1. **NIMBLE Overview**

1.1 What is the Nimble?

Apache Mynewt offers the world’s first fully open-source Bluetooth Low Energy (BLE) or Bluetooth Smart stack fully compliant with Bluetooth 5 specifications with support for Bluetooth Mesh. It is called NimBLE.

BLE technology operates in the unlicensed industrial, scientific and medical (ISM) band at 2.4 to 2.485 GHz which is available in most countries. It uses a spread spectrum, frequency hopping, full-duplex signal. BLE FHSS employs 40 2-MHz-wide channels to ensure greater reliability over longer distances. It also features 0-dBm (1 mW) power output and a typical maximum range of 50 meters. With Bluetooth 5 specification range can be increased 4 times and speed 2 time.

1.2 Components

A Bluetooth low energy stack (NimBLE included) consists of two components with several subcomponents:

* **Controller**
  + **Physical Layer**: adaptive frequency-hopping Gaussian Frequency Shift Keying (GFSK) radio using 40 RF channels (0-39), with 2 MHz spacing.
  + **Link Layer**: with one of five states (Standby, Advertising, Scanning, Initiating, Connection states) active at any time
* **Host**
  + **Logical Link Control and Adaptation Protocol (L2CAP)**: provides logical channels, named L2CAP channels, which are multiplexed over one or more logical links to provide packet segmentation and reassembly, flow control, error control, streaming, QoS etc.
  + **Security Manager (SM)**: uses Security Manager Protocol (SMP) for pairing and transport specific key distribution for securing radio communication
  + **Attribute protocol (ATT)**: allows a device (Server) to expose certain pieces of data, known as Attributes, to another device (Client)
  + **Generic Attribute Profile (GATT)**: a framework for using the ATT protocol to exchange attributes encapsulated as Characteristics or Services
  + **Generic Access Profile (GAP)**: a base profile which all Bluetooth devices implement, which in the case of LE, defines the Physical Layer, Link Layer, L2CAP, Security Manager, Attribute Protocol and Generic Attribute Profile.
  + **Host Controller Interface (HCI)**: the interface between the host and controller either through software API or by a hardware interface such as SPI, UART or USB.

1.3 Nimble security

The Bluetooth Low Energy security model includes five distinct security concepts as listed below. For detailed specifications, see BLUETOOTH SPECIFICATION Version 4.2 [Vol 1, Part A].

* **Pairing**: The process for creating one or more shared secret keys. In LE a single link key is generated by combining contributions from each device into a link key used during pairing.
* **Bonding**: The act of storing the keys created during pairing for use in subsequent connections in order to form a trusted device pair.
* **Device authentication**: Verification that the two devices have the same keys (verify device identity)
* **Encryption**: Keeps message confidential. Encryption in Bluetooth LE uses AES-CCM cryptography and is performed in the Controller.
* **Message integrity**: Protects against message forgeries.

Bluetooth LE uses four association models depending on the I/O capabilities of the devices.

* **Just Works**: designed for scenarios where at least one of the devices does not have a display capable of displaying a six digit number nor does it have a keyboard capable of entering six decimal digits.
* **Numeric Comparison**: designed for scenarios where both devices are capable of displaying a six digit number and both are capable of having the user enter “yes” or “no”. A good example of this model is the cell phone / PC scenario.
* **Out of Band**: designed for scenarios where an Out of Band mechanism is used to both discover the devices as well as to exchange or transfer cryptographic numbers used in the pairing process.
* **Passkey Entry**: designed for the scenario where one device has input capability but does not have the capability to display six digits and the other device has output capabilities. A good example of this model is the PC and keyboard scenario.

2. **Nimble API**

# 2.1 Configure device address

2.1.1 Syscfg value set

The NimBLE controller package exports a [syscfg](http://mynewt.apache.org/latest/os/modules/sysinitconfig/sysinitconfig.html) setting called BLE\_PUBLIC\_DEV\_ADDR. This setting can be overridden at the application or target level to configure a public Bluetooth address. For example, a target can assign the public address *11:22:33:44:55:66* as follows:

Syscfg.h:

#ifndef MYNEWT\_VAL\_BLE\_PUBLIC\_DEV\_ADDR

#define MYNEWT\_VAL\_BLE\_PUBLIC\_DEV\_ADDR (uint8\_t[6]){0x66, 0x55, 0x44, 0x33, 0x22, 0x11})

#endif

This setting takes the form of a C expression. Specifically, the value is a designated initializer expressing a six-byte array. Also note that the bytes are reversed, as an array is inherently little-endian, while addresses are generally expressed in big-endian.

2.1.2 Configure a random address at runtime

Random addresses get configured through the NimBLE host. The following two functions are used in random address configuration:

**ble\_hs\_id\_gen\_rnd**: Generates a new random address.

**ble\_hs\_id\_set\_rnd**: Sets the device’s random address.

2.2 HAL API

Initialize the nimble hal layer module，include HW, OS, host and controller.

**void nimble\_hal\_init（void）;**

Ensure functions only gets called one time during nimble initialization.

**void nimble\_sysinit\_start(void);**

**void nimble\_sysinit\_end(void);**

2.3 Host API

2.3.1 Host GAP API

Searches for a connection with the specified handle. If a matching connection is found, the supplied connection descriptor is filled correspondingly.

@param handle The connection handle to search for.

@param out\_desc On success, this is populated with information relating to

the matching connection. Pass NULL if you don't need this information.

@return 0 on success, BLE\_HS\_ENOTCONN if no matching connection was

found.

**int ble\_gap\_conn\_find(uint16\_t handle, struct ble\_gap\_conn\_desc out\_desc);**

This function configures and start advertising procedure.

@param own\_addr\_type The type of address the stack should use for itself.

Valid values are:

- BLE\_OWN\_ADDR\_PUBLIC

- BLE\_OWN\_ADDR\_RANDOM

- BLE\_OWN\_ADDR\_RPA\_PUBLIC\_DEFAULT

- BLE\_OWN\_ADDR\_RPA\_RANDOM\_DEFAULT

@param direct\_addr The peer's address for directed advertising. his

parameter shall be non-NULL if directed advertising is

being used.

@param duration\_ms The duration of the advertisement procedure. On

expiration, the procedure ends and a

BLE\_GAP\_EVENT\_ADV\_COMPLETE event is reported. Units are

milliseconds. Specify BLE\_HS\_FOREVER for no expiration.

@param adv\_params Additional arguments specifying the particulars of the

advertising procedure.

@param cb The callback to associate with this advertising

procedure. If advertising ends, the event is reported

through this callback. If advertising results in a

connection, the connection inherits this callback as its event-reporting mechanism.

@param cb\_arg The optional argument to pass to the callback function.

@return 0 on success, error code on failure.

**int ble\_gap\_adv\_start(uint8\_t own\_addr\_type,**

**const ble\_addr\_t direct\_addr,**

**int32\_t duration\_ms,**

**const struct ble\_gap\_adv\_params adv\_params,**

**ble\_gap\_event\_fn cb, void cb\_arg);**

Stops the currently-active advertising procedure. A success return

code indicates that advertising has been fully aborted and a new advertising procedure can be initiated immediately.

**int ble\_gap\_adv\_stop(void);**

Indicates whether an advertisement procedure is currently in progress.

@return 0 if no advertisement procedure in progress, 1 otherwise.

**int ble\_gap\_adv\_active(void);**

Configures the data to include in subsequent advertisements.

@param data Buffer containing the advertising data.

@param data\_len The size of the advertising data, in bytes.

@return 0 on succes, BLE\_HS\_EBUSY if advertising is in progress,

other error code on failure.

int ble\_gap\_adv\_set\_data(const uint8\_t data, int data\_len);

Configures the data to include in subsequent scan responses.

@param data Buffer containing the scan response data.

@param data\_len The size of the response data, in bytes.

@return 0 on succes, BLE\_HS\_EBUSY if advertising is in progress,

other error code on failure.

**int ble\_gap\_adv\_rsp\_set\_data(const uint8\_t data, int data\_len);**

Configures the fields to include in subsequent advertisements. This is a

convenience wrapper for ble\_gap\_adv\_set\_data().

@param adv\_fields Specifies the advertisement data.

@return 0 on success,

BLE\_HS\_EBUSY if advertising is in progress,

BLE\_HS\_EMSGSIZE if the specified data is too large to

fit in an advertisement,

other error code on failure.

**int ble\_gap\_adv\_set\_fields(const struct ble\_hs\_adv\_fields rsp\_fields);**

Configures the fields to include in subsequent scan responses. This is a

convenience wrapper for ble\_gap\_adv\_rsp\_set\_data().

@param adv\_fields Specifies the scan response data.

@return 0 on success,

BLE\_HS\_EBUSY if advertising is in progress,

BLE\_HS\_EMSGSIZE if the specified data is too large to

fit in a scan response,

other error code on failure.

**int ble\_gap\_adv\_rsp\_set\_fields(const struct ble\_hs\_adv\_fields rsp\_fields);**

Performs the Limited or General Discovery Procedures.

@param own\_addr\_type The type of address the stack should use for

itself when sending scan requests. Valid

values are:

- BLE\_ADDR\_TYPE\_PUBLIC

- BLE\_ADDR\_TYPE\_RANDOM

- BLE\_ADDR\_TYPE\_RPA\_PUB\_DEFAULT

- BLE\_ADDR\_TYPE\_RPA\_RND\_DEFAULT

This parameter is ignored unless active

scanning is being used.

@param duration\_ms The duration of the discovery procedure.

On expiration, the procedure ends and a

BLE\_GAP\_EVENT\_DISC\_COMPLETE event is

reported. Units are milliseconds. Specify

BLE\_HS\_FOREVER for no expiration.

@param disc\_params Additional arguments specifying the particulars

of the discovery procedure.

@param cb The callback to associate with this discovery

procedure. Advertising reports and

discovery termination events are reported

through this callback.

@param cb\_arg The optional argument to pass to the callback

function.

@return 0 on success; nonzero on failure.

**int ble\_gap\_disc(uint8\_t own\_addr\_type, int32\_t duration\_ms,**

**const struct ble\_gap\_disc\_params disc\_params,**

**ble\_gap\_event\_fn cb, void cb\_arg);**

r reset, the variable x has the value 5. Initialized variable of this

Cancels the discovery procedure currently in progress. A success return

code indicates that scanning has been fully aborted; a new discovery or

connect procedure can be initiated immediately.

@return 0 on success;

BLE\_HS\_EALREADY if there is no discovery

procedure to cancel;

Other nonzero on unexpected error.

int ble\_gap\_disc\_cancel(void);

Indicates whether a discovery procedure is currently in progress.

@return 0: No discovery procedure in progress;

1: Discovery procedure in progress.

**int ble\_gap\_disc\_active(void);**

Initiates a connect procedure.

@param own\_addr\_type The type of address the stack should use for

itself during connection establishment.

- BLE\_OWN\_ADDR\_PUBLIC

- BLE\_OWN\_ADDR\_RANDOM

- BLE\_OWN\_ADDR\_RPA\_PUBLIC\_DEFAULT

- BLE\_OWN\_ADDR\_RPA\_RANDOM\_DEFAULT

@param peer\_addr The address of the peer to connect to.

If this parameter is NULL, the white list

is used.

@param duration\_ms The duration of the discovery procedure.

On expiration, the procedure ends and a

BLE\_GAP\_EVENT\_DISC\_COMPLETE event is

reported. Units are milliseconds.

@param conn\_params Additional arguments specifying the particulars

of the connect procedure. Specify null for

default values.

@param cb The callback to associate with this connect

procedure. When the connect procedure

completes, the result is reported through

this callback. If the connect procedure

succeeds, the connection inherits this

callback as its event-reporting mechanism.

@param cb\_arg The optional argument to pass to the callback

function.

@return 0 on success;

BLE\_HS\_EALREADY if a connection attempt is

already in progress;

BLE\_HS\_EBUSY if initiating a connection is not

possible because scanning is in progress;

BLE\_HS\_EDONE if the specified peer is already

connected;

Other nonzero on error.

**int ble\_gap\_connect(uint8\_t own\_addr\_type, const ble\_addr\_t peer\_addr,**

**int32\_t duration\_ms,**

**const struct ble\_gap\_conn\_params params,**

**ble\_gap\_event\_fn cb, void cb\_arg);**

Aborts a connect procedure in progress.

@return 0 on success;

BLE\_HS\_EALREADY if there is no active connect

procedure.

Other nonzero on error.

**int ble\_gap\_conn\_cancel(void);**

Indicates whether a connect procedure is currently in progress.

@return 0: No connect procedure in progress;

1: Connect procedure in progress.

**int ble\_gap\_conn\_active(void);**

Terminates an established connection.

@param conn\_handle The handle corresponding to the connection to

terminate.

@param hci\_reason The HCI error code to indicate as the reason

for termination.

@return 0 on success;

BLE\_HS\_ENOTCONN if there is no connection with

the specified handle;

Other nonzero on failure.

**int ble\_gap\_terminate(uint16\_t conn\_handle, uint8\_t hci\_reason);**

Initiates the GAP security procedure.

Depending on connection role and stored security information this function

will start appropriate security procedure (pairing or encryption).

@param conn\_handle The handle corresponding to the connection to secure.

@return 0 on success;

BLE\_HS\_ENOTCONN if the there is no connection

with the specified handle;

BLE\_HS\_EALREADY if an security procedure for

this connection is already in progress;

Other nonzero on error.

**int ble\_gap\_security\_initiate(uint16\_t conn\_handle);**

2.3.2 Host GATT API

Adjusts a host configuration object's settings to accommodate the specified

service definition array. This function adds the counts to the appropriate

fields in the supplied configuration object without clearing them first, so

it can be called repeatedly with different inputs to calculate totals. Be

sure to zero the GATT server settings prior to the first call to this

function.

@param defs The service array containing the resource

definitions to be counted.

@return 0 on success;

BLE\_HS\_EINVAL if the svcs array contains an

invalid resource definition.

**int ble\_gatts\_count\_cfg(const struct ble\_gatt\_svc\_def defs);**

Queues a set of service definitions for registration. All services queued

in this manner get registered when ble\_gatts\_start() is called.

@param svcs An array of service definitions to queue for

registration. This array must be

terminated with an entry whose 'type'

equals 0.

@return 0 on success;

BLE\_HS\_ENOMEM on heap exhaustion.

int ble\_gattc\_disc\_all\_chrs(uint16\_t conn\_handle, uint16\_t start\_handle,

uint16\_t end\_handle, ble\_gatt\_chr\_fn cb,

void cb\_arg);

Initiates GATT procedure: Discover All Primary Services.

@param conn\_handle The connection over which to execute the

procedure.

@param cb The function to call to report procedure status

updates; null for no callback.

@param cb\_arg The optional argument to pass to the callback

function.

**int ble\_gattc\_disc\_all\_svcs(uint16\_t conn\_handle,**

**ble\_gatt\_disc\_svc\_fn cb, void cb\_arg);**

Initiates GATT procedure: Discover All Characteristics of a Service.

@param conn\_handle The connection over which to execute the

procedure.

@param start\_handle The handle to begin the search at (generally

the service definition handle).

@param end\_handle The handle to end the search at (generally the

last handle in the service).

@param cb The function to call to report procedure status

updates; null for no callback.

@param cb\_arg The optional argument to pass to the callback

function.

@return 0 on success; nonzero on failure.

**int ble\_gattc\_disc\_all\_chrs(uint16\_t conn\_handle, uint16\_t start\_handle,**

**uint16\_t end\_handle, ble\_gatt\_chr\_fn cb,**

**void cb\_arg);**

Initiates GATT procedure: Exchange MTU.

@param conn\_handle The connection over which to execute the

procedure.

@param cb The function to call to report procedure status

updates; null for no callback.

@param cb\_arg The optional argument to pass to the callback

function.

@return 0 on success; nonzero on failure.

**int ble\_gattc\_exchange\_mtu(uint16\_t conn\_handle,**

**ble\_gatt\_mtu\_fn cb, void cb\_arg);**

Initiates GATT procedure: Discover Characteristics by UUID.

@param conn\_handle The connection over which to execute the procedure.

@param start\_handle The handle to begin the search at (generally

the service definition handle).

@param end\_handle The handle to end the search at (generally the

last handle in the service).

@param chr\_uuid128 The 128-bit UUID of the characteristic to

discover.

@param cb The function to call to report procedure status

updates; null for no callback.

@param cb\_arg The optional argument to pass to the callback

function.

@return 0 on success; nonzero on failure.

**int ble\_gattc\_disc\_chrs\_by\_uuid(uint16\_t conn\_handle, uint16\_t start\_handle,**

**uint16\_t end\_handle, const ble\_uuid\_t uuid,**

**ble\_gatt\_chr\_fn cb, void cb\_arg);**

**3. Nimble Data Struct**

**3.1 BLE HOST Struct**

@brief Bluetooth Host main configuration structure

Those can be used by application to configure stack.

struct **ble\_hs\_cfg** {

An optional callback that gets executed upon registration of each GATT

resource (service, characteristic, or descriptor).

**ble\_gatt\_register\_fn gatts\_register\_cb;**

An optional argument that gets passed to the GATT registration

callback.

**void gatts\_register\_arg;**

Security Manager Local Input Output Capabilities

**uint8\_t sm\_io\_cap;**

@brief Security Manager OOB flag

If set proper flag in Pairing Request/Response will be set.

**unsigned sm\_oob\_data\_flag:1;**

@brief Security Manager Bond flag

If set proper flag in Pairing Request/Response will be set. This results

in storing keys distributed during bonding.

**unsigned sm\_bonding:1;**

@brief Security Manager MITM flag

If set proper flag in Pairing Request/Response will be set. This results

in requiring Man-In-The-Middle protection when pairing.

**unsigned sm\_mitm:1;**

@brief Security Manager Secure Connections flag

If set proper flag in Pairing Request/Response will be set. This results

in using LE Secure Connections for pairing if also supported by remote

device. Fallback to legacy pairing if not supported by remote.

**unsigned sm\_sc:1;**

@brief Security Manager Key Press Notification flag

Currently unsupported and should not be set.

**unsigned sm\_keypress:1;**

@brief Security Manager Local Key Distribution Mask

**uint8\_t sm\_our\_key\_dist;**

@brief Security Manager Remote Key Distribution Mask

**uint8\_t sm\_their\_key\_dist;**

@brief Stack reset callback

This callback is executed when the host resets itself and the controller

due to fatal error.

**ble\_hs\_reset\_fn reset\_cb;**

@brief Stack sync callback

This callback is executed when the host and controller become synced.

This happens at startup and after a reset.

**ble\_hs\_sync\_fn sync\_cb;**

XXX: These need to go away. Instead, the nimble host package should

require the host-store API (not yet implemented)..

Storage Read callback handles read of security material

**ble\_store\_read\_fn store\_read\_cb;**

Storage Write callback handles write of security material

**ble\_store\_write\_fn store\_write\_cb;**

Storage Delete callback handles deletion of security material

**ble\_store\_delete\_fn store\_delete\_cb;**

@brief Storage Status callback.

This callback gets executed when a persistence operation cannot be

performed or a persistence failure is imminent. For example, if is

insufficient storage capacity for a record to be persisted, this

function gets called to give the application the opportunity to make

room.

**ble\_store\_status\_fn store\_status\_cb;**

An optional argument that gets passed to the storage status callback.

**void store\_status\_arg;**

}

**3.2 NIMBLE GAP Data Struct**

Connection security state

struct **ble\_gap\_sec\_state** {

If connection is encrypted

**unsigned encrypted:1;**

If connection is authenticated

**unsigned authenticated:1;**

If connection is bonded (security information is stored)

**unsigned bonded:1;**

Size of a key used for encryption

**unsigned key\_size:5;**

};

Advertising parameters

struct **ble\_gap\_adv\_params** {

Advertising mode. Can be one of following constants:

- BLE\_GAP\_CONN\_MODE\_NON (non-connectable; 3.C.9.3.2).

- BLE\_GAP\_CONN\_MODE\_DIR (directed-connectable; 3.C.9.3.3).

- BLE\_GAP\_CONN\_MODE\_UND (undirected-connectable; 3.C.9.3.4).

**uint8\_t conn\_mode;**

Discoverable mode. Can be one of following constants:

- BLE\_GAP\_DISC\_MODE\_NON (non-discoverable; 3.C.9.2.2).

- BLE\_GAP\_DISC\_MODE\_LTD (limited-discoverable; 3.C.9.2.3).

- BLE\_GAP\_DISC\_MODE\_GEN (general-discoverable; 3.C.9.2.4).

**uint8\_t disc\_mode;**

Minimum advertising interval, if 0 stack use sane defaults

**uint16\_t itvl\_min;**

Maximum advertising interval, if 0 stack use sane defaults

**uint16\_t itvl\_max;**

Advertising channel map , if 0 stack use sane defaults

**uint8\_t channel\_map;**

Advertising Filter policy

**uint8\_t filter\_policy;**

If do High Duty cycle for Directed Advertising

**uint8\_t high\_duty\_cycle:1;**

};

@brief Connection descriptor

struct **ble\_gap\_conn\_desc** {

Connection security state

**struct ble\_gap\_sec\_state sec\_state;**

Local identity address

**ble\_addr\_t our\_id\_addr;**

Peer identity address

**ble\_addr\_t peer\_id\_addr;**

Local over-the-air address

**ble\_addr\_t our\_ota\_addr;**

Peer over-the-air address

**ble\_addr\_t peer\_ota\_addr;**

Connection handle

**uint16\_t conn\_handle;**

Connection interval

**uint16\_t conn\_itvl;**

Connection latency

**uint16\_t conn\_latency;**

Connection supervision timeout

**uint16\_t supervision\_timeout;**

Connection Role

Possible values BLE\_GAP\_ROLE\_SLAVE or BLE\_GAP\_ROLE\_MASTER

**uint8\_t role;**

Master clock accuracy

**uint8\_t master\_clock\_accuracy;**

};

struct **ble\_gap\_conn\_params** {

uint16\_t scan\_itvl;

uint16\_t scan\_window;

uint16\_t itvl\_min;

uint16\_t itvl\_max;

uint16\_t latency;

uint16\_t supervision\_timeout;

uint16\_t min\_ce\_len;

uint16\_t max\_ce\_len;

};

struct **ble\_gap\_disc\_params** {

uint16\_t itvl;

uint16\_t window;

uint8\_t filter\_policy;

uint8\_t limited:1;

uint8\_t passive:1;

uint8\_t filter\_duplicates:1;

};

struct **ble\_gap\_upd\_params** {

uint16\_t itvl\_min;

uint16\_t itvl\_max;

uint16\_t latency;

uint16\_t supervision\_timeout;

uint16\_t min\_ce\_len;

uint16\_t max\_ce\_len;

};

struct **ble\_gap\_disc\_desc** {

Common fields.

uint8\_t event\_type;

uint8\_t length\_data;

ble\_addr\_t addr;

int8\_t rssi;

uint8\_t data;

LE direct advertising report fields; direct\_addr is BLE\_ADDR\_ANY if

direct address fields are not present.

ble\_addr\_t direct\_addr;

};

Represents a GAP-related event. When such an event occurs, the host

notifies the application by passing an instance of this structure to an

application-specified callback.

struct **ble\_gap\_event** {

Indicates the type of GAP event that occurred. This is one of the

BLE\_GAP\_EVENT codes.

**uint8\_t type;**

A discriminated union containing additional details concerning the GAP

event. The 'type' field indicates which member of the union is valid.

union {

Represents a connection attempt. Valid for the following event

types:

o BLE\_GAP\_EVENT\_CONNECT

struct {

The status of the connection attempt;

o 0: the connection was successfully established.

o BLE host error code: the connection attempt failed for

the specified reason.

int status;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **connect**;

Represents a terminated connection. Valid for the following event

types:

o BLE\_GAP\_EVENT\_DISCONNECT

struct {

A BLE host return code indicating the reason for the

disconnect.

int reason;

Information about the connection prior to termination.

struct ble\_gap\_conn\_desc conn;

} **disconnect**;

Represents an advertising report received during a discovery

procedure. Valid for the following event types:

o BLE\_GAP\_EVENT\_DISC

struct ble\_gap\_disc\_desc disc;

#if MYNEWT\_VAL(BLE\_EXT\_ADV)

Represents an extended advertising report received during a discovery

procedure. Valid for the following event types:

o BLE\_GAP\_EVENT\_EXT\_DISC

struct ble\_gap\_ext\_disc\_desc ext\_disc;

#endif

Represents a completed discovery procedure. Valid for the following

event types:

o BLE\_GAP\_EVENT\_DISC\_COMPLETE

struct {

The reason the discovery procedure stopped. Typical reason

codes are:

o 0: Duration expired.

o BLE\_HS\_EPREEMPTED: Host aborted procedure to configure a

peer's identity.

int reason;

} **disc\_complete**;

Represents a completed advertise procedure. Valid for the following

event types:

o BLE\_GAP\_EVENT\_ADV\_COMPLETE

struct {

The reason the advertise procedure stopped. Typical reason

codes are:

o 0: Terminated due to connection.

o BLE\_HS\_ETIMEOUT: Duration expired.

o BLE\_HS\_EPREEMPTED: Host aborted procedure to configure a

peer's identity.

int reason;

#if MYNEWT\_VAL(BLE\_EXT\_ADV)

Advertising instance

uint8\_t instance;

The handle of the relevant connection - valid if reason=0

uint16\_t conn\_handle;

Number of completed extended advertising events

This field is only valid if non-zero max\_events was passed to

ble\_gap\_ext\_adv\_start() and advertising completed due to duration

timeout or max events transmitted.

uint8\_t num\_ext\_adv\_events;

#endif

} **adv\_complete**;

Represents an attempt to update a connection's parameters. If the

attempt was successful, the connection's descriptor reflects the

updated parameters.

Valid for the following event types:

o BLE\_GAP\_EVENT\_CONN\_UPDATE

struct {

The result of the connection update attempt;

o 0: the connection was successfully updated.

o BLE host error code: the connection update attempt failed

for the specified reason.

int status;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **conn\_update**;

Represents a peer's request to update the connection parameters.

This event is generated when a peer performs any of the following

procedures:

o L2CAP Connection Parameter Update Procedure

o Link-Layer Connection Parameters Request Procedure

To reject the request, return a non-zero HCI error code. The value

returned is the reject reason given to the controller.

Valid for the following event types:

o BLE\_GAP\_EVENT\_L2CAP\_UPDATE\_REQ

o BLE\_GAP\_EVENT\_CONN\_UPDATE\_REQ

struct {

Indicates the connection parameters that the peer would like to

use.

const struct ble\_gap\_upd\_params peer\_params;

Indicates the connection parameters that the local device would

like to use. The application callback should fill this in. By

default, this struct contains the requested parameters (i.e.,

it is a copy of 'peer\_params').

struct ble\_gap\_upd\_params self\_params;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **conn\_update\_req**;

Represents a failed attempt to terminate an established connection.

Valid for the following event types:

o BLE\_GAP\_EVENT\_TERM\_FAILURE

struct {

A BLE host return code indicating the reason for the failure.

int status;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **term\_failure**;

Represents an attempt to change the encrypted state of a

connection. If the attempt was successful, the connection

descriptor reflects the updated encrypted state.

Valid for the following event types:

o BLE\_GAP\_EVENT\_ENC\_CHANGE

struct {

Indicates the result of the encryption state change attempt;

o 0: the encrypted state was successfully updated;

o BLE host error code: the encryption state change attempt

failed for the specified reason.

int status;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **enc\_change**;

Represents a passkey query needed to complete a pairing procedure.

Valid for the following event types:

o BLE\_GAP\_EVENT\_PASSKEY\_ACTION

struct {

Contains details about the passkey query.

struct ble\_gap\_passkey\_params params;

The handle of the relevant connection.

uint16\_t conn\_handle;

} **passkey**;

Represents a received ATT notification or indication.

Valid for the following event types:

o BLE\_GAP\_EVENT\_NOTIFY\_RX

struct {

The contents of the notification or indication. If the

application wishes to retain this mbuf for later use, it must

set this pointer to NULL to prevent the stack from freeing it.

struct os\_mbuf om;

The handle of the relevant ATT attribute.

uint16\_t attr\_handle;

The handle of the relevant connection.

uint16\_t conn\_handle;

Whether the received command is a notification or an

indication;

o 0: Notification;

o 1: Indication.

uint8\_t indication:1;

} **notify\_rx**;

Represents a transmitted ATT notification or indication, or a

completed indication transaction.

Valid for the following event types:

o BLE\_GAP\_EVENT\_NOTIFY\_TX

struct {

The status of the notification or indication transaction;

o 0: Command successfully sent;

o BLE\_HS\_EDONE: Confirmation (indication ack) received;

o BLE\_HS\_ETIMEOUT: Confirmation (indication ack) never

received;

o Other return code: Error.

int status;

The handle of the relevant connection.

uint16\_t conn\_handle;

The handle of the relevant characterstic value.

uint16\_t attr\_handle;

Whether the transmitted command is a notification or an

indication;

o 0: Notification;

o 1: Indication.

uint8\_t indication:1;

} **notify\_tx**;

Represents a state change in a peer's subscription status. In this

comment, the term "update" is used to refer to either a notification

or an indication. This event is triggered by any of the following

occurrences:

o Peer enables or disables updates via a CCCD write.

o Connection is about to be terminated and the peer is

subscribed to updates.

o Peer is now subscribed to updates after its state was restored

from persistence. This happens when bonding is restored.

Valid for the following event types:

o BLE\_GAP\_EVENT\_SUBSCRIBE

struct {

The handle of the relevant connection.

uint16\_t conn\_handle;

The value handle of the relevant characteristic.

uint16\_t attr\_handle;

One of the BLE\_GAP\_SUBSCRIBE\_REASON codes.

uint8\_t reason;

Whether the peer was previously subscribed to notifications.

uint8\_t prev\_notify:1;

Whether the peer is currently subscribed to notifications.

uint8\_t cur\_notify:1;

Whether the peer was previously subscribed to indications.

uint8\_t prev\_indicate:1;

Whether the peer is currently subscribed to indications.

uint8\_t cur\_indicate:1;

} **subscribe**;

Represents a change in an L2CAP channel's MTU.

Valid for the following event types:

o BLE\_GAP\_EVENT\_MTU

struct {

The handle of the relevant connection.

uint16\_t conn\_handle;

Indicates the channel whose MTU has been updated; either

BLE\_L2CAP\_CID\_ATT or the ID of a connection-oriented channel.

uint16\_t channel\_id;

/The channel's new MTU.

uint16\_t value;

} **mtu**;

Represents a change in peer's identity. This is issued after

successful pairing when Identity Address Information was received.

Valid for the following event types:

o BLE\_GAP\_EVENT\_IDENTITY\_RESOLVED

struct {

The handle of the relevant connection.

uint16\_t conn\_handle;

} **identity\_resolved**;

Represents a peer's attempt to pair despite a bond already existing.

The application has two options for handling this event type:

o Retry: Return BLE\_GAP\_REPEAT\_PAIRING\_RETRY after deleting the

conflicting bond. The stack will verify the bond has

been deleted and continue the pairing procedure. If

the bond is still present, this event will be reported

again.

o Ignore: Return BLE\_GAP\_REPEAT\_PAIRING\_IGNORE. The stack will

silently ignore the pairing request.

Valid for the following event types:

o BLE\_GAP\_EVENT\_REPEAT\_PAIRING

struct **ble\_gap\_repeat\_pairing repeat\_pairing**;

Represents a change of PHY. This is issue after successful

change on PHY.

struct {

int status;

uint16\_t conn\_handle;

Indicates enabled TX/RX PHY. Possible values:

o BLE\_GAP\_LE\_PHY\_1M

o BLE\_GAP\_LE\_PHY\_2M

o BLE\_GAP\_LE\_PHY\_CODED

uint8\_t tx\_phy;

uint8\_t rx\_phy;

} **phy\_updated**;

};

}

**3.3** **NIMBLE GATT Struct**

BLE gatt error struct.

struct **ble\_gatt\_error** {

uint16\_t status;

uint16\_t att\_handle;

};

BLE gatt sevices struct.

struct **ble\_gatt\_svc** {

uint16\_t start\_handle;

uint16\_t end\_handle;

ble\_uuid\_any\_t uuid;

};

BLE gatt attribution struct.

struct **ble\_gatt\_attr** {

uint16\_t handle;

uint16\_t offset;

struct os\_mbuf om;

};

BLE gatt characteristic struct.

struct **ble\_gatt\_chr** {

uint16\_t def\_handle;

uint16\_t val\_handle;

uint8\_t properties;

ble\_uuid\_any\_t uuid;

};

BLE gatt description for callback.

struct **ble\_gatt\_dsc** {

uint16\_t handle;

ble\_uuid\_any\_t uuid;

};

BLE gatt MTU callback function type.

**typedef int ble\_gatt\_mtu\_fn(uint16\_t conn\_handle,**

**const struct ble\_gatt\_error error,**

**uint16\_t mtu, void arg);**

BLE gatt discovery services callback function type.

**typedef int** **ble\_gatt\_disc\_svc\_fn(uint16\_t conn\_handle,**

**const struct ble\_gatt\_error error,**

**const struct ble\_gatt\_svc service,**

**void arg);**

The host will free the attribute mbuf automatically after the callback is

executed. The application can take ownership of the mbuf and prevent it

from being freed by assigning NULL to attr->om.

**typedef int ble\_gatt\_attr\_fn(uint16\_t conn\_handle,**

**const struct ble\_gatt\_error error,**

**struct ble\_gatt\_attr attr,**

**void arg);**

BLE gatt discovery characteristic callback function type.

**typedef int ble\_gatt\_chr\_fn(uint16\_t conn\_handle,**

**const struct ble\_gatt\_error error,**

**const struct ble\_gatt\_chr chr, void arg);**

BLE gatt discovery descriptions callback function type.

**typedef int ble\_gatt\_dsc\_fn(uint16\_t conn\_handle,**

**const struct ble\_gatt\_error error,**

**uint16\_t chr\_def\_handle,**

**const struct ble\_gatt\_dsc dsc,**

**void arg);**

Callback function type after characteristic is read or written.

**typedef int ble\_gatt\_access\_fn(uint16\_t conn\_handle, uint16\_t attr\_handle,**

**struct ble\_gatt\_access\_ctxt ctxt, void arg);**

struct **ble\_gatt\_chr\_def** {

Pointer to characteristic UUID; use BLE\_UUIDxx\_DECLARE macros to declare

proper UUID; NULL if there are no more characteristics in the service.

**const ble\_uuid\_t uuid;**

Callback that gets executed when this characteristic is read or

written.

**ble\_gatt\_access\_fn access\_cb;**

Optional argument for callback.

**void arg;**

Array of this characteristic's descriptors. NULL if no descriptors.

Do not include CCCD; it gets added automatically if this

characteristic's notify or indicate flag is set.

**struct ble\_gatt\_dsc\_def descriptors;**

Specifies the set of permitted operations for this characteristic.

**ble\_gatt\_chr\_flags flags;**

Specifies minimum required key size to access this characteristic.

**uint8\_t min\_key\_size;**

At registration time, this is filled in with the characteristic's value

attribute handle.

**uint16\_t val\_handle;**

};

struct **ble\_gatt\_svc\_def** {

One of the following:

o BLE\_GATT\_SVC\_TYPE\_PRIMARY - primary service

o BLE\_GATT\_SVC\_TYPE\_SECONDARY - secondary service

o 0 - No more services in this array.

**uint8\_t type;**

Pointer to service UUID; use BLE\_UUIDxx\_DECLARE macros to declare

proper UUID; NULL if there are no more characteristics in the service.

**const ble\_uuid\_t uuid;**

Array of pointers to other service definitions. These services are

reported as "included services" during service discovery. Terminate the

array with NULL.

**const struct ble\_gatt\_svc\_def includes;**

Array of characteristic definitions corresponding to characteristics

belonging to this service.

**const struct ble\_gatt\_chr\_def characteristics;**

};