanning parameters can be configured with the Set Scan Parameters command before issuing this command.

To cancel on an ongoing discovery process use the End Procedure command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x02	method	Message ID
4	**uint8**	**mode**	see:GAP Discover Mode

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x02	method	Message ID
4 - 5	**uint16**	**result**	**0**: Scan procedure was successfully started **Non-zero**: An error occurred

Table: EVENTS

Event	Description
gap scan_response	Discovered device scan response

C Functions

```
/* Function */
void ble_cmd_gap_discover(
    uint8 mode
);

/* Callback */
struct ble_msg_gap_discover_rsp_t{
    uint16 result
}
void ble_rsp_gap_discover(
    const struct ble_msg_gap_discover_rsp_t * msg
)
```

BGScript Functions

```
call gap_discover(mode)(result)
```

End Procedure

This command ends the current GAP discovery procedure and stop the scanning of advertising devices.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x04	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x04	method	Message ID
4 - 5	**uint16**	**result**	**0**: the command was successful **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_gap_end_procedure(
    void
);

/* Callback */
struct ble_msg_gap_end_procedure_rsp_t{
    uint16 result
}
void ble_rsp_gap_end_procedure(
    const struct ble_msg_gap_end_procedure_rsp_t * msg
)
```

BGScript Functions

```
call gap_end_procedure()(result)
```

Set Adv Data

This commands set advertisement or scan response data used in the advertisement and scan response packets. The command allows application specific data to be broadcasts either in advertisement or scan response packets.

The data set with this command is only used when the GAP discoverable mode is set to gap_user_data.

Notice that advertisement or scan response data must be formatted in accordance to the Bluetooth Core Specification. See BLUETOOTH SPECIFICATION Version 4.0 [Vol 3 - Part C - Chapter 11].

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x09	method	Message ID
4	**uint8**	**set_scanrsp**	Advertisement data type **0 : sets advertisement data** **: sets scan response data**
5	**uint8array**	**adv_data**	Advertisement data to send

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x09	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_gap_set_adv_data(
    uint8 set_scanrsp,
    uint8 adv_data_len,
    const uint8* adv_data_data
);

/* Callback */
struct ble_msg_gap_set_adv_data_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_adv_data(
    const struct ble_msg_gap_set_adv_data_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_adv_data(set_scanrsp, adv_data_len, adv_data_data)(result)
```

Set Adv Parameters

This command is used to set the advertising parameters.

Example: If the minimum advertisement interval is 40ms and the maximum advertisement interval is 100ms then the real advertisement interval will be mostly the middle value (70ms) plus a randomly added 20ms delay, which needs to be added according to the Bluetooth specification.

> If you are currently advertising, then any changes set using this command will not take effect until you stop and re-start advertising.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x08	method	Message ID
4 - 5	**uint16**	**adv_interval_min**	Minimum advertisement interval in units of 625us **Range: 0x20 to 0x4000** **Default: 0x200** (320ms) **Explanation:** 0x200 = 512 512 * 625us = 320000us = 320ms
6 - 7	**uint16**	**adv_interval_max**	Maximum advertisement interval in units of 625us. **Range: 0x20 to 0x4000** **Default: 0x200** (320ms)
8	**uint8**	**adv_channels**	A bit mask to identify which of the three advertisement channels are used. **Examples:** **0x07:** All three channels are used **0x03**: Advertisement channels 37 and 38 are used. **0x04:** Only advertisement channel 39 is used

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x08	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was successfully executed **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_gap_set_adv_parameters(
    uint16 adv_interval_min,
    uint16 adv_interval_max,
    uint8 adv_channels
);

/* Callback */
struct ble_msg_gap_set_adv_parameters_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_adv_parameters(
    const struct ble_msg_gap_set_adv_parameters_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_adv_parameters(adv_interval_min, adv_interval_max, adv_channels)(result)
```

Set Directed Connectable Mode

This command sets device to Directed Connectable mode.

In this mode the device uses fast advertisement procedure for the first 1.28 seconds, after which the device enters a non-connectable mode. If the device implements the Peripheral Preferred Connection Parameters characteristic in its GAP service the parameters defined by this characteristic will be used for the connection.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x07	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x0A	method	Message ID
4 - 9	**bd_addr**	**address**	Bluetooth address to connect to
10	**uint8**	**addr_type**	Address type to connect see:enum gap_address_type

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x0A	method	Message ID
4 - 5	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
connection status	Sent after connection is established

C Functions

```
/* Function */
void ble_cmd_gap_set_directed_connectable_mode(
    bd_addr address,
    uint8 addr_type
);

/* Callback */
struct ble_msg_gap_set_directed_connectable_mode_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_directed_connectable_mode(
    const struct ble_msg_gap_set_directed_connectable_mode_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_directed_connectable_mode(address, addr_type)(result)
```

Set Filtering

This command can be used to set scan, connection, and advertising filtering parameters based on the local devices white list. See also Whitelist Append command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x06	method	Message ID
4	**uint8**	**scan_policy**	see: enum gap_scan_policy
5	**uint8**	**adv_policy**	see: enum gap_advertising_policy
6	**uint8**	**scan_duplicate_filtering**	**0:** Do not filter duplicate advertisers **1:** Filter duplicates

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x06	method	Message ID
4 - 5	**uint16**	**result**	**0**: The command was successfully executed **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_gap_set_filtering(
    uint8 scan_policy,
    uint8 adv_policy,
    uint8 scan_duplicate_filtering
);

/* Callback */
struct ble_msg_gap_set_filtering_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_filtering(
    const struct ble_msg_gap_set_filtering_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_filtering(scan_policy, adv_policy, scan_duplicate_filtering)(result)
```

Set Mode

This command configures the current GAP discoverability and connectability mode. It can be used to enable advertisements and/or allow connection. The command can also be used used to stop advertising.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x01	method	Message ID
4	**uint8**	**discover**	see:GAP Discoverable Mode
5	**uint8**	**connect**	see:GAP Connectable Mode

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x01	method	Message ID
4 - 5	**uint16**	**result**	**0**: the command was executed successfully **Non-zero**: An error occurred

Table: EVENTS

Event	Description
connection status	Sent if device was connectable and master connected to device

C Functions

```
/* Function */
void ble_cmd_gap_set_mode(
    uint8 discover,
    uint8 connect
);

/* Callback */
struct ble_msg_gap_set_mode_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_mode(
    const struct ble_msg_gap_set_mode_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_mode(discover, connect)(result)
```

Set Privacy Flags

This command sets GAP central/peripheral privacy flags.

By setting for example peripheral_privacy to 1, the *Bluetooth* stack will automatically generate a resolvable random private address for the advertising packets every time the Set Mode command is used to enter advertising mode.

> ⚠ It is not recommended to adjust peripheral privacy unless mandatory by the application, because not all *Bluetooth* implementations can decode resolvable private addresses.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x00	method	Message ID
4	**uint8**	**peripheral_privacy**	**1**: enable peripheral privacy **0**: disable peripheral privacy Any other value will have no effect on flag
5	**uint8**	**central_privacy**	**1**: enable central privacy **0**: disable central privacy Any other value will have no effect on flag

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x00	method	Message ID

C Functions

```
/* Function */
void ble_cmd_gap_set_privacy_flags(
    uint8 peripheral_privacy,
    uint8 central_privacy
);

/* Callback *
void ble_rsp_gap_set_privacy_flags(
    const void *nul
)
```

BGScript Functions

```
call gap_set_privacy_flags(peripheral_privacy, central_privacy)
```

Set Scan Parameters

This command sets the scan parameters which affect how other *Bluetooth* Smart devices are discovered.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x07	method	Message ID
4 - 5	**uint16**	**scan_interval**	Scan interval defines the interval when scanning is re-started in units of 625us **Range: 0x4 - 0x4000** **Default: 0x4B** (75ms) After every scan interval the scanner will change the frequency it operates at at it will cycle through all the three advertisements channels in a round robin fashion. According to the *Bluetooth* specification all three channels must be used by a scanner.
6 - 7	**uint16**	**scan_window**	Scan Window defines how long time the scanner will listen on a certain frequency and try to pick up advertisement packets. Scan window is defined as units of 625us **Range: 0x4 - 0x4000** **Default: 0x32** (50 ms) Scan windows must be equal or smaller than scan interval If scan window is equal to the scan interval value, then the *Bluetooth* module will be scanning at a 100% duty cycle. If scan window is half of the scan interval value, then the *Bluetooth* module will be scanning at a 50% duty cycle.
8	**uint8**	**active**	**1:** Active scanning is used. When an advertisement packet is received the *Bluetooth* stack will send a scan request packet to the advertiser to try and read the scan response data. **0:** Passive scanning is used. No scan request is made.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x07	method	Message ID
4 - 5	**uint16**	**result**	**0**: The command was executed successfully **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_gap_set_scan_parameters(
    uint16 scan_interval,
    uint16 scan_window,
    uint8 active
);

/* Callback */
struct ble_msg_gap_set_scan_parameters_rsp_t{
    uint16 result
}
void ble_rsp_gap_set_scan_parameters(
    const struct ble_msg_gap_set_scan_parameters_rsp_t * msg
)
```

BGScript Functions

```
call gap_set_scan_parameters(scan_interval, scan_window, active)(result)
```

5.4.2 Enumerations

Generic Access Profile class enumerations

AD_FLAGS

Scan header flags

Table: VALUES

Value	Name	Description
0x01	GAP_AD_FLAG_LIMITED_DISCOVERABLE	Limited discoverability
0x02	GAP_AD_FLAG_GENERAL_DISCOVERABLE	General discoverability
0x04	GAP_AD_FLAG_BREDR_NOT_SUPPORTED	BR/EDR not supported
0x10	GAP_AD_FLAG_SIMULTANEOUS_LEBREDR_CTRL	BR/EDR controller
0x20	GAP_AD_FLAG_SIMULTANEOUS_LEBREDR_HOST	BE/EDR host
0x1f	GAP_AD_FLAG_MASK	-

AD Type Flags

Table: VALUES

Value	Name	Description
0	gap_ad_type_none	
1	gap_ad_type_flags	
2	gap_ad_type_services_16bit_more	
3	gap_ad_type_services_16bit_all	
4	gap_ad_type_services_32bit_more	
5	gap_ad_type_services_32bit_all	
6	gap_ad_type_services_128bit_more	
7	gap_ad_type_services_128bit_all	
8	gap_ad_type_localname_short	
9	gap_ad_type_localname_complete	
10	gap_ad_type_txpower	

Advertising policy

Advertising policy

Table: VALUES

Value	Name	Description
0	gap_adv_policy_all	Respond to scan requests from any master, allow connection from any master (default)
1	gap_adv_policy_whitelist_scan	Respond to scan requests from whitelist only, allow connection from any
2	gap_adv_policy_whitelist_connect	Respond to scan requests from any, allow connection from whitelist only
3	gap_adv_policy_whitelist_all	Respond to scan requests from whitelist only, allow connection from whitelist only

Bluetooth Address Types

Bluetooth address types

Table: VALUES

Value	Name	Description
0	gap_address_type_public	Public Address
1	gap_address_type_random	Random Address

GAP Connectable Mode

GAP connectable modes

Table: VALUES

Value	Name	Description
0	gap_non_connectable	Not connectable
1	gap_directed_connectable	Directed Connectable
2	gap_undirected_connectable	Undirected connectable
3	gap_scannable_non_connectable	Same as non-connectable, but also supports ADV_SCAN_IND packets. Device accepts scan requests (active scanning) but is not connectable.

GAP Discoverable Mode

GAP discoverable modes

Table: VALUES

Value	Name	Description
0	gap_non_discoverable	Non-discoverable mode: the *LE Limited Discoverable Mode* and the *LE General Discoverable Mode* bits are NOT set in the *Flags* AD type. A master can still connect to the advertising slave in this mode.
1	gap_limited_discoverable	Discoverable using limited scanning mode: the advertisement packets will carry the *LE Limited Discoverable Mode* bit set in the *Flags* AD type.
2	gap_general_discoverable	Discoverable using general scanning mode: the advertisement packets will carry the *LE General Discoverable Mode* bit set in the *Flags* AD type.
3	gap_broadcast	Same as gap_non_discoverable above.
4	gap_user_data	In this advertisement the advertisement and scan response data defined by user will be used. The user is responsible of building the advertisement data so that it also contains the appropriate desired *Flags* AD type.
0x80	gap_enhanced_broadcasting	When turning the most highest bit on in GAP discoverable mode, the devices that perform send a scan request packet to the advertiser are reported back to the application through Scan Response event. This is so called Enhanced Broadcasting mode.

GAP Discover Mode

GAP Discoverable modes

Table: VALUES

Value	Name	Description
0	gap_discover_limited	Discover only limited discoverable devices, that is, Slaves which have the *LE Limited Discoverable Mode* bit set in the *Flags* AD type of their advertisement packets.
1	gap_discover_generic	Discover limited and generic discoverable devices, that is, Slaves which have the *LE Limited Discoverable Mode* or the *LE General Discoverable Mode* bit set in the *Flags* AD type of their advertisement packets.
2	gap_discover_observation	Discover all devices regardless of the *Flags* AD type, so also devices in non-discoverable mode will be reported to host.

SCAN_HEADER_FLAGS

Scan header flags

Table: VALUES

Value	Name	Description
0	GAP_SCAN_HEADER_ADV_IND	Connectable undirected advertising event
1	GAP_SCAN_HEADER_ADV_DIRECT_IND	Connectable directed advertising event
2	GAP_SCAN_HEADER_ADV_NONCONN_IND	Non-connectable undirected advertising event
3	GAP_SCAN_HEADER_SCAN_REQ	Scanner wants information from Advertiser
4	GAP_SCAN_HEADER_SCAN_RSP	Advertiser gives more information to Scanner
5	GAP_SCAN_HEADER_CONNECT_REQ	Initiator wants to connect to Advertiser
6	GAP_SCAN_HEADER_ADV_DISCOVER_IND	Non-connectable undirected advertising event

Scan Policy

Scan Policy

Table: VALUES

Value	Name	Description
0	gap_scan_policy_all	Accept All advertisement Packets (default)
1	gap_scan_policy_whitelist	Ignore advertisement packets from remote slaves not in the running whitelist

5.4.3 Events

Generic Access Profile class events

Scan Response

This is a scan response event.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x0B	lolen	Minimum payload length
2	0x06	class	Message class: Generic Access Profile
3	0x00	method	Message ID
4	**int8**	**rssi**	RSSI value (dBm) **Range:** -103 to -38
5	**uint8**	**packet_type**	Scan response header **0**: Connectable Advertisement packet **2**: Non Connectable Advertisement packet **4**: Scan response packet **6**: Discoverable advertisement packet
6 - 11	**bd_addr**	**sender**	*Advertisers Bluetooth* address
12	**uint8**	**address_type**	Advertiser address type **1**: random address **0**: public address
13	**uint8**	**bond**	Bond handle if there is known bond for this device, 0xff otherwise
14	**uint8array**	**data**	Scan response data

C Functions

```
/* Callback */
struct ble_msg_gap_scan_response_evt_t{
    int8 rssi,
    uint8 packet_type,
    bd_addr sender,
    uint8 address_type,
    uint8 bond,
    uint8 data_len,
    const uint8* data_data
}
void ble_evt_gap_scan_response(
    const struct ble_msg_gap_scan_response_evt_t * msg
)
```

BGScript Functions

```
event gap_scan_response(rssi, packet_type, sender, address_type, bond, data_len, data_data)
```

5.5 Hardware

The Hardware class provides methods to access the local devices hardware interfaces such as : A/D converters, IO and timers, I2C interface etc.

5.5.1 Commands

Hardware class commands

ADC Read

This command reads the devices local A/D converter.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x02	method	Message ID
4	**uint8**	**input**	Selects the ADC input. **0x0**: AIN0 (pin 0 of port P0, denoted as A0 in the ADC row of datasheet's table 3) **0x1**: AIN1 **0x2**: AIN2 **0x3**: AIN3 **0x4**: AIN4 **0x5**: AIN5 **0x6**: AIN6 **0x7**: AIN7 **0x8**: AIN0--AIN1 differential **0x9**: AIN2--AIN3 differential **0xa**: AIN4--AIN5 differential **0xb**: AIN6--AIN7 differential **0xc**: GND **0xd**: Reserved **0xe**: Temperature sensor **0xf**: VDD/3
5	**uint8**	**decimation**	Select resolution and conversion rate for conversion, result is always stored in MSB bits. **0**: 7 effective bits **1**: 9 effective bits **2**: 10 effective bits **3**: 12 effective bits
6	**uint8**	**reference_selection**	Selects the reference for the ADC. Reference corresponds to the maximum allowed input value. **0**: Internal reference (1.24V) **1**: External reference on AIN7 pin **2**: AVDD pin **3**: External reference on AIN6--AIN7 differential input

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x02	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was executed successfully **Non-zero**: An error occurred

Table: EVENTS

Event	Description
hardare adc result	ADC read operation has completed

C Functions

```
/* Function */
void ble_cmd_hardware_adc_read(
    uint8 input,
    uint8 decimation,
    uint8 reference_selection
);

/* Callback */
struct ble_msg_hardware_adc_read_rsp_t{
    uint16 result
}
void ble_rsp_hardware_adc_read(
    const struct ble_msg_hardware_adc_read_rsp_t * msg
)
```

BGScript Functions

```
call hardware_adc_read(input, decimation, reference_selection)(result)
```

Analog Comparator Config IRQ

This command configures analog comparator interrupts. Before enabling this interrupt, analog comparator has to be first enabled with Analog Comparator Enable command.

Analog comparator interrupts are generated by default on rising edge, i.e. when condition $V_+ > V_-$ becomes true. It is also possible to configure the opposite functionality, i.e. interrupts are generated on falling edge when $V_+ < V_-$ becomes true. The interrupt direction may be configured with Io Port Irq Direction command, by setting I/O-port 0 direction. Please note that this configuration affects both analog comparator interrupt direction and all I/O-port 0 pin interrupt directions.

Analog comparator interrupts are automatically disabled once triggered , so that a high frequency signal doesn't cause unintended consequences. Continuous operation may be achieved by re-enabling the interrupt as soon as the Analog Comparator Status event has been received.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x12	method	Message ID
4	**uint8**	**enabled**	**1**: enable interrupts **0**: disable interrupts

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x12	method	Message ID
4 - 5	**uint16**	**result**	Command result Zero on success, error code otherwise

Table: EVENTS

event	Description
hardware_analog_comparator_status	Sent after comparator output changes in the configured direction

C Functions

```
/* Function */
void ble_cmd_hardware_analog_comparator_config_irq(
    uint8 enabled
);

/* Callback */
struct ble_msg_hardware_analog_comparator_config_irq_rsp_t{
    uint16 result
}
void ble_rsp_hardware_analog_comparator_config_irq(
    const struct ble_msg_hardware_analog_comparator_config_irq_rsp_t * msg
)
```

BGScript Functions

```
call hardware_analog_comparator_config_irq(enabled)(result)
```

Analog Comparator Enable

Enables or disables the analog comparator. Analog comparator has to be enabled prior using any other analog comparator commands.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x10	method	Message ID
4	**uint8**	**enable**	**1**: enable **0**: disable

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x10	method	Message ID

C Functions

```
/* Function */
void ble_cmd_hardware_analog_comparator_enable(
    uint8 enable
);

/* Callback *
void ble_rsp_hardware_analog_comparator_enable(
    const void *nul
)
```

BGScript Functions

```
call hardware_analog_comparator_enable(enable)
```

Analog Comparator Read

The command reads analog comparator output. Before using this command, analog comparator has to be enabled with Analog Comparator Enable command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x11	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x11	method	Message ID
4 - 5	**uint16**	**result**	Command result Zero on success, error code otherwise
6	**uint8**	**output**	Analog comparator output **1**: if $V_+ > V_-$ **0**: if $V_+ < V_-$

C Functions

```
/* Function */
void ble_cmd_hardware_analog_comparator_read(
    void
);

/* Callback */
struct ble_msg_hardware_analog_comparator_read_rsp_t{
    uint16 result,
    uint8 output
}
void ble_rsp_hardware_analog_comparator_read(
    const struct ble_msg_hardware_analog_comparator_read_rsp_t * msg
)
```

BGScript Functions

```
call hardware_analog_comparator_read()(result, output)
```

I2C Read

The command reads data from I2C bus.

BLE112 module: uses bit-bang method and only master-mode is supported in current firmwares, I2C CLK is fixed to P1_7 and I2C DATA to P1_6 (pull-up must be enabled on both pins).

BLE113 and BLE121LR modules: only master-mode is supported in current firmware build, I2C pins are 14 (I2C CLK) and 15 (I2C DATA) as seen in the datasheet, operates at 267kHz.

To convert a 7-bit I2C address to an 8-bit one, shift left by one bit. For example, a 7-bit address of 0x40 (dec 64) would be used as 0x80 (dec 128).

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0A	method	Message ID
4	**uint8**	**address**	I2C's 8-bit slave address according to the note above. Keep read/write bit (LSB) set to zero, as the firmware will set it automatically.
5	**uint8**	**stop**	If nonzero Send I2C stop condition after transmission
6	**uint8**	**length**	Number of bytes to read

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0A	method	Message ID
4 - 5	**uint16**	**result**	Command result
6	**uint8array**	**data**	Data read

C Functions

```
/* Function */
void ble_cmd_hardware_i2c_read(
    uint8 address,
    uint8 stop,
    uint8 length
);

/* Callback */
struct ble_msg_hardware_i2c_read_rsp_t{
    uint16 result,
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_hardware_i2c_read(
    const struct ble_msg_hardware_i2c_read_rsp_t * msg
)
```

BGScript Functions

```
call hardware_i2c_read(address, stop, length)(result, data_len, data_data)
```

I2C Write

Write data to I2C bus.

BLE112: uses bit-bang method, only master-mode is supported in current firmwares, I2C CLK is fixed to P1_7 and I2C DATA to P1_6 (pull-up must be enabled on both pins).

BLE113: only master-mode is supported in current firmwares, I2C pins are 14 (I2C CLK) and 15 (I2C DATA) as seen in the datasheet, operates at 267kHz.

> To convert a 7-bit address to an 8-bit one, shift left by one bit. For example, a 7-bit address of 0x40 (dec 64) would be used as 0x80 (dec 128).

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0B	method	Message ID
4	**uint8**	**address**	I2C's 8-bit slave address according to the note above. Keep read/write bit (LSB) set to zero, as the firmware will set it automatically.
5	**uint8**	**stop**	If nonzero Send I2C stop condition after transmission
6	**uint8array**	**data**	Data to write

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0B	method	Message ID
4	**uint8**	**written**	Bytes written

C Functions

```
/* Function */
void ble_cmd_hardware_i2c_write(
    uint8 address,
    uint8 stop,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_hardware_i2c_write_rsp_t{
    uint8 written
}
void ble_rsp_hardware_i2c_write(
    const struct ble_msg_hardware_i2c_write_rsp_t * msg
)
```

BGScript Functions

```
call hardware_i2c_write(address, stop, data_len, data_data)(written)
```

IO Port Config Direction

The command configiures I/O-port directions

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x03	method	Message ID
4	**uint8**	**port**	I/0 PORT index: **0, 1 or 2**
5	**uint8**	**direction**	Bitmask for each individual pin direction **bit0** means input (default) **bit1** means output **Example:** To configure ll port's pins as output use **0xff**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x03	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was executed successfully **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_config_direction(
    uint8 port,
    uint8 direction
);

/* Callback */
struct ble_msg_hardware_io_port_config_direction_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_config_direction(
    const struct ble_msg_hardware_io_port_config_direction_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_config_direction(port, direction)(result)
```

IO Port Config Function

This command configures the I/O-ports function.

If bit is set in function parameter then the corresponding I/O port is set to peripheral function, otherwise it is general purpose I/O pin.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x04	method	Message ID
4	**uint8**	**port**	I/O port: **0,1 or 2**
5	**uint8**	**function**	peripheral selection bit for pins

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x04	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was executed successfully **Non-zero**: An error occurred

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_config_function(
    uint8 port,
    uint8 function
);

/* Callback */
struct ble_msg_hardware_io_port_config_function_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_config_function(
    const struct ble_msg_hardware_io_port_config_function_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_config_function(port, function)(result)
```

IO Port Config IRQ

> **Deprecation warning**
>
> This command is deprecated in and Io Port Irq Enable and Io Port Irq Direction commands should be used instead.

This command configures the locals I/O-port interrupts.

> Currently interrupts can not be enabled on I/O-port 2.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x00	method	Message ID
4	**uint8**	**port**	I/O port selection **Values: 0 or 1**
5	**uint8**	**enable_bits**	A bit mask which tells which I/O generate an interrupt **bit 0**: Interrupt is enabled **bit 1**: Interrupt is disabled
6	**uint8**	**falling_edge**	Interrupt sense for port. **0 : rising edge** **1 : falling edge** **Note**: affects all IRQ enabled pins on the port

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x00	method	Message ID
4 - 5	**uint16**	**result**	error code, 0-success

Table: EVENTS

Event	Description
hardware io_port_status	Sent after pin change edge detected, and pin irq is enabled

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_config_irq(
    uint8 port,
    uint8 enable_bits,
    uint8 falling_edge
);

/* Callback */
struct ble_msg_hardware_io_port_config_irq_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_config_irq(
    const struct ble_msg_hardware_io_port_config_irq_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_config_irq(port, enable_bits, falling_edge)(result)
```

IO Port Config Pull

Configure I/O-port pull-up/pull-down

⚠ Pins P1_0 and P1_1 do not have pull-up/pull-down.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x05	method	Message ID
4	**uint8**	**port**	I/O port select: **0, 1 or 2**
5	**uint8**	**tristate_mask**	If this bit is set, disabled pull on pin
6	**uint8**	**pull_up**	**1:** pull all port's pins up **0:** pull all port's pins down

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x05	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was successful **Otherwise:** An error occurred

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_config_pull(
    uint8 port,
    uint8 tristate_mask,
    uint8 pull_up
);

/* Callback */
struct ble_msg_hardware_io_port_config_pull_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_config_pull(
    const struct ble_msg_hardware_io_port_config_pull_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_config_pull(port, tristate_mask, pull_up)(result)
```

IO Port IRQ Direction

Set I/O-port interrupt direction. The direction applies for every pin in the given I/O-port.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0F	method	Message ID
4	**uint8**	**port**	I/O Port Values: **0 or 1**
5	**uint8**	**falling_edge**	Interrupt edge direction for port **0**: rising edge **1**: falling edge

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0F	method	Message ID
4 - 5	**uint16**	**result**	Command result Zero on success, error code otherwise

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_irq_direction(
    uint8 port,
    uint8 falling_edge
);

/* Callback */
struct ble_msg_hardware_io_port_irq_direction_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_irq_direction(
    const struct ble_msg_hardware_io_port_irq_direction_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_irq_direction(port, falling_edge)(result)
```

IO Port Irq Enable

Enable I/O-port interrupts. When enabled, I/O-port interrupts are triggered on either rising or falling edge. The direction when the interrupt occurs may be configured with IO Port Irq Direction command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0E	method	Message ID
4	**uint8**	**port**	I/O Port **Value: 0 or 1**
5	**uint8**	**enable_bits**	Interrupt enable mask for pins **bit0** means interrupt is disabled **bit1** means interrupt is enabled **Example:** To enable interrupts an all pins use **0xFF**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0E	method	Message ID
4 - 5	**uint16**	**result**	Command result Zero on success, error code otherwise

Table: EVENTS

event	Description
hardware_io_port_status	Sent after pin change edge is detected and pin IRQ is enabled

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_irq_enable(
    uint8 port,
    uint8 enable_bits
);

/* Callback */
struct ble_msg_hardware_io_port_irq_enable_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_irq_enable(
    const struct ble_msg_hardware_io_port_irq_enable_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_irq_enable(port, enable_bits)(result)
```

IO Port Read

Read I/O-port

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x07	method	Message ID
4	**uint8**	**port**	I/O port to read Values: **0,1 or 2**
5	**uint8**	**mask**	Bit mask to tell which I/O pins to read **Example:** To read the status of all IO pins use **0xFF**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x07	method	Message ID
4 - 5	**uint16**	**result**	error code, 0-success
6	**uint8**	**port**	I/O port read
7	**uint8**	**data**	I/O port pin state

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_read(
    uint8 port,
    uint8 mask
);

/* Callback */
struct ble_msg_hardware_io_port_read_rsp_t{
    uint16 result,
    uint8 port,
    uint8 data
}
void ble_rsp_hardware_io_port_read(
    const struct ble_msg_hardware_io_port_read_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_read(port, mask)(result, port, data)
```

IO Port Write

Write I/O-port statuses

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x06	method	Message ID
4	**uint8**	**port**	I/O port to write to **Values: 0,1 or 2**
5	**uint8**	**mask**	Bit mask to tell which I/O pins to write **Example:** To write the status of all IO pins use **0xFF**
6	**uint8**	**data**	Bit mask to tell which state to write **bit0**: I/O is disabled **bit1**: I/O is enabled **Example:** To enable all IO pins use **0xFF**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x06	method	Message ID
4 - 5	**uint16**	**result**	error code, 0-success

C Functions

```
/* Function */
void ble_cmd_hardware_io_port_write(
    uint8 port,
    uint8 mask,
    uint8 data
);

/* Callback */
struct ble_msg_hardware_io_port_write_rsp_t{
    uint16 result
}
void ble_rsp_hardware_io_port_write(
    const struct ble_msg_hardware_io_port_write_rsp_t * msg
)
```

BGScript Functions

```
call hardware_io_port_write(port, mask, data)(result)
```

Set RXgain

This command sets the radio receiver (RX) sensitivity to either high (default) or standard. The exact sensitivity value is dependent on the used hardware (refer to the appropriate data sheet).

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x13	method	Message ID
4	**uint8**	**gain**	**0**: standard gain **1**: high gain (default)

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x13	method	Message ID

C Functions

```
/* Function */
void ble_cmd_hardware_set_rxgain(
    uint8 gain
);

/* Callback *
void ble_rsp_hardware_set_rxgain(
    const void *nul
)
```

BGScript Functions

```
call hardware_set_rxgain(gain)
```

Set Soft Timer

This command configures the local software timer. The timer is 22 bits so the maximum value with BLE112 is 2^22 = 4194304/32768Hz = 128 seconds. With BLED112 USB dongle the maximum value is 2^22 = 4194304/32000Hz = 131 seconds.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x06	lolen	Minimum payload length

Byte	Type	Name	Description
2	0x07	class	Message class: Hardware
3	0x01	method	Message ID
4 - 7	**uint32**	**time**	Timer interrupt period in units of local crystal frequency. **time :** 1/32768 seconds for modules where the external sleep oscillator must be enabled. **time :** 1/32000 seconds for the dongle where internal RC oscillator is used. If time is 0, scheduled timer is removed.
8	**uint8**	**handle**	Handle that is sent back within triggered event at timeout
9	**uint8**	**single_shot**	Timer mode. **0 :** Repeating timeout: the timer event is triggered at intervals defined with **time**. The stack only supports one repeating timer at a time for reliability purposes. Starting a repeating soft timer removes the current one if any. **1 :** Single timeout: the timer event is triggered only once after a period defined with **time**.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x01	method	Message ID
4 - 5	**uint16**	**result**	Command result

Table: EVENTS

Event	Description
hardware soft_timer	Sent after specified interval

C Functions

```
/* Function */
void ble_cmd_hardware_set_soft_timer(
    uint32 time,
    uint8 handle,
    uint8 single_shot
);

/* Callback */
struct ble_msg_hardware_set_soft_timer_rsp_t{
    uint16 result
}
void ble_rsp_hardware_set_soft_timer(
    const struct ble_msg_hardware_set_soft_timer_rsp_t * msg
)
```

BGScript Functions

```
call hardware_set_soft_timer(time, handle, single_shot)(result)
```

Set TXpower

Re-configure TX output power.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0C	method	Message ID
4	**uint8**	**power**	TX output power level to use **Range:** **0 to 15** with the BLE112 and the BLED112 **0 to 14** with the BLE113 **0 to 9** with the BLE121LR For more information, refer to the <txpower> tag in the hardware.xml configuration file.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0C	method	Message ID

C Functions

```
/* Function */
void ble_cmd_hardware_set_txpower(
    uint8 power
);

/* Callback *
void ble_rsp_hardware_set_txpower(
    const void *nul
)
```

BGScript Functions

```
call hardware_set_txpower(power)
```

SPI Config

The command configures the SPI interface

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x06	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x08	method	Message ID
4	**uint8**	**channel**	USART channel Values: **0 or 1**
5	**uint8**	**polarity**	Clock polarity Values: **0 or 1**
6	**uint8**	**phase**	Clock phase Values: **0 or 1**
7	**uint8**	**bit_order**	Endianness **0: LSB** **1: MSB**
8	**uint8**	**baud_e**	baud rate exponent value
9	**uint8**	**baud_m**	baud rate mantissa value

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x08	method	Message ID
4 - 5	**uint16**	**result**	error code, 0-success

C Functions

```
/* Function */
void ble_cmd_hardware_spi_config(
    uint8 channel,
    uint8 polarity,
    uint8 phase,
    uint8 bit_order,
    uint8 baud_e,
    uint8 baud_m
);

/* Callback */
struct ble_msg_hardware_spi_config_rsp_t{
    uint16 result
}
void ble_rsp_hardware_spi_config(
    const struct ble_msg_hardware_spi_config_rsp_t * msg
)
```

BGScript Functions

```
call hardware_spi_config(channel, polarity, phase, bit_order, baud_e, baud_m)(result)
```

SPI Transfer

This command is used to transfer SPI data when in master mode. Maximum of 64 bytes can be transferred at a time.

> ⚠ Slave select pin is not controlled automatically when transferring data while in SPI master mode, so it must be controlled by the application using normal GPIO control commands like IO Port Write command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x09	method	Message ID
4	**uint8**	**channel**	SPI channel **Value:** 0 or 1
5	**uint8array**	**data**	Data to transmit Maximum length is 64 bytes

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x09	method	Message ID
4 - 5	**uint16**	**result**	error code **0**: Success **Otherwise**: an error occurred
6	**uint8**	**channel**	SPI channel used **Value:** 0 or 1
7	**uint8array**	**data**	data received from SPI

C Functions

```
/* Function */
void ble_cmd_hardware_spi_transfer(
    uint8 channel,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_hardware_spi_transfer_rsp_t{
    uint16 result,
    uint8 channel,
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_hardware_spi_transfer(
    const struct ble_msg_hardware_spi_transfer_rsp_t * msg
)
```

BGScript Functions

```
call hardware_spi_transfer(channel, data_len, data_data)(result, channel, data_len, data_data)
```

Timer Comparator

Set comparator for timer channel.

This command may be used to generate e.g. PWM signals with hardware timer. More information on different comparator modes and their usage may be found from Texas Instruments CC2540 User's Guide (SWRU191B), section 9.8 Output Compare Mode.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0D	method	Message ID
4	**uint8**	**timer**	Timer
5	**uint8**	**channel**	Timer channel
6	**uint8**	**mode**	Comparator mode
7 - 8	**uint16**	**comparator_value**	Comparator value

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x0D	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_hardware_timer_comparator(
    uint8 timer,
    uint8 channel,
    uint8 mode,
    uint16 comparator_value
);

/* Callback */
struct ble_msg_hardware_timer_comparator_rsp_t{
    uint16 result
}
void ble_rsp_hardware_timer_comparator(
    const struct ble_msg_hardware_timer_comparator_rsp_t * msg
)
```

BGScript Functions

```
call hardware_timer_comparator(timer, channel, mode, comparator_value)(result)
```

USB Enable

This command activates (enable) or deactivates USB controller on the BLE112 Bluetooth Smart module. The USB controller is activated by default when USB is set on in the hardware configuration. On the other hand, the USB controller cannot be activated if the USB is not set on in the hardware configuration.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x14	method	Message ID
4	**uint8**	**enable**	**1:** enable USB **0:** disable USB

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x14	method	Message ID
4 - 5	**uint16**	**result**	Command result **0:** command executed succesfully **Otherwise**: an error occurred

C Functions

```
/* Function */
void ble_cmd_hardware_usb_enable(
    uint8 enable
);

/* Callback */
struct ble_msg_hardware_usb_enable_rsp_t{
    uint16 result
}
void ble_rsp_hardware_usb_enable(
    const struct ble_msg_hardware_usb_enable_rsp_t * msg
)
```

BGScript Functions

```
call hardware_usb_enable(enable)(result)
```

5.5.2 Events

Hardware class events

ADC Result

This events is produced when an A/D converter result is received.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x02	method	Message ID
4	**uint8**	**input**	A/D input from which value is received from **0x0**: AIN0 (pin 0 of port P0, denoted as A0 in the ADC row of datasheet's table 3) **0x1**: AIN1 **0x2**: AIN2 **0x3**: AIN3 **0x4**: AIN4 **0x5**: AIN5 **0x6**: AIN6 **0x7**: AIN7 **0x8**: AIN0--AIN1 differential **0x9**: AIN2--AIN3 differential **0xa**: AIN4--AIN5 differential **0xb**: AIN6--AIN7 differential **0xc**: GND **0xd**: Reserved **0xe**: Temperature sensor **0xf**: VDD/3
5 - 6	**int16**	**value**	A/D value. In the example case of 12 effective bits decimation, you will need to read the left-most 12 bits of the value to interpret it. It is a 12-bit 2's complement value left-aligned to the MSB of the 16-bit container, which means that negative values (which are uncommon but not impossible) are 0x8000 or higher, and positive values are 0x7FF0 or lower. Since it is only 12 bits, the last nibble will always be 0 (0xnnn0). You can divide the value by 16 (that is, bit-shift 4 bits to the right) to obtain the expected 12-bit value.

C Functions

```
/* Callback */
struct ble_msg_hardware_adc_result_evt_t{
    uint8 input,
    int16 value
}
void ble_evt_hardware_adc_result(
    const struct ble_msg_hardware_adc_result_evt_t * msg
)
```

BGScript Functions

```
event hardware_adc_result(input, value)
```

Analog Comparator Status

This event is produced when analog comparator output changes in the configured direction.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x05	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x03	method	Message ID
4 - 7	**uint32**	**timestamp**	Value of internal timer **Range:** 0 to 2^24-1
8	**uint8**	**output**	Analog comparator output **1:** if $V_+ > V_-$ **0:** if $V_+ < V_-$

C Functions

```
/* Callback */
struct ble_msg_hardware_analog_comparator_status_evt_t{
    uint32 timestamp,
    uint8 output
}
void ble_evt_hardware_analog_comparator_status(
    const struct ble_msg_hardware_analog_comparator_status_evt_t * msg
)
```

BGScript Functions

```
event hardware_analog_comparator_status(timestamp, output)
```

IO Port Status

This event is produced when I/O port status changes.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x07	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x00	method	Message ID
4 - 7	**uint32**	**timestamp**	Value of internal timer **Range** : 0 to 2^24-1
8	**uint8**	**port**	I/O port

Byte	Type	Name	Description
9	**uint8**	**irq**	I/O flags Tells which port caused interrupt (bitmask).
10	**uint8**	**state**	Current status of all I/Os in **port** (bitmask).

C Functions

```
/* Callback */
struct ble_msg_hardware_io_port_status_evt_t{
    uint32 timestamp,
    uint8 port,
    uint8 irq,
    uint8 state
}
void ble_evt_hardware_io_port_status(
    const struct ble_msg_hardware_io_port_status_evt_t * msg
)
```

BGScript Functions

```
event hardware_io_port_status(timestamp, port, irq, state)
```

Soft Timer

This event is produced when software timer interrupt is generated.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x01	lolen	Minimum payload length
2	0x07	class	Message class: Hardware
3	0x01	method	Message ID
4	**uint8**	**handle**	The software timer handle

C Functions

```
/* Callback */
struct ble_msg_hardware_soft_timer_evt_t{
    uint8 handle
}
void ble_evt_hardware_soft_timer(
    const struct ble_msg_hardware_soft_timer_evt_t * msg
)
```

BGScript Functions

```
event hardware_soft_timer(handle)
```

5.6 Persistent Store

The Persistent Store (PS) class provides methods to read write and dump the local devices parameters (PS keys). The persistent store is an abstract data storage on the local devices flash where an application can store data for future use.

5.6.1 Commands

Persistent Store class commands

Erase Page

The command erases a flash page which is allocated for user-data. Every page on the flash is 2kB in size starting from the first page indexed as 0.

When flash page is erased all bytes inside that page are set to 0xFF.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x06	method	Message ID
4	**uint8**	**page**	Index of memory page to erase **0: First 2kB flash page** **1: Next 2kB flash page** **etc.**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x06	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_flash_erase_page(
    uint8 page
);

/* Callback */
struct ble_msg_flash_erase_page_rsp_t{
    uint16 result
}
void ble_rsp_flash_erase_page(
    const struct ble_msg_flash_erase_page_rsp_t * msg
)
```

BGScript Functions

```
call flash_erase_page(page)(result)
```

PS Defrag

This command defragments the Persistent Store.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x00	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x00	method	Message ID

C Functions

```
/* Function */
void ble_cmd_flash_ps_defrag(
    void
);

/* Callback *
void ble_rsp_flash_ps_defrag(
    const void *nul
)
```

BGScript Functions

```
call flash_ps_defrag()
```

PS Dump

This command dumps all Persistent Store keys.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x01	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x01	method	Message ID

Table: EVENTS

Event	Description
flash ps_key	PS Key contents

C Functions

```
/* Function */
void ble_cmd_flash_ps_dump(
    void
);

/* Callback *
void ble_rsp_flash_ps_dump(
    const void *nul
)
```

BGScript Functions

```
call flash_ps_dump()
```

PS Erase All

This command erases all Persistent Store keys.

> ⚠ The software needs to be restarted after using this command. During the reset the device will generate missing encryption keys and update bonding cache.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x02	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x02	method	Message ID

C Functions

```
/* Function */
void ble_cmd_flash_ps_erase_all(
    void
);

/* Callback *
void ble_rsp_flash_ps_erase_all(
    const void *nul
)
```

BGScript Functions

```
call flash_ps_erase_all()
```

PS Erase

This command erases a Persistent Store key given as parameter.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x05	method	Message ID
4 - 5	**uint16**	**key**	Key to erase **Values: 0x8000 to 0x807F**

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x00	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x05	method	Message ID

C Functions

```
/* Function */
void ble_cmd_flash_ps_erase(
    uint16 key
);

/* Callback *
void ble_rsp_flash_ps_erase(
    const void *nul
)
```

BGScript Functions

```
call flash_ps_erase(key)
```

PS Load

This command reads a Persistent Store key from the local device.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x04	method	Message ID
4 - 5	**uint16**	**key**	Key to load **Values**: 0x8000 to 0x807F

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x04	method	Message ID
4 - 5	**uint16**	**result**	**0:** the read was successful **Otherwise**: An error occurred
6	**uint8array**	**value**	Key's value

C Functions

```
/* Function */
void ble_cmd_flash_ps_load(
    uint16 key
);

/* Callback */
struct ble_msg_flash_ps_load_rsp_t{
    uint16 result,
    uint8 value_len,
    const uint8* value_data
}
void ble_rsp_flash_ps_load(
    const struct ble_msg_flash_ps_load_rsp_t * msg
)
```

BGScript Functions

```
call flash_ps_load(key)(result, value_len, value_data)
```

PS Save

This command saves a Persistent Store (PS) key to the local device. The size of a single PS-key is 32 bytes and a total of 128 keys are available.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x03	method	Message ID
4 - 5	**uint16**	**key**	Key to save. **Values**: 0x8000 to 0x807F can be used for persistent storage of user data.
6	**uint8array**	**value**	Value of the key

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x03	method	Message ID
4 - 5	**uint16**	**result**	0 : the write was successful

C Functions

```
/* Function */
void ble_cmd_flash_ps_save(
    uint16 key,
    uint8 value_len,
    const uint8* value_data
);

/* Callback */
struct ble_msg_flash_ps_save_rsp_t{
    uint16 result
}
void ble_rsp_flash_ps_save(
    const struct ble_msg_flash_ps_save_rsp_t * msg
)
```

BGScript Functions

```
call flash_ps_save(key, value_len, value_data)(result)
```

Read Data

This command can be used to read data from user data area.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x08	method	Message ID
4 - 7	**uint32**	**address**	Offset in the user data space to start reading from in bytes. **0**: 1st byte
8	**uint8**	**length**	Length to read in bytes

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x08	method	Message ID
4	**uint8array**	**data**	Data read from flash. length is set to 0 if read address was invalid

C Functions

```
/* Function */
void ble_cmd_flash_read_data(
    uint32 address,
    uint8 length
);

/* Callback */
struct ble_msg_flash_read_data_rsp_t{
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_flash_read_data(
    const struct ble_msg_flash_read_data_rsp_t * msg
)
```

BGScript Functions

```
call flash_read_data(address, length)(data_len, data_data)
```

Write Data

This command can be used to write data to user data area.

Bits on the flash can only be turned from 1 to 0. To turn the bits from 0 to 1 the Erase Page command must be used. Notice that the erase page will erase the full 2kB flash page.

The amount of flash reserved for the user data needs to be defined in the application configuration file (config.xml).

The amount of available user flash depends on the hardware version and whether it has 128kbB or 256kB flash and also how much flash is left after the Bluetooth Smart stack, BGScript application and the GATT database. The BGBuild xompiler will show the flash consumption in it's output.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x07	method	Message ID
4 - 7	**uint32**	**address**	Offset in the user data space to write in bytes **0**: 1st byte
8	**uint8array**	**data**	Data to write

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x07	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_flash_write_data(
    uint32 address,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_flash_write_data_rsp_t{
    uint16 result
}
void ble_rsp_flash_write_data(
    const struct ble_msg_flash_write_data_rsp_t * msg
)
```

BGScript Functions

```
call flash_write_data(address, data_len, data_data)(result)
```

5.6.2 Events

Persistent Store class events

PS Key

This event is produced during a Persistent Store key dump which in launched with command PS Dump.

The event reporting a PS Key with address of 0xFFFF and empty value is always sent: it is meant to indicate that all existing PS Keys have been read.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x01	class	Message class: Persistent Store
3	0x00	method	Message ID
4 - 5	**uint16**	**key**	Persistent Store key ID **Values**: 0x8000 to 0x807F **0xFFFF**: All keys have been dumped
6	**uint8array**	**value**	Key value

C Functions

```
/* Callback */
struct ble_msg_flash_ps_key_evt_t{
    uint16 key,
    uint8 value_len,
    const uint8* value_data
}
void ble_evt_flash_ps_key(
    const struct ble_msg_flash_ps_key_evt_t * msg
)
```

BGScript Functions

```
event flash_ps_key(key, value_len, value_data)
```

5.7 Security Manager

The Security Manager (SM) class provides access to the Bluetooth low energy Security Manager and methods such as : bonding management and modes and encryption control.

5.7.1 Commands

Security Manager class commands

Delete Bonding

This command deletes a bonding from the local security database. There can be a maximum of 8 bonded devices stored at the same time, and one of them must be deleted if you need bonding with a 9th device.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x02	method	Message ID
4	**uint8**	**handle**	Bonding handle of a device. This handle can be obtained for example from events like: Scan Response Status If handle is 0xFF, all bondings have been deleted

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x02	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_sm_delete_bonding(
    uint8 handle
);

/* Callback */
struct ble_msg_sm_delete_bonding_rsp_t{
    uint16 result
}
void ble_rsp_sm_delete_bonding(
    const struct ble_msg_sm_delete_bonding_rsp_t * msg
)
```

BGScript Functions

```
call sm_delete_bonding(handle)(result)
```

Encrypt Start

This command starts the encryption for a given connection.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x00	method	Message ID
4	**uint8**	**handle**	Connection handle
5	**uint8**	**bonding**	Create bonding if devices are not already bonded **0**: Do not create bonding **1**: Creating bonding

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x03	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x00	method	Message ID
4	**uint8**	**handle**	Connection handle
5 - 6	**uint16**	**result**	0 : the encryption was started successfully

Table: EVENTS

Event	Description
sm_bonding_fail	Sent if encryption or bonding fails
connection_status	Sent when connectino is encrypted

C Functions

```
/* Function */
void ble_cmd_sm_encrypt_start(
    uint8 handle,
    uint8 bonding
);

/* Callback */
struct ble_msg_sm_encrypt_start_rsp_t{
    uint8 handle,
    uint16 result
}
void ble_rsp_sm_encrypt_start(
    const struct ble_msg_sm_encrypt_start_rsp_t * msg
)
```

BGScript Functions

```
call sm_encrypt_start(handle, bonding)(handle, result)
```

Get Bonds

This command lists all bonded devices. There can be a maximum of 8 bonded devices. The information related to the bonded devices is stored in the Flash memory, so it is persistent across resets and power-cycles.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x05	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x05	method	Message ID
4	**uint8**	**bonds**	Num of currently bonded devices

Table: EVENTS

Event	Description
sm bond_status	Bond status for each bonded device

C Functions

```
/* Function */
void ble_cmd_sm_get_bonds(
    void
);

/* Callback */
struct ble_msg_sm_get_bonds_rsp_t{
    uint8 bonds
}
void ble_rsp_sm_get_bonds(
    const struct ble_msg_sm_get_bonds_rsp_t * msg
)
```

BGScript Functions

```
call sm_get_bonds()(bonds)
```

Passkey Entry

This command is used to enter a passkey required for Man-in-the-Middle pairing. It should be sent as a response to Passkey Request event.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x04	method	Message ID
4	**uint8**	**handle**	Connection Handle
5 - 8	**uint32**	**passkey**	Passkey **Range**: 000000-999999

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x02	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x04	method	Message ID
4 - 5	**uint16**	**result**	Command result

' 0

C Functions

```
/* Function */
void ble_cmd_sm_passkey_entry(
    uint8 handle,
    uint32 passkey
);

/* Callback */
struct ble_msg_sm_passkey_entry_rsp_t{
    uint16 result
}
void ble_rsp_sm_passkey_entry(
    const struct ble_msg_sm_passkey_entry_rsp_t * msg
)
```

BGScript Functions

```
call sm_passkey_entry(handle, passkey)(result)
```

Set Bondable Mode

Set device to bondable mode

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x01	method	Message ID
4	**uint8**	**bondable**	Enables or disables bonding mode **0** : the device is not bondable **1** : the device is bondable

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x01	method	Message ID

C Functions

```
/* Function */
void ble_cmd_sm_set_bondable_mode(
    uint8 bondable
);

/* Callback *
void ble_rsp_sm_set_bondable_mode(
    const void *nul
)
```

BGScript Functions

```
call sm_set_bondable_mode(bondable)
```

Set OoB Data

This commands sets the Out-of-Band encryption data for a device.
Device does not allow any other kind of pairing except OoB if the OoB data is set.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x06	method	Message ID
4	**uint8array**	**oob**	The OoB data to set, which must be 16 or 0 octets long. If the data is empty it clears the previous OoB data.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x06	method	Message ID

C Functions

```
/* Function */
void ble_cmd_sm_set_oob_data(
    uint8 oob_len,
    const uint8* oob_data
);

/* Callback *
void ble_rsp_sm_set_oob_data(
    const void *nul
)
```

BGScript Functions

```
call sm_set_oob_data(oob_len, oob_data)
```

Set Parameters

This command is used to configure the local Security Manager and its features.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x03	method	Message ID
4	**uint8**	**mitm**	**1**: Man-in-the-middle protection required **0**: No Man-in-the-middle protection **Default:** 0
5	**uint8**	**min_key_size**	Minimum key size in Bytes **Range**: 7-16 **Default**: 7 (56bits)
6	**uint8**	**io_capabilities**	Configures the local devices I/O capabilities. See: SMP IO Capabilities for options. **Default**: No Input and No Output

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x03	method	Message ID

C Functions

```
/* Function */
void ble_cmd_sm_set_parameters(
    uint8 mitm,
    uint8 min_key_size,
    uint8 io_capabilities
);

/* Callback *
void ble_rsp_sm_set_parameters(
    const void *nul
)
```

BGScript Functions

```
call sm_set_parameters(mitm, min_key_size, io_capabilities)
```

Whitelist Bonds--sm

This command will add all bonded devices with a known public or static address to the local devices white list. Previous entries in the white list will be first cleared.

⚠ This command can't be used while advertising, scanning or being connected.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x07	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x07	method	Message ID
4 - 5	**uint16**	**result**	Command result
6	**uint8**	**count**	Number of whitelisted bonds

C Functions

```
/* Function */
void ble_cmd_sm_whitelist_bonds(
    void
);

/* Callback */
struct ble_msg_sm_whitelist_bonds_rsp_t{
    uint16 result,
    uint8 count
}
void ble_rsp_sm_whitelist_bonds(
    const struct ble_msg_sm_whitelist_bonds_rsp_t * msg
)
```

BGScript Functions

```
call sm_whitelist_bonds()(result, count)
```

5.7.2 Enumerations

Security Manager commands

Bonding Keys

Bonding information stored

Table: VALUES

Value	Name	Description
0x01	sm_bonding_key_ltk	LTK saved in master
0x02	sm_bonding_key_addr_public	Public Address
0x04	sm_bonding_key_addr_static	Static Address
0x08	sm_bonding_key_irk	Identity resolving key for resolvable private addresses
0x10	sm_bonding_key_edivrand	EDIV+RAND received from slave
0x20	sm_bonding_key_csrk	Connection signature resolving key
0x40	sm_bonding_key_masterid	EDIV+RAND sent to master

Table: VALUES

Value	Name	Description
1	sm_bonding_key_ltk	LTK saved in master
2	sm_bonding_key_addr_public	Public Address
4	sm_bonding_key_addr_static	Static Address
8	sm_bonding_key_irk	Identity resolving key for resolvable private addresses
16	sm_bonding_key_edivrand	EDIV+RAND received from slave
32	sm_bonding_key_csrk	Connection signature resolving key
64	sm_bonding_key_masterid	EDIV+RAND sent to master

SMP IO Capabilities

Security Manager I/O Capabilities

Table: VALUES

Value	Name	Description
0	sm_io_capability_displayonly	Display Only
1	sm_io_capability_displayyesno	Display with Yes/No-buttons
2	sm_io_capability_keyboardonly	Keyboard Only
3	sm_io_capability_noinputnooutput	No Input and No Output
4	sm_io_capability_keyboarddisplay	Display with Keyboard

5.7.3 Events

Security Manager class events

Bonding Fail

This event indicates the bonding has failed for a connection.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x03	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x01	method	Message ID
4	**uint8**	**handle**	Connection handle
5 - 6	**uint16**	**result**	Encryption status, describes error that occurred during bonding See: Security Manager Protocol Errors

C Functions

```
/* Callback */
struct ble_msg_sm_bonding_fail_evt_t{
    uint8 handle,
    uint16 result
}
void ble_evt_sm_bonding_fail(
    const struct ble_msg_sm_bonding_fail_evt_t * msg
)
```

BGScript Functions

```
event sm_bonding_fail(handle, result)
```

Bond Status

This event outputs bonding status information.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x04	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x04	method	Message ID
4	**uint8**	**bond**	Bonding handle
5	**uint8**	**keysize**	Encryption key size used in long-term key
6	**uint8**	**mitm**	Was Man-in-the-Middle mode was used in pairing **0:** No MITM used **1:** MITM was used
7	**uint8**	**keys**	Keys stored for bonding See: Bonding Keys

C Functions

```
/* Callback */
struct ble_msg_sm_bond_status_evt_t{
    uint8 bond,
    uint8 keysize,
    uint8 mitm,
    uint8 keys
}
void ble_evt_sm_bond_status(
    const struct ble_msg_sm_bond_status_evt_t * msg
)
```

BGScript Functions

```
event sm_bond_status(bond, keysize, mitm, keys)
```

Passkey Display

This event tells a passkey should be printed to the user for bonding. This passkey must be entered in the remote device for bonding to be successful.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x05	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x02	method	Message ID
4	**uint8**	**handle**	*Bluetooth* connection handle
5 - 8	**uint32**	**passkey**	Passkey range: 000000-999999

C Functions

```
/* Callback */
struct ble_msg_sm_passkey_display_evt_t{
    uint8 handle,
    uint32 passkey
}
void ble_evt_sm_passkey_display(
    const struct ble_msg_sm_passkey_display_evt_t * msg
)
```

BGScript Functions

```
event sm_passkey_display(handle, passkey)
```

Passkey Request

This event indicates the Security Manager requests the user to enter passkey. The passkey the user needs to enter is displayed by the remote device.

Use Passkey Entry command to respond to request

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x01	lolen	Minimum payload length
2	0x05	class	Message class: Security Manager
3	0x03	method	Message ID
4	**uint8**	**handle**	Connection handle

C Functions

```
/* Callback */
struct ble_msg_sm_passkey_request_evt_t{
    uint8 handle
}
void ble_evt_sm_passkey_request(
    const struct ble_msg_sm_passkey_request_evt_t * msg
)
```

BGScript Functions

```
event sm_passkey_request(handle)
```

5.8 System

The System class provides access to the local device and contains functions for example to query the local *Bluetooth* address, read firmware version, read radio packet counters etc.

5.8.1 Commands

System class commands

Address Get

This command reads the local devices public Bluetooth address.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x02	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x06	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x02	method	Message ID
4 - 9	**bd_addr**	**address**	Bluetooth address of the local device

C Functions

```
/* Function */
void ble_cmd_system_address_get(
    void
);

/* Callback */
struct ble_msg_system_address_get_rsp_t{
    bd_addr address
}
void ble_rsp_system_address_get(
    const struct ble_msg_system_address_get_rsp_t * msg
)
```

BGScript Functions

```
call system_address_get()(address)
```


AES Decrypt

This command decrypts the given data using the AES algorithm with the predefined key set with command Aes Setkey.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x11	method	Message ID
4	**uint8array**	**data**	Data to be decrypted Maximum size is 16 bytes, will be zero padded if less.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x11	method	Message ID
4	**uint8array**	**data**	Decrypted data

C Functions

```
/* Function */
void ble_cmd_system_aes_decrypt(
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_system_aes_decrypt_rsp_t{
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_system_aes_decrypt(
    const struct ble_msg_system_aes_decrypt_rsp_t * msg
)
```

BGScript Functions

```
call system_aes_decrypt(data_len, data_data)(data_len, data_data)
```

AES Encrypt

This command encrypts the given data using the AES algorithm with the predefined with command Aes Setkey.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x10	method	Message ID
4	**uint8array**	**data**	Data to be encrypted Maximum size is 16 bytes, will be zero padded if less.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x10	method	Message ID
4	**uint8array**	**data**	Encrypted data

C Functions

```
/* Function */
void ble_cmd_system_aes_encrypt(
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_system_aes_encrypt_rsp_t{
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_system_aes_encrypt(
    const struct ble_msg_system_aes_encrypt_rsp_t * msg
)
```

BGScript Functions

```
call system_aes_encrypt(data_len, data_data)(data_len, data_data)
```

AES Setkey

This command defines the encryption key that will be used with the AES encrypt and decrypt commands.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0F	method	Message ID

Byte	Type	Name	Description
4	**uint8array**	**key**	Encryption key Key size is 16 bytes, will be zero padded if less.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0F	method	Message ID

C Functions

```
/* Function */
void ble_cmd_system_aes_setkey(
    uint8 key_len,
    const uint8* key_data
);

/* Callback *
void ble_rsp_system_aes_setkey(
    const void *nul
)
```

BGScript Functions

```
call system_aes_setkey(key_len, key_data)
```

Endpoint RX

Read data from an endpoint (i.e., data souce, e.g., UART), error is returned if endpoint does not have enough data.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0D	method	Message ID
4	**uint8**	**endpoint**	Endpoint index to read data from
5	**uint8**	**size**	Size of data to read

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length

Byte	Type	Name	Description
2	0x00	class	Message class: System
3	0x0D	method	Message ID
4 - 5	**uint16**	**result**	Command result
6	**uint8array**	**data**	Data read from endpoint

C Functions

```
/* Function */
void ble_cmd_system_endpoint_rx(
    uint8 endpoint,
    uint8 size
);

/* Callback */
struct ble_msg_system_endpoint_rx_rsp_t{
    uint16 result,
    uint8 data_len,
    const uint8* data_data
}
void ble_rsp_system_endpoint_rx(
    const struct ble_msg_system_endpoint_rx_rsp_t * msg
)
```

BGScript Functions

```
call system_endpoint_rx(endpoint, size)(result, data_len, data_data)
```

Endpoint Set Watermarks

Set watermarks on both input and output sides of an endpoint. This is used to enable and disable the following events: Endpoint Watermark Tx and Endpoint Watermark Rx.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0E	method	Message ID
4	**uint8**	**endpoint**	Endpoint index to set watermarks.
5	**uint8**	**rx**	Watermark position on receive buffer **0xFF** : watermark is not modified **0x00** : disables watermark **1-63** : sets watermark position
6	**uint8**	**tx**	Watermark position on transmit buffer **0xFF** : watermark is not modified **0x00** : disables watermark **1-63** : sets watermark position

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0E	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_system_endpoint_set_watermarks(
    uint8 endpoint,
    uint8 rx,
    uint8 tx
);

/* Callback */
struct ble_msg_system_endpoint_set_watermarks_rsp_t{
    uint16 result
}
void ble_rsp_system_endpoint_set_watermarks(
    const struct ble_msg_system_endpoint_set_watermarks_rsp_t * msg
)
```

BGScript Functions

```
call system_endpoint_set_watermarks(endpoint, rx, tx)(result)
```

Endpoint TX

Send data to endpoint, error is returned if endpoint does not have enough space

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x09	method	Message ID
4	**uint8**	**endpoint**	Endpoint index to send data to
5	**uint8array**	**data**	data to send

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x09	method	Message ID
4 - 5	**uint16**	**result**	Command result

C Functions

```
/* Function */
void ble_cmd_system_endpoint_tx(
    uint8 endpoint,
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_system_endpoint_tx_rsp_t{
    uint16 result
}
void ble_rsp_system_endpoint_tx(
    const struct ble_msg_system_endpoint_tx_rsp_t * msg
)
```

BGScript Functions

```
call system_endpoint_tx(endpoint, data_len, data_data)(result)
```

Get Connections

This command reads the number of supported connections from the local device.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x06	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x06	method	Message ID
4	**uint8**	**maxconn**	Max supported connections

Table: EVENTS

Event	Description
connection status	Connection status for each connection handle

C Functions

```
/* Function */
void ble_cmd_system_get_connections(
    void
);

/* Callback */
struct ble_msg_system_get_connections_rsp_t{
    uint8 maxconn
}
void ble_rsp_system_get_connections(
    const struct ble_msg_system_get_connections_rsp_t * msg
)
```

BGScript Functions

```
call system_get_connections()(maxconn)
```

Get Counters

Read packet counters and resets them, also returns available packet buffers.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x05	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x05	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x05	method	Message ID
4	**uint8**	**txok**	Acknowledgements received for sent packets
5	**uint8**	**txretry**	Number of packets retransmitted
6	**uint8**	**rxok**	packets received where crc was ok
7	**uint8**	**rxfail**	packets received where crc failed
8	**uint8**	**mbuf**	Available packet buffers

C Functions

```
/* Function */
void ble_cmd_system_get_counters(
    void
);

/* Callback */
struct ble_msg_system_get_counters_rsp_t{
    uint8 txok,
    uint8 txretry,
    uint8 rxok,
    uint8 rxfail,
    uint8 mbuf
}
void ble_rsp_system_get_counters(
    const struct ble_msg_system_get_counters_rsp_t * msg
)
```

BGScript Functions

```
call system_get_counters()(txok, txretry, rxok, rxfail, mbuf)
```

Get Info

This command reads the local devices software and hardware versions.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x08	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x0C	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x08	method	Message ID
4 - 5	**uint16**	**major**	Major software version
6 - 7	**uint16**	**minor**	Minor software version
8 - 9	**uint16**	**patch**	Patch ID
10 - 11	**uint16**	**build**	Build version
12 - 13	**uint16**	**ll_version**	Link layer version
14	**uint8**	**protocol_version**	BGAPI protocol version
15	**uint8**	**hw**	Hardware version

C Functions

```
/* Function */
void ble_cmd_system_get_info(
    void
);

/* Callback */
struct ble_msg_system_get_info_rsp_t{
    uint16 major,
    uint16 minor,
    uint16 patch,
    uint16 build,
    uint16 ll_version,
    uint8 protocol_version,
    uint8 hw
}
void ble_rsp_system_get_info(
    const struct ble_msg_system_get_info_rsp_t * msg
)
```

BGScript Functions

```
call system_get_info()(major, minor, patch, build, ll_version, protocol_version, hw)
```

Hello

This command can be used to test if the local device is functional. Similar to a typical "AT" -> "OK" test.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x01	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x01	method	Message ID

C Functions

```
/* Function */
void ble_cmd_system_hello(
    void
);

/* Callback *
void ble_rsp_system_hello(
    const void *nul
)
```

BGScript Functions

```
call system_hello()
```

Reset

This command resets the local device immediately. The command does not have a response.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x00	method	Message ID
4	**uint8**	**boot_in_dfu**	Selects the boot mode **0 : boot to main program** **1 : boot to DFU**

Table: EVENTS

Event	Description
system boot	Sent when device has completed reset

C Functions

```
/* Function */
void ble_cmd_system_reset(
    uint8 boot_in_dfu
);
```

BGScript Functions

```
call system_reset(boot_in_dfu)
```

Whitelist Append

Add an entry to the running white list. By the white list you can define for example the remote devices which are allowed to establish a connection. See also Set Filtering and Connect Selective (if the white list is empty they will not be active). Do not use this command while advertising, scanning, or while being connected. The current list is discarded upon reset or power-cycle.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x07	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0A	method	Message ID
4 - 9	**bd_addr**	**address**	Bluetooth device address to add to the running white list Maximum of 8 can be stored before you must clear or remove entires
10	**uint8**	**address_type**	Bluetooth address type

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0A	method	Message ID
4 - 5	**uint16**	**result**	

C Functions

```
/* Function */
void ble_cmd_system_whitelist_append(
    bd_addr address,
    uint8 address_type
);

/* Callback */
struct ble_msg_system_whitelist_append_rsp_t{
    uint16 result
}
void ble_rsp_system_whitelist_append(
    const struct ble_msg_system_whitelist_append_rsp_t * msg
)
```

BGScript Functions

```
call system_whitelist_append(address, address_type)(result)
```

Whitelist Clear

Delete all entries on the white list at once.

⚠ Do not use this command while advertising or while being connected.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0C	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0C	method	Message ID

C Functions

```
/* Function */
void ble_cmd_system_whitelist_clear(
    void
);

/* Callback *
void ble_rsp_system_whitelist_clear(
    const void *nul
)
```

BGScript Functions

```
call system_whitelist_clear()
```

Whitelist Remove

Remove an entry from the running white list.

Do not use this command while advertising or while being connected.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x07	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0B	method	Message ID
4 - 9	**bd_addr**	**address**	Bluetooth device address to remove from the running white list
10	**uint8**	**address_type**	Bluetooth address type

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x0B	method	Message ID
4 - 5	**uint16**	**result**	

C Functions

```
/* Function */
void ble_cmd_system_whitelist_remove(
    bd_addr address,
    uint8 address_type
);

/* Callback */
struct ble_msg_system_whitelist_remove_rsp_t{
    uint16 result
}
void ble_rsp_system_whitelist_remove(
    const struct ble_msg_system_whitelist_remove_rsp_t * msg
)
```

BGScript Functions

```
call system_whitelist_remove(address, address_type)(result)
```

5.8.2 Enumerations

System class enumerations

Endpoints

Data Endpoints used in data routing and interface configuration

Table: VALUES

Value	Name	Description
0	system_endpoint_api	Command Parser
1	system_endpoint_test	Radio Test
2	system_endpoint_script	BGScript (not used)
3	system_endpoint_usb	USB Interface
4	system_endpoint_uart0	USART 0
5	system_endpoint_uart1	USART 1

5.8.3 Events

System class events

Boot

This event is produced when the device boots up and is ready to receive commands

This event is not sent over USB interface.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x0C	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x00	method	Message ID
4 - 5	**uint16**	**major**	Major software version
6 - 7	**uint16**	**minor**	Minor software version
8 - 9	**uint16**	**patch**	Patch ID
10 - 11	**uint16**	**build**	Build version
12 - 13	**uint16**	**ll_version**	Link layer version
14	**uint8**	**protocol_version**	BGAPI protocol version
15	**uint8**	**hw**	Hardware version

C Functions

```
/* Callback */
struct ble_msg_system_boot_evt_t{
    uint16 major,
    uint16 minor,
    uint16 patch,
    uint16 build,
    uint16 ll_version,
    uint8 protocol_version,
    uint8 hw
}
void ble_evt_system_boot(
    const struct ble_msg_system_boot_evt_t * msg
)
```

BGScript Functions

```
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
```

Endpoint Watermark RX

This event is generated if the receive (incoming) buffer of the endpoint has been filled with a number of bytes equal or higher than the value defined by the command Endpoint Set Watermarks. Data from the receive buffer can then be read (and consequently cleared) with the command Endpoint Rx.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x02	method	Message ID
4	**uint8**	**endpoint**	Endpoint index where data was received
5	**uint8**	**data**	Received data size

C Functions

```
/* Callback */
struct ble_msg_system_endpoint_watermark_rx_evt_t{
    uint8 endpoint,
    uint8 data
}
void ble_evt_system_endpoint_watermark_rx(
    const struct ble_msg_system_endpoint_watermark_rx_evt_t * msg
)
```

BGScript Functions

```
event system_endpoint_watermark_rx(endpoint, data)
```

Endpoint Watermark TX

This event is generated when the transmit (outgoing) buffer of the endpoint has free space for a number of bytes equal or higher than the value defined by the command Endpoint Set Watermarks. When there is enough free space, data can be sent out of the endpoint by the command Endpoint Tx.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x03	method	Message ID
4	**uint8**	**endpoint**	Endpoint index where data was sent
5	**uint8**	**data**	Space available

C Functions

```
/* Callback */
struct ble_msg_system_endpoint_watermark_tx_evt_t{
    uint8 endpoint,
    uint8 data
}
void ble_evt_system_endpoint_watermark_tx(
    const struct ble_msg_system_endpoint_watermark_tx_evt_t * msg
)
```

BGScript Functions

```
event system_endpoint_watermark_tx(endpoint, data)
```

No License Key

This error is produced when no valid license key found form the *Bluetooth* Smart hardware. When there is no valid license key the Bluetooth radio will not be operational.

A new license key can be requested from the Bluegiga Technical Support.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x00	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x05	method	Message ID

C Functions

```
/* Callback *
void ble_evt_system_no_license_key(
    const void *nul
)
```

BGScript Functions

```
event system_no_license_ke)
```

Protocol Error

A protocol error was detected in BGAPI command parser. This event is triggered if a BGAPI command from the host contains syntax error(s), or if a command is sent only partially sent. The the BGAPI parser has a 1 second command timeout and if a valid command is not transmitted within this timeout an error is raised and the partial command will be ignored.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x02	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x06	method	Message ID
4 - 5	**uint16**	**reason**	Reason for failure See: BGAPI Errors

C Functions

```
/* Callback */
struct ble_msg_system_protocol_error_evt_t{
    uint16 reason
}
void ble_evt_system_protocol_error(
    const struct ble_msg_system_protocol_error_evt_t * msg
)
```

BGScript Functions

```
event system_protocol_error(reason)
```

Script Failure

A BGScript failure has been detected and this event is raised.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x04	lolen	Minimum payload length
2	0x00	class	Message class: System
3	0x04	method	Message ID
4 - 5	**uint16**	**address**	Address where failure was detected
6 - 7	**uint16**	**reason**	Reason for failure

C Functions

```
/* Callback */
struct ble_msg_system_script_failure_evt_t{
    uint16 address,
    uint16 reason
}
void ble_evt_system_script_failure(
    const struct ble_msg_system_script_failure_evt_t * msg
)
```

BGScript Functions

```
event system_script_failure(address, reason)
```

5.9 Testing

The Testing API provides access to functions which can be used to put the local device into a test mode required for *Bluetooth* conformance testing.

5.9.1 Commands

Channel Mode

Set channel quality measurement mode. This command defines the kind of information reported by the response to the command Get Channel Map.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x06	method	Message ID
4	**uint8**	**mode**	**0** : Last RSSI from packet (when this mode is selected, the command Get Channel Map must be issued while a connection exists) **1:** Accumulate error counter (when this mode is selected, the command Get Channel Map must be issued while a connection exists) **2:** Channel Sweep (when this mode is selected, the command Get Channel Map must be issued while no connection exists)

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x06	method	Message ID

C Functions

```
/* Function */
void ble_cmd_test_channel_mode(
    uint8 mode
);

/* Callback *
void ble_rsp_test_channel_mode(
    const void *nul
)
```

BGScript Functions

```
call test_channel_mode(mode)
```

Get Channel Map

This command can be used to read the Channel Quality Map. Channel Quality Map is cleared after the response to this command is sent. Measurements are entered into the Channel Quality Map as packets are received over the different channels during a normal connection.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x04	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: response
1	0x01	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x04	method	Message ID
4	**uint8array**	**channel_map**	Channel quality map measurements. The 37 bytes reported by this response, one per each channel, carry the information defined via the Channel Mode configuration command.

C Functions

```
/* Function */
void ble_cmd_test_get_channel_map(
    void
);

/* Callback */
struct ble_msg_test_get_channel_map_rsp_t{
    uint8 channel_map_len,
    const uint8* channel_map_data
}
void ble_rsp_test_get_channel_map(
    const struct ble_msg_test_get_channel_map_rsp_t * msg
)
```

BGScript Functions

```
call test_get_channel_map()(channel_map_len, channel_map_data)
```


Phy End

This command ends a PHY test and report received packets.

PHY - testing commands implement Direct test mode from Bluetooth Core Specification, Volume 6, Part F.

These commands are meant to be used when testing against separate Bluetooth tester.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x02	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x02	method	Message ID
4 - 5	**uint16**	**counter**	Received packet counter

C Functions

```
/* Function */
void ble_cmd_test_phy_end(
    void
);

/* Callback */
struct ble_msg_test_phy_end_rsp_t{
    uint16 counter
}
void ble_rsp_test_phy_end(
    const struct ble_msg_test_phy_end_rsp_t * msg
)
```

BGScript Functions

```
call test_phy_end()(counter)
```

Phy Rx

This commands starts a PHY receive test. Valid packets received can be read by Phy End command.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length

Byte	Type	Name	Description
2	0x08	class	Message class: Testing
3	0x01	method	Message ID
4	**uint8**	**channel**	Bluetooth channel to use **Values: 0x00 - 0x27** Channel is (Frequency-2402)/2 Frequency Range 2402 MHz to 2480 MHz **Examples:** **0x0 :** 2402MHz **0x13**: 2441MHz **0x27:** 2480MHz

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x01	method	Message ID

C Functions

```
/* Function */
void ble_cmd_test_phy_rx(
    uint8 channel
);

/* Callback *
void ble_rsp_test_phy_rx(
    const void *nul
)
```

BGScript Functions

```
call test_phy_rx(channel)
```

Phy Tx

This command start PHY packet transmission and the radio starts to send one packet at every 625us. If a carrier wave is specified as type then the radio just broadcasts continuous carrier wave.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x03	lolen	Minimum payload length
2	0x08	class	Message class: Testing

Byte	Type	Name	Description
3	0x00	method	Message ID
4	**uint8**	**channel**	RF channel to use **Values: 0x00 - 0x27** channel is (Frequency-2402)/2 Frequency Range 2402 MHz to 2480 MHz
5	**uint8**	**length**	Payload data length as octetes **Values: 0x00 - 0x25**
6	**uint8**	**type**	Packet Payload data contents **0:** PRBS9 pseudo-random data **1:** 11110000 sequence **2:** 10101010 sequence **3:** broadcast carrier wave

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x08	class	Message class: Testing
3	0x00	method	Message ID

C Functions

```
/* Function */
void ble_cmd_test_phy_tx(
    uint8 channel,
    uint8 length,
    uint8 type
);

/* Callback *
void ble_rsp_test_phy_tx(
    const void *nul
)
```

BGScript Functions

```
call test_phy_tx(channel, length, type)
```

5.10 Device Firmware Upgrade

The commands and events in the DFU (Device firmware upgrade) can be used to perform a firmware upgrade to the local device for example over the UART interface.

T commands in this class are only available when the module has been booted into DFU mode with the reset command.

5.10.1 Commands

Device Firmware Upgrade commands

Flash Set Address

After the device has been boot into DFU mode, and if the UART bootloader is used (defined in project configuration file), this command can be used to start the DFU firmware upgrade.

The UART DFU process:

1. Boot device to DFU mode with : Reset command.
2. Wait for DFU Boot event
3. Send command Flash Set Address to start the firmware update.
4. Upload the firmware with Flash Upload commands until all the data has been uploaded.
5. Send Flash Upload Finish to when all the data has been uploaded.
6. Finalize the DFU firmware update with command: Reset.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x04	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x01	method	Message ID
4 - 7	**uint32**	**address**	The offset in the flash where to start flashing. **Always use:** 0x1000

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x01	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was successful

C Functions

```
/* Function */
void ble_cmd_dfu_flash_set_address(
    uint32 address
);

/* Callback */
struct ble_msg_dfu_flash_set_address_rsp_t{
    uint16 result
}
void ble_rsp_dfu_flash_set_address(
    const struct ble_msg_dfu_flash_set_address_rsp_t * msg
)
```

BGScript Functions

```
call dfu_flash_set_address(address)(result)
```

Flash Upload

This command is used repeatedly to upload the new binary firmware image to module over the UART interface. The address on the flash will be updated automatically.

When all data is uploaded finalize the upload with command: Flash Upload Finish.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x02	method	Message ID
4	**uint8array**	**data**	An array of data which will be written into the flash. Up to 64 bytes can be send at a time.

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x02	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was successful

C Functions

```
/* Function */
void ble_cmd_dfu_flash_upload(
    uint8 data_len,
    const uint8* data_data
);

/* Callback */
struct ble_msg_dfu_flash_upload_rsp_t{
    uint16 result
}
void ble_rsp_dfu_flash_upload(
    const struct ble_msg_dfu_flash_upload_rsp_t * msg
)
```

BGScript Functions

```
call dfu_flash_upload(data_len, data_data)(result)
```

Flash Upload Finish

This command tells to the device that the uploading of DFU data has finished. After this command the issue still Reset command to restart the Bluetooth module in normal mode.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x00	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x03	method	Message ID

Table: RESPONSE

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x02	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x03	method	Message ID
4 - 5	**uint16**	**result**	**0**: Command was successful

C Functions

```
/* Function */
void ble_cmd_dfu_flash_upload_finish(
    void
);

/* Callback */
struct ble_msg_dfu_flash_upload_finish_rsp_t{
    uint16 result
}
void ble_rsp_dfu_flash_upload_finish(
    const struct ble_msg_dfu_flash_upload_finish_rsp_t * msg
)
```

BGScript Functions

```
call dfu_flash_upload_finish()(result)
```

Reset

This command resets the *Bluetooth* module or the dongle. This command does not have a response, but triggers the normal boot event Boot or DFU boot event (Boot) if the DFU option is used and UART boot loader is installed.

There are three available boot loaders: USB for DFU upgrades using the USB-DFU protocol over the USB interface, UART for DFU upgrades using the BGAPI protocol over the UART interface, and OTA for the Over-the-Air upgrades.

Table: COMMAND

Byte	Type	Name	Description
0	0x00	hilen	Message type: command
1	0x01	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x00	method	Message ID
4	**uint8**	**dfu**	Whether or not to boot into DFU mode: **0:** Reboot normally **1:** Reboot into DFU mode for communication with the currently installed boot loader (UART, USB or OTA)

C Functions

```
/* Function */
void ble_cmd_dfu_reset(
    uint8 dfu
);
```

BGScript Functions

```
call dfu_reset(dfu)
```

5.10.2 Events

Device Firmware Upgrade events

Boot

Device has booted up in DFU mode and is ready to receive commands.

Table: EVENT

Byte	Type	Name	Description
0	0x80	hilen	Message type: event
1	0x04	lolen	Minimum payload length
2	0x09	class	Message class: Device Firmware Upgrade
3	0x00	method	Message ID
4 - 7	**uint32**	**version**	DFU protocol version **1:** DFUv1 protocol

C Functions

```
/* Callback */
struct ble_msg_dfu_boot_evt_t{
    uint32 version
}
void ble_evt_dfu_boot(
    const struct ble_msg_dfu_boot_evt_t * msg
)
```

BGScript Functions

```
event dfu_boot(version)
```

5.11 Error Codes

This section describes the error codes the API commands may produce.

5.11.1 BGAPI Errors

Errors related to BGAPI protocol

Invalid Parameter (0x0180)

Command contained invalid parameter

Device in Wrong State (0x0181)

Device is in wrong state to receive command

Out Of Memory (0x0182)

Device has run out of memory

Feature Not Implemented (0x0183)

Feature is not implemented

Command Not Recognized (0x0184)

Command was not recognized

Timeout (0x0185)

Command or Procedure failed due to timeout

Not Connected (0x0186)

Connection handle passed is to command is not a valid handle

flow (0x0187)

Command would cause either underflow or overflow error

User Attribute (0x0188)

User attribute was accessed through API which is not supported

Invalid License Key (0x0189)

No valid license key found

Command Too Long (0x018A)

Command maximum length exceeded

Out of Bonds (0x018B)

Bonding procedure can't be started because device has no space left for bond.

5.11.2 Bluetooth Errors

Bluetooth errors

Authentication Failure (0x0205)

Pairing or authentication failed due to incorrect results in the pairing or authentication procedure. This could be due to an incorrect PIN or Link Key

Pin or Key Missing (0x0206)

Pairing failed because of missing PIN, or authentication failed because of missing Key.

Memory Capacity Exceeded (0x0207)

Controller is out of memory.

Connection Timeout (0x0208)

Link supervision timeout has expired.

Connection Limit Exceeded (0x0209)

Controller is at limit of connections it can support.

Command Disallowed (0x020C)

Command requested cannot be executed because the Controller is in a state where it cannot process this command at this time.

Invalid Command Parameters (0x0212)

Command contained invalid parameters.

Remote User Terminated Connection (0x0213)

User on the remote device terminated the connection.

Connection Terminated by Local Host (0x0216)

Local device terminated the connection.

LL Response Timeout (0x0222)

Connection terminated due to link-layer procedure timeout.

LL Instant Passed (0x0228)

Received link-layer control packet where instant was in the past.

Controller Busy (0x023A)

Operation was rejected because the controller is busy and unable to process the request.

Unacceptable Connection Interval (0x023B)

The Unacceptable Connection Interval error code indicates that the remote device terminated the connection because of an unacceptable connection interval.

Directed Advertising Timeout (0x023C)

Directed advertising completed without a connection being created.

MIC Failure (0x023D)

Connection was terminated because the Message Integrity Check (MIC) failed on a received packet.

Connection Failed to be Established (0x023E)

LL initiated a connection but the connection has failed to be established. Controller did not receive any packets from remote end.

5.11.3 Security Manager Protocol Errors

Errors from Security Manager Protocol

Passkey Entry Failed (0x0301)

The user input of passkey failed, for example, the user cancelled the operation

OOB Data is not available (0x0302)

Out of Band data is not available for authentication

Authentication Requirements (0x0303)

The pairing procedure cannot be performed as authentication requirements cannot be met due to IO capabilities of one or both devices

Confirm Value Failed (0x0304)

The confirm value does not match the calculated compare value

Pairing Not Supported (0x0305)

Pairing is not supported by the device

Encryption Key Size (0x0306)

The resultant encryption key size is insufficient for the security requirements of this device

Command Not Supported (0x0307)

The SMP command received is not supported on this device

Unspecified Reason (0x0308)

Pairing failed due to an unspecified reason

Repeated Attempts (0x0309)

Pairing or authentication procedure is disallowed because too little time has elapsed since last pairing request or security request

Invalid Parameters (0x030A)

The Invalid Parameters error code indicates: the command length is invalid or a parameter is outside of the specified range.

5.11.4 Attribute Protocol Errors

Errors from Attribute Protocol

Invalid Handle (0x0401)

The attribute handle given was not valid on this server

Read Not Permitted (0x0402)

The attribute cannot be read

Write Not Permitted (0x0403)

The attribute cannot be written

Invalid PDU (0x0404)

The attribute PDU was invalid

Insufficient Authentication (0x0405)

The attribute requires authentication before it can be read or written.

Request Not Supported (0x0406)

Attribute Server does not support the request received from the client.

Invalid Offset (0x0407)

Offset specified was past the end of the attribute

Insufficient Authorization (0x0408)

The attribute requires authorization before it can be read or written.

Prepare Queue Full (0x0409)

Too many prepare writes have been queueud

Attribute Not Found (0x040A)

No attribute found within the given attribute handle range.

Attribute Not Long (0x040B)

The attribute cannot be read or written using the Read Blob Request

Insufficient Encryption Key Size (0x040C)

The Encryption Key Size used for encrypting this link is insufficient.

Invalid Attribute Value Length (0x040D)

The attribute value length is invalid for the operation

Unlikely Error (0x040E)

The attribute request that was requested has encountered an error that was unlikely, and therefore could not be completed as requested.

Insufficient Encryption (0x040F)

The attribute requires encryption before it can be read or written.

Unsupported Group Type (0x0410)

The attribute type is not a supported grouping attribute as defined by a higher layer specification.

Insufficient Resources (0x0411)

Insufficient Resources to complete the request

Application Error Codes (0x0480)

Application error code defined by a higher layer specification.

Contact information

Sales:	sales@bluegiga.com
Technical support:	http://www.bluegiga.com/support/
Orders:	orders@bluegiga.com
WWW:	http://www.bluegiga.com
Head Office / Finland:	Phone: +358-9-4355 060 Fax: +358-9-4355 0660 Sinikalliontie 5 A 02630 ESPOO FINLAND
Head address / Finland:	P.O. Box 120 02631 ESPOO FINLAND
Sales Office / USA:	Phone: +1 770 291 2181 Fax: +1 770 291 2183 Bluegiga Technologies, Inc. 3235 Satellite Boulevard, Building 400, Suite 300 Duluth, GA, 30096, USA
Sales Office / Hong-Kong:	Phone: +852 3182 7321 Fax: +852 3972 5777 Bluegiga Technologies, Inc. Unit 10-18, 32/F, Tower 1, Millennium City 1, 388 Kwun Tong Road, Kwun Tong, Kowloon, Hong Kong