NXP Semiconductors

User's Guide

Document identifier: EFBTPALAPIRM

Rev. 1, 23 March 2021

EdgeFast BT PAL Documentation

CONTENTS:

1 Bluetooth **1**

1.1 Connection Management 1

1.1.1 API Reference 1

1.2 Data Buffers 25

1.2.1 API Reference 25

1.3 Generic Access Profile (GAP) 27

1.3.1 API Reference 27

1.4 Generic Attribute Profile (GATT) 63

1.4.1 API Reference 65

1.4.1.1 GATT Server 70

1.4.1.2 GATT Client 83

1.5 Hands Free Profile (HFP) 91

1.5.1 API Reference 91

1.6 Logical Link Control and Adaptation Protocol (L2CAP) 107

1.6.1 API Reference 108

1.7 Serial Port Emulation (RFCOMM) 115

1.7.1 API Reference 115

1.8 Service Discovery Protocol (SDP) 118

1.8.1 API Reference 118

1.9 Advance Audio Distribution Profile (A2DP) 128

1.9.1 API Reference 128

1.10 Serial Port Profile (SPP) 136

1.10.1 API Reference 136

1.11 Universal Unique Identifiers (UUIDs) 139

1.11.1 API Reference 139

1.12 services 156

1.12.1 HTTP Proxy Service (HPS) 156

1.12.1.1 API Reference 156

1.12.2 Health Thermometer Service (HTS) 158

1.12.2.1 API Reference 158

1.12.3 Internet Protocol Support Profile (IPSP) 159

1.12.3.1 API Reference 159

1.12.4 Proximity Reporter (PXR) 160

1.12.4.1 API Reference 160

Index **163**

CHAPTER

ONE

BLUETOOTH

1.1 Connection Management

The Zephyr Bluetooth stack uses an abstraction called `bt_conn` to represent connections to other devices. The internals of this struct are not exposed to the application, but a limited amount of information (such as the remote address) can be acquired using the `bt_conn_get_info()` API. Connection objects are reference counted, and the application is expected to use the `bt_conn_ref()` API whenever storing a connection pointer for a longer period of time, since this ensures that the object remains valid (even if the connection would get disconnected). Similarly the `bt_conn_unref()` API is to be used when releasing a reference to a connection.

An application may track connections by registering a `bt_conn_cb` struct using the `bt_conn_cb_register()` API. This struct lets the application define callbacks for connection & disconnection events, as well as other events related to a connection such as a change in the security level or the connection parameters. When acting as a central the application will also get hold of the connection object through the return value of the `bt_conn_create_le()` API.

1.1.1 API Reference

group **`bt_conn`**
Connection management.

Defines

`BT_LE_CONN_PARAM_INIT`(*int_min*, *int_max*, *lat*, *to*)
Initialize connection parameters.

Parameters

- `int_min`: Minimum Connection Interval (N * 1.25 ms)
- `int_max`: Maximum Connection Interval (N * 1.25 ms)
- `lat`: Connection Latency
- `to`: Supervision Timeout (N * 10 ms)

`BT_LE_CONN_PARAM`(*int_min*, *int_max*, *lat*, *to*)
Helper to declare connection parameters inline

Parameters

- `int_min`: Minimum Connection Interval (N * 1.25 ms)
- `int_max`: Maximum Connection Interval (N * 1.25 ms)
- `lat`: Connection Latency
- `to`: Supervision Timeout (N * 10 ms)

`BT_LE_CONN_PARAM_DEFAULT`
Default LE connection parameters: Connection Interval: 30-50 ms Latency: 0 Timeout: 4 s

`BT_CONN_LE_PHY_PARAM_INIT`(*_pref_tx_phy*, *_pref_rx_phy*)
Initialize PHY parameters

Parameters

- `_pref_tx_phy`: Bitmask of preferred transmit PHYs.
- `_pref_rx_phy`: Bitmask of preferred receive PHYs.

`BT_CONN_LE_PHY_PARAM`(*_pref_tx_phy*, *_pref_rx_phy*)
Helper to declare PHY parameters inline

Parameters

- `_pref_tx_phy`: Bitmask of preferred transmit PHYs.
- `_pref_rx_phy`: Bitmask of preferred receive PHYs.

`BT_CONN_LE_PHY_PARAM_1M`
Only LE 1M PHY

`BT_CONN_LE_PHY_PARAM_2M`
Only LE 2M PHY

`BT_CONN_LE_PHY_PARAM_CODED`
Only LE Coded PHY.

`BT_CONN_LE_PHY_PARAM_ALL`
All LE PHYs.

`BT_CONN_LE_DATA_LEN_PARAM_INIT`(*_tx_max_len*, *_tx_max_time*)
Initialize transmit data length parameters

Parameters

- `_tx_max_len`: Maximum Link Layer transmission payload size in bytes.
- `_tx_max_time`: Maximum Link Layer transmission payload time in us.

`BT_CONN_LE_DATA_LEN_PARAM`(*_tx_max_len*, *_tx_max_time*)
Helper to declare transmit data length parameters inline

Parameters

- `_tx_max_len`: Maximum Link Layer transmission payload size in bytes.
- `_tx_max_time`: Maximum Link Layer transmission payload time in us.

BT_LE_DATA_LEN_PARAM_DEFAULT
Default LE data length parameters.

BT_LE_DATA_LEN_PARAM_MAX
Maximum LE data length parameters.

BT_CONN_LE_CREATE_PARAM_INIT(*_options*, *_interval*, *_window*)
Initialize create connection parameters.

Parameters

- `_options`: Create connection options.
- `_interval`: Create connection scan interval (N * 0.625 ms).
- `_window`: Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_PARAM(*_options*, *_interval*, *_window*)
Helper to declare create connection parameters inline

Parameters

- `_options`: Create connection options.
- `_interval`: Create connection scan interval (N * 0.625 ms).
- `_window`: Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_CONN
Default LE create connection parameters. Scan continuously by setting scan interval equal to scan window.

BT_CONN_LE_CREATE_CONN_AUTO
Default LE create connection using whitelist parameters. Scan window: 30 ms. Scan interval: 60 ms.

BT_PASSKEY_INVALID
Special passkey value that can be used to disable a previously set fixed passkey.

BT_BR_CONN_PARAM_INIT(*role_switch*)
Initialize BR/EDR connection parameters.

Parameters

- `role_switch`: True if role switch is allowed

BT_BR_CONN_PARAM(*role_switch*)
Helper to declare BR/EDR connection parameters inline

Parameters

- `role_switch`: True if role switch is allowed

BT_BR_CONN_PARAM_DEFAULT
Default BR/EDR connection parameters: Role switch allowed

Typedefs

`typedef enum` *_bt_security* **`bt_security_t`**

Enums

`enum` **[anonymous]**
Connection PHY options

Values:

`enumerator BT_CONN_LE_PHY_OPT_NONE`
Convenience value when no options are specified.

`enumerator BT_CONN_LE_PHY_OPT_CODED_S2`
LE Coded using S=2 coding preferred when transmitting.

`enumerator BT_CONN_LE_PHY_OPT_CODED_S8`
LE Coded using S=8 coding preferred when transmitting.

`enum` **[anonymous]**
Connection Type

Values:

`enumerator BT_CONN_TYPE_LE`
LE Connection Type

`enumerator BT_CONN_TYPE_BR`
BR/EDR Connection Type

`enumerator BT_CONN_TYPE_SCO`
SCO Connection Type

`enumerator BT_CONN_TYPE_ISO`
ISO Connection Type

`enumerator BT_CONN_TYPE_ALL`
All Connection Type

`enum` **[anonymous]**
Connection role (master or slave)

Values:

`enumerator BT_CONN_ROLE_MASTER`

`enumerator BT_CONN_ROLE_SLAVE`

`enum bt_conn_le_tx_power_phy`
Values:

`enumerator BT_CONN_LE_TX_POWER_PHY_NONE`
Convenience macro for when no PHY is set.

`enumerator BT_CONN_LE_TX_POWER_PHY_1M`
LE 1M PHY

`enumerator BT_CONN_LE_TX_POWER_PHY_2M`
LE 2M PHY

`enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S8`
LE Coded PHY using S=8 coding.

enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S2
LE Coded PHY using S=2 coding.

enum [anonymous]
Values:

enumerator BT_CONN_LE_OPT_NONE
Convenience value when no options are specified.

enumerator BT_CONN_LE_OPT_CODED
Enable LE Coded PHY.

Enable scanning on the LE Coded PHY.

enumerator BT_CONN_LE_OPT_NO_1M
Disable LE 1M PHY.

Disable scanning on the LE 1M PHY.

Note Requires *BT_CONN_LE_OPT_CODED*.

enum _bt_security
Security level.

Values:

enumerator BT_SECURITY_L0
Level 0: Only for BR/EDR special cases, like SDP

enumerator BT_SECURITY_L1
Level 1: No encryption and no authentication.

enumerator BT_SECURITY_L2
Level 2: Encryption and no authentication (no MITM).

enumerator BT_SECURITY_L3
Level 3: Encryption and authentication (MITM).

enumerator BT_SECURITY_L4
Level 4: Authenticated Secure Connections and 128-bit key.

enumerator BT_SECURITY_FORCE_PAIR
Bit to force new pairing procedure, bit-wise OR with requested security level.

enum bt_security_err
Values:

enumerator BT_SECURITY_ERR_SUCCESS
Security procedure successful.

enumerator BT_SECURITY_ERR_AUTH_FAIL
Authentication failed.

enumerator BT_SECURITY_ERR_PIN_OR_KEY_MISSING
PIN or encryption key is missing.

enumerator BT_SECURITY_ERR_OOB_NOT_AVAILABLE
OOB data is not available.

enumerator BT_SECURITY_ERR_AUTH_REQUIREMENT
The requested security level could not be reached.

enumerator BT_SECURITY_ERR_PAIR_NOT_SUPPORTED
Pairing is not supported

enumerator BT_SECURITY_ERR_PAIR_NOT_ALLOWED
Pairing is not allowed.

enumerator BT_SECURITY_ERR_INVALID_PARAM
Invalid parameters.

enumerator BT_SECURITY_ERR_UNSPECIFIED
Pairing failed but the exact reason could not be specified.

Functions

struct bt_conn ***bt_conn_ref**(**struct** bt_conn **conn*)
Increment a connection's reference count.

Increment the reference count of a connection object.

Note Will return NULL if the reference count is zero.

Return Connection object with incremented reference count, or NULL if the reference count is zero.

Parameters

- `conn`: Connection object.

void **bt_conn_unref**(**struct** bt_conn **conn*)
Decrement a connection's reference count.

Decrement the reference count of a connection object.

Parameters

- `conn`: Connection object.

void **bt_conn_foreach**(int *type*, void (**func*)) **struct** bt_conn *conn, void *data
, void **data*Iterate through all existing connections.

Parameters

- `type`: Connection Type
- `func`: Function to call for each connection.
- `data`: Data to pass to the callback function.

struct bt_conn ***bt_conn_lookup_addr_le**(uint8_t *id*, **const** *bt_addr_le_t* **peer*)
Look up an existing connection by address.

Look up an existing connection based on the remote address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Return Connection object or NULL if not found.

Parameters

- `id`: Local identity (in most cases BT_ID_DEFAULT).
- `peer`: Remote address.

const *bt_addr_le_t* ***bt_conn_get_dst*** (**const struct** bt_conn **conn*)

Get destination (peer) address of a connection.

Return Destination address.

Parameters

- `conn`: Connection object.

const *bt_addr_t* ***bt_conn_get_dst_br*** (**const struct** bt_conn **conn*)

Get destination (peer) address of a BR connection.

Return Destination address.

Parameters

- `conn`: Connection object.

uint8_t **bt_conn_index** (**struct** bt_conn **conn*)

Get array index of a connection.

This function is used to map bt_conn to index of an array of connections. The array has CONFIG_BT_MAX_CONN elements.

Return Index of the connection object. The range of the returned value is 0..CONFIG_BT_MAX_CONN-1

Parameters

- `conn`: Connection object.

int **bt_conn_get_info** (**const struct** bt_conn **conn*, **struct** *bt_conn_info* **info*)

Get connection info.

Return Zero on success or (negative) error code on failure.

Parameters

- `conn`: Connection object.
- `info`: Connection info object.

int **bt_conn_get_remote_info** (**struct** bt_conn **conn*, **struct** *bt_conn_remote_info* **remote_info*)

Get connection info for the remote device.

Note In order to retrieve the remote version (version, manufacturer and subversion) `CONFIG_BT_REMOTE_VERSION` must be enabled

The remote information is exchanged directly after the connection has been established. The application can be notified about when the remote information is available through the remote_info_available callback.

Return Zero on success or (negative) error code on failure.

-EBUSY The remote information is not yet available.

Parameters

- `conn`: Connection object.
- `remote_info`: Connection remote info object.

int **bt_conn_le_get_tx_power_level**(**struct** bt_conn **conn*, **struct** *bt_conn_le_tx_power* **tx_power_level*)

Get connection transmit power level.

Return Zero on success or (negative) error code on failure.

-ENOBUFS HCI command buffer is not available.

Parameters

- `conn`: Connection object.
- `tx_power_level`: Transmit power level descriptor.

int **bt_conn_le_param_update**(**struct** bt_conn **conn*, **const struct** *bt_le_conn_param* **param*)

Update the connection parameters.

If the local device is in the peripheral role then updating the connection parameters will be delayed. This delay can be configured by through the `CONFIG_BT_CONN_PARAM_UPDATE_TIMEOUT` option.

Return Zero on success or (negative) error code on failure.

Parameters

- `conn`: Connection object.
- `param`: Updated connection parameters.

int **bt_conn_le_data_len_update**(**struct** bt_conn **conn*, **const struct** *bt_conn_le_data_len_param* **param*)

Update the connection transmit data length parameters.

Return Zero on success or (negative) error code on failure.

Parameters

- `conn`: Connection object.
- `param`: Updated data length parameters.

int **bt_conn_le_phy_update**(**struct** bt_conn **conn*, **const struct** *bt_conn_le_phy_param* **param*)

Update the connection PHY parameters.

Update the preferred transmit and receive PHYs of the connection. Use *BT_GAP_LE_PHY_NONE* to indicate no preference.

Return Zero on success or (negative) error code on failure.

Parameters

- `conn`: Connection object.
- `param`: Updated connection parameters.

int **bt_conn_disconnect**(**struct** bt_conn **conn*, uint8_t *reason*)

Disconnect from a remote device or cancel pending connection.

Disconnect an active connection with the specified reason code or cancel pending outgoing connection.

The disconnect reason for a normal disconnect should be: BT_HCI_ERR_REMOTE_USER_TERM_CONN.

The following disconnect reasons are accepted:

- BT_HCI_ERR_AUTH_FAIL
- BT_HCI_ERR_REMOTE_USER_TERM_CONN
- BT_HCI_ERR_REMOTE_LOW_RESOURCES
- BT_HCI_ERR_REMOTE_POWER_OFF
- BT_HCI_ERR_UNSUPP_REMOTE_FEATURE
- BT_HCI_ERR_PAIRING_NOT_SUPPORTED
- BT_HCI_ERR_UNACCEPT_CONN_PARAM

Return Zero on success or (negative) error code on failure.

Parameters

- `conn`: Connection to disconnect.
- `reason`: Reason code for the disconnection.

int **bt_conn_le_create**(**const** *bt_addr_le_t* **peer*, **const struct** *bt_conn_le_create_param* **create_param*, **const struct** *bt_le_conn_param* **conn_param*, **struct** bt_conn ***conn*)

Initiate an LE connection to a remote device.

Allows initiate new LE link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

This uses the General Connection Establishment procedure.

Return Zero on success or (negative) error code on failure.

Parameters

- `[in] peer`: Remote address.
- `[in] create_param`: Create connection parameters.
- `[in] conn_param`: Initial connection parameters.
- `[out] conn`: Valid connection object on success.

int **bt_conn_le_create_auto**(**const struct** *bt_conn_le_create_param* **create_param*, **const struct** *bt_le_conn_param* **conn_param*)

Automatically connect to remote devices in whitelist.

This uses the Auto Connection Establishment procedure. The procedure will continue until a single connection is established or the procedure is stopped through *bt_conn_create_auto_stop*. To establish connections to all devices in the whitelist the procedure should be started again in the connected callback after a new connection has been established.

Return Zero on success or (negative) error code on failure.

-ENOMEM No free connection object available.

Parameters

- `create_param`: Create connection parameters
- `conn_param`: Initial connection parameters.

int **`bt_conn_create_auto_stop`**(void)

Stop automatic connect creation.

Return Zero on success or (negative) error code on failure.

int **`bt_le_set_auto_conn`**(**`const`** *bt_addr_le_t* **addr*, **`const struct`** *bt_le_conn_param* **param*)

Automatically connect to remote device if it's in range.

This function enables/disables automatic connection initiation. Every time the device loses the connection with peer, this connection will be re-established if connectable advertisement from peer is received.

Note Auto connect is disabled during explicit scanning.

Return Zero on success or error code otherwise.

Parameters

- `addr`: Remote Bluetooth address.
- `param`: If non-NULL, auto connect is enabled with the given parameters. If NULL, auto connect is disabled.

int **`bt_conn_set_security`**(**`struct`** bt_conn **conn*, *bt_security_t* *sec*)

Set security level for a connection.

This function enable security (encryption) for a connection. If the device has bond information for the peer with sufficiently strong key encryption will be enabled. If the connection is already encrypted with sufficiently strong key this function does nothing.

If the device has no bond information for the peer and is not already paired then the pairing procedure will be initiated. If the device has bond information or is already paired and the keys are too weak then the pairing procedure will be initiated.

This function may return error if required level of security is not possible to achieve due to local or remote device limitation (e.g., input output capabilities), or if the maximum number of paired devices has been reached.

This function may return error if the pairing procedure has already been initiated by the local device or the peer device.

Note When `CONFIG_BT_SMP_SC_ONLY` is enabled then the security level will always be level 4.

When `CONFIG_BT_SMP_OOB_LEGACY_PAIR_ONLY` is enabled then the security level will always be level 3.

Return 0 on success or negative error

Parameters

- `conn`: Connection object.

- `sec`: Requested security level.

bt_security_t **`bt_conn_get_security`**(**`struct`** bt_conn **conn*)

Get security level for a connection.

Return Connection security level

uint8_t **`bt_conn_enc_key_size`**(**`struct`** bt_conn **conn*)

Get encryption key size.

This function gets encryption key size. If there is no security (encryption) enabled 0 will be returned.

Return Encryption key size.

Parameters

- `conn`: Existing connection object.

void **`bt_conn_cb_register`**(**`struct`** *bt_conn_cb* **cb*)

Register connection callbacks.

Register callbacks to monitor the state of connections.

Parameters

- `cb`: Callback struct. Must point to memory that remains valid.

void **`bt_set_bondable`**(bool *enable*)

Enable/disable bonding.

Set/clear the Bonding flag in the Authentication Requirements of SMP Pairing Request/Response data. The initial value of this flag depends on BT_BONDABLE Kconfig setting. For the vast majority of applications calling this function shouldn't be needed.

Parameters

- `enable`: Value allowing/disallowing to be bondable.

void **`bt_set_oob_data_flag`**(bool *enable*)

Allow/disallow remote OOB data to be used for pairing.

Set/clear the OOB data flag for SMP Pairing Request/Response data. The initial value of this flag depends on BT_OOB_DATA_PRESENT Kconfig setting.

Parameters

- `enable`: Value allowing/disallowing remote OOB data.

int **`bt_le_oob_set_legacy_tk`**(**`struct`** bt_conn **conn*, **`const`** uint8_t **tk*)

Set OOB Temporary Key to be used for pairing.

This function allows to set OOB data for the LE legacy pairing procedure. The function should only be called in response to the oob_data_request() callback provided that the legacy method is user pairing.

Return Zero on success or -EINVAL if NULL

Parameters

- `conn`: Connection object
- `tk`: Pointer to 16 byte long TK array

int **`bt_le_oob_set_sc_data`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_le_oob_sc_data *oobd_local*, **`const struct`** *bt_le_oob_sc_data *oobd_remote*)

Set OOB data during LE Secure Connections (SC) pairing procedure.

This function allows to set OOB data during the LE SC pairing procedure. The function should only be called in response to the oob_data_request() callback provided that LE SC method is used for pairing.

The user should submit OOB data according to the information received in the callback. This may yield three different configurations: with only local OOB data present, with only remote OOB data present or with both local and remote OOB data present.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `conn`: Connection object
- `oobd_local`: Local OOB data or NULL if not present
- `oobd_remote`: Remote OOB data or NULL if not present

int **`bt_le_oob_get_sc_data`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_le_oob_sc_data **oobd_local*, **`const struct`** *bt_le_oob_sc_data **oobd_remote*)

Get OOB data used for LE Secure Connections (SC) pairing procedure.

This function allows to get OOB data during the LE SC pairing procedure that were set by the *bt_le_oob_set_sc_data()* API.

Note The OOB data will only be available as long as the connection object associated with it is valid.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `conn`: Connection object
- `oobd_local`: Local OOB data or NULL if not set
- `oobd_remote`: Remote OOB data or NULL if not set

int **`bt_passkey_set`**(unsigned int *passkey*)

Set a fixed passkey to be used for pairing.

This API is only available when the CONFIG_BT_FIXED_PASSKEY configuration option has been enabled.

Sets a fixed passkey to be used for pairing. If set, the pairing_confim() callback will be called for all incoming pairings.

Return 0 on success or a negative error code on failure.

Parameters

- `passkey`: A valid passkey (0 - 999999) or BT_PASSKEY_INVALID to disable a previously set fixed passkey.

int **`bt_conn_auth_cb_register`**(**const struct** *bt_conn_auth_cb* **cb*)

Register authentication callbacks.

Register callbacks to handle authenticated pairing. Passing NULL unregisters a previous callbacks structure.

Return Zero on success or negative error code otherwise

Parameters

- `cb`: Callback struct.

int **`bt_conn_auth_passkey_entry`**(**struct** bt_conn **conn*, unsigned int *passkey*)

Reply with entered passkey.

This function should be called only after passkey_entry callback from *bt_conn_auth_cb* structure was called.

Return Zero on success or negative error code otherwise

Parameters

- `conn`: Connection object.
- `passkey`: Entered passkey.

int **`bt_conn_auth_cancel`**(**struct** bt_conn **conn*)

Cancel ongoing authenticated pairing.

This function allows to cancel ongoing authenticated pairing.

Return Zero on success or negative error code otherwise

Parameters

- `conn`: Connection object.

int **`bt_conn_auth_passkey_confirm`**(**struct** bt_conn **conn*)

Reply if passkey was confirmed to match by user.

This function should be called only after passkey_confirm callback from *bt_conn_auth_cb* structure was called.

Return Zero on success or negative error code otherwise

Parameters

- `conn`: Connection object.

int **`bt_conn_auth_pairing_confirm`**(**struct** bt_conn **conn*)

Reply if incoming pairing was confirmed by user.

This function should be called only after pairing_confirm callback from *bt_conn_auth_cb* structure was called if user confirmed incoming pairing.

Return Zero on success or negative error code otherwise

Parameters

- `conn`: Connection object.

int **`bt_conn_auth_pincode_entry`**(**`struct`** bt_conn **conn*, **`const`** char **pin*)

Reply with entered PIN code.

This function should be called only after PIN code callback from *bt_conn_auth_cb* structure was called. It's for legacy 2.0 devices.

Return Zero on success or negative error code otherwise

Parameters

- `conn`: Connection object.
- `pin`: Entered PIN code.

`struct` bt_conn ***`bt_conn_create_br`**(**`const`** *bt_addr_t* **peer*, **`const`** **`struct`** *bt_br_conn_param* **param*)

Initiate an BR/EDR connection to a remote device.

Allows initiate new BR/EDR link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Return Valid connection object on success or NULL otherwise.

Parameters

- `peer`: Remote address.
- `param`: Initial connection parameters.

`struct` bt_conn ***`bt_conn_create_sco`**(**`const`** *bt_addr_t* **peer*)

Initiate an SCO connection to a remote device.

Allows initiate new SCO link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Return Valid connection object on success or NULL otherwise.

Parameters

- `peer`: Remote address.

`struct bt_le_conn_param`

#include <conn.h> Connection parameters for LE connections

`struct bt_conn_le_phy_info`

#include <conn.h> Connection PHY information for LE connections

Public Members

uint8_t **`rx_phy`**

Connection transmit PHY

`struct bt_conn_le_phy_param`

#include <conn.h> Preferred PHY parameters for LE connections

Public Members

uint8_t **`pref_tx_phy`**

Connection PHY options.

uint8_t **`pref_rx_phy`**

Bitmask of preferred transmit PHYs

`struct bt_conn_le_data_len_info`

#include <conn.h> Connection data length information for LE connections

Public Members

uint16_t **`tx_max_len`**

Maximum Link Layer transmission payload size in bytes.

uint16_t **`tx_max_time`**

Maximum Link Layer transmission payload time in us.

uint16_t **`rx_max_len`**

Maximum Link Layer reception payload size in bytes.

uint16_t **`rx_max_time`**

Maximum Link Layer reception payload time in us.

`struct bt_conn_le_data_len_param`

#include <conn.h> Connection data length parameters for LE connections

Public Members

uint16_t **`tx_max_len`**

Maximum Link Layer transmission payload size in bytes.

uint16_t **`tx_max_time`**

Maximum Link Layer transmission payload time in us.

`struct bt_conn_le_info`

#include <conn.h> LE Connection Info Structure

Public Members

const *bt_addr_le_t* ***src**
Source (Local) Identity Address

const *bt_addr_le_t* ***dst**
Destination (Remote) Identity Address or remote Resolvable Private Address (RPA) before identity has been resolved.

const *bt_addr_le_t* ***local**
Local device address used during connection setup.

const *bt_addr_le_t* ***remote**
Remote device address used during connection setup.

uint16_t **latency**
Connection interval

uint16_t **timeout**
Connection slave latency

struct *bt_conn_le_phy_info* ***phy**
Connection supervision timeout

struct bt_conn_br_info
#include <conn.h> BR/EDR Connection Info Structure

struct bt_conn_info
#include <conn.h> Connection Info Structure

Public Members

uint8_t **type**
Connection Type.

uint8_t **role**
Connection Role.

uint8_t **id**
Which local identity the connection was created with

union *bt_conn_info*.**[anonymous] [anonymous]**
Connection Type specific Info.

union *bt_conn_info*.**__unnamed__**
Connection Type specific Info.

Public Members

struct *bt_conn_le_info* **le**
LE Connection specific Info.

struct *bt_conn_br_info* **br**
BR/EDR Connection specific Info.

struct bt_conn_le_remote_info
#include <conn.h> LE Connection Remote Info Structure

Public Members

const uint8_t ***features**
Remote LE feature set (bitmask).

struct bt_conn_br_remote_info
#include <conn.h> BR/EDR Connection Remote Info structure

Public Members

const uint8_t ***features**
Remote feature set (pages of bitmasks).

uint8_t **num_pages**
Number of pages in the remote feature set.

struct bt_conn_remote_info
#include <conn.h> Connection Remote Info Structure.

Note The version, manufacturer and subversion fields will only contain valid data if `CONFIG_BT_REMOTE_VERSION` is enabled.

Public Members

uint8_t **type**
Connection Type

uint8_t **version**
Remote Link Layer version

uint16_t **manufacturer**
Remote manufacturer identifier

uint16_t **subversion**
Per-manufacturer unique revision

union *bt_conn_remote_info*.**__unnamed__**

Public Members

struct *bt_conn_le_remote_info* **le**
LE connection remote info

struct *bt_conn_br_remote_info* **br**
BR/EDR connection remote info

struct bt_conn_le_tx_power
#include <conn.h> LE Transmit Power Level Structure

Public Members

uint8_t **`phy`**
Input: 1M, 2M, Coded S2 or Coded S8

int8_t **`current_level`**
Output: current transmit power level

int8_t **`max_level`**
Output: maximum transmit power level

`struct bt_conn_le_create_param`
#include <conn.h>

Public Members

uint32_t **`options`**
Bit-field of create connection options.

uint16_t **`interval`**
Scan interval (N * 0.625 ms)

uint16_t **`window`**
Scan window (N * 0.625 ms)

uint16_t **`interval_coded`**
Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval

uint16_t **`window_coded`**
Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

uint16_t **`timeout`**
Connection initiation timeout (N * 10 MS)

Set zero to use the default `CONFIG_BT_CREATE_CONN_TIMEOUT` timeout.

Note Unused in *bt_conn_le_create_auto*

`struct bt_conn_cb`
#include <conn.h> Connection callback structure.

This structure is used for tracking the state of a connection. It is registered with the help of the *bt_conn_cb_register()* API. It's permissible to register multiple instances of this *bt_conn_cb* type, in case different modules of an application are interested in tracking the connection state. If a callback is not of interest for an instance, it may be set to NULL and will as a consequence not be used for that instance.

Public Members

void (***connected**)(**struct** bt_conn *conn, uint8_t err)

A new connection has been established.

This callback notifies the application of a new connection. In case the err parameter is non-zero it means that the connection establishment failed.

`err` can mean either of the following:

- BT_HCI_ERR_UNKNOWN_CONN_ID Creating the connection started by *bt_conn_le_create* was canceled either by the user through *bt_conn_disconnect* or by the timeout in the host through *bt_conn_le_create_param* timeout parameter, which defaults to `CONFIG_BT_CREATE_CONN_TIMEOUT` seconds.
- `BT_HCI_ERR_ADV_TIMEOUT` High duty cycle directed connectable advertiser started by *bt_le_adv_start* failed to be connected within the timeout.

Parameters

- `conn`: New connection object.
- `err`: HCI error. Zero for success, non-zero otherwise.

void (***disconnected**)(**struct** bt_conn *conn, uint8_t reason)

A connection has been disconnected.

This callback notifies the application that a connection has been disconnected.

When this callback is called the stack still has one reference to the connection object. If the application in this callback tries to start either a connectable advertiser or create a new connection this might fail because there are no free connection objects available. To avoid this issue it is recommended to either start connectable advertise or create a new connection using k_work_submit or increase `CONFIG_BT_MAX_CONN` .

Parameters

- `conn`: Connection object.
- `reason`: HCI reason for the disconnection.

bool (***le_param_req**)(**struct** bt_conn *conn, **struct** *bt_le_conn_param* *param)

LE connection parameter update request.

This callback notifies the application that a remote device is requesting to update the connection parameters. The application accepts the parameters by returning true, or rejects them by returning false. Before accepting, the application may also adjust the parameters to better suit its needs.

It is recommended for an application to have just one of these callbacks for simplicity. However, if an application registers multiple it needs to manage the potentially different requirements for each callback. Each callback gets the parameters as returned by previous callbacks, i.e. they are not necessarily the same ones as the remote originally sent.

Return true to accept the parameters, or false to reject them.

Parameters

- `conn`: Connection object.
- `param`: Proposed connection parameters.

void (***le_param_updated**)(**struct** bt_conn *conn, uint16_t interval, uint16_t latency, uint16_t timeout)

The parameters for an LE connection have been updated.

This callback notifies the application that the connection parameters for an LE connection have been updated.

Parameters
- `conn`: Connection object.
- `interval`: Connection interval.
- `latency`: Connection latency.
- `timeout`: Connection supervision timeout.

void (***identity_resolved**)(**struct** bt_conn *conn, **const** *bt_addr_le_t* *rpa, **const** *bt_addr_le_t* *identity)

Remote Identity Address has been resolved.

This callback notifies the application that a remote Identity Address has been resolved

Parameters
- `conn`: Connection object.
- `rpa`: Resolvable Private Address.
- `identity`: Identity Address.

void (***security_changed**)(**struct** bt_conn *conn, *bt_security_t* level, **enum** *bt_security_err* err)

The security level of a connection has changed.

This callback notifies the application that the security of a connection has changed.

The security level of the connection can either have been increased or remain unchanged. An increased security level means that the pairing procedure has been performed or the bond information from a previous connection has been applied. If the security level remains unchanged this means that the encryption key has been refreshed for the connection.

Parameters
- `conn`: Connection object.
- `level`: New security level of the connection.
- `err`: Security error. Zero for success, non-zero otherwise.

void (***remote_info_available**)(**struct** bt_conn *conn, **struct** *bt_conn_remote_info* *remote_info)

Remote information procedures has completed.

This callback notifies the application that the remote information has been retrieved from the remote peer.

Parameters
- `conn`: Connection object.
- `remote_info`: Connection information of remote device.

void (***le_phy_updated**)(**struct** bt_conn *conn, **struct** *bt_conn_le_phy_info* *param)

The PHY of the connection has changed.

This callback notifies the application that the PHY of the connection has changed.

Parameters
- `conn`: Connection object.
- `info`: Connection LE PHY information.

void (***le_data_len_updated**)(**struct** bt_conn *conn, **struct** *bt_conn_le_data_len_info* *info)

The data length parameters of the connection has changed.

This callback notifies the application that the maximum Link Layer payload length or transmission time has changed.

Parameters
- `conn`: Connection object.
- `info`: Connection data length information.

`struct bt_conn_oob_info`
#include <conn.h> Info Structure for OOB pairing

Public Types

`enum` [anonymous]
Type of OOB pairing method

Values:

`enumerator BT_CONN_OOB_LE_LEGACY`
LE legacy pairing

`enumerator BT_CONN_OOB_LE_SC`
LE SC pairing

Public Members

`enum` *bt_conn_oob_info*.***[anonymous]*** **`type`**
Type of OOB pairing method

`union` *`bt_conn_oob_info`*.**`__unnamed__`**

Public Members

`struct` *bt_conn_oob_info*.**[anonymous]**.**[anonymous]** **`lesc`**
LE Secure Connections OOB pairing parameters

`struct` *`bt_conn_oob_info.__unnamed__`*.**`lesc`**
LE Secure Connections OOB pairing parameters

Public Members

`enum` *bt_conn_oob_info*.**[anonymous]**.**[anonymous]**.**[anonymous]** **`oob_config`**
OOB data configuration

`struct bt_conn_pairing_feat`
#include <conn.h> Pairing request and pairing response info structure.

This structure is the same for both smp_pairing_req and smp_pairing_rsp and a subset of the packet data, except for the initial Code octet. It is documented in Core Spec. Vol. 3, Part H, 3.5.1 and 3.5.2.

Public Members

uint8_t **io_capability**
IO Capability, Core Spec. Vol 3, Part H, 3.5.1, Table 3.4

uint8_t **oob_data_flag**
OOB data flag, Core Spec. Vol 3, Part H, 3.5.1, Table 3.5

uint8_t **auth_req**
AuthReq, Core Spec. Vol 3, Part H, 3.5.1, Fig. 3.3

uint8_t **max_enc_key_size**
Maximum Encryption Key Size, Core Spec. Vol 3, Part H, 3.5.1

uint8_t **init_key_dist**
Initiator Key Distribution/Generation, Core Spec. Vol 3, Part H, 3.6.1, Fig. 3.11

uint8_t **resp_key_dist**
Responder Key Distribution/Generation, Core Spec. Vol 3, Part H 3.6.1, Fig. 3.11

struct bt_conn_auth_cb
#include <conn.h> Authenticated pairing callback structure

Public Members

enum *bt_security_err* (***pairing_accept**)(**struct** bt_conn *conn, **const struct** *bt_conn_pairing_feat* ***const** feat)
Query to proceed incoming pairing or not.

On any incoming pairing req/rsp this callback will be called for the application to decide whether to allow for the pairing to continue.

The pairing info received from the peer is passed to assist making the decision.

As this callback is synchronous the application should return a response value immediately. Otherwise it may affect the timing during pairing. Hence, this information should not be conveyed to the user to take action.

The remaining callbacks are not affected by this, but do notice that other callbacks can be called during the pairing. Eg. if pairing_confirm is registered both will be called for Just-Works pairings.

This callback may be unregistered in which case pairing continues as if the Kconfig flag was not set.

This callback is not called for BR/EDR Secure Simple Pairing (SSP).

Parameters
- `conn`: Connection where pairing is initiated.
- `feat`: Pairing req/resp info.

void (***passkey_display**)(**struct** bt_conn *conn, unsigned int passkey)
Display a passkey to the user.

When called the application is expected to display the given passkey to the user, with the expectation that the passkey will then be entered on the peer device. The passkey will be in the range of 0 - 999999, and is expected to be padded with zeroes so that six digits are always shown. E.g. the value 37 should be shown as 000037.

This callback may be set to NULL, which means that the local device lacks the ability do display a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop displaying the passkey.

Parameters
- `conn`: Connection where pairing is currently active.
- `passkey`: Passkey to show to the user.

void (***passkey_entry**)(**struct** bt_conn *conn)

Request the user to enter a passkey.

When called the user is expected to enter a passkey. The passkey must be in the range of 0 - 999999, and should be expected to be zero-padded, as that's how the peer device will typically be showing it (e.g. 37 would be shown as 000037).

Once the user has entered the passkey its value should be given to the stack using the *bt_conn_auth_passkey_entry()* API.

This callback may be set to NULL, which means that the local device lacks the ability to enter a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to enter a passkey.

Parameters
- `conn`: Connection where pairing is currently active.

void (***passkey_confirm**)(**struct** bt_conn *conn, unsigned int passkey)

Request the user to confirm a passkey.

When called the user is expected to confirm that the given passkey is also shown on the peer device.. The passkey will be in the range of 0 - 999999, and should be zero-padded to always be six digits (e.g. 37 would be shown as 000037).

Once the user has confirmed the passkey to match, the *bt_conn_auth_passkey_confirm()* API should be called. If the user concluded that the passkey doesn't match the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a passkey.

Parameters
- `conn`: Connection where pairing is currently active.
- `passkey`: Passkey to be confirmed.

void (***oob_data_request**)(**struct** bt_conn *conn, **struct** *bt_conn_oob_info* *info)

Request the user to provide Out of Band (OOB) data.

When called the user is expected to provide OOB data. The required data are indicated by the information structure.

For LE Secure Connections OOB pairing, the user should provide local OOB data, remote OOB data or both depending on their availability. Their value should be given to the stack using the *bt_le_oob_set_sc_data()* API.

This callback must be set to non-NULL in order to support OOB pairing.

Parameters
- `conn`: Connection where pairing is currently active.
- `info`: OOB pairing information.

void (***cancel**)(**struct** bt_conn *conn)

Cancel the ongoing user request.

This callback will be called to notify the application that it should cancel any previous user request (passkey display, entry or confirmation).

This may be set to NULL, but must always be provided whenever the passkey_display, passkey_entry passkey_confirm or pairing_confirm callback has been provided.

Parameters
- `conn`: Connection where pairing is currently active.

void (***`pairing_confirm`**)(**`struct`** bt_conn *conn)

Request confirmation for an incoming pairing.

This callback will be called to confirm an incoming pairing request where none of the other user callbacks is applicable.

If the user decides to accept the pairing the *bt_conn_auth_pairing_confirm()* API should be called. If the user decides to reject the pairing the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a pairing request. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a pairing request.

Parameters
- `conn`: Connection where pairing is currently active.

void (***`pincode_entry`**)(**`struct`** bt_conn *conn, bool highsec)

Request the user to enter a passkey.

This callback will be called for a BR/EDR (Bluetooth Classic) connection where pairing is being performed. Once called the user is expected to enter a PIN code with a length between 1 and 16 digits. If the *highsec* parameter is set to true the PIN code must be 16 digits long.

Once entered, the PIN code should be given to the stack using the *bt_conn_auth_pincode_entry()* API.

This callback may be set to NULL, however in that case pairing over BR/EDR will not be possible. If provided, the cancel callback must be provided as well.

Parameters
- `conn`: Connection where pairing is currently active.
- `highsec`: true if 16 digit PIN is required.

void (***`pairing_complete`**)(**`struct`** bt_conn *conn, bool bonded)

notify that pairing procedure was complete.

This callback notifies the application that the pairing procedure has been completed.

Parameters
- `conn`: Connection object.
- `bonded`: Bond information has been distributed during the pairing procedure.

void (***`pairing_failed`**)(**`struct`** bt_conn *conn, **`enum`** *bt_security_err* reason)

notify that pairing process has failed.

Parameters
- `conn`: Connection object.
- `reason`: Pairing failed reason

void (***bond_deleted**) (uint8_t id, **const** *bt_addr_le_t* *peer)
Notify that bond has been deleted.

This callback notifies the application that the bond information for the remote peer has been deleted

Parameters
- `id`: Which local identity had the bond.
- `peer`: Remote address.

struct bt_br_conn_param
#include <conn.h> Connection parameters for BR/EDR connections

1.2 Data Buffers

1.2.1 API Reference

group **bt_buf**
Data buffers.

Defines

BT_BUF_RESERVE

BT_BUF_SIZE (*size*)

BT_BUF_RX_SIZE
Data size neeed for HCI RX buffers

Enums

enum bt_buf_type
Possible types of buffers passed around the Bluetooth stack

Values:

enumerator BT_BUF_CMD
HCI command

enumerator BT_BUF_EVT
HCI event

enumerator BT_BUF_ACL_OUT
Outgoing ACL data

enumerator BT_BUF_ACL_IN
Incoming ACL data

enumerator BT_BUF_ISO_OUT
Outgoing ISO data

enumerator BT_BUF_ISO_IN
Incoming ISO data

enumerator BT_BUF_H4
H:4 data

Functions

struct net_buf ***bt_buf_get_rx**(**enum** *bt_buf_type type*, k_timeout_t *timeout*)

Allocate a buffer for incoming data

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before bt_recv_prio().

Return A new buffer.

Parameters

- `type`: Type of buffer. Only BT_BUF_EVT and BT_BUF_ACL_IN are allowed.
- `timeout`: Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

struct net_buf ***bt_buf_get_tx**(**enum** *bt_buf_type type*, k_timeout_t *timeout*, **const** void **data*, size_t *size*)

Allocate a buffer for outgoing data

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before bt_send().

Return A new buffer.

Parameters

- `type`: Type of buffer. Only BT_BUF_CMD, BT_BUF_ACL_OUT or BT_BUF_H4, when operating on H:4 mode, are allowed.
- `timeout`: Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.
- `data`: Initial data to append to buffer.
- `size`: Initial data size.

struct net_buf ***bt_buf_get_cmd_complete**(k_timeout_t *timeout*)

Allocate a buffer for an HCI Command Complete/Status Event

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before bt_recv_prio().

Return A new buffer.

Parameters

- `timeout`: Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

struct net_buf ***bt_buf_get_evt**(uint8_t *evt*, bool *discardable*, k_timeout_t *timeout*)

Allocate a buffer for an HCI Event

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before bt_recv_prio() or bt_recv().

Return A new buffer.

Parameters

- `evt`: HCI event code

- `discardable`: Whether the driver considers the event discardable.
- `timeout`: Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

static inline void **bt_buf_set_type**(**struct** net_buf **buf*, **enum** *bt_buf_type* *type*)

Set the buffer type

Parameters

- `buf`: Bluetooth buffer
- `type`: The BT_* type to set the buffer to

static inline enum *bt_buf_type* **bt_buf_get_type**(**struct** net_buf **buf*)

Get the buffer type

Return The BT_* type to of the buffer

Parameters

- `buf`: Bluetooth buffer

struct bt_buf_data

#include <buf.h> This is a base type for bt_buf user data.

1.3 Generic Access Profile (GAP)

1.3.1 API Reference

group **bt_gap**

Generic Access Profile.

Defines

BT_ID_DEFAULT

Convenience macro for specifying the default identity. This helps make the code more readable, especially when only one identity is supported.

BT_DATA(*_type*, *_data*, *_data_len*)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- `_type`: Type of advertising data field
- `_data`: Pointer to the data field payload
- `_data_len`: Number of bytes behind the _data pointer

BT_DATA_BYTES(*_type*, *_bytes...*)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- `_type`: Type of advertising data field
- `_bytes`: Variable number of single-byte parameters

BT_LE_ADV_PARAM_INIT(*_options*, *_int_min*, *_int_max*, *_peer*)

Initialize advertising parameters.

Parameters

- `_options`: Advertising Options
- `_int_min`: Minimum advertising interval
- `_int_max`: Maximum advertising interval
- `_peer`: Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

BT_LE_ADV_PARAM(*_options*, *_int_min*, *_int_max*, *_peer*)

Helper to declare advertising parameters inline.

Parameters

- `_options`: Advertising Options
- `_int_min`: Minimum advertising interval
- `_int_max`: Maximum advertising interval
- `_peer`: Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

BT_LE_ADV_CONN_DIR(*_peer*)

BT_LE_ADV_CONN

BT_LE_ADV_CONN_NAME

BT_LE_ADV_CONN_DIR_LOW_DUTY(*_peer*)

BT_LE_ADV_NCONN

Non-connectable advertising with private address

BT_LE_ADV_NCONN_NAME

Non-connectable advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_ADV_NCONN_IDENTITY

Non-connectable advertising with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_NCONN

Non-connectable extended advertising with private address

BT_LE_EXT_ADV_NCONN_NAME

Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_NCONN_IDENTITY
Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_CODED_NCONN
Non-connectable extended advertising on coded PHY with private address

BT_LE_EXT_ADV_CODED_NCONN_NAME
Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_CODED_NCONN_IDENTITY
Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_START_PARAM_INIT(*_timeout*, *_n_evts*)
Helper to initialize extended advertising start parameters inline

Parameters

- `_timeout`: Advertiser timeout
- `_n_evts`: Number of advertising events

BT_LE_EXT_ADV_START_PARAM(*_timeout*, *_n_evts*)
Helper to declare extended advertising start parameters inline

Parameters

- `_timeout`: Advertiser timeout
- `_n_evts`: Number of advertising events

BT_LE_EXT_ADV_START_DEFAULT

BT_LE_PER_ADV_PARAM_INIT(*_int_min*, *_int_max*, *_options*)
Helper to declare periodic advertising parameters inline

Parameters

- `_int_min`: Minimum periodic advertising interval
- `_int_max`: Maximum periodic advertising interval
- `_options`: Periodic advertising properties bitfield.

BT_LE_PER_ADV_PARAM(*_int_min*, *_int_max*, *_options*)
Helper to declare periodic advertising parameters inline

Parameters

- `_int_min`: Minimum periodic advertising interval
- `_int_max`: Maximum periodic advertising interval
- `_options`: Periodic advertising properties bitfield.

BT_LE_PER_ADV_DEFAULT

BT_LE_SCAN_PARAM_INIT(*_type*, *_options*, *_interval*, *_window*)
Initialize scan parameters.

Parameters

- `_type`: Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- `_options`: Scan options
- `_interval`: Scan Interval (N * 0.625 ms)
- `_window`: Scan Window (N * 0.625 ms)

BT_LE_SCAN_PARAM(*_type*, *_options*, *_interval*, *_window*)

Helper to declare scan parameters inline.

Parameters

- `_type`: Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- `_options`: Scan options
- `_interval`: Scan Interval (N * 0.625 ms)
- `_window`: Scan Window (N * 0.625 ms)

BT_LE_SCAN_ACTIVE

Helper macro to enable active scanning to discover new devices.

BT_LE_SCAN_PASSIVE

Helper macro to enable passive scanning to discover new devices.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

BT_LE_SCAN_CODED_ACTIVE

Helper macro to enable active scanning to discover new devices. Include scanning on Coded PHY in addition to 1M PHY.

BT_LE_SCAN_CODED_PASSIVE

Helper macro to enable passive scanning to discover new devices. Include scanning on Coded PHY in addition to 1M PHY.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

Typedefs

typedef void (***bt_ready_cb_t**)(int err)

Callback for notifying that Bluetooth has been enabled.

Parameters

- `err`: zero on success or (negative) error code otherwise.

typedef void **bt_le_scan_cb_t**(**const** *bt_addr_le_t* **addr*, int8_t *rssi*, uint8_t *adv_type*, **struct** net_buf_simple **buf*)

Callback type for reporting LE scan results.

A function of this type is given to the *bt_le_scan_start()* function and will be called for any discovered LE device.

Parameters

- `addr`: Advertiser LE address and type.
- `rssi`: Strength of advertiser signal.
- `adv_type`: Type of advertising response from advertiser.
- `buf`: Buffer containing advertiser data.

typedef void **bt_br_discovery_cb_t**(**struct** *bt_br_discovery_result* **results*, size_t *count*)

Callback type for reporting BR/EDR discovery (inquiry) results.

A callback of this type is given to the *bt_br_discovery_start()* function and will be called at the end of the discovery with information about found devices populated in the results array.

Parameters

- `results`: Storage used for discovery results
- `count`: Number of valid discovery results.

Enums

enum [anonymous]

Advertising options

Values:

enumerator BT_LE_ADV_OPT_NONE

Convenience value when no options are specified.

enumerator BT_LE_ADV_OPT_CONNECTABLE

Advertise as connectable.

Advertise as connectable. If not connectable then the type of advertising is determined by providing scan response data. The advertiser address is determined by the type of advertising and/or enabling privacy `CONFIG_BT_PRIVACY` .

enumerator BT_LE_ADV_OPT_ONE_TIME

Advertise one time.

Don't try to resume connectable advertising after a connection. This option is only meaningful when used together with BT_LE_ADV_OPT_CONNECTABLE. If set the advertising will be stopped when *bt_le_adv_stop()* is called or when an incoming (slave) connection happens. If this option is not set the stack will take care of keeping advertising enabled even as connections occur. If Advertising directed or the advertiser was started with *bt_le_ext_adv_start* then this behavior is the default behavior and this flag has no effect.

enumerator BT_LE_ADV_OPT_USE_IDENTITY

Advertise using identity address.

Advertise using the identity address as the advertiser address.

Warning This will compromise the privacy of the device, so care must be taken when using this option.

Note The address used for advertising will not be the same as returned by *bt_le_oob_get_local*, instead *bt_id_get* should be used to get the LE address.

enumerator BT_LE_ADV_OPT_USE_NAME

Advertise using GAP device name

enumerator BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY
Low duty cycle directed advertising.

Use low duty directed advertising mode, otherwise high duty mode will be used.

enumerator BT_LE_ADV_OPT_DIR_ADDR_RPA
Directed advertising to privacy-enabled peer.

Enable use of Resolvable Private Address (RPA) as the target address in directed advertisements. This is required if the remote device is privacy-enabled and supports address resolution of the target address in directed advertisement. It is the responsibility of the application to check that the remote device supports address resolution of directed advertisements by reading its Central Address Resolution characteristic.

enumerator BT_LE_ADV_OPT_FILTER_SCAN_REQ
Use whitelist to filter devices that can request scan response data.

enumerator BT_LE_ADV_OPT_FILTER_CONN
Use whitelist to filter devices that can connect.

enumerator BT_LE_ADV_OPT_NOTIFY_SCAN_REQ
Notify the application when a scan response data has been sent to an active scanner.

enumerator BT_LE_ADV_OPT_SCANNABLE
Support scan response data.

When used together with *BT_LE_ADV_OPT_EXT_ADV* then this option cannot be used together with the *BT_LE_ADV_OPT_CONNECTABLE* option. When used together with *BT_LE_ADV_OPT_EXT_ADV* then scan response data must be set.

enumerator BT_LE_ADV_OPT_EXT_ADV
Advertise with extended advertising.

This options enables extended advertising in the advertising set. In extended advertising the advertising set will send a small header packet on the three primary advertising channels. This small header points to the advertising data packet that will be sent on one of the 37 secondary advertising channels. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 2M PHY. Connections will be established on LE 2M PHY.

Without this option the advertiser will send advertising data on the three primary advertising channels.

Note Enabling this option requires extended advertising support in the peer devices scanning for advertisement packets.

enumerator BT_LE_ADV_OPT_NO_2M
Disable use of LE 2M PHY on the secondary advertising channel.

Disabling the use of LE 2M PHY could be necessary if scanners don't support the LE 2M PHY. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 1M PHY. Connections will be established on LE 1M PHY.

Note Cannot be set if BT_LE_ADV_OPT_CODED is set.

Requires *BT_LE_ADV_OPT_EXT_ADV*.

enumerator BT_LE_ADV_OPT_CODED
Advertise on the LE Coded PHY (Long Range).

The advertiser will send both primary and secondary advertising on the LE Coded PHY. This gives the advertiser increased range with the trade-off of lower data rate and higher power consumption. Connections will be established on LE Coded PHY.

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

`enumerator BT_LE_ADV_OPT_ANONYMOUS`
Advertise without a device address (identity or RPA).

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

`enumerator BT_LE_ADV_OPT_USE_TX_POWER`
Advertise with transmit power.

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

`enumerator BT_LE_ADV_OPT_DISABLE_CHAN_37`
Disable advertising on channel index 37.

`enumerator BT_LE_ADV_OPT_DISABLE_CHAN_38`
Disable advertising on channel index 38.

`enumerator BT_LE_ADV_OPT_DISABLE_CHAN_39`
Disable advertising on channel index 39.

`enum` [anonymous]
Periodic Advertising options

Values:

`enumerator BT_LE_PER_ADV_OPT_NONE`
Convenience value when no options are specified.

`enumerator BT_LE_PER_ADV_OPT_USE_TX_POWER`
Advertise with transmit power.

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

`enum` [anonymous]
Periodic advertising sync options

Values:

`enumerator BT_LE_PER_ADV_SYNC_OPT_NONE`
Convenience value when no options are specified.

`enumerator BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST`
Use the periodic advertising list to sync with advertiser.

When this option is set, the address and SID of the parameters are ignored.

`enumerator BT_LE_PER_ADV_SYNC_OPT_REPORTING_INITIALLY_DISABLED`
Disables periodic advertising reports.

No advertisement reports will be handled until enabled.

`enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOA`
Sync with Angle of Arrival (AoA) constant tone extension

`enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_1US`
Sync with Angle of Departure (AoD) 1 us constant tone extension

`enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_2US`
Sync with Angle of Departure (AoD) 2 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_OPT_SYNC_ONLY_CONST_TONE_EXT
Do not sync to packets without a constant tone extension

enum [anonymous]
Periodic Advertising Sync Transfer options

Values:

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_NONE
Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOA
No Angle of Arrival (AoA)

Do not sync with Angle of Arrival (AoA) constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_1US
No Angle of Departure (AoD) 1 us.

Do not sync with Angle of Departure (AoD) 1 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_2US
No Angle of Departure (AoD) 2.

Do not sync with Angle of Departure (AoD) 2 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_ONLY_CTE
Only sync to packets with constant tone extension

enum [anonymous]
Values:

enumerator BT_LE_SCAN_OPT_NONE
Convenience value when no options are specified.

enumerator BT_LE_SCAN_OPT_FILTER_DUPLICATE
Filter duplicates.

enumerator BT_LE_SCAN_OPT_FILTER_WHITELIST
Filter using whitelist.

enumerator BT_LE_SCAN_OPT_CODED
Enable scan on coded PHY (Long Range).

enumerator BT_LE_SCAN_OPT_NO_1M
Disable scan on 1M phy.

Note Requires *BT_LE_SCAN_OPT_CODED*.

enum [anonymous]
Values:

enumerator BT_LE_SCAN_TYPE_PASSIVE
Scan without requesting additional information from advertisers.

enumerator BT_LE_SCAN_TYPE_ACTIVE
Scan and request additional information from advertisers.

Functions

int **bt_enable**(*bt_ready_cb_t cb*)

Enable Bluetooth.

Enable Bluetooth. Must be the called before any calls that require communication with the local Bluetooth hardware.

Return Zero on success or (negative) error code otherwise.

Parameters

- `cb`: Callback to notify completion or NULL to perform the enabling synchronously.

int **bt_set_name**(**const** char **name*)

Set Bluetooth Device Name.

Set Bluetooth GAP Device Name.

Return Zero on success or (negative) error code otherwise.

Parameters

- `name`: New name

const char ***bt_get_name**(void)

Get Bluetooth Device Name.

Get Bluetooth GAP Device Name.

Return Bluetooth Device Name

int **bt_set_id_addr**(**const** *bt_addr_le_t *addr*)

Set the local Identity Address.

Allows setting the local Identity Address from the application. This API must be called before calling *bt_enable()*. Calling it at any other time will cause it to fail. In most cases the application doesn't need to use this API, however there are a few valid cases where it can be useful (such as for testing).

At the moment, the given address must be a static random address. In the future support for public addresses may be added.

Deprecated:

in 2.5 release, replace with bt_id_create before bt_enable.

Return Zero on success or (negative) error code otherwise.

void **bt_id_get**(*bt_addr_le_t *addrs*, size_t **count*)

Get the currently configured identities.

Returns an array of the currently configured identity addresses. To make sure all available identities can be retrieved, the number of elements in the *addrs* array should be CONFIG_BT_ID_MAX. The identity identifier that some APIs expect (such as advertising parameters) is simply the index of the identity in the *addrs* array.

Note Deleted identities may show up as BT_LE_ADDR_ANY in the returned array.

Parameters

- `addrs`: Array where to store the configured identities.
- `count`: Should be initialized to the array size. Once the function returns it will contain the number of returned identities.

int **`bt_id_create`** (*bt_addr_le_t *addr*, uint8_t **irk*)

Create a new identity.

Create a new identity using the given address and IRK. This function can be called before calling *bt_enable()*, in which case it can be used to override the controller's public address (in case it has one). However, the new identity will only be stored persistently in flash when this API is used after *bt_enable()*. The reason is that the persistent settings are loaded after *bt_enable()* and would therefore cause potential conflicts with the stack blindly overwriting what's stored in flash. The identity will also not be written to flash in case a pre-defined address is provided, since in such a situation the app clearly has some place it got the address from and will be able to repeat the procedure on every power cycle, i.e. it would be redundant to also store the information in flash.

Generating random static address or random IRK is not supported when calling this function before *bt_enable()*.

If the application wants to have the stack randomly generate identities and store them in flash for later recovery, the way to do it would be to first initialize the stack (using bt_enable), then call settings_load(), and after that check with *bt_id_get()* how many identities were recovered. If an insufficient amount of identities were recovered the app may then call *bt_id_create()* to create new ones.

Return Identity identifier (>= 0) in case of success, or a negative error code on failure.

Parameters

- `addr`: Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new random static address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).
- `irk`: Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy `CONFIG_BT_PRIVACY` is not enabled this parameter must be NULL.

int **`bt_id_reset`** (uint8_t *id*, *bt_addr_le_t *addr*, uint8_t **irk*)

Reset/reclaim an identity for reuse.

The semantics of the *addr* and *irk* parameters of this function are the same as with *bt_id_create()*. The difference is the first *id* parameter that needs to be an existing identity (if it doesn't exist this function will return an error). When given an existing identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then create a new identity in the same slot, based on the *addr* and *irk* parameters.

Note the default identity (BT_ID_DEFAULT) cannot be reset, i.e. this API will return an error if asked to do that.

Return Identity identifier (>= 0) in case of success, or a negative error code on failure.

Parameters

- `id`: Existing identity identifier.

- addr: Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new static random address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).
- irk: Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy CONFIG_BT_PRIVACY is not enabled this parameter must be NULL.

int **bt_id_delete** (uint8_t *id*)

Delete an identity.

When given a valid identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then flag is as deleted, so that it can not be used for any operations. To take back into use the slot the identity was occupying the *bt_id_reset()* API needs to be used.

Note the default identity (BT_ID_DEFAULT) cannot be deleted, i.e. this API will return an error if asked to do that.

Return 0 in case of success, or a negative error code on failure.

Parameters

- id: Existing identity identifier.

int **bt_le_adv_start** (**const struct** *bt_le_adv_param* **param*, **const struct** *bt_data* **ad*, size_t *ad_len*, **const struct** *bt_data* **sd*, size_t *sd_len*)

Start advertising.

Set advertisement data, scan response data, advertisement parameters and start advertising.

When the advertisement parameter peer address has been set the advertising will be directed to the peer. In this case advertisement data and scan response data parameters are ignored. If the mode is high duty cycle the timeout will be *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

Return Zero on success or (negative) error code otherwise.

-ENOMEM No free connection objects available for connectable advertiser.

-ECONNREFUSED When connectable advertising is requested and there is already maximum number of connections established in the controller. This error code is only guaranteed when using Zephyr controller, for other controllers code returned in this case may be -EIO.

Parameters

- param: Advertising parameters.
- ad: Data to be used in advertisement packets.
- ad_len: Number of elements in ad
- sd: Data to be used in scan response packets.
- sd_len: Number of elements in sd

int **bt_le_adv_update_data** (**const struct** *bt_data* **ad*, size_t *ad_len*, **const struct** *bt_data* **sd*, size_t *sd_len*)

Update advertising.

Update advertisement and scan response data.

Return Zero on success or (negative) error code otherwise.

Parameters

- `ad`: Data to be used in advertisement packets.
- `ad_len`: Number of elements in ad
- `sd`: Data to be used in scan response packets.
- `sd_len`: Number of elements in sd

int **`bt_le_adv_stop`**(void)

Stop advertising.

Stops ongoing advertising.

Return Zero on success or (negative) error code otherwise.

int **`bt_le_ext_adv_create`**(**`const struct`** *bt_le_adv_param* **param*, **`const struct`** *bt_le_ext_adv_cb* **cb*, **`struct`** bt_le_ext_adv ***adv*)

Create advertising set.

Create a new advertising set and set advertising parameters. Advertising parameters can be updated with *bt_le_ext_adv_update_param*.

Return Zero on success or (negative) error code otherwise.

Parameters

- `[in] param`: Advertising parameters.
- `[in] cb`: Callback struct to notify about advertiser activity. Can be NULL. Must point to valid memory during the lifetime of the advertising set.
- `[out] adv`: Valid advertising set object on success.

int **`bt_le_ext_adv_start`**(**`struct`** bt_le_ext_adv **adv*, **`struct`** *bt_le_ext_adv_start_param* **param*)

Start advertising with the given advertising set.

If the advertiser is limited by either the timeout or number of advertising events the application will be notified by the advertiser sent callback once the limit is reached. If the advertiser is limited by both the timeout and the number of advertising events then the limit that is reached first will stop the advertiser.

Parameters

- `adv`: Advertising set object.
- `param`: Advertise start parameters.

int **`bt_le_ext_adv_stop`**(**`struct`** bt_le_ext_adv **adv*)

Stop advertising with the given advertising set.

Stop advertising with a specific advertising set. When using this function the advertising sent callback will not be called.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.

int **`bt_le_ext_adv_set_data`**(**`struct`** bt_le_ext_adv **adv*, **`const struct`** *bt_data* **ad*, size_t *ad_len*, **`const struct`** *bt_data* **sd*, size_t *sd_len*)

Set an advertising set's advertising or scan response data.

Set advertisement data or scan response data. If the advertising set is currently advertising then the advertising data will be updated in subsequent advertising events.

When both *BT_LE_ADV_OPT_EXT_ADV* and *BT_LE_ADV_OPT_SCANNABLE* are enabled then advertising data is ignored. When *BT_LE_ADV_OPT_SCANNABLE* is not enabled then scan response data is ignored.

If the advertising set has been configured to send advertising data on the primary advertising channels then the maximum data length is *BT_GAP_ADV_MAX_ADV_DATA_LEN* bytes. If the advertising set has been configured for extended advertising, then the maximum data length is defined by the controller with the maximum possible of *BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN* bytes.

Note Not all scanners support extended data length advertising data.

When updating the advertising data while advertising the advertising data and scan response data length must be smaller or equal to what can be fit in a single advertising packet. Otherwise the advertiser must be stopped.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.
- `ad`: Data to be used in advertisement packets.
- `ad_len`: Number of elements in ad
- `sd`: Data to be used in scan response packets.
- `sd_len`: Number of elements in sd

int **`bt_le_ext_adv_update_param`**(**`struct`** bt_le_ext_adv **adv*, **`const struct`** *bt_le_adv_param* **param*)

Update advertising parameters.

Update the advertising parameters. The function will return an error if the advertiser set is currently advertising. Stop the advertising set before calling this function.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.
- `param`: Advertising parameters.

int **`bt_le_ext_adv_delete`**(**`struct`** bt_le_ext_adv **adv*)

Delete advertising set.

Delete advertising set. This will free up the advertising set and make it possible to create a new advertising set.

Return Zero on success or (negative) error code otherwise.

uint8_t **`bt_le_ext_adv_get_index`**(**`struct`** bt_le_ext_adv **adv*)

Get array index of an advertising set.

This function is used to map bt_adv to index of an array of advertising sets. The array has CONFIG_BT_EXT_ADV_MAX_ADV_SET elements.

Return Index of the advertising set object. The range of the returned value is 0..CONFIG_BT_EXT_ADV_MAX_ADV_SET-1

Parameters

- `adv`: Advertising set.

int **`bt_le_ext_adv_get_info`**(**`const struct`** bt_le_ext_adv **adv*, **`struct`** *bt_le_ext_adv_info* **info*)

Get advertising set info.

Return Zero on success or (negative) error code on failure.

Parameters

- `adv`: Advertising set object
- `info`: Advertising set info object

int **`bt_le_per_adv_set_param`**(**`struct`** bt_le_ext_adv **adv*, **`const struct`** *bt_le_per_adv_param* **param*)

Set or update the periodic advertising parameters.

The periodic advertising parameters can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.
- `param`: Advertising parameters.

int **`bt_le_per_adv_set_data`**(**`const struct`** bt_le_ext_adv **adv*, **`const struct`** *bt_data* **ad*, size_t *ad_len*)

Set or update the periodic advertising data.

The periodic advertisement data can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.
- `ad`: Advertising data.
- `ad_len`: Advertising data length.

int **bt_le_per_adv_start**(**struct** bt_le_ext_adv **adv*)

Starts periodic advertising.

Enabling the periodic advertising can be done independently of extended advertising, but both periodic advertising and extended advertising shall be enabled before any periodic advertising data is sent. The periodic advertising and extended advertising can be enabled in any order.

Once periodic advertising has been enabled, it will continue advertising until *bt_le_per_adv_stop()* has been called, or if the advertising set is deleted by *bt_le_ext_adv_delete()*. Calling *bt_le_ext_adv_stop()* will not stop the periodic advertising.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.

int **bt_le_per_adv_stop**(**struct** bt_le_ext_adv **adv*)

Stops periodic advertising.

Disabling the periodic advertising can be done independently of extended advertising. Disabling periodic advertising will not disable extended advertising.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: Advertising set object.

uint8_t **bt_le_per_adv_sync_get_index**(**struct** bt_le_per_adv_sync **per_adv_sync*)

Get array index of an periodic advertising sync object.

This function is get the index of an array of periodic advertising sync objects. The array has CONFIG_BT_PER_ADV_SYNC_MAX elements.

Return Index of the periodic advertising sync object. The range of the returned value is 0..CONFIG_BT_PER_ADV_SYNC_MAX-1

Parameters

- `per_adv_sync`: The periodic advertising sync object.

int **bt_le_per_adv_sync_create**(**const struct** *bt_le_per_adv_sync_param* **param*, **struct** bt_le_per_adv_sync ***out_sync*)

Create a periodic advertising sync object.

Create a periodic advertising sync object that can try to synchronize to periodic advertising reports from an advertiser. Scan shall either be disabled or extended scan shall be enabled.

Return Zero on success or (negative) error code otherwise.

Parameters

- `[in] param`: Periodic advertising sync parameters.
- `[out] out_sync`: Periodic advertising sync object on.

int **bt_le_per_adv_sync_delete**(**struct** bt_le_per_adv_sync **per_adv_sync*)

Delete periodic advertising sync.

Delete the periodic advertising sync object. Can be called regardless of the state of the sync. If the syncing is currently syncing, the syncing is cancelled. If the sync has been established, it is terminated. The periodic advertising sync object will be invalidated afterwards.

If the state of the sync object is syncing, then a new periodic advertising sync object may not be created until the controller has finished canceling this object.

Return Zero on success or (negative) error code otherwise.

Parameters

- `per_adv_sync`: The periodic advertising sync object.

void **bt_le_per_adv_sync_cb_register**(**struct** *bt_le_per_adv_sync_cb* **cb*)

Register periodic advertising sync callbacks.

Adds the callback structure to the list of callback structures for periodic adverising syncs.

This callback will be called for all periodic advertising sync activity, such as synced, terminated and when data is received.

Parameters

- `cb`: Callback struct. Must point to memory that remains valid.

int **bt_le_per_adv_sync_recv_enable**(**struct** bt_le_per_adv_sync **per_adv_sync*)

Enables receiving periodic advertising reports for a sync.

If the sync is already receiving the reports, -EALREADY is returned.

Return Zero on success or (negative) error code otherwise.

Parameters

- `per_adv_sync`: The periodic advertising sync object.

int **bt_le_per_adv_sync_recv_disable**(**struct** bt_le_per_adv_sync **per_adv_sync*)

Disables receiving periodic advertising reports for a sync.

If the sync report receiving is already disabled, -EALREADY is returned.

Return Zero on success or (negative) error code otherwise.

Parameters

- `per_adv_sync`: The periodic advertising sync object.

int **bt_le_per_adv_sync_transfer**(**const struct** bt_le_per_adv_sync **per_adv_sync*, **const struct** bt_conn **conn*, uint16_t *service_data*)

Transfer the periodic advertising sync information to a peer device.

This will allow another device to quickly synchronize to the same periodic advertising train that this device is currently synced to.

Return Zero on success or (negative) error code otherwise.

Parameters

- `per_adv_sync`: The periodic advertising sync to transfer.
- `conn`: The peer device that will receive the sync information.
- `service_data`: Application service data provided to the remote host.

int **`bt_le_per_adv_set_info_transfer`**(**`const struct`** bt_le_ext_adv **adv*, **`const struct`** bt_conn **conn*, uint16_t *service_data*)

Transfer the information about a periodic advertising set.

This will allow another device to quickly synchronize to periodic advertising set from this device.

Return Zero on success or (negative) error code otherwise.

Parameters

- `adv`: The periodic advertising set to transfer info of.
- `conn`: The peer device that will receive the information.
- `service_data`: Application service data provided to the remote host.

int **`bt_le_per_adv_sync_transfer_subscribe`**(**`const struct`** bt_conn **conn*, **`const struct`** *bt_le_per_adv_sync_transfer_param* **param*)

Subscribe to periodic advertising sync transfers (PASTs).

Sets the parameters and allow other devices to transfer periodic advertising syncs.

Return Zero on success or (negative) error code otherwise.

Parameters

- `conn`: The connection to set the parameters for. If NULL default parameters for all connections will be set. Parameters set for specific connection will always have precedence.
- `param`: The periodic advertising sync transfer parameters.

int **`bt_le_per_adv_sync_transfer_unsubscribe`**(**`const struct`** bt_conn **conn*)

Unsubscribe from periodic advertising sync transfers (PASTs).

Remove the parameters that allow other devices to transfer periodic advertising syncs.

Return Zero on success or (negative) error code otherwise.

Parameters

- `conn`: The connection to remove the parameters for. If NULL default parameters for all connections will be removed. Unsubscribing for a specific device, will still allow other devices to transfer periodic advertising syncs.

int **`bt_le_per_adv_list_add`**(**`const`** *bt_addr_le_t* **addr*, uint8_t *sid*)

Add a device to the periodic advertising list.

Add peer device LE address to the periodic advertising list. This will make it possibly to automatically create a periodic advertising sync to this device.

Return Zero on success or (negative) error code otherwise.

Parameters

- `addr`: Bluetooth LE identity address.
- `sid`: The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

int **`bt_le_per_adv_list_remove`**(**`const`** *bt_addr_le_t* **addr*, uint8_t *sid*)

Remove a device from the periodic advertising list.

Removes peer device LE address from the periodic advertising list.

Return Zero on success or (negative) error code otherwise.

Parameters

- `addr`: Bluetooth LE identity address.
- `sid`: The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

int **`bt_le_per_adv_list_clear`**(void)

Clear the periodic advertising list.

Clears the entire periodic advertising list.

Return Zero on success or (negative) error code otherwise.

int **`bt_le_scan_start`**(**`const struct`** *bt_le_scan_param* **param*, *bt_le_scan_cb_t* *cb*)

Start (LE) scanning.

Start LE scanning with given parameters and provide results through the specified callback.

Note The LE scanner by default does not use the Identity Address of the local device when `CONFIG_BT_PRIVACY` is disabled. This is to prevent the active scanner from disclosing the identity information when requesting additional information from advertisers. In order to enable directed advertiser reports then `CONFIG_BT_SCAN_WITH_IDENTITY` must be enabled.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `param`: Scan parameters.
- `cb`: Callback to notify scan results. May be NULL if callback registration through *bt_le_scan_cb_register* is preferred.

int **`bt_le_scan_stop`**(void)

Stop (LE) scanning.

Stops ongoing LE scanning.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

void **bt_le_scan_cb_register**(**struct** *bt_le_scan_cb* **cb*)

Register scanner packet callbacks.

Adds the callback structure to the list of callback structures that monitors scanner activity.

This callback will be called for all scanner activity, regardless of what API was used to start the scanner.

Parameters

- `cb`: Callback struct. Must point to memory that remains valid.

void **bt_le_scan_cb_unregister**(**struct** *bt_le_scan_cb* **cb*)

Unregister scanner packet callbacks.

Remove the callback structure from the list of scanner callbacks.

Parameters

- `cb`: Callback struct. Must point to memory that remains valid.

int **bt_le_whitelist_add**(**const** *bt_addr_le_t* **addr*)

Add device (LE) to whitelist.

Add peer device LE address to the whitelist.

Note The whitelist cannot be modified when an LE role is using the whitelist, i.e advertiser or scanner using a whitelist or automatic connecting to devices using whitelist.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `addr`: Bluetooth LE identity address.

int **bt_le_whitelist_rem**(**const** *bt_addr_le_t* **addr*)

Remove device (LE) from whitelist.

Remove peer device LE address from the whitelist.

Note The whitelist cannot be modified when an LE role is using the whitelist, i.e advertiser or scanner using a whitelist or automatic connecting to devices using whitelist.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `addr`: Bluetooth LE identity address.

int **bt_le_whitelist_clear**(void)

Clear whitelist.

Clear all devices from the whitelist.

Note The whitelist cannot be modified when an LE role is using the whitelist, i.e advertiser or scanner using a whitelist or automatic connecting to devices using whitelist.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **`bt_le_set_chan_map`** (uint8_t *chan_map*[5])

Set (LE) channel map.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `chan_map`: Channel map.

void **`bt_data_parse`** (**`struct`** net_buf_simple **ad*, bool (**func*)) **`struct`** *bt_data* *data, void *user_data

, void **user_data*Helper for parsing advertising (or EIR or OOB) data.

A helper for parsing the basic data types used for Extended Inquiry Response (EIR), Advertising Data (AD), and OOB data blocks. The most common scenario is to call this helper on the advertising data received in the callback that was given to *bt_le_scan_start()*.

Parameters

- `ad`: Advertising data as given to the bt_le_scan_cb_t callback.
- `func`: Callback function which will be called for each element that's found in the data. The callback should return true to continue parsing, or false to stop parsing.
- `user_data`: User data to be passed to the callback.

int **`bt_le_oob_get_local`** (uint8_t *id*, **`struct`** *bt_le_oob* **oob*)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy `CONFIG_BT_PRIVACY` is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for `CONFIG_BT_RPA_TIMEOUT` seconds. This address will be used for advertising started by *bt_le_adv_start*, active scanning and connection creation.

Note If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback. In addition when extended advertising `CONFIG_BT_EXT_ADV` is not enabled or not supported by the controller:
- Advertiser is enabled using a Random Static Identity Address for a different local identity.
- The local identity conflicts with the local identity used by other roles.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `[in] id`: Local identity, in most cases BT_ID_DEFAULT.
- `[out] oob`: LE OOB information

int **`bt_le_ext_adv_oob_get_local`** (**`struct`** bt_le_ext_adv **adv*, **`struct`** *bt_le_oob* **oob*)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy `CONFIG_BT_PRIVACY` is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for `CONFIG_BT_RPA_TIMEOUT` seconds. This address will be used by the advertising set.

Note When generating OOB information for multiple advertising set all OOB information needs to be generated at the same time.

If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

Parameters

- `[in] adv`: The advertising set object
- `[out] oob`: LE OOB information

int **`bt_br_discovery_start`**(**`const struct`** *bt_br_discovery_param *param*, **`struct`** *bt_br_discovery_result *results*, size_t *count*, *bt_br_discovery_cb_t cb*)

Start BR/EDR discovery.

Start BR/EDR discovery (inquiry) and provide results through the specified callback. When bt_br_discovery_cb_t is called it indicates that discovery has completed. If more inquiry results were received during session than fits in provided result storage, only ones with highest RSSI will be reported.

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error

Parameters

- `param`: Discovery parameters.
- `results`: Storage for discovery results.
- `count`: Number of results in storage. Valid range: 1-255.
- `cb`: Callback to notify discovery results.

int **`bt_br_discovery_stop`**(void)

Stop BR/EDR discovery.

Stops ongoing BR/EDR discovery. If discovery was stopped by this call results won't be reported

Return Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **`bt_br_oob_get_local`**(**`struct`** *bt_br_oob *oob*)

Get BR/EDR local Out Of Band information.

This function allows to get local controller information that are useful for Out Of Band pairing or connection creation process.

Parameters

- `oob`: Out Of Band information

int **`bt_br_set_discoverable`**(bool *enable*)

Enable/disable set controller in discoverable state.

Allows make local controller to listen on INQUIRY SCAN channel and responds to devices making general inquiry. To enable this state it's mandatory to first be in connectable state.

Return Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

Parameters

- `enable`: Value allowing/disallowing controller to become discoverable.

int **`bt_br_set_connectable`**(bool *enable*)

Enable/disable set controller in connectable state.

Allows make local controller to be connectable. It means the controller start listen to devices requests on PAGE SCAN channel. If disabled also resets discoverability if was set.

Return Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

Parameters

- `enable`: Value allowing/disallowing controller to be connectable.

int **`bt_unpair`**(uint8_t *id*, **`const`** *bt_addr_le_t* **addr*)

Clear pairing information.

Return 0 on success or negative error value on failure.

Parameters

- `id`: Local identity (mostly just BT_ID_DEFAULT).
- `addr`: Remote address, NULL or BT_ADDR_LE_ANY to clear all remote devices.

void **`bt_foreach_bond`**(uint8_t *id*, void (**func*)) **`const struct`** *bt_bond_info* *info, void *user_data

, void **user_data*Iterate through all existing bonds.

Parameters

- `id`: Local identity (mostly just BT_ID_DEFAULT).
- `func`: Function to call for each bond.
- `user_data`: Data to pass to the callback function.

`struct bt_le_ext_adv_sent_info`

#include <bluetooth.h>

Public Members

uint8_t **num_sent**
The number of advertising events completed.

struct bt_le_ext_adv_connected_info
#include <bluetooth.h>

Public Members

struct bt_conn ***conn**
Connection object of the new connection

struct bt_le_ext_adv_scanned_info
#include <bluetooth.h>

Public Members

bt_addr_le_t ***addr**
Active scanner LE address and type

struct bt_le_ext_adv_cb
#include <bluetooth.h>

Public Members

void (***sent**)(**struct** bt_le_ext_adv *adv, **struct** *bt_le_ext_adv_sent_info* *info)
The advertising set has finished sending adv data.

This callback notifies the application that the advertising set has finished sending advertising data. The advertising set can either have been stopped by a timeout or because the specified number of advertising events has been reached.

Parameters
- `adv`: The advertising set object.
- `info`: Information about the sent event.

void (***connected**)(**struct** bt_le_ext_adv *adv, **struct** *bt_le_ext_adv_connected_info* *info)
The advertising set has accepted a new connection.

This callback notifies the application that the advertising set has accepted a new connection.

Parameters
- `adv`: The advertising set object.
- `info`: Information about the connected event.

void (***scanned**)(**struct** bt_le_ext_adv *adv, **struct** *bt_le_ext_adv_scanned_info* *info)
The advertising set has sent scan response data.

This callback notifies the application that the advertising set has has received a Scan Request packet, and has sent a Scan Response packet.

Parameters
- `adv`: The advertising set object.

- `addr`: Information about the scanned event.

`struct bt_data`

#include <bluetooth.h> Bluetooth data.

Description of different data types that can be encoded into advertising data. Used to form arrays that are passed to the *bt_le_adv_start()* function.

`struct bt_le_adv_param`

#include <bluetooth.h> LE Advertising Parameters.

Public Members

uint8_t **`id`**

Local identity.

Note When extended advertising `CONFIG_BT_EXT_ADV` is not enabled or not supported by the controller it is not possible to scan and advertise simultaneously using two different random addresses.

It is not possible to have multiple connectable advertising sets advertising simultaneously using different identities.

uint8_t **`sid`**

Advertising Set Identifier, valid range 0x00 - 0x0f.

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

uint8_t **`secondary_max_skip`**

Secondary channel maximum skip count.

Maximum advertising events the advertiser can skip before it must send advertising data on the secondary advertising channel.

Note Requires *BT_LE_ADV_OPT_EXT_ADV*

uint32_t **`options`**

Bit-field of advertising options

uint32_t **`interval_min`**

Minimum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval. The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

uint32_t **`interval_max`**

Maximum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval. The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

`const` *bt_addr_le_t* ***`peer`**

Directed advertising to peer.

When this parameter is set the advertiser will send directed advertising to the remote device.

The advertising type will either be high duty cycle, or low duty cycle if the BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY option is enabled. When using *BT_LE_ADV_OPT_EXT_ADV* then only low duty cycle is allowed.

In case of connectable high duty cycle if the connection could not be established within the timeout the connected() callback will be called with the status set to BT_HCI_ERR_ADV_TIMEOUT.

struct bt_le_per_adv_param
#include <bluetooth.h>

Public Members

uint16_t **interval_min**
Minimum Periodic Advertising Interval (N * 1.25 ms)

uint16_t **interval_max**
Maximum Periodic Advertising Interval (N * 1.25 ms)

uint32_t **options**
Bit-field of periodic advertising options

struct bt_le_ext_adv_start_param
#include <bluetooth.h>

Public Members

uint16_t **timeout**
Advertiser timeout (N * 10 ms).

Application will be notified by the advertiser sent callback. Set to zero for no timeout.

When using high duty cycle directed connectable advertising then this parameters must be set to a non-zero value less than or equal to the maximum of *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

If privacy `CONFIG_BT_PRIVACY` is enabled then the timeout must be less than `CONFIG_BT_RPA_TIMEOUT`.

uint8_t **num_events**
Number of advertising events.

Application will be notified by the advertiser sent callback. Set to zero for no limit.

struct bt_le_ext_adv_info
#include <bluetooth.h> Advertising set info structure.

Public Members

int8_t **tx_power**
Currently selected Transmit Power (dBM).

struct bt_le_per_adv_sync_synced_info
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**
Advertiser LE address and type.

uint8_t **sid**
Advertiser SID

uint16_t **interval**
Periodic advertising interval (N * 1.25 ms)

uint8_t **phy**
Advertiser PHY

bool **recv_enabled**
True if receiving periodic advertisements, false otherwise.

uint16_t **service_data**
Service Data provided by the peer when sync is transferred.

Will always be 0 when the sync is locally created.

struct bt_conn ***conn**
Peer that transferred the periodic advertising sync.

Will always be 0 when the sync is locally created.

struct bt_le_per_adv_sync_term_info
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**
Advertiser LE address and type.

uint8_t **sid**
Advertiser SID

struct bt_le_per_adv_sync_recv_info
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**
Advertiser LE address and type.

uint8_t **sid**
Advertiser SID

int8_t **tx_power**
The TX power of the advertisement.

int8_t **rssi**
The RSSI of the advertisement excluding any CTE.

uint8_t **cte_type**
The Constant Tone Extension (CTE) of the advertisement

struct bt_le_per_adv_sync_state_info
#include <bluetooth.h>

Public Members

bool **recv_enabled**
True if receiving periodic advertisements, false otherwise.

struct bt_le_per_adv_sync_cb
#include <bluetooth.h>

Public Members

void (***synced**)(**struct** bt_le_per_adv_sync *sync, **struct** *bt_le_per_adv_sync_synced_info* *info)
The periodic advertising has been successfully synced.

This callback notifies the application that the periodic advertising set has been successfully synced, and will now start to receive periodic advertising reports.

Parameters
- `sync`: The periodic advertising sync object.
- `info`: Information about the sync event.

void (***term**)(**struct** bt_le_per_adv_sync *sync, **const struct** *bt_le_per_adv_sync_term_info* *info)
The periodic advertising sync has been terminated.

This callback notifies the application that the periodic advertising sync has been terminated, either by local request, remote request or because due to missing data, e.g. by being out of range or sync.

Parameters
- `sync`: The periodic advertising sync object.

void (***recv**)(**struct** bt_le_per_adv_sync *sync, **const struct** *bt_le_per_adv_sync_recv_info* *info, **struct** net_buf_simple *buf)
Periodic advertising data received.

This callback notifies the application of an periodic advertising report.

Parameters
- `sync`: The advertising set object.
- `info`: Information about the periodic advertising event.
- `buf`: Buffer containing the periodic advertising data.

void (***state_changed**)(**struct** bt_le_per_adv_sync *sync, **const struct** *bt_le_per_adv_sync_state_info* *info)
The periodic advertising sync state has changed.

This callback notifies the application about changes to the sync state. Initialize sync and termination is handled by their individual callbacks, and won't be notified here.

Parameters
- `sync`: The periodic advertising sync object.
- `info`: Information about the state change.

struct bt_le_per_adv_sync_param
#include <bluetooth.h>

Public Members

bt_addr_le_t **addr**
Periodic Advertiser Address.

Only valid if not using the periodic advertising list

uint8_t **sid**
Advertiser SID.

Only valid if not using the periodic advertising list

uint32_t **options**
Bit-field of periodic advertising sync options.

uint16_t **skip**
Maximum event skip.

Maximum number of periodic advertising events that can be skipped after a successful receive

uint16_t **timeout**
Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

struct bt_le_per_adv_sync_transfer_param
#include <bluetooth.h>

Public Members

uint16_t **skip**
Maximum event skip.

The number of periodic advertising packets that can be skipped after a successful receive.

uint16_t **timeout**
Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

uint32_t **options**
Periodic Advertising Sync Transfer options

struct bt_le_scan_param
#include <bluetooth.h> LE scan parameters

Public Members

uint8_t **type**
Scan type (BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE)

uint32_t **options**
Bit-field of scanning options.

uint16_t **interval**
Scan interval (N * 0.625 ms)

uint16_t **window**
Scan window (N * 0.625 ms)

uint16_t **timeout**

Scan timeout (N * 10 ms)

Application will be notified by the scan timeout callback. Set zero to disable timeout.

uint16_t **interval_coded**

Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval.

uint16_t **window_coded**

Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

struct bt_le_scan_recv_info

#include <bluetooth.h> LE advertisement packet information

Public Members

const *bt_addr_le_t* ***addr**

Advertiser LE address and type.

If advertiser is anonymous then this address will be *BT_ADDR_LE_ANY*.

uint8_t **sid**

Advertising Set Identifier.

int8_t **rssi**

Strength of advertiser signal.

int8_t **tx_power**

Transmit power of the advertiser.

uint8_t **adv_type**

Advertising packet type.

uint16_t **adv_props**

Advertising packet properties.

uint16_t **interval**

Periodic advertising interval.

If 0 there is no periodic advertising.

uint8_t **primary_phy**

Primary advertising channel PHY.

uint8_t **secondary_phy**

Secondary advertising channel PHY.

struct bt_le_scan_cb

#include <bluetooth.h> Listener context for (LE) scanning.

Public Members

void (***recv**) (**const struct** *bt_le_scan_recv_info* *info, **struct** net_buf_simple *buf)

Advertisement packet received callback.

Parameters

- `info`: Advertiser packet information.
- `buf`: Buffer containing advertiser data.

void (***timeout**) (void)

The scanner has stopped scanning after scan timeout.

struct bt_le_oob_sc_data

#include <bluetooth.h> LE Secure Connections pairing Out of Band data.

Public Members

uint8_t **r**[16]

Random Number.

uint8_t **c**[16]

Confirm Value.

struct bt_le_oob

#include <bluetooth.h> LE Out of Band information.

Public Members

bt_addr_le_t **addr**

LE address. If privacy is enabled this is a Resolvable Private Address.

struct *bt_le_oob_sc_data* **le_sc_data**

LE Secure Connections pairing Out of Band data.

struct bt_br_discovery_result

#include <bluetooth.h> BR/EDR discovery result structure.

Public Members

uint8_t **_priv**[4]

private

bt_addr_t **addr**

Remote device address

int8_t **rssi**

RSSI from inquiry

uint8_t **cod**[3]

Class of Device

uint8_t **eir**[240]

Extended Inquiry Response

struct bt_br_discovery_param

#include <bluetooth.h> BR/EDR discovery parameters

Public Members

uint8_t **length**
Maximum length of the discovery in units of 1.28 seconds. Valid range is 0x01 - 0x30.

bool **limited**
True if limited discovery procedure is to be used.

struct bt_br_oob
#include <bluetooth.h>

Public Members

bt_addr_t **addr**
BR/EDR address.

struct bt_bond_info
#include <bluetooth.h> Information about a bond with a remote device.

Public Members

bt_addr_le_t **addr**
Address of the remote device.

group **bt_addr**
Bluetooth device address definitions and utilities.

Defines

BT_ADDR_LE_PUBLIC

BT_ADDR_LE_RANDOM

BT_ADDR_LE_PUBLIC_ID

BT_ADDR_LE_RANDOM_ID

BT_ADDR_ANY
Bluetooth device "any" address, not a valid address

BT_ADDR_NONE
Bluetooth device "none" address, not a valid address

BT_ADDR_LE_ANY
Bluetooth LE device "any" address, not a valid address

BT_ADDR_LE_NONE
Bluetooth LE device "none" address, not a valid address

BT_ADDR_IS_RPA(*a*)
Check if a Bluetooth LE random address is resolvable private address.

BT_ADDR_IS_NRPA(*a*)
Check if a Bluetooth LE random address is a non-resolvable private address.

BT_ADDR_IS_STATIC(*a*)
Check if a Bluetooth LE random address is a static address.

BT_ADDR_SET_RPA(*a*)
Set a Bluetooth LE random address as a resolvable private address.

BT_ADDR_SET_NRPA(*a*)
Set a Bluetooth LE random address as a non-resolvable private address.

BT_ADDR_SET_STATIC(*a*)
Set a Bluetooth LE random address as a static address.

BT_ADDR_STR_LEN
Recommended length of user string buffer for Bluetooth address.

The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

BT_ADDR_LE_STR_LEN
Recommended length of user string buffer for Bluetooth LE address.

The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

Functions

static inline int **bt_addr_cmp**(**const** *bt_addr_t* **a*, **const** *bt_addr_t* **b*)
Compare Bluetooth device addresses.

Return negative value if $a < b$, 0 if $a == b$, else positive

Parameters

- `a`: First Bluetooth device address to compare
- `b`: Second Bluetooth device address to compare

static inline int **bt_addr_le_cmp**(**const** *bt_addr_le_t* **a*, **const** *bt_addr_le_t* **b*)
Compare Bluetooth LE device addresses.

Return negative value if $a < b$, 0 if $a == b$, else positive

Parameters

- `a`: First Bluetooth LE device address to compare
- `b`: Second Bluetooth LE device address to compare

static inline void **bt_addr_copy**(*bt_addr_t* **dst*, **const** *bt_addr_t* **src*)
Copy Bluetooth device address.

Parameters

- `dst`: Bluetooth device address destination buffer.
- `src`: Bluetooth device address source buffer.

static inline void **bt_addr_le_copy**(*bt_addr_le_t* **dst*, **const** *bt_addr_le_t* **src*)
Copy Bluetooth LE device address.

Parameters

- `dst`: Bluetooth LE device address destination buffer.
- `src`: Bluetooth LE device address source buffer.

int **`bt_addr_le_create_nrpa`**(*bt_addr_le_t *addr*)
Create a Bluetooth LE random non-resolvable private address.

int **`bt_addr_le_create_static`**(*bt_addr_le_t *addr*)
Create a Bluetooth LE random static address.

`static inline` bool **`bt_addr_le_is_rpa`**(**`const`** *bt_addr_le_t *addr*)
Check if a Bluetooth LE address is a random private resolvable address.

Return true if address is a random private resolvable address.

Parameters

- `addr`: Bluetooth LE device address.

`static inline` bool **`bt_addr_le_is_identity`**(**`const`** *bt_addr_le_t *addr*)
Check if a Bluetooth LE address is valid identity address.

Valid Bluetooth LE identity addresses are either public address or random static address.

Return true if address is a valid identity address.

Parameters

- `addr`: Bluetooth LE device address.

`static inline` int **`bt_addr_to_str`**(**`const`** *bt_addr_t *addr*, char **str*, size_t *len*)
Converts binary Bluetooth address to string.

Return Number of successfully formatted bytes from binary address.

Parameters

- `addr`: Address of buffer containing binary Bluetooth address.
- `str`: Address of user buffer with enough room to store formatted string containing binary address.
- `len`: Length of data to be copied to user string buffer. Refer to BT_ADDR_STR_LEN about recommended value.

`static inline` int **`bt_addr_le_to_str`**(**`const`** *bt_addr_le_t *addr*, char **str*, size_t *len*)
Converts binary LE Bluetooth address to string.

Return Number of successfully formatted bytes from binary address.

Parameters

- `addr`: Address of buffer containing binary LE Bluetooth address.
- `str`: Address of user buffer with enough room to store formatted string containing binary LE address.
- `len`: Length of data to be copied to user string buffer. Refer to BT_ADDR_LE_STR_LEN about recommended value.

int **bt_addr_from_str**(**const** char **str*, *bt_addr_t* **addr*)
Convert Bluetooth address from string to binary.

Return Zero on success or (negative) error code otherwise.

Parameters

- `[in] str`: The string representation of a Bluetooth address.
- `[out] addr`: Address of buffer to store the Bluetooth address

int **bt_addr_le_from_str**(**const** char **str*, **const** char **type*, *bt_addr_le_t* **addr*)
Convert LE Bluetooth address from string to binary.

Return Zero on success or (negative) error code otherwise.

Parameters

- `[in] str`: The string representation of an LE Bluetooth address.
- `[in] type`: The string representation of the LE Bluetooth address type.
- `[out] addr`: Address of buffer to store the LE Bluetooth address

struct bt_addr_t
#include <addr.h> Bluetooth Device Address

struct bt_addr_le_t
#include <addr.h> Bluetooth LE Device Address

group **bt_gap_defines**
Bluetooth Generic Access Profile defines and Assigned Numbers.

Defines

BT_COMP_ID_LF
Company Identifiers (see Bluetooth Assigned Numbers)

BT_DATA_FLAGS
EIR/AD data type definitions

BT_DATA_UUID16_SOME

BT_DATA_UUID16_ALL

BT_DATA_UUID32_SOME

BT_DATA_UUID32_ALL

BT_DATA_UUID128_SOME

BT_DATA_UUID128_ALL

BT_DATA_NAME_SHORTENED

BT_DATA_NAME_COMPLETE

BT_DATA_TX_POWER

BT_DATA_SM_TK_VALUE

BT_DATA_SM_OOB_FLAGS

BT_DATA_SOLICIT16

BT_DATA_SOLICIT128

BT_DATA_SVC_DATA16

BT_DATA_GAP_APPEARANCE

BT_DATA_LE_BT_DEVICE_ADDRESS

BT_DATA_LE_ROLE

BT_DATA_SOLICIT32

BT_DATA_SVC_DATA32

BT_DATA_SVC_DATA128

BT_DATA_LE_SC_CONFIRM_VALUE

BT_DATA_LE_SC_RANDOM_VALUE

BT_DATA_URI

BT_DATA_MESH_PROV

BT_DATA_MESH_MESSAGE

BT_DATA_MESH_BEACON

BT_DATA_BIG_INFO

BT_DATA_BROADCAST_CODE

BT_DATA_MANUFACTURER_DATA

BT_LE_AD_LIMITED

BT_LE_AD_GENERAL

BT_LE_AD_NO_BREDR

BT_GAP_SCAN_FAST_INTERVAL

BT_GAP_SCAN_FAST_WINDOW

BT_GAP_SCAN_SLOW_INTERVAL_1

BT_GAP_SCAN_SLOW_WINDOW_1

BT_GAP_SCAN_SLOW_INTERVAL_2

BT_GAP_SCAN_SLOW_WINDOW_2

BT_GAP_ADV_FAST_INT_MIN_1

BT_GAP_ADV_FAST_INT_MAX_1

BT_GAP_ADV_FAST_INT_MIN_2

BT_GAP_ADV_FAST_INT_MAX_2

BT_GAP_ADV_SLOW_INT_MIN

BT_GAP_ADV_SLOW_INT_MAX

BT_GAP_INIT_CONN_INT_MIN

BT_GAP_INIT_CONN_INT_MAX

BT_GAP_ADV_MAX_ADV_DATA_LEN
Maximum advertising data length.

BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN
Maximum extended advertising data length.

Note The maximum advertising data length that can be sent by an extended advertiser is defined by the controller.

BT_GAP_TX_POWER_INVALID

BT_GAP_RSSI_INVALID

BT_GAP_SID_INVALID

BT_GAP_NO_TIMEOUT

BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT

BT_GAP_DATA_LEN_DEFAULT

BT_GAP_DATA_LEN_MAX

BT_GAP_DATA_TIME_DEFAULT

BT_GAP_DATA_TIME_MAX

BT_GAP_SID_MAX

BT_GAP_PER_ADV_MAX_MAX_SKIP

BT_GAP_PER_ADV_MAX_MAX_TIMEOUT

Enums

enum [anonymous]
LE PHY types

Values:

enumerator BT_GAP_LE_PHY_NONE
Convenience macro for when no PHY is set.

enumerator BT_GAP_LE_PHY_1M
LE 1M PHY

enumerator BT_GAP_LE_PHY_2M
LE 2M PHY

enumerator BT_GAP_LE_PHY_CODED
LE Coded PHY

enum [anonymous]
Advertising PDU types

Values:

enumerator BT_GAP_ADV_TYPE_ADV_IND
Scannable and connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_DIRECT_IND
Directed connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_SCAN_IND
Non-connectable and scannable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_NONCONN_IND
Non-connectable and non-scannable advertising.

enumerator BT_GAP_ADV_TYPE_SCAN_RSP
Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_TYPE_EXT_ADV
Extended advertising, see advertising properties.

enum [anonymous]
Advertising PDU properties

Values:

enumerator BT_GAP_ADV_PROP_CONNECTABLE
Connectable advertising.

enumerator BT_GAP_ADV_PROP_SCANNABLE
Scannable advertising.

enumerator BT_GAP_ADV_PROP_DIRECTED
Directed advertising.

enumerator BT_GAP_ADV_PROP_SCAN_RESPONSE
Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_PROP_EXT_ADV
Extended advertising.

enum [anonymous]
Constant Tone Extension (CTE) types

Values:

enumerator BT_GAP_CTE_AOA
Angle of Arrival

enumerator BT_GAP_CTE_AOD_1US
Angle of Departure with 1 us slots

enumerator BT_GAP_CTE_AOD_2US
Angle of Departure with 2 us slots

enumerator BT_GAP_CTE_NONE
No extensions

1.4 Generic Attribute Profile (GATT)

GATT layer manages the service database providing APIs for service registration and attribute declaration.

Services can be registered using `bt_gatt_service_register()` API which takes the `bt_gatt_service` struct that provides the list of attributes the service contains. The helper macro `BT_GATT_SERVICE()` can be used to declare a service.

Attributes can be declared using the `bt_gatt_attr` struct or using one of the helper macros:

BT_GATT_PRIMARY_SERVICE Declares a Primary Service.

BT_GATT_SECONDARY_SERVICE Declares a Secondary Service.

BT_GATT_INCLUDE_SERVICE Declares a Include Service.

BT_GATT_CHARACTERISTIC Declares a Characteristic.

BT_GATT_DESCRIPTOR Declares a Descriptor.

BT_GATT_ATTRIBUTE Declares an Attribute.

BT_GATT_CCC Declares a Client Characteristic Configuration.

BT_GATT_CEP Declares a Characteristic Extended Properties.

BT_GATT_CUD Declares a Characteristic User Format.

Each attribute contain a `uuid`, which describes their type, a `read` callback, a `write` callback and a set of permission. Both read and write callbacks can be set to NULL if the attribute permission don't allow their respective operations.

Note: Attribute `read` and `write` callbacks are called directly from RX Thread thus it is not recommended to block for long periods of time in them.

Attribute value changes can be notified using *bt_gatt_notify()* API, alternatively there is *bt_gatt_notify_cb()* where is is possible to pass a callback to be called when it is necessary to know the exact instant when the data has been transmitted over the air. Indications are supported by *bt_gatt_indicate()* API.

Client procedures can be enabled with the configuration option: `CONFIG_BT_GATT_CLIENT`

Discover procedures can be initiated with the use of *bt_gatt_discover()* API which takes the *bt_gatt_discover_params* struct which describes the type of discovery. The parameters also serves as a filter when setting the `uuid` field only attributes which matches will be discovered, in contrast setting it to NULL allows all attributes to be discovered.

Note: Caching discovered attributes is not supported.

Read procedures are supported by *bt_gatt_read()* API which takes the *bt_gatt_read_params* struct as parameters. In the parameters one or more attributes can be set, though setting multiple handles requires the option: `CONFIG_BT_GATT_READ_MULTIPLE`

Write procedures are supported by *bt_gatt_write()* API and takes *bt_gatt_write_params* struct as parameters. In case the write operation don't require a response *bt_gatt_write_without_response()* or *bt_gatt_write_without_response_cb()* APIs can be used, with the later working similarly to *bt_gatt_notify_cb()*.

Subscriptions to notification and indication can be initiated with use of *bt_gatt_subscribe()* API which takes *bt_gatt_subscribe_params* as parameters. Multiple subscriptions to the same attribute are supported so there could be multiple `notify` callback being triggered for the same attribute. Subscriptions can be removed with use of *bt_gatt_unsubscribe()* API.

Note: When subscriptions are removed `notify` callback is called with the data set to NULL.

1.4.1 API Reference

group **`bt_gatt`**

Generic Attribute Profile (GATT)

Defines

`BT_GATT_ERR`(*_att_err*)

Construct error return value for attribute read and write callbacks.

Return Appropriate error code for the attribute callbacks.

Parameters

- `_att_err`: ATT error code

`BT_GATT_CHRC_BROADCAST`

Characteristic broadcast property.

Characteristic Properties Bit field values If set, permits broadcasts of the Characteristic Value using Server Characteristic Configuration Descriptor.

`BT_GATT_CHRC_READ`

Characteristic read property.

If set, permits reads of the Characteristic Value.

`BT_GATT_CHRC_WRITE_WITHOUT_RESP`

Characteristic write without response property.

If set, permits write of the Characteristic Value without response.

`BT_GATT_CHRC_WRITE`

Characteristic write with response property.

If set, permits write of the Characteristic Value with response.

`BT_GATT_CHRC_NOTIFY`

Characteristic notify property.

If set, permits notifications of a Characteristic Value without acknowledgment.

`BT_GATT_CHRC_INDICATE`

Characteristic indicate property.

If set, permits indications of a Characteristic Value with acknowledgment.

`BT_GATT_CHRC_AUTH`

Characteristic Authenticated Signed Writes property.

If set, permits signed writes to the Characteristic Value.

`BT_GATT_CHRC_EXT_PROP`

Characteristic Extended Properties property.

If set, additional characteristic properties are defined in the Characteristic Extended Properties Descriptor.

`BT_GATT_CEP_RELIABLE_WRITE`

Characteristic Extended Properties Bit field values

`BT_GATT_CEP_WRITABLE_AUX`

BT_GATT_CCC_NOTIFY
Client Characteristic Configuration Notification.

Client Characteristic Configuration Values If set, changes to Characteristic Value shall be notified.

BT_GATT_CCC_INDICATE
Client Characteristic Configuration Indication.

If set, changes to Characteristic Value shall be indicated.

Enums

enum [anonymous]
GATT attribute permission bit field values

Values:

enumerator BT_GATT_PERM_NONE
No operations supported, e.g. for notify-only

enumerator BT_GATT_PERM_READ
Attribute read permission.

enumerator BT_GATT_PERM_WRITE
Attribute write permission.

enumerator BT_GATT_PERM_READ_ENCRYPT
Attribute read permission with encryption.

If set, requires encryption for read access.

enumerator BT_GATT_PERM_WRITE_ENCRYPT
Attribute write permission with encryption.

If set, requires encryption for write access.

enumerator BT_GATT_PERM_READ_AUTHEN
Attribute read permission with authentication.

If set, requires encryption using authenticated link-key for read access.

enumerator BT_GATT_PERM_WRITE_AUTHEN
Attribute write permission with authentication.

If set, requires encryption using authenticated link-key for write access.

enumerator BT_GATT_PERM_PREPARE_WRITE
Attribute prepare write permission.

If set, allows prepare writes with use of BT_GATT_WRITE_FLAG_PREPARE passed to write callback.

enum [anonymous]
GATT attribute write flags

Values:

enumerator BT_GATT_WRITE_FLAG_PREPARE
Attribute prepare write flag.

If set, write callback should only check if the device is authorized but no data shall be written.

enumerator BT_GATT_WRITE_FLAG_CMD

Attribute write command flag.

If set, indicates that write operation is a command (Write without response) which doesn't generate any response.

struct bt_gatt_attr

#include <gatt.h> GATT Attribute structure.

Public Members

struct *bt_uuid* ***uuid**

Attribute UUID

ssize_t (***read**) (**struct** bt_conn *conn, **const struct** *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Attribute read callback.

The callback can also be used locally to read the contents of the attribute in which case no connection will be set.

Return Number fo bytes read, or in case of an error *BT_GATT_ERR()* with a specific ATT error code.

Parameters
- `conn`: The connection that is requesting to read
- `attr`: The attribute that's being read
- `buf`: Buffer to place the read result in
- `len`: Length of data to read
- `offset`: Offset to start reading from

ssize_t (***write**) (**struct** bt_conn *conn, **const struct** *bt_gatt_attr* *attr, **const** void *buf, uint16_t len, uint16_t offset, uint8_t flags)

Attribute write callback.

Return Number of bytes written, or in case of an error *BT_GATT_ERR()* with a specific ATT error code.

Parameters
- `conn`: The connection that is requesting to write
- `attr`: The attribute that's being written
- `buf`: Buffer with the data to write
- `len`: Number of bytes in the buffer
- `offset`: Offset to start writing from
- `flags`: Flags (BT_GATT_WRITE_*)

void ***user_data**

Attribute user data

uint16_t **handle**

Attribute handle

uint8_t **perm**

Attribute permissions

struct bt_gatt_service_static

#include <gatt.h> GATT Service structure.

Public Members

struct *bt_gatt_attr* ***attrs**
Service Attributes

size_t **attr_count**
Service Attribute count

struct bt_gatt_service
#include <gatt.h> GATT Service structure.

Public Members

struct *bt_gatt_attr* ***attrs**
Service Attributes

size_t **attr_count**
Service Attribute count

struct bt_gatt_service_val
#include <gatt.h> Service Attribute Value.

Public Members

struct *bt_uuid* ***uuid**
Service UUID.

uint16_t **end_handle**
Service end handle.

struct bt_gatt_include
#include <gatt.h> Include Attribute Value.

Public Members

struct *bt_uuid* ***uuid**
Service UUID.

uint16_t **start_handle**
Service start handle.

uint16_t **end_handle**
Service end handle.

struct bt_gatt_chrc
#include <gatt.h> Characteristic Attribute Value.

Public Members

struct *bt_uuid* ***uuid**

Characteristic UUID.

uint16_t **value_handle**

Characteristic Value handle.

uint8_t **properties**

Characteristic properties.

struct bt_gatt_cep

#include <gatt.h> Characteristic Extended Properties Attribute Value.

Public Members

uint16_t **properties**

Characteristic Extended properties

struct bt_gatt_ccc

#include <gatt.h> Client Characteristic Configuration Attribute Value

Public Members

uint16_t **flags**

Client Characteristic Configuration flags

struct bt_gatt_cpf

#include <gatt.h> GATT Characteristic Presentation Format Attribute Value.

Public Members

uint8_t **format**

Format of the value of the characteristic

int8_t **exponent**

Exponent field to determine how the value of this characteristic is further formatted

uint16_t **unit**

Unit of the characteristic

uint8_t **name_space**

Name space of the description

uint16_t **description**

Description of the characteristic as defined in a higher layer profile

1.4.1.1 GATT Server

group **`bt_gatt_server`**

Defines

`BT_GATT_SERVICE_DEFINE`(*_name*, *...*)
Statically define and register a service.

Helper macro to statically define and register a service.

Parameters

- `_name`: Service name.

`_BT_GATT_ATTRS_ARRAY_DEFINE`(*n*, *_instances*, *_attrs_def*)

`_BT_GATT_SERVICE_ARRAY_ITEM`(*_n*, *_*)

`BT_GATT_SERVICE_INSTANCE_DEFINE`(*_name*, *_instances*, *_instance_num*, *_attrs_def*)
Statically define service structure array.

Helper macro to statically define service structure array. Each element of the array is linked to the service attribute array which is also defined in this scope using _attrs_def macro.

Parameters

- `_name`: Name of service structure array.
- `_instances`: Array of instances to pass as user context to the attribute callbacks.
- `_instance_num`: Number of elements in instance array.
- `_attrs_def`: Macro provided by the user that defines attribute array for the serivce. This macro should accept single parameter which is the instance context.

`BT_GATT_SERVICE`(*_attrs*)
Service Structure Declaration Macro.

Helper macro to declare a service structure.

Parameters

- `_attrs`: Service attributes.

`BT_GATT_PRIMARY_SERVICE`(*_service*)
Primary Service Declaration Macro.

Helper macro to declare a primary service attribute.

Parameters

- `_service`: Service attribute value.

`BT_GATT_SECONDARY_SERVICE`(*_service*)
Secondary Service Declaration Macro.

Helper macro to declare a secondary service attribute.

Parameters

- `_service`: Service attribute value.

`BT_GATT_INCLUDE_SERVICE`(*_service_incl*)
Include Service Declaration Macro.

Helper macro to declare database internal include service attribute.

Parameters

- `_service_incl`: the first service attribute of service to include

`BT_GATT_CHRC_INIT`(*_uuid*, *_handle*, *_props*)

`BT_GATT_CHARACTERISTIC`(*_uuid*, *_props*, *_perm*, *_read*, *_write*, *_value*)
Characteristic and Value Declaration Macro.

Helper macro to declare a characteristic attribute along with its attribute value.

Parameters

- `_uuid`: Characteristic attribute uuid.
- `_props`: Characteristic attribute properties.
- `_perm`: Characteristic Attribute access permissions.
- `_read`: Characteristic Attribute read callback.
- `_write`: Characteristic Attribute write callback.
- `_value`: Characteristic Attribute value.

`BT_GATT_CCC_MAX`

`BT_GATT_CCC_INITIALIZER`(*_changed*, *_write*, *_match*)
Initialize Client Characteristic Configuration Declaration Macro.

Helper macro to initialize a Managed CCC attribute value.

Parameters

- `_changed`: Configuration changed callback.
- `_write`: Configuration write callback.
- `_match`: Configuration match callback.

`BT_GATT_CCC_MANAGED`(*_ccc*, *_perm*)
Managed Client Characteristic Configuration Declaration Macro.

Helper macro to declare a Managed CCC attribute.

Parameters

- `_ccc`: CCC attribute user data, shall point to a *_bt_gatt_ccc*.
- `_perm`: CCC access permissions.

BT_GATT_CCC(*_changed*, *_perm*)

Client Characteristic Configuration Declaration Macro.

Helper macro to declare a CCC attribute.

Parameters

- `_changed`: Configuration changed callback.
- `_perm`: CCC access permissions.

BT_GATT_CEP(*_value*)

Characteristic Extended Properties Declaration Macro.

Helper macro to declare a CEP attribute.

Parameters

- `_value`: Descriptor attribute value.

BT_GATT_CUD(*_value*, *_perm*)

Characteristic User Format Descriptor Declaration Macro.

Helper macro to declare a CUD attribute.

Parameters

- `_value`: User description NULL-terminated C string.
- `_perm`: Descriptor attribute access permissions.

BT_GATT_CPF(*_value*)

Characteristic Presentation Format Descriptor Declaration Macro.

Helper macro to declare a CPF attribute.

Parameters

- `_value`: Descriptor attribute value.

BT_GATT_DESCRIPTOR(*_uuid*, *_perm*, *_read*, *_write*, *_value*)

Descriptor Declaration Macro.

Helper macro to declare a descriptor attribute.

Parameters

- `_uuid`: Descriptor attribute uuid.
- `_perm`: Descriptor attribute access permissions.
- `_read`: Descriptor attribute read callback.
- `_write`: Descriptor attribute write callback.
- `_value`: Descriptor attribute value.

BT_GATT_ATTRIBUTE(*_uuid*, *_perm*, *_read*, *_write*, *_value*)
Attribute Declaration Macro.

Helper macro to declare an attribute.

Parameters

- `_uuid`: Attribute uuid.
- `_perm`: Attribute access permissions.
- `_read`: Attribute read callback.
- `_write`: Attribute write callback.
- `_value`: Attribute value.

Typedefs

typedef uint8_t (***bt_gatt_attr_func_t**)(**const struct** *bt_gatt_attr* *attr, uint16_t handle, void *user_data)
Attribute iterator callback.

Return BT_GATT_ITER_CONTINUE if should continue to the next attribute.

BT_GATT_ITER_STOP to stop.

Parameters

- `attr`: Attribute found.
- `handle`: Attribute handle found.
- `user_data`: Data given.

typedef void (***bt_gatt_complete_func_t**)(**struct** bt_conn *conn, void *user_data)
Notification complete result callback.

Parameters

- `conn`: Connection object.
- `user_data`: Data passed in by the user.

typedef void (***bt_gatt_indicate_func_t**)(**struct** bt_conn *conn, **struct** *bt_gatt_indicate_params* *params, uint8_t err)
Indication complete result callback.

Parameters

- `conn`: Connection object.
- `params`: Indication params object.
- `err`: ATT error code

typedef void (***bt_gatt_indicate_params_destroy_t**)(**struct** *bt_gatt_indicate_params* *params)

Enums

enum [anonymous]

Values:

enumerator BT_GATT_ITER_STOP

enumerator BT_GATT_ITER_CONTINUE

Functions

int **bt_gatt_service_register**(**struct** *bt_gatt_service* **svc*)

Register GATT service.

Register GATT service. Applications can make use of macros such as BT_GATT_PRIMARY_SERVICE, BT_GATT_CHARACTERISTIC, BT_GATT_DESCRIPTOR, etc.

When using `CONFIG_BT_SETTINGS` then all services that should have bond configuration loaded, i.e. CCC values, must be registered before calling settings_load.

When using `CONFIG_BT_GATT_CACHING` and `CONFIG_BT_SETTINGS` then all services that should be included in the GATT Database Hash calculation should be added before calling settings_load. All services registered after settings_load will trigger a new database hash calculation and a new hash stored.

Return 0 in case of success or negative value in case of error.

Parameters

- `svc`: Service containing the available attributes

int **bt_gatt_service_unregister**(**struct** *bt_gatt_service* **svc*)

Unregister GATT service. *.

Return 0 in case of success or negative value in case of error.

Parameters

- `svc`: Service to be unregistered.

void **bt_gatt_foreach_attr_type**(uint16_t *start_handle*, uint16_t *end_handle*, **const struct** *bt_uuid* **uuid*, **const** void **attr_data*, uint16_t *num_matches*, *bt_gatt_attr_func_t func*, void **user_data*)

Attribute iterator by type.

Iterate attributes in the given range matching given UUID and/or data.

Parameters

- `start_handle`: Start handle.
- `end_handle`: End handle.
- `uuid`: UUID to match, passing NULL skips UUID matching.
- `attr_data`: Attribute data to match, passing NULL skips data matching.
- `num_matches`: Number matches, passing 0 makes it unlimited.
- `func`: Callback function.

- `user_data`: Data to pass to the callback.

static inline void **bt_gatt_foreach_attr**(uint16_t *start_handle*, uint16_t *end_handle*, *bt_gatt_attr_func_t* *func*, void **user_data*)

Attribute iterator.

Iterate attributes in the given range.

Parameters

- `start_handle`: Start handle.
- `end_handle`: End handle.
- `func`: Callback function.
- `user_data`: Data to pass to the callback.

struct *bt_gatt_attr* ***bt_gatt_attr_next**(**const struct** *bt_gatt_attr* **attr*)

Iterate to the next attribute.

Iterate to the next attribute following a given attribute.

Return The next attribute or NULL if it cannot be found.

Parameters

- `attr`: Current Attribute.

uint16_t **bt_gatt_attr_get_handle**(**const struct** *bt_gatt_attr* **attr*)

Get Attribute handle.

Return Handle of the corresponding attribute or zero if the attribute could not be found.

Parameters

- `attr`: Attribute object.

uint16_t **bt_gatt_attr_value_handle**(**const struct** *bt_gatt_attr* **attr*)

Get the handle of the characteristic value descriptor.

Note The user_data of the attribute must of type *bt_gatt_chrc*.

Return the handle of the corresponding Characteristic Value. The value will be zero (the invalid handle) if `attr` was not a characteristic attribute.

Parameters

- `attr`: A Characteristic Attribute.

ssize_t **bt_gatt_attr_read**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, void **buf*, uint16_t *buf_len*, uint16_t *offset*, **const** void **value*, uint16_t *value_len*)

Generic Read Attribute value helper.

Read attribute value from local database storing the result into buffer.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value.
- `buf_len`: Buffer length.
- `offset`: Start offset.
- `value`: Attribute value.
- `value_len`: Length of the attribute value.

ssize_t **`bt_gatt_attr_read_service`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr *attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Service Attribute helper.

Read service attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a *bt_uuid*.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value read.
- `len`: Buffer length.
- `offset`: Start offset.

ssize_t **`bt_gatt_attr_read_included`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr *attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Include Attribute helper.

Read include service attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a *bt_gatt_include*.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value read.
- `len`: Buffer length.
- `offset`: Start offset.

ssize_t **`bt_gatt_attr_read_chrc`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr *attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Characteristic Attribute helper.

Read characteristic attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a *bt_gatt_chrc*.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value read.
- `len`: Buffer length.
- `offset`: Start offset.

ssize_t **bt_gatt_attr_read_ccc**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Client Characteristic Configuration Attribute helper.

Read CCC attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a *_bt_gatt_ccc*.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value read.
- `len`: Buffer length.
- `offset`: Start offset.

ssize_t **bt_gatt_attr_write_ccc**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, **const** void **buf*, uint16_t *len*, uint16_t *offset*, uint8_t *flags*)

Write Client Characteristic Configuration Attribute helper.

Write value in the buffer into CCC attribute.

Note Only use this with attributes which user_data is a *_bt_gatt_ccc*.

Return number of bytes written in case of success or negative values in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Attribute to read.
- `buf`: Buffer to store the value read.
- `len`: Buffer length.
- `offset`: Start offset.
- `flags`: Write flags.

ssize_t **bt_gatt_attr_read_cep**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Characteristic Extended Properties Attribute helper.

Read CEP attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a *bt_gatt_cep*.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object
- `attr`: Attribute to read
- `buf`: Buffer to store the value read
- `len`: Buffer length
- `offset`: Start offset

ssize_t **`bt_gatt_attr_read_cud`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Characteristic User Description Descriptor Attribute helper.

Read CUD attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a NULL-terminated C string.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object
- `attr`: Attribute to read
- `buf`: Buffer to store the value read
- `len`: Buffer length
- `offset`: Start offset

ssize_t **`bt_gatt_attr_read_cpf`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

Read Characteristic Presentation format Descriptor Attribute helper.

Read CPF attribute value from local database storing the result into buffer after encoding it.

Note Only use this with attributes which user_data is a bt_gatt_pf.

Return number of bytes read in case of success or negative values in case of error.

Parameters

- `conn`: Connection object
- `attr`: Attribute to read
- `buf`: Buffer to store the value read
- `len`: Buffer length
- `offset`: Start offset

int **`bt_gatt_notify_cb`**(**`struct`** bt_conn **conn*, **`struct`** *bt_gatt_notify_params* **params*)

Notify attribute value change.

This function works in the same way as *bt_gatt_notify*. With the addition that after sending the notification the callback function will be called.

The callback is run from System Workqueue context.

Alternatively it is possible to notify by UUID by setting it on the parameters, when using this method the attribute given is used as the start range when looking up for possible matches.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Notification parameters.

int **`bt_gatt_notify_multiple`**(**`struct`** bt_conn **conn*, uint16_t *num_params*, **`struct`** *bt_gatt_notify_params* **params*)

Notify multiple attribute value change.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `num_params`: Number of notification parameters.
- `params`: Array of notification parameters.

`static inline` int **`bt_gatt_notify`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr* **attr*, **`const`** void **data*, uint16_t *len*)

Notify attribute value change.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `attr`: Characteristic or Characteristic Value attribute.
- `data`: Pointer to Attribute data.
- `len`: Attribute value length.

`static inline` int **`bt_gatt_notify_uuid`**(**`struct`** bt_conn **conn*, **`const struct`** *bt_uuid* **uuid*, **`const struct`** *bt_gatt_attr* **attr*, **`const`** void **data*, uint16_t *len*)

Notify attribute value change by UUID.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only on the given connection.

The attribute object is the starting point for the search of the UUID.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `uuid`: The UUID. If the server contains multiple services with the same UUID, then the first occurrence, starting from the attr given, is used.
- `attr`: Pointer to an attribute that serves as the starting point for the search of a match for the UUID.
- `data`: Pointer to Attribute data.
- `len`: Attribute value length.

int **bt_gatt_indicate**(**struct** bt_conn **conn*, **struct** bt_gatt_indicate_params **params*)

Indicate attribute value change.

Send an indication of attribute value change. if connection is NULL indicate all peer that have notification enabled via CCC otherwise do a direct indication only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

The callback is run from System Workqueue context.

Alternatively it is possible to indicate by UUID by setting it on the parameters, when using this method the attribute given is used as the start range when looking up for possible matches.

Note This procedure is asynchronous therefore the parameters need to remains valid while it is active. The procedure is active until the destroy callback is run.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Indicate parameters.

bool **bt_gatt_is_subscribed**(**struct** bt_conn **conn*, **const struct** bt_gatt_attr **attr*, uint16_t *ccc_value*)

Check if connection have subscribed to attribute.

Check if connection has subscribed to attribute value change.

The attribute object can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC, or the Client Characteristic Configuration Descriptor (CCCD) which is created by BT_GATT_CCC.

Return true if the attribute object has been subscribed.

Parameters

- `conn`: Connection object.
- `attr`: Attribute object.

- `ccc_value`: The subscription type, either notifications or indications.

uint16_t **bt_gatt_get_mtu**(**struct** bt_conn **conn*)

Get ATT MTU for a connection.

Get negotiated ATT connection MTU, note that this does not equal the largest amount of attribute data that can be transferred within a single packet.

Return MTU in bytes

Parameters

- `conn`: Connection object.

struct bt_gatt_ccc_cfg

#include <gatt.h> GATT CCC configuration entry.

Public Members

uint8_t **id**

Local identity, BT_ID_DEFAULT in most cases.

bt_addr_le_t **peer**

Remote peer address.

uint16_t **value**

Configuration value.

struct _bt_gatt_ccc

#include <gatt.h> Internal representation of CCC value

Public Members

struct *bt_gatt_ccc_cfg* **cfg**[(**CONFIG_BT_MAX_PAIRED** + **CONFIG_BT_MAX_CONN**)]

Configuration for each connection

uint16_t **value**

Highest value of all connected peer's subscriptions

void (***cfg_changed**)(**const struct** *bt_gatt_attr* *attr, uint16_t value)

CCC attribute changed callback.

Parameters

- `attr`: The attribute that's changed value
- `value`: New value

ssize_t (***cfg_write**)(**struct** bt_conn *conn, **const struct** *bt_gatt_attr* *attr, uint16_t value)

CCC attribute write validation callback.

Return Number of bytes to write, or in case of an error *BT_GATT_ERR()* with a specific error code.

Parameters

- `conn`: The connection that is requesting to write
- `attr`: The attribute that's being written
- `value`: CCC value to write

bool (***cfg_match**) (**struct** bt_conn *conn, **const struct** *bt_gatt_attr* *attr)

CCC attribute match handler.

Indicate if it is OK to send a notification or indication to the subscriber.

Return true if application has approved notification/indication, false if application does not approve.

Parameters

- `conn`: The connection that is being checked
- `attr`: The attribute that's being checked

struct bt_gatt_notify_params

#include <gatt.h>

Public Members

struct *bt_uuid* ***uuid**

Notification Attribute UUID type

struct *bt_gatt_attr* ***attr**

Notification Attribute object

const void ***data**

Notification Value data

uint16_t **len**

Notification Value length

bt_gatt_complete_func_t **func**

Notification Value callback

void ***user_data**

Notification Value callback user data

struct bt_gatt_indicate_params

#include <gatt.h> GATT Indicate Value parameters.

Public Members

struct *bt_uuid* ***uuid**

Notification Attribute UUID type

struct *bt_gatt_attr* ***attr**

Indicate Attribute object

bt_gatt_indicate_func_t **func**

Indicate Value callback

bt_gatt_indicate_params_destroy_t **destroy**

Indicate operation complete callback

const void ***data**

Indicate Value data

uint16_t **len**

Indicate Value length

uint8_t **_ref**

Private reference counter

1.4.1.2 GATT Client

group **bt_gatt_client**

Typedefs

typedef uint8_t (***bt_gatt_discover_func_t**)(**struct** bt_conn *conn, **const struct** *bt_gatt_attr* *attr, **struct** *bt_gatt_discover_params* *params)

Discover attribute callback function.

If discovery procedure has completed this callback will be called with attr set to NULL. This will not happen if procedure was stopped by returning BT_GATT_ITER_STOP.

Parameters

- `conn`: Connection object.
- `attr`: Attribute found, or NULL if not found.
- `params`: Discovery parameters given.

The attribute object as well as its UUID and value objects are temporary and must be copied to in order to cache its information. Only the following fields of the attribute contains valid information:

- uuid UUID representing the type of attribute.
- handle Handle in the remote database.
- user_data The value of the attribute. Will be NULL when discovering descriptors

To be able to read the value of the discovered attribute the user_data must be cast to an appropriate type.

- *bt_gatt_service_val* when UUID is *BT_UUID_GATT_PRIMARY* or *BT_UUID_GATT_SECONDARY*.
- *bt_gatt_include* when UUID is *BT_UUID_GATT_INCLUDE*.
- *bt_gatt_chrc* when UUID is *BT_UUID_GATT_CHRC*.

Return BT_GATT_ITER_CONTINUE to continue discovery procedure.

BT_GATT_ITER_STOP to stop discovery procedure.

typedef uint8_t (***bt_gatt_read_func_t**)(**struct** bt_conn *conn, uint8_t err, **struct** *bt_gatt_read_params* *params, **const** void *data, uint16_t length)

Read callback function.

Return BT_GATT_ITER_CONTINUE if should continue to the next attribute.

BT_GATT_ITER_STOP to stop.

Parameters

- `conn`: Connection object.
- `err`: ATT error code.
- `params`: Read parameters used.

- `data`: Attribute value data. NULL means read has completed.
- `length`: Attribute value length.

typedef void (***bt_gatt_write_func_t**)(**struct** bt_conn *conn, uint8_t err, **struct** *bt_gatt_write_params* *params)

Write callback function.

Parameters

- `conn`: Connection object.
- `err`: ATT error code.
- `params`: Write parameters used.

typedef uint8_t (***bt_gatt_notify_func_t**)(**struct** bt_conn *conn, **struct** *bt_gatt_subscribe_params* *params, **const** void *data, uint16_t length)

Notification callback function.

Return BT_GATT_ITER_CONTINUE to continue receiving value notifications. BT_GATT_ITER_STOP to unsubscribe from value notifications.

Parameters

- `conn`: Connection object. May be NULL, indicating that the peer is being unpaired
- `params`: Subscription parameters.
- `data`: Attribute value data. If NULL then subscription was removed.
- `length`: Attribute value length.

Enums

enum [anonymous]

GATT Discover types

Values:

enumerator BT_GATT_DISCOVER_PRIMARY

Discover Primary Services.

enumerator BT_GATT_DISCOVER_SECONDARY

Discover Secondary Services.

enumerator BT_GATT_DISCOVER_INCLUDE

Discover Included Services.

enumerator BT_GATT_DISCOVER_CHARACTERISTIC

Discover Characteristic Values.

Discover Characteristic Value and its properties.

enumerator BT_GATT_DISCOVER_DESCRIPTOR

Discover Descriptors.

Discover Attributes which are not services or characteristics.

Note The use of this type of discover is not recommended for discovering in ranges across multiple services/characteristics as it may incur in extra round trips.

`enumerator BT_GATT_DISCOVER_ATTRIBUTE`

Discover Attributes.

Discover Attributes of any type.

Note The use of this type of discover is not recommended for discovering in ranges across multiple services/characteristics as it may incur in more round trips.

`enum` [anonymous]

Subscription flags

Values:

`enumerator BT_GATT_SUBSCRIBE_FLAG_VOLATILE`

Persistence flag.

If set, indicates that the subscription is not saved on the GATT server side. Therefore, upon disconnection, the subscription will be automatically removed from the client's subscriptions list and when the client reconnects, it will have to issue a new subscription.

`enumerator BT_GATT_SUBSCRIBE_FLAG_NO_RESUB`

No resubscribe flag.

By default when BT_GATT_SUBSCRIBE_FLAG_VOLATILE is unset, the subscription will be automatically renewed when the client reconnects, as a workaround for GATT servers that do not persist subscriptions.

This flag will disable the automatic resubscription. It is useful if the application layer knows that the GATT server remembers subscriptions from previous connections and wants to avoid renewing the subscriptions.

`enumerator BT_GATT_SUBSCRIBE_FLAG_WRITE_PENDING`

Write pending flag.

If set, indicates write operation is pending waiting remote end to respond.

`enumerator BT_GATT_SUBSCRIBE_NUM_FLAGS`

Functions

int **`bt_gatt_exchange_mtu`**(`struct` bt_conn **conn*, `struct` *bt_gatt_exchange_params* **params*)

Exchange MTU.

This client procedure can be used to set the MTU to the maximum possible size the buffers can hold.

Note Shall only be used once per connection.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Exchange MTU parameters.

int **`bt_gatt_discover`**(**`struct`** bt_conn **conn*, **`struct`** *bt_gatt_discover_params* **params*)

GATT Discover function.

This procedure is used by a client to discover attributes on a server.

Primary Service Discovery: Procedure allows to discover specific Primary Service based on UUID. Include Service Discovery: Procedure allows to discover all Include Services within specified range. Characteristic Discovery: Procedure allows to discover all characteristics within specified handle range as well as discover characteristics with specified UUID. Descriptors Discovery: Procedure allows to discover all characteristic descriptors within specified range.

For each attribute found the callback is called which can then decide whether to continue discovering or stop.

Note This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Discover parameters.

int **`bt_gatt_read`**(**`struct`** bt_conn **conn*, **`struct`** *bt_gatt_read_params* **params*)

Read Attribute Value by handle.

This procedure read the attribute value and return it to the callback.

When reading attributes by UUID the callback can be called multiple times depending on how many instances of given the UUID exists with the start_handle being updated for each instance.

If an instance does contain a long value which cannot be read entirely the caller will need to read the remaining data separately using the handle and offset.

Note This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Read parameters.

int **`bt_gatt_write`**(**`struct`** bt_conn **conn*, **`struct`** *bt_gatt_write_params* **params*)

Write Attribute Value by handle.

This procedure write the attribute value and return the result in the callback.

Note This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Write parameters.

int **bt_gatt_write_without_response_cb**(**struct** bt_conn **conn*, uint16_t *handle*, **const** void **data*, uint16_t *length*, bool *sign*, *bt_gatt_complete_func_t* *func*, void **user_data*)

Write Attribute Value by handle without response with callback.

This function works in the same way as *bt_gatt_write_without_response*. With the addition that after sending the write the callback function will be called.

The callback is run from System Workqueue context.

Note By using a callback it also disable the internal flow control which would prevent sending multiple commands without waiting for their transmissions to complete, so if that is required the caller shall not submit more data until the callback is called.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `handle`: Attribute handle.
- `data`: Data to be written.
- `length`: Data length.
- `sign`: Whether to sign data
- `func`: Transmission complete callback.
- `user_data`: User data to be passed back to callback.

static inline int **bt_gatt_write_without_response**(**struct** bt_conn **conn*, uint16_t *handle*, **const** void **data*, uint16_t *length*, bool *sign*)

Write Attribute Value by handle without response.

This procedure write the attribute value without requiring an acknowledgment that the write was successfully performed

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `handle`: Attribute handle.
- `data`: Data to be written.
- `length`: Data length.
- `sign`: Whether to sign data

int **bt_gatt_subscribe**(**struct** bt_conn **conn*, **struct** *bt_gatt_subscribe_params* **params*)

Subscribe Attribute Value Notification.

This procedure subscribe to value notification using the Client Characteristic Configuration handle. If notification received subscribe value callback is called to return notified value. One may then decide whether to unsubscribe directly from this callback. Notification callback with NULL data will not be called if subscription was removed by this method.

Note Notifications are asynchronous therefore the parameters need to remain valid while subscribed.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Subscribe parameters.

int **`bt_gatt_resubscribe`**(uint8_t *id*, **`const`** *bt_addr_le_t* **peer*, **`struct`** *bt_gatt_subscribe_params* **params*)

Resubscribe Attribute Value Notification subscription.

Resubscribe to Attribute Value Notification when already subscribed from a previous connection. The GATT server will remember subscription from previous connections when bonded, so resubscribing can be done without performing a new subscribe procedure after a power cycle.

Note Notifications are asynchronous therefore the parameters need to remain valid while subscribed.

Return 0 in case of success or negative value in case of error.

Parameters

- `id`: Local identity (in most cases BT_ID_DEFAULT).
- `peer`: Remote address.
- `params`: Subscribe parameters.

int **`bt_gatt_unsubscribe`**(**`struct`** bt_conn **conn*, **`struct`** *bt_gatt_subscribe_params* **params*)

Unsubscribe Attribute Value Notification.

This procedure unsubscribe to value notification using the Client Characteristic Configuration handle. Notification callback with NULL data will be called if subscription was removed by this call, until then the parameters cannot be reused.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `params`: Subscribe parameters.

void **`bt_gatt_cancel`**(**`struct`** bt_conn **conn*, void **params*)

Cancel GATT pending request.

Parameters

- `conn`: Connection object.
- `params`: Requested params address.

`struct bt_gatt_exchange_params`

#include <gatt.h> GATT Exchange MTU parameters.

Public Members

void (***func**)(**struct** bt_conn *conn, uint8_t err, **struct** *bt_gatt_exchange_params* *params)
Response callback

struct bt_gatt_discover_params
#include <gatt.h> GATT Discover Attributes parameters.

Public Members

struct *bt_uuid* ***uuid**
Discover UUID type

bt_gatt_discover_func_t **func**
Discover attribute callback

uint16_t **end_handle**
Discover end handle

uint8_t **type**
Discover type

union *bt_gatt_discover_params*.**__unnamed__**

Public Members

struct *bt_gatt_discover_params*.**[anonymous]**.**[anonymous] _included**

uint16_t **start_handle**
Discover start handle

struct *bt_gatt_discover_params*.*__unnamed__*.**_included**

Public Members

uint16_t **attr_handle**
Include service attribute declaration handle

uint16_t **start_handle**
Included service start handle

uint16_t **end_handle**
Included service end handle

struct bt_gatt_read_params
#include <gatt.h> GATT Read parameters.

Public Members

bt_gatt_read_func_t **func**
Read attribute callback.

size_t **handle_count**
If equals to 1 single.handle and single.offset are used. If >1 Read Multiple Characteristic Values is performed and handles are used. If equals to 0 by_uuid is used for Read Using Characteristic UUID.

union *bt_gatt_read_params*.**__unnamed__**

Public Members

struct *bt_gatt_read_params*.**[anonymous].[anonymous] single**

uint16_t ***handles**
Handles to read in Read Multiple Characteristic Values.

struct *bt_gatt_read_params*.**[anonymous].[anonymous] by_uuid**

struct *bt_gatt_read_params*.*__unnamed__*.**single**

Public Members

uint16_t **handle**
Attribute handle.

uint16_t **offset**
Attribute data offset.

struct *bt_gatt_read_params*.*__unnamed__*.**by_uuid**

Public Members

uint16_t **start_handle**
First requested handle number.

uint16_t **end_handle**
Last requested handle number.

struct *bt_uuid* ***uuid**
2 or 16 octet UUID.

struct bt_gatt_write_params
#include <gatt.h> GATT Write parameters.

Public Members

bt_gatt_write_func_t **func**
Response callback

uint16_t **handle**
Attribute handle

uint16_t **offset**
Attribute data offset

const void ***data**
: Data to be written

uint16_t **length**
: Length of the data

struct bt_gatt_subscribe_params
#include <gatt.h> GATT Subscribe parameters.

Public Functions

ATOMIC_DEFINE (flags, BT_GATT_SUBSCRIBE_NUM_FLAGS)
: Subscription flags

Public Members

bt_gatt_notify_func_t **notify**
: Notification value callback

bt_gatt_write_func_t **write**
: Subscribe CCC write request response callback

uint16_t **value_handle**
: Subscribe value handle

uint16_t **ccc_handle**
: Subscribe CCC handle

uint16_t **value**
: Subscribe value

1.5 Hands Free Profile (HFP)

1.5.1 API Reference

group **bt_hfp**
Hands Free AG Profile (HFP AG)

Hands Free Profile (HFP)

Defines

HFP_HF_DIGIT_ARRAY_SIZE

HFP_HF_MAX_OPERATOR_NAME_LEN

HFP_HF_CMD_OK

HFP_HF_CMD_ERROR

HFP_HF_CMD_CME_ERROR

HFP_HF_CMD_UNKNOWN_ERROR

Typedefs

typedef enum *_hf_volume_type_t* **hf_volume_type_t**
bt hfp ag volume type

typedef enum *_hfp_ag_call_status_t* **hfp_ag_call_status_t**
bt hf call status

typedef struct *_hfp_ag_get_config* **hfp_ag_get_config**
bt ag configure setting

typedef struct *_hfp_ag_cind_t* **hfp_ag_cind_t**
bt hf call status

typedef int (***bt_hfp_ag_discover_callback**) (**struct** bt_conn *conn, uint8_t channel)
hfp_ag discover callback function

Parameters

- `conn`: Pointer to bt_conn structure.
- `channel`: the server channel of hfp ag

typedef enum *_hf_volume_type_t* **hf_volume_type_t**
bt hfp ag volume type

typedef enum *_hf_multiparty_call_option_t* **hf_multiparty_call_option_t**
bt hfp ag volume type

typedef struct *_hf_waiting_call_state_t* **hf_waiting_call_state_t**

Enums

enum _hf_volume_type_t
bt hfp ag volume type

Values:

enumerator hf_volume_type_speaker

enumerator hf_volume_type_mic

enumerator hf_volume_type_speaker

enumerator hf_volume_type_mic

enum _hfp_ag_call_status_t
bt hf call status

Values:

enumerator hfp_ag_call_call_end

enumerator hfp_ag_call_call_active

enumerator hfp_ag_call_call_incoming

enumerator hfp_ag_call_call_outgoing

enum hfp_ag_call_setup_status_t
bt ag call setup status

Values:

enumerator HFP_AG_CALL_SETUP_STATUS_IDLE

enumerator HFP_AG_CALL_SETUP_STATUS_INCOMING

enumerator HFP_AG_CALL_SETUP_STATUS_OUTGOING_DIALING

enumerator HFP_AG_CALL_SETUP_STATUS_OUTGOING_ALERTING

enum bt_hfp_hf_at_cmd

Values:

enumerator BT_HFP_HF_ATA

enumerator BT_HFP_HF_AT_CHUP

enum _hf_volume_type_t

bt hfp ag volume type

Values:

enumerator hf_volume_type_speaker

enumerator hf_volume_type_mic

enumerator hf_volume_type_speaker

enumerator hf_volume_type_mic

enum _hf_multiparty_call_option_t

bt hfp ag volume type

Values:

enumerator hf_multiparty_call_option_one

enumerator hf_multiparty_call_option_two

enumerator hf_multiparty_call_option_three

enumerator hf_multiparty_call_option_four

enumerator hf_multiparty_call_option_five

Functions

int **bt_hfp_ag_init**(void)

BT HFP AG Initialize

This function called to initialize bt hfp ag

Return 0 in case of success or otherwise in case of error.

int **bt_hfp_ag_deinit**(void)

BT HFP AG Deinitialize

This function called to initialize bt hfp ag

Return 0 in case of success or otherwise in case of error.

int **bt_hfp_ag_connect**(**struct** bt_conn **conn*, *hfp_ag_get_config* **config*, **struct** *bt_hfp_ag_cb* **cb*, **struct** bt_hfp_ag ***phfp_ag*)

hfp ag Connect.

This function is to be called after the conn parameter is obtained by performing a GAP procedure. The API is to be used to establish hfp ag connection between devices. This function only establish RFCOM connection. After connection success, the callback that is registered by bt_hfp_ag_register_connect_callback is called.

Return 0 in case of success or otherwise in case of error.

Parameters

- `conn`: Pointer to bt_conn structure.
- `config`: bt hfp ag congigure
- `cb`: bt hfp ag congigure
- `phfp_ag`: Pointer to pointer of bt hfp ag Connection object

int **bt_hfp_ag_disconnect**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag DisConnect.

This function is to be called after the conn parameter is obtained by performing a GAP procedure. The API is to be used to establish hfp ag connection between devices. This function only establish RFCOM connection. After connection success, the callback that is registered by bt_hfp_ag_register_connect_callback is called.

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **bt_hfp_ag_discover**(**struct** bt_conn **conn*, *bt_hfp_ag_discover_callback* *discoverCallback*)

hfp ag discover

This function is to be called after the conn parameter is obtained by performing a GAP procedure. The API is to be used to establish hfp ag connection between devices.

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `discoverCallback`: pointer to discover callback function,defined in application

void **bt_hfp_ag_open_audio**(**struct** bt_hfp_ag **hfp_ag*, uint8_t *codec*)

hfp ag open audio for codec

This function is to open audio codec for hfp funciton

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

void **bt_hfp_ag_close_audio**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag close audio for codec

This function is to close audio codec for hfp funciton

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **`bt_hfp_ag_register_supp_features`**(**`struct`** bt_hfp_ag **hfp_ag*, uint32_t *supported_features*)

configure hfp ag supported features.

if the function is not called, will use default supported featureshfp ag to configure hfp ag supported features

This function is to be configure hfp ag supported features. If the function is not called, will use default supported features

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `supported_features`: suppported features of hfp ag

uint32_t **`bt_hfp_ag_get_peer_supp_features`**(**`struct`** bt_hfp_ag **hfp_ag*)

hfp ag to get peer hfp hp support feautes

This function is to be called to get hfp hp support feautes

Return the supported feature of hfp ag

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **`bt_hfp_ag_register_cind_features`**(**`struct`** bt_hfp_ag **hfp_ag*, char **cind*)

hfp ag to configure hfp ag supported features

This function is to be configure hfp ag cind setting supported features. If the function is not called, will use default supported features

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `cind`: pointer to hfp ag cwind

int **`bt_hfp_ag_send_disable_voice_recognition`**(**`struct`** bt_hfp_ag **hfp_ag*)

hfp ag to disable voice recognition

This function is o disabl voice recognition

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object int **bt_hfp_ag_send_enable_voice_recognition**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag to enable voice recognition

This function is used to enable voice recognition

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag Connection object

int **bt_hfp_ag_send_disable_voice_ecnr**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag to disable noise reduction and echo canceling

This function is o noise reduction and echo canceling

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **bt_hfp_ag_send_enable_voice_ecnr**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag to enable noise reduction and echo canceling

This function is to enable noise reduction and echo canceling

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **bt_hfp_ag_set_cops**(**struct** bt_hfp_ag **hfp_ag*, char **name*)

hfp ag to set the name of the currently selected Network operator by AG

This function is to set the name of the currently selected Network operator by AG

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `name`: the name of the currently selected Network operator by AG

int **bt_hfp_ag_set_volume_control**(**struct** bt_hfp_ag **hfp_ag*, *hf_volume_type_t* *type*, int *value*)

hfp ag to set volue of hfp hp

This function is to set volue of hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `type`: the hfp hp volume type

- `value`: the volue of volume

int **`bt_hfp_ag_set_inband_ring_tone`**(**`struct`** bt_hfp_ag **hfp_ag*, int *value*)
hfp ag to set inband ring tone support

This function is to set inband ring tone support

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: the inband ring tone type

int **`bt_hfp_ag_set_phnum_tag`**(**`struct`** bt_hfp_ag **hfp_ag*, char **name*)
hfp ag to set the attach a phone number to a voice Tag

This function is to set the attach a phone number to a voice Tag

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `name`: the name of attach a phone number to a voice Tag

void **`bt_hfp_ag_call_status_pl`**(**`struct`** bt_hfp_ag **hfp_ag*, *hfp_ag_call_status_t status*)
hfp ag to set the call status

This function is to set the call status

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `status`: the ag call status

int **`bt_hfp_ag_handle_btrh`**(**`struct`** bt_hfp_ag **hfp_ag*, uint8_t *option*)
hfp ag to set the status of the "Response and Hold" state of the AG.

This function is to hfp ag to set the status of the "Response and Hold" state of the AG.

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `option`: the hfp ag "Response and Hold" state of the AG

int **`bt_hfp_ag_handle_indicator_enable`**(**`struct`** bt_hfp_ag **hfp_ag*, uint8_t *index*, uint8_t *enable*)
hfp ag to set the status of the "Response and Hold" state of the AG.

This function is to hfp ag to set the status of the "Response and Hold" state of the AG.

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `item`: 1 for Enhanced Safety, 2 for Battery Level
- `enable`: 1 for enable

void **`bt_hfp_ag_send_callring`**(**struct** bt_hfp_ag **hfp_ag*)

hfp ag to set ring command to hfp hp

This function is hfp ag to set ring command to hfp hp

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object

int **`bt_hfp_ag_send_call_indicator`**(**struct** bt_hfp_ag **hfp_ag*, uint8_t *value*)

hfp ag set call indicator to hfp hp

This function is hfp ag set call indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of call indicator

int **`bt_hfp_ag_send_callsetup_indicator`**(**struct** bt_hfp_ag **hfp_ag*, uint8_t *value*)

hfp ag set call setup indicator to hfp hp

This function is hfp ag set call setup indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of call setup indicator

int **`bt_hfp_ag_send_service_indicator`**(**struct** bt_hfp_ag **hfp_ag*, uint8_t *value*)

hfp ag set service indicator to hfp hp

This function is hfp ag set service indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of service indicator

int **`bt_hfp_ag_send_signal_indicator`**(**struct** bt_hfp_ag **hfp_ag*, uint8_t *value*)

hfp ag set signal strength indicator to hfp hp

This function is hfp ag set signal strength indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of signal strength indicator

int **`bt_hfp_ag_send_roaming_indicator`**(**`struct`** bt_hfp_ag **hfp_ag*, uint8_t *value*)
hfp ag set roaming indicator to hfp hp

This function is hfp ag set roaming indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of roaming indicator

int **`bt_hfp_ag_send_battery_indicator`**(**`struct`** bt_hfp_ag **hfp_ag*, uint8_t *value*)
hfp ag set battery level indicator to hfp hp

This function is hfp ag set battery level indicator to hfp hp

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of battery level indicator

int **`bt_hfp_ag_send_ccwa_indicator`**(**`struct`** bt_hfp_ag **hfp_ag*, char **number*)
hfp ag set ccwa indicator to hfp hp

This function is hfp ag set ccwa indicator to hfp hp for mutiple call

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of battery level indicator

int **`bt_hfp_ag_codec_selector`**(**`struct`** bt_hfp_ag **hfp_ag*, uint8_t *value*)
hfp ag set codec selector to hfp hp

This function is hfp ag set odec selector to hfp hp for codec negotiation

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `value`: value of codec selector

int **bt_hfp_ag_unknown_at_response**(**struct** bt_hfp_ag **hfp_ag*, uint8_t **unknow_at_rsp*, uint16_t *unknow_at_rsplen*)

hfp ag set unknown at command response to hfp fp

This function is hfp ag set unknown at command response to hfp fp, the command is not supported on hfp ag profile, Need handle the unknown command on application

Return 0 in case of success or otherwise in case of error.

Parameters

- `phfp_ag`: pointer to bt hfp ag connection object
- `unknow_at_rsp`: string of unkown at command response
- `unknow_at_rsplen`: string length of unkown at command response

int **bt_hfp_hf_register**(**struct** *bt_hfp_hf_cb* **cb*)

Register HFP HF profile.

Register Handsfree profile callbacks to monitor the state and get the required HFP details to display.

Return 0 in case of success or negative value in case of error.

Parameters

- `cb`: callback structure.

int **bt_hfp_hf_send_cmd**(**struct** bt_conn **conn*, **enum** *bt_hfp_hf_at_cmd* *cmd*)

Handsfree client Send AT.

Send specific AT commands to handsfree client profile.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **bt_hfp_hf_start_voice_recognition**(**struct** bt_conn **conn*)

Handsfree to enable voice recognition in the AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **bt_hfp_hf_stop_voice_recognition**(**struct** bt_conn **conn*)

Handsfree to Disable voice recognition in the AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **bt_hfp_hf_volume_update**(**struct** bt_conn **conn*, *hf_volume_type_t* *type*, int *volume*)
Handsfree to update Volume with AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `type`: volume control target, speaker or microphone
- `volume`: gain of the speaker of microphone, ranges 0 to 15

int **bt_hfp_hf_dial**(**struct** bt_conn **conn*, **const** char **number*)
Place a call with a specified number, if number is NULL, last called number is called. As a precondition to use this API, Service Level Connection shall exist with AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `number`: number string of the call. If NULL, the last number is called(aka re-dial)

int **bt_hfp_hf_dial_memory**(**struct** bt_conn **conn*, int *location*)
Place a call with number specified by location(speed dial). As a precondition, to use this API, Service Level Connection shall exist with AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `location`: location of the number in the memory

int **bt_hfp_hf_last_dial**(**struct** bt_conn **conn*)
Place a call with number specified by location(speed dial). As a precondition, to use this API, Service Level connection shall exist with AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **bt_hfp_hf_multiparty_call_option**(**struct** bt_conn **conn*, *hf_multiparty_call_option_t* *option*)
Place a call with number specified by location(speed dial). As a precondition, to use this API, Service Level Connection shall exist with AG.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **`bt_hfp_hf_enable_clip_notification`**(**`struct`** bt_conn **conn*)
Enable the CLIP notification.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **`bt_hfp_hf_disable_clip_notification`**(**`struct`** bt_conn **conn*)
Disable the CLIP notification.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **`bt_hfp_hf_enable_call_waiting_notification`**(**`struct`** bt_conn **conn*)
Enable the call waiting notification.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **`bt_hfp_hf_disable_call_waiting_notification`**(**`struct`** bt_conn **conn*)
Disable the call waiting notification.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

int **`bt_hfp_hf_get_last_voice_tag_number`**(**`struct`** bt_conn **conn*)
Get the last voice tag nubmer, the mubmer will be fill callback event voicetag_phnum.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.

`struct _hfp_ag_get_config`
#include <hfp_ag.h> bt ag configure setting

`struct _hfp_ag_cind_t`
#include <hfp_ag.h> bt hf call status

`struct bt_hfp_ag_cb`
#include <hfp_ag.h> HFP profile application callback.

Public Members

void (***connected**)(**struct** bt_hfp_ag *hfp_ag)
AG connected callback to application

If this callback is provided it will be called whenever the connection completes.

Parameters
- `hfp_ag`: bt hfp ag Connection object.

void (***disconnected**)(**struct** bt_hfp_ag *hfp_ag)
AG disconnected callback to application

If this callback is provided it will be called whenever the connection gets disconnected, including when a connection gets rejected or cancelled or any error in SLC establisment.

Parameters
- `hfp_ag`: bt hfp ag Connection object.

void (***volume_control**)(**struct** bt_hfp_ag *hfp_ag, *hf_volume_type_t* type, int value)
AG volume_control Callback

This callback provides volume_control indicator value to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `type`: the hfp volue type, for speaker or mic.
- `value`: service indicator value received from the AG.

void (***hfu_brsf**)(**struct** bt_hfp_ag *hfp_ag, uint32_t value)
AG remote support feature Callback

This callback provides the remote hfp unit supported feature

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: call indicator he remote hfp unit supported feature received from the AG.

void (***ata_response**)(**struct** bt_hfp_ag *hfp_ag)
AG remote call is answered Callback

This callback provides call indicator the call is answered to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.

void (***chup_response**)(**struct** bt_hfp_ag *hfp_ag)
AG remote call is answered Callback

This callback provides call indicator the call is rejected to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.

void (***dial**)(**struct** bt_hfp_ag *hfp_ag, char *number)
AG remote call is answered Callback

This callback provides call indicator the call is rejected to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: call information.

void (***brva**)(**struct** bt_hfp_ag *hfp_ag, uint32_t value)
AG remote voice recognition activation Callback

This callback provides call indicator voice recognition activation of peer HF to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: voice recognition activation information.

void (***nrec**)(**struct** bt_hfp_ag *hfp_ag, uint32_t value)
AG remote noise reduction and echo canceling Callback

This callback provides call indicator voice recognition activation of peer HF to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: Noise Reduction and Echo Canceling information.

void (***codec_negotiate**)(**struct** bt_hfp_ag *hfp_ag, uint32_t value)
AG remote codec negotiate Callback

This callback provides codec negotiate information of peer HF to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: codec index of peer HF.

void (***chld**)(**struct** bt_hfp_ag *hfp_ag, uint8_t option, uint8_t index)
AG multiparty call status indicator Callback

This callback provides multiparty call status indicator Callback of peer HF to the application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `option`: Multiparty call option.
- `index`: Multiparty call index.

void (***unkown_at**)(**struct** bt_hfp_ag *hfp_ag, char *value, uint32_t length)
AG unkown at Callback

This callback provides AG unkown at value to the application, the unkown at command could be handled by application

Parameters
- `hfp_ag`: bt hfp ag Connection object.
- `value`: unknow AT string buffer
- `length`: unknow AT string length.

struct bt_hfp_hf_cmd_complete
#include <hfp_hf.h> HFP HF Command completion field.

struct _hf_waiting_call_state_t

#include <hfp_hf.h>

struct bt_hfp_hf_cb

#include <hfp_hf.h> HFP profile application callback.

Public Members

void (***connected**) (**struct** bt_conn *conn)

HF connected callback to application

If this callback is provided it will be called whenever the connection completes.

Parameters

- `conn`: Connection object.

void (***disconnected**) (**struct** bt_conn *conn)

HF disconnected callback to application

If this callback is provided it will be called whenever the connection gets disconnected, including when a connection gets rejected or cancelled or any error in SLC establisment.

Parameters

- `conn`: Connection object.

void (***service**) (**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides service indicator value to the application

Parameters

- `conn`: Connection object.
- `value`: service indicator value received from the AG.

void (***call**) (**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call indicator value to the application

Parameters

- `conn`: Connection object.
- `value`: call indicator value received from the AG.

void (***call_setup**) (**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call setup indicator value to the application

Parameters

- `conn`: Connection object.
- `value`: call setup indicator value received from the AG.

void (***call_held**) (**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call held indicator value to the application

Parameters
- `conn`: Connection object.
- `value`: call held indicator value received from the AG.

void (***signal**)(**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides signal indicator value to the application

Parameters
- `conn`: Connection object.
- `value`: signal indicator value received from the AG.

void (***roam**)(**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides roaming indicator value to the application

Parameters
- `conn`: Connection object.
- `value`: roaming indicator value received from the AG.

void (***battery**)(**struct** bt_conn *conn, uint32_t value)

HF indicator Callback

This callback battery service indicator value to the application

Parameters
- `conn`: Connection object.
- `value`: battery indicator value received from the AG.

void (***voicetag_phnum**)(**struct** bt_conn *conn, char *number)

HF voice tag phnum indicator Callback

This callback voice tag phnum indicator to the application

Parameters
- `conn`: Connection object.
- `voice`: tag phnum value received from the AG.

void (***call_phnum**)(**struct** bt_conn *conn, char *number)

HF calling phone number string indication callback to application

If this callback is provided it will be called whenever there is an incoming call and bt_hfp_hf_enable_clip_notification is called.

Parameters
- `conn`: Connection object.
- `char`: to phone number string.

void (***waiting_call**)(**struct** bt_conn *conn, *hf_waiting_call_state_t* *wcs)

HF waiting call indication callback to application

If this callback is provided it will be called in waiting call state

Parameters

- `conn`: Connection object.
- `pointer`: to waiting call state information.

void (***ring_indication**)(**struct** bt_conn *conn)

HF incoming call Ring indication callback to application

If this callback is provided it will be called whenever there is an incoming call.

Parameters
- `conn`: Connection object.

void (***cmd_complete_cb**)(**struct** bt_conn *conn, **struct** *bt_hfp_hf_cmd_complete* *cmd)

HF notify command completed callback to application

The command sent from the application is notified about its status

Parameters
- `conn`: Connection object.
- `cmd`: structure contains status of the command including cme.

1.6 Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP layer enables connection-oriented channels which can be enable with the configuration option: `CONFIG_BT_L2CAP_DYNAMIC_CHANNEL`. This channels support segmentation and reassembly transparently, they also support credit based flow control making it suitable for data streams.

Channels instances are represented by the *`bt_l2cap_chan`* struct which contains the callbacks in the *`bt_l2cap_chan_ops`* struct to inform when the channel has been connected, disconnected or when the encryption has changed. In addition to that it also contains the `recv` callback which is called whenever an incoming data has been received. Data received this way can be marked as processed by returning 0 or using *`bt_l2cap_chan_recv_complete()`* API if processing is asynchronous.

Note: The `recv` callback is called directly from RX Thread thus it is not recommended to block for long periods of time.

For sending data the *`bt_l2cap_chan_send()`* API can be used noting that it may block if no credits are available, and resuming as soon as more credits are available.

Servers can be registered using *`bt_l2cap_server_register()`* API passing the *`bt_l2cap_server`* struct which informs what `psm` it should listen to, the required security level `sec_level`, and the callback `accept` which is called to authorize incoming connection requests and allocate channel instances.

Client channels can be initiated with use of *`bt_l2cap_chan_connect()`* API and can be disconnected with the *`bt_l2cap_chan_disconnect()`* API. Note that the later can also disconnect channel instances created by servers.

1.6.1 API Reference

group **bt_l2cap**

L2CAP.

Defines

BT_L2CAP_HDR_SIZE

L2CAP header size, used for buffer size calculations

BT_L2CAP_BUF_SIZE(*mtu*)

Helper to calculate needed outgoing buffer size, useful e.g. for creating buffer pools.

Return Needed buffer size to match the requested L2CAP MTU.

Parameters

- `mtu`: Needed L2CAP MTU.

BT_L2CAP_LE_CHAN(*_ch*)

Helper macro getting container object of type *bt_l2cap_le_chan* address having the same container chan member address as object in question.

Return Address of in memory *bt_l2cap_le_chan* object type containing the address of in question object.

Parameters

- `_ch`: Address of object of *bt_l2cap_chan* type

BT_L2CAP_CFG_OPT_MTU

configuration parameter options type

BT_L2CAP_CFG_OPT_FUSH_TIMEOUT

BT_L2CAP_CFG_OPT_QOS

BT_L2CAP_CFG_OPT_RETRANS_FC

BT_L2CAP_CFG_OPT_FCS

BT_L2CAP_CFG_OPT_EXT_FLOW_SPEC

BT_L2CAP_CFG_OPT_EXT_WIN_SIZE

BT_L2CAP_MODE_BASIC

L2CAP Operation Modes

BT_L2CAP_MODE_RTM

BT_L2CAP_MODE_FC

BT_L2CAP_MODE_ERTM

BT_L2CAP_MODE_SM

BT_L2CAP_FEATURE_FC

L2CAP Extended Feature Mask values

BT_L2CAP_FEATURE_RTM

BT_L2CAP_FEATURE_QOS

BT_L2CAP_FEATURE_ERTM

BT_L2CAP_FEATURE_SM

BT_L2CAP_FEATURE_FCS

BT_L2CAP_FEATURE_EFS_BR_EDR

BT_L2CAP_FEATURE_FIXED_CHANNELS

BT_L2CAP_FEATURE_EXTENDED_WINDOW_SIZE

BT_L2CAP_FEATURE_UCD

BT_L2CAP_CHAN_SEND_RESERVE
Headroom needed for outgoing buffers.

Typedefs

typedef void (***bt_l2cap_chan_destroy_t**)(**struct** *bt_l2cap_chan* *chan)
Channel destroy callback.

Parameters

- `chan`: Channel object.

typedef enum *bt_l2cap_chan_state* **bt_l2cap_chan_state_t**

typedef enum *bt_l2cap_chan_status* **bt_l2cap_chan_status_t**

Enums

enum bt_l2cap_chan_state
Life-span states of L2CAP CoC channel.

Used only by internal APIs dealing with setting channel to proper state depending on operational context.

Values:

enumerator BT_L2CAP_DISCONNECTED
Channel disconnected

enumerator BT_L2CAP_CONNECT
Channel in connecting state

enumerator BT_L2CAP_CONFIG
Channel in config state, BR/EDR specific

enumerator BT_L2CAP_CONNECTED
Channel ready for upper layer traffic on it

enumerator BT_L2CAP_DISCONNECT
Channel in disconnecting state

enum bt_l2cap_chan_status
Status of L2CAP channel.

Values:

enumerator BT_L2CAP_STATUS_OUT
Channel output status

enumerator BT_L2CAP_STATUS_SHUTDOWN
Channel shutdown status.

Once this status is notified it means the channel will no longer be able to transmit or receive data.

enumerator BT_L2CAP_STATUS_ENCRYPT_PENDING
Channel encryption pending status.

enumerator BT_L2CAP_NUM_STATUS

Functions

int **bt_l2cap_server_register**(**struct** *bt_l2cap_server* **server*)
Register L2CAP server.

Register L2CAP server for a PSM, each new connection is authorized using the accept() callback which in case of success shall allocate the channel structure to be used by the new connection.

For fixed, SIG-assigned PSMs (in the range 0x0001-0x007f) the PSM should be assigned to server->psm before calling this API. For dynamic PSMs (in the range 0x0080-0x00ff) server->psm may be pre-set to a given value (this is however not recommended) or be left as 0, in which case upon return a newly allocated value will have been assigned to it. For dynamically allocated values the expectation is that it's exposed through a GATT service, and that's how L2CAP clients discover how to connect to the server.

Return 0 in case of success or negative value in case of error.

Parameters

- `server`: Server structure.

int **bt_l2cap_br_server_register**(**struct** *bt_l2cap_server* **server*)
Register L2CAP server on BR/EDR oriented connection.

Register L2CAP server for a PSM, each new connection is authorized using the accept() callback which in case of success shall allocate the channel structure to be used by the new connection.

Return 0 in case of success or negative value in case of error.

Parameters

- `server`: Server structure.

int **bt_l2cap_ecred_chan_connect**(**struct** bt_conn **conn*, **struct** *bt_l2cap_chan* ***chans*, uint16_t *psm*)
Connect Enhanced Credit Based L2CAP channels.

Connect up to 5 L2CAP channels by PSM, once the connection is completed each channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `chans`: Array of channel objects.
- `psm`: Channel PSM to connect to.

int **bt_l2cap_chan_connect**(**struct** bt_conn **conn*, **struct** *bt_l2cap_chan* **chan*, uint16_t *psm*)

Connect L2CAP channel.

Connect L2CAP channel by PSM, once the connection is completed channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead. Channel object passed (over an address of it) as second parameter shouldn't be instantiated in application as standalone. Instead of, application should create transport dedicated L2CAP objects, i.e. type of *bt_l2cap_le_chan* for LE and/or type of *bt_l2cap_br_chan* for BR/EDR. Then pass to this API the location (address) of *bt_l2cap_chan* type object which is a member of both transport dedicated objects.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `chan`: Channel object.
- `psm`: Channel PSM to connect to.

int **bt_l2cap_chan_disconnect**(**struct** *bt_l2cap_chan* **chan*)

Disconnect L2CAP channel.

Disconnect L2CAP channel, if the connection is pending it will be canceled and as a result the channel disconnected() callback is called. Regarding to input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

Return 0 in case of success or negative value in case of error.

Parameters

- `chan`: Channel object.

int **bt_l2cap_chan_send**(**struct** *bt_l2cap_chan* **chan*, **struct** net_buf **buf*)

Send data to L2CAP channel.

Send data from buffer to the channel. If credits are not available, buf will be queued and sent as and when credits are received from peer. Regarding to first input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

Return Bytes sent in case of success or negative value in case of error.

int **bt_l2cap_chan_recv_complete**(**struct** *bt_l2cap_chan* **chan*, **struct** net_buf **buf*)

Complete receiving L2CAP channel data.

Complete the reception of incoming data. This shall only be called if the channel recv callback has returned -EINPROGRESS to process some incoming data. The buffer shall contain the original user_data as that is used for storing the credits/segments used by the packet.

Return 0 in case of success or negative value in case of error.

Parameters

- `chan`: Channel object.
- `buf`: Buffer containing the data.

struct bt_l2cap_chan
#include <l2cap.h> L2CAP Channel structure.

Public Members

struct bt_conn ***conn**
Channel connection reference

struct *bt_l2cap_chan_ops* ***ops**
Channel operations reference

struct bt_l2cap_le_endpoint
#include <l2cap.h> LE L2CAP Endpoint structure.

Public Members

uint16_t **cid**
Endpoint Channel Identifier (CID)

uint16_t **mtu**
Endpoint Maximum Transmission Unit

uint16_t **mps**
Endpoint Maximum PDU payload Size

uint16_t **init_credits**
Endpoint initial credits

atomic_t **credits**
Endpoint credits

struct bt_l2cap_le_chan
#include <l2cap.h> LE L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* **chan**
Common L2CAP channel reference object

struct *bt_l2cap_le_endpoint* **rx**
Channel Receiving Endpoint

struct *bt_l2cap_le_endpoint* **tx**
Channel Transmission Endpoint

struct k_fifo **tx_queue**
Channel Transmission queue

struct net_buf ***tx_buf**
Channel Pending Transmission buffer

struct k_work **tx_work**
Channel Transmission work

struct net_buf ***_sdu**
Segment SDU packet from upper layer

struct bt_l2cap_br_endpoint
#include <l2cap.h> BREDR L2CAP Endpoint structure.

Public Members

uint16_t **cid**
Endpoint Channel Identifier (CID)

uint16_t **mtu**
Endpoint Maximum Transmission Unit

struct bt_l2cap_br_chan
#include <l2cap.h> BREDR L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* **chan**
Common L2CAP channel reference object

struct *bt_l2cap_br_endpoint* **rx**
Channel Receiving Endpoint

struct *bt_l2cap_br_endpoint* **tx**
Channel Transmission Endpoint

struct bt_l2cap_qos
#include <l2cap.h> QUALITY OF SERVICE (QOS) OPTION

struct bt_l2cap_retrans_fc
#include <l2cap.h> RETRANSMISSION AND FLOW CONTROL OPTION

struct bt_l2cap_ext_flow_spec
#include <l2cap.h> EXTENDED FLOW SPECIFICATION OPTION

struct bt_l2cap_cfg_options
#include <l2cap.h> L2CAP configuration parameter options.

struct bt_l2cap_chan_ops
#include <l2cap.h> L2CAP Channel operations structure.

Public Members

void (***connected**)(**struct** *bt_l2cap_chan* *chan)
Channel connected callback.

If this callback is provided it will be called whenever the connection completes.

Parameters
- `chan`: The channel that has been connected

void (***disconnected**)(**struct** *bt_l2cap_chan* *chan)
Channel disconnected callback.

If this callback is provided it will be called whenever the channel is disconnected, including when a connection gets rejected.

Parameters
- `chan`: The channel that has been Disconnected

void (***encrypt_change**)(**struct** *bt_l2cap_chan* *chan, uint8_t hci_status)

Channel encrypt_change callback.

If this callback is provided it will be called whenever the security level changed (indirectly link encryption done) or authentication procedure fails. In both cases security initiator and responder got the final status (HCI status) passed by related to encryption and authentication events from local host's controller.

Parameters
- `chan`: The channel which has made encryption status changed.
- `status`: HCI status of performed security procedure caused by channel security requirements. The value is populated by HCI layer and set to 0 when success and to non-zero (reference to HCI Error Codes) when security/authentication failed.

struct net_buf *(***alloc_buf**)(**struct** *bt_l2cap_chan* *chan)

Channel alloc_buf callback.

If this callback is provided the channel will use it to allocate buffers to store incoming data. Channels that requires segmentation must set this callback.

Return Allocated buffer.

Parameters
- `chan`: The channel requesting a buffer.

int (***recv**)(**struct** *bt_l2cap_chan* *chan, **struct** net_buf *buf)

Channel recv callback.

Return 0 in case of success or negative value in case of error.

-EINPROGRESS in case where user has to confirm once the data has been processed by calling *bt_l2cap_chan_recv_complete* passing back the buffer received with its original user_data which contains the number of segments/credits used by the packet.

Parameters
- `chan`: The channel receiving data.
- `buf`: Buffer containing incoming data.

void (***sent**)(**struct** *bt_l2cap_chan* *chan)

Channel sent callback.

If this callback is provided it will be called whenever a SDU has been completely sent.

Parameters
- `chan`: The channel which has sent data.

void (***status**)(**struct** *bt_l2cap_chan* *chan, atomic_t *status)

Channel status callback.

If this callback is provided it will be called whenever the channel status changes.

Parameters
- `chan`: The channel which status changed
- `status`: The channel status

struct bt_l2cap_server

#include <l2cap.h> L2CAP Server structure.

Public Members

uint16_t **psm**

Server PSM.

Possible values: 0 A dynamic value will be auto-allocated when *bt_l2cap_server_register()* is called.

0x0001-0x007f Standard, Bluetooth SIG-assigned fixed values.

0x0080-0x00ff Dynamically allocated. May be pre-set by the application before server registration (not recommended however), or auto-allocated by the stack if the app gave 0 as the value.

bt_security_t **sec_level**

Required minimim security level

int (***accept**)(**struct** bt_conn *conn, **struct** *bt_l2cap_chan* **chan)

Server accept callback.

This callback is called whenever a new incoming connection requires authorization.

Return 0 in case of success or negative value in case of error.

-ENOMEM if no available space for new channel.

-EACCES if application did not authorize the connection.

-EPERM if encryption key size is too short.

Parameters

- `conn`: The connection that is requesting authorization
- `chan`: Pointer to received the allocated channel

1.7 Serial Port Emulation (RFCOMM)

1.7.1 API Reference

group **bt_rfcomm**

RFCOMM.

Typedefs

typedef enum *bt_rfcomm_role* **bt_rfcomm_role_t**

Enums

enum [anonymous]

Values:

enumerator BT_RFCOMM_CHAN_HFP_HF

enumerator BT_RFCOMM_CHAN_HFP_AG

enumerator BT_RFCOMM_CHAN_HSP_AG

enumerator BT_RFCOMM_CHAN_HSP_HS

enumerator BT_RFCOMM_CHAN_SPP

enum bt_rfcomm_role

Role of RFCOMM session and dlc. Used only by internal APIs.

Values:

enumerator BT_RFCOMM_ROLE_ACCEPTOR

enumerator BT_RFCOMM_ROLE_INITIATOR

Functions

int **bt_rfcomm_server_register**(**struct** *bt_rfcomm_server* **server*)

Register RFCOMM server.

(defined(CONFIG_BT_RFCOMM_ENABLE_CONTROL_CMD) && (CONFIG_BT_RFCOMM_ENABLE_CONTROL_CMD > 0))Register RFCOMM server for a channel, each new connection is authorized using the accept() callback which in case of success shall allocate the dlc structure to be used by the new connection.

Return 0 in case of success or negative value in case of error.

Parameters

- `server`: Server structure.

int **bt_rfcomm_dlc_connect**(**struct** bt_conn **conn*, **struct** *bt_rfcomm_dlc* **dlc*, uint8_t *channel*)

Connect RFCOMM channel.

Connect RFCOMM dlc by channel, once the connection is completed dlc connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Connection object.
- `dlc`: Dlc object.
- `channel`: Server channel to connect to.

int **bt_rfcomm_dlc_send**(**struct** *bt_rfcomm_dlc* **dlc*, **struct** net_buf **buf*)

Send data to RFCOMM.

Send data from buffer to the dlc. Length should be less than or equal to mtu.

Return Bytes sent in case of success or negative value in case of error.

Parameters

- `dlc`: Dlc object.
- `buf`: Data buffer.

int **bt_rfcomm_dlc_disconnect**(**struct** *bt_rfcomm_dlc* **dlc*)

Disconnect RFCOMM dlc.

Disconnect RFCOMM dlc, if the connection is pending it will be canceled and as a result the dlc disconnected() callback is called.

Return 0 in case of success or negative value in case of error.

Parameters

- `dlc`: Dlc object.

struct net_buf ***bt_rfcomm_create_pdu**(**struct** net_buf_pool **pool*)

Allocate the buffer from pool after reserving head room for RFCOMM, L2CAP and ACL headers.

(defined(CONFIG_BT_RFCOMM_ENABLE_CONTROL_CMD) && (CONFIG_BT_RFCOMM_ENABLE_CONTROL_CMD > 0))

Return New buffer.

Parameters

- `pool`: Which pool to take the buffer from.

struct bt_rfcomm_dlc_ops

#include <rfcomm.h> RFCOMM DLC operations structure.

Public Members

void (***connected**)(**struct** *bt_rfcomm_dlc* *dlc)

DLC connected callback

If this callback is provided it will be called whenever the connection completes.

Parameters

- `dlc`: The dlc that has been connected

void (***disconnected**)(**struct** *bt_rfcomm_dlc* *dlc)

DLC disconnected callback

If this callback is provided it will be called whenever the dlc is disconnected, including when a connection gets rejected or cancelled (both incoming and outgoing)

Parameters

- `dlc`: The dlc that has been Disconnected

void (***recv**)(**struct** *bt_rfcomm_dlc* *dlc, **struct** net_buf *buf)

DLC recv callback

Parameters

- `dlc`: The dlc receiving data.
- `buf`: Buffer containing incoming data.

void (***sent**)(**struct** *bt_rfcomm_dlc* *dlc, **struct** net_buf *buf)

DLC sent callback

Parameters

- `dlc`: The dlc receiving data.
- `buf`: Buffer containing sending data.

struct bt_rfcomm_dlc

#include <rfcomm.h> RFCOMM DLC structure.

struct bt_rfcomm_server
#include <rfcomm.h>

Public Members

uint8_t **channel**
Server Channel

int (***accept**)(**struct** bt_conn *conn, **struct** *bt_rfcomm_dlc* **dlc)
Server accept callback

This callback is called whenever a new incoming connection requires authorization.

Return 0 in case of success or negative value in case of error.
Parameters
- `conn`: The connection that is requesting authorization
- `dlc`: Pointer to received the allocated dlc

1.8 Service Discovery Protocol (SDP)

1.8.1 API Reference

group **bt_sdp**
Service Discovery Protocol (SDP)

Defines

BT_SDP_SDP_SERVER_SVCLASS

BT_SDP_BROWSE_GRP_DESC_SVCLASS

BT_SDP_PUBLIC_BROWSE_GROUP

BT_SDP_SERIAL_PORT_SVCLASS

BT_SDP_LAN_ACCESS_SVCLASS

BT_SDP_DIALUP_NET_SVCLASS

BT_SDP_IRMC_SYNC_SVCLASS

BT_SDP_OBEX_OBJPUSH_SVCLASS

BT_SDP_OBEX_FILETRANS_SVCLASS

BT_SDP_IRMC_SYNC_CMD_SVCLASS

BT_SDP_HEADSET_SVCLASS

BT_SDP_CORDLESS_TELEPHONY_SVCLASS

BT_SDP_AUDIO_SOURCE_SVCLASS

BT_SDP_AUDIO_SINK_SVCLASS

BT_SDP_AV_REMOTE_TARGET_SVCLASS

BT_SDP_ADVANCED_AUDIO_SVCLASS

BT_SDP_AV_REMOTE_SVCLASS

BT_SDP_AV_REMOTE_CONTROLLER_SVCLASS

BT_SDP_INTERCOM_SVCLASS

BT_SDP_FAX_SVCLASS

BT_SDP_HEADSET_AGW_SVCLASS

BT_SDP_WAP_SVCLASS

BT_SDP_WAP_CLIENT_SVCLASS

BT_SDP_PANU_SVCLASS

BT_SDP_NAP_SVCLASS

BT_SDP_GN_SVCLASS

BT_SDP_DIRECT_PRINTING_SVCLASS

BT_SDP_REFERENCE_PRINTING_SVCLASS

BT_SDP_IMAGING_SVCLASS

BT_SDP_IMAGING_RESPONDER_SVCLASS

BT_SDP_IMAGING_ARCHIVE_SVCLASS

BT_SDP_IMAGING_REFOBJS_SVCLASS

BT_SDP_HANDSFREE_SVCLASS

BT_SDP_HANDSFREE_AGW_SVCLASS

BT_SDP_DIRECT_PRT_REFOBJS_SVCLASS

BT_SDP_REFLECTED_UI_SVCLASS

BT_SDP_BASIC_PRINTING_SVCLASS

BT_SDP_PRINTING_STATUS_SVCLASS

BT_SDP_HID_SVCLASS

BT_SDP_HCR_SVCLASS

BT_SDP_HCR_PRINT_SVCLASS

BT_SDP_HCR_SCAN_SVCLASS

BT_SDP_CIP_SVCLASS

BT_SDP_VIDEO_CONF_GW_SVCLASS

BT_SDP_UDI_MT_SVCLASS

BT_SDP_UDI_TA_SVCLASS

BT_SDP_AV_SVCLASS

BT_SDP_SAP_SVCLASS

BT_SDP_PBAP_PCE_SVCLASS

BT_SDP_PBAP_PSE_SVCLASS

BT_SDP_PBAP_SVCLASS

BT_SDP_MAP_MSE_SVCLASS

BT_SDP_MAP_MCE_SVCLASS

BT_SDP_MAP_SVCLASS

BT_SDP_GNSS_SVCLASS

BT_SDP_GNSS_SERVER_SVCLASS

BT_SDP_MPS_SC_SVCLASS

BT_SDP_MPS_SVCLASS

BT_SDP_PNP_INFO_SVCLASS

BT_SDP_GENERIC_NETWORKING_SVCLASS

BT_SDP_GENERIC_FILETRANS_SVCLASS

BT_SDP_GENERIC_AUDIO_SVCLASS

BT_SDP_GENERIC_TELEPHONY_SVCLASS

BT_SDP_UPNP_SVCLASS

BT_SDP_UPNP_IP_SVCLASS

BT_SDP_UPNP_PAN_SVCLASS

BT_SDP_UPNP_LAP_SVCLASS

BT_SDP_UPNP_L2CAP_SVCLASS

BT_SDP_VIDEO_SOURCE_SVCLASS

BT_SDP_VIDEO_SINK_SVCLASS

BT_SDP_VIDEO_DISTRIBUTION_SVCLASS

BT_SDP_HDP_SVCLASS

BT_SDP_HDP_SOURCE_SVCLASS

BT_SDP_HDP_SINK_SVCLASS

BT_SDP_GENERIC_ACCESS_SVCLASS

BT_SDP_GENERIC_ATTRIB_SVCLASS

BT_SDP_APPLE_AGENT_SVCLASS

BT_SDP_SERVER_RECORD_HANDLE

BT_SDP_ATTR_RECORD_HANDLE

BT_SDP_ATTR_SVCLASS_ID_LIST

BT_SDP_ATTR_RECORD_STATE

BT_SDP_ATTR_SERVICE_ID

BT_SDP_ATTR_PROTO_DESC_LIST

BT_SDP_ATTR_BROWSE_GRP_LIST

BT_SDP_ATTR_LANG_BASE_ATTR_ID_LIST

BT_SDP_ATTR_SVCINFO_TTL

BT_SDP_ATTR_SERVICE_AVAILABILITY

BT_SDP_ATTR_PROFILE_DESC_LIST

BT_SDP_ATTR_DOC_URL

BT_SDP_ATTR_CLNT_EXEC_URL

BT_SDP_ATTR_ICON_URL

BT_SDP_ATTR_ADD_PROTO_DESC_LIST

BT_SDP_ATTR_GROUP_ID

BT_SDP_ATTR_IP_SUBNET

BT_SDP_ATTR_VERSION_NUM_LIST

BT_SDP_ATTR_SUPPORTED_FEATURES_LIST

BT_SDP_ATTR_GOEP_L2CAP_PSM

BT_SDP_ATTR_SVCDB_STATE

BT_SDP_ATTR_MPSD_SCENARIOS

BT_SDP_ATTR_MPMD_SCENARIOS

BT_SDP_ATTR_MPS_DEPENDENCIES

BT_SDP_ATTR_SERVICE_VERSION

BT_SDP_ATTR_EXTERNAL_NETWORK

BT_SDP_ATTR_SUPPORTED_DATA_STORES_LIST

BT_SDP_ATTR_DATA_EXCHANGE_SPEC

BT_SDP_ATTR_NETWORK

BT_SDP_ATTR_FAX_CLASS1_SUPPORT

BT_SDP_ATTR_REMOTE_AUDIO_VOLUME_CONTROL

BT_SDP_ATTR_MCAP_SUPPORTED_PROCEDURES

BT_SDP_ATTR_FAX_CLASS20_SUPPORT

BT_SDP_ATTR_SUPPORTED_FORMATS_LIST

BT_SDP_ATTR_FAX_CLASS2_SUPPORT

BT_SDP_ATTR_AUDIO_FEEDBACK_SUPPORT

BT_SDP_ATTR_NETWORK_ADDRESS

BT_SDP_ATTR_WAP_GATEWAY

BT_SDP_ATTR_HOMEPAGE_URL

BT_SDP_ATTR_WAP_STACK_TYPE

BT_SDP_ATTR_SECURITY_DESC

BT_SDP_ATTR_NET_ACCESS_TYPE

BT_SDP_ATTR_MAX_NET_ACCESSRATE

BT_SDP_ATTR_IP4_SUBNET

BT_SDP_ATTR_IP6_SUBNET

BT_SDP_ATTR_SUPPORTED_CAPABILITIES

BT_SDP_ATTR_SUPPORTED_FEATURES

BT_SDP_ATTR_SUPPORTED_FUNCTIONS

BT_SDP_ATTR_TOTAL_IMAGING_DATA_CAPACITY

BT_SDP_ATTR_SUPPORTED_REPOSITORIES

BT_SDP_ATTR_MAS_INSTANCE_ID

BT_SDP_ATTR_SUPPORTED_MESSAGE_TYPES

BT_SDP_ATTR_PBAP_SUPPORTED_FEATURES

BT_SDP_ATTR_MAP_SUPPORTED_FEATURES

BT_SDP_ATTR_SPECIFICATION_ID

BT_SDP_ATTR_VENDOR_ID

BT_SDP_ATTR_PRODUCT_ID

BT_SDP_ATTR_VERSION

BT_SDP_ATTR_PRIMARY_RECORD

BT_SDP_ATTR_VENDOR_ID_SOURCE

BT_SDP_ATTR_HID_DEVICE_RELEASE_NUMBER

BT_SDP_ATTR_HID_PARSER_VERSION

BT_SDP_ATTR_HID_DEVICE_SUBCLASS

BT_SDP_ATTR_HID_COUNTRY_CODE

BT_SDP_ATTR_HID_VIRTUAL_CABLE

BT_SDP_ATTR_HID_RECONNECT_INITIATE

BT_SDP_ATTR_HID_DESCRIPTOR_LIST

BT_SDP_ATTR_HID_LANG_ID_BASE_LIST

BT_SDP_ATTR_HID_SDP_DISABLE

BT_SDP_ATTR_HID_BATTERY_POWER

BT_SDP_ATTR_HID_REMOTE_WAKEUP

BT_SDP_ATTR_HID_PROFILE_VERSION

BT_SDP_ATTR_HID_SUPERVISION_TIMEOUT

BT_SDP_ATTR_HID_NORMALLY_CONNECTABLE

BT_SDP_ATTR_HID_BOOT_DEVICE

BT_SDP_PRIMARY_LANG_BASE

BT_SDP_ATTR_SVCNAME_PRIMARY

BT_SDP_ATTR_SVCDESC_PRIMARY

BT_SDP_ATTR_PROVNAME_PRIMARY

BT_SDP_DATA_NIL

BT_SDP_UINT8

BT_SDP_UINT16

BT_SDP_UINT32

BT_SDP_UINT64

BT_SDP_UINT128

BT_SDP_INT8

BT_SDP_INT16

BT_SDP_INT32

BT_SDP_INT64

BT_SDP_INT128

BT_SDP_UUID_UNSPEC

BT_SDP_UUID16

BT_SDP_UUID32

BT_SDP_UUID128

BT_SDP_TEXT_STR_UNSPEC

BT_SDP_TEXT_STR8

BT_SDP_TEXT_STR16

BT_SDP_TEXT_STR32

BT_SDP_BOOL

BT_SDP_SEQ_UNSPEC

BT_SDP_SEQ8

BT_SDP_SEQ16

BT_SDP_SEQ32

BT_SDP_ALT_UNSPEC

BT_SDP_ALT8

BT_SDP_ALT16

BT_SDP_ALT32

BT_SDP_URL_STR_UNSPEC

BT_SDP_URL_STR8

BT_SDP_URL_STR16

BT_SDP_URL_STR32

BT_SDP_TYPE_DESC_MASK

BT_SDP_SIZE_DESC_MASK

BT_SDP_SIZE_INDEX_OFFSET

BT_SDP_ARRAY_8 (...)
Declare an array of 8-bit elements in an attribute.

BT_SDP_ARRAY_16 (...)
Declare an array of 16-bit elements in an attribute.

BT_SDP_ARRAY_32 (...)
Declare an array of 32-bit elements in an attribute.

BT_SDP_TYPE_SIZE(*_type*)
Declare a fixed-size data element header.

Parameters

- `_type`: Data element header containing type and size descriptors.

BT_SDP_TYPE_SIZE_VAR(*_type*, *_size*)
Declare a variable-size data element header.

Parameters

- `_type`: Data element header containing type and size descriptors.
- `_size`: The actual size of the data.

BT_SDP_DATA_ELEM_LIST(...)
Declare a list of data elements.

BT_SDP_NEW_SERVICE
SDP New Service Record Declaration Macro.

Helper macro to declare a new service record. Default attributes: Record Handle, Record State, Language Base, Root Browse Group

BT_SDP_LIST(*_att_id*, *_type_size*, *_data_elem_seq*)
Generic SDP List Attribute Declaration Macro.

Helper macro to declare a list attribute.

Parameters

- `_att_id`: List Attribute ID.
- `_data_elem_seq`: Data element sequence for the list.
- `_type_size`: SDP type and size descriptor.

BT_SDP_SERVICE_ID(*_uuid*)
SDP Service ID Attribute Declaration Macro.

Helper macro to declare a service ID attribute.

Parameters

- `_uuid`: Service ID 16bit UUID.

BT_SDP_SERVICE_NAME(*_name*)
SDP Name Attribute Declaration Macro.

Helper macro to declare a service name attribute.

Parameters

- `_name`: Service name as a string (up to 256 chars).

BT_SDP_SUPPORTED_FEATURES(*_features*)
SDP Supported Features Attribute Declaration Macro.

Helper macro to declare supported features of a profile/protocol.

Parameters

- `_features`: Feature mask as 16bit unsigned integer.

BT_SDP_RECORD(*_attrs*)
SDP Service Declaration Macro.

Helper macro to declare a service.

Parameters

- `_attrs`: List of attributes for the service record.

Typedefs

typedef uint8_t (***bt_sdp_discover_func_t**)(**struct** bt_conn *conn, **struct** *bt_sdp_client_result* *result)
Callback type reporting to user that there is a resolved result on remote for given UUID and the result record buffer can be used by user for further inspection.

A function of this type is given by the user to the *bt_sdp_discover_params* object. It'll be called on each valid record discovery completion for given UUID. When UUID resolution gives back no records then NULL is passed to the user. Otherwise user can get valid record(s) and then the internal hint 'next record' is set to false saying the UUID resolution is complete or the hint can be set by caller to true meaning that next record is available for given UUID. The returned function value allows the user to control retrieving follow-up resolved records if any. If the user doesn't want to read more resolved records for given UUID since current record data fulfills its requirements then should return BT_SDP_DISCOVER_UUID_STOP. Otherwise returned value means more subcall iterations are allowable.

Return BT_SDP_DISCOVER_UUID_STOP in case of no more need to read next record data and continue discovery for given UUID. By returning BT_SDP_DISCOVER_UUID_CONTINUE user allows this discovery continuation.

Parameters

- `conn`: Connection object identifying connection to queried remote.
- `result`: Object pointing to logical unparsed SDP record collected on base of response driven by given UUID.

Enums

enum [anonymous]

Helper enum to be used as return value of bt_sdp_discover_func_t. The value informs the caller to perform further pending actions or stop them.

Values:

enumerator BT_SDP_DISCOVER_UUID_STOP

enumerator BT_SDP_DISCOVER_UUID_CONTINUE

enum bt_sdp_proto

Protocols to be asked about specific parameters.

Values:

enumerator BT_SDP_PROTO_RFCOMM

enumerator BT_SDP_PROTO_L2CAP

Functions

int **bt_sdp_register_service**(**struct** *bt_sdp_record* **service*)

Register a Service Record.

Register a Service Record. Applications can make use of macros such as BT_SDP_DECLARE_SERVICE, BT_SDP_LIST, BT_SDP_SERVICE_ID, BT_SDP_SERVICE_NAME, etc. A service declaration must start with BT_SDP_NEW_SERVICE.

Return 0 in case of success or negative value in case of error.

Parameters

- `service`: Service record declared using BT_SDP_DECLARE_SERVICE.

int **bt_sdp_discover**(**struct** bt_conn **conn*, **const struct** *bt_sdp_discover_params* **params*)

Allows user to start SDP discovery session.

The function performs SDP service discovery on remote server driven by user delivered discovery parameters. Discovery session is made as soon as no SDP transaction is ongoing between peers and if any then this one is queued to be processed at discovery completion of previous one. On the service discovery completion the callback function will be called to get feedback to user about findings.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Object identifying connection to remote.
- `params`: SDP discovery parameters.

int **bt_sdp_discover_cancel**(**struct** bt_conn **conn*, **const struct** *bt_sdp_discover_params* **params*)

Release waiting SDP discovery request.

It can cancel valid waiting SDP client request identified by SDP discovery parameters object.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Object identifying connection to remote.
- `params`: SDP discovery parameters.

int **`bt_sdp_get_proto_param`**(**`const struct`** net_buf **buf*, **`enum`** *bt_sdp_proto proto*, uint16_t **param*)

Give to user parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Protocol Descriptor List attribute.

Return 0 on success when specific parameter associated with given protocol value is found, or negative if error occurred during processing.

Parameters

- `buf`: Original buffered raw record data.
- `proto`: Known protocol to be checked like RFCOMM or L2CAP.
- `param`: On success populated by found parameter value.

int **`bt_sdp_get_addl_proto_param`**(**`const struct`** net_buf **buf*, **`enum`** *bt_sdp_proto proto*, uint8_t *param_index*, uint16_t **param*)

Get additional parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Additional Protocol Descriptor List attribute.

Return 0 on success when a specific parameter associated with a given protocol value is found, or negative if error occurred during processing.

Parameters

- `buf`: Original buffered raw record data.
- `proto`: Known protocol to be checked like RFCOMM or L2CAP.
- `param_index`: There may be more than one parameter realted to the given protocol UUID. This function returns the result that is indexed by this parameter. It's value is from 0, 0 means the first matched result, 1 means the second matched result.
- `[out] param`: On success populated by found parameter value.

int **`bt_sdp_get_profile_version`**(**`const struct`** net_buf **buf*, uint16_t *profile*, uint16_t **version*)

Get profile version.

Helper API extracting remote profile version number. To get it proper generic profile parameter needs to be selected usually listed in SDP Interoperability Requirements section for given profile specification.

Return 0 on success, negative value if error occurred during processing.

Parameters

- `buf`: Original buffered raw record data.
- `profile`: Profile family identifier the profile belongs.

- `version`: On success populated by found version number.

int **`bt_sdp_get_features`**(**`const struct`** net_buf **buf*, uint16_t **features*)

Get SupportedFeatures attribute value.

Allows if exposed by remote retrieve SupportedFeature attribute.

Return 0 on success if feature found and valid, negative in case any error

Parameters

- `buf`: Buffer holding original raw record data from remote.
- `features`: On success object to be populated with SupportedFeature mask.

`struct bt_sdp_data_elem`

#include <sdp.h> SDP Generic Data Element Value.

`struct bt_sdp_attribute`

#include <sdp.h> SDP Attribute Value.

`struct bt_sdp_record`

#include <sdp.h> SDP Service Record Value.

`struct bt_sdp_client_result`

#include <sdp.h> Generic SDP Client Query Result data holder.

`struct bt_sdp_discover_params`

#include <sdp.h> Main user structure used in SDP discovery of remote.

Public Members

`struct` *bt_uuid* ***`uuid`**

UUID (service) to be discovered on remote SDP entity

bt_sdp_discover_func_t **`func`**

Discover callback to be called on resolved SDP record

`struct` net_buf_pool ***`pool`**

Memory buffer enabled by user for SDP query results

1.9 Advance Audio Distribution Profile (A2DP)

1.9.1 API Reference

group **`bt_a2dp`**

Advance Audio Distribution Profile (A2DP)

Defines

BT_A2DP_SBC_IE_LENGTH
SBC IE length

BT_A2DP_MPEG_1_2_IE_LENGTH
MPEG1,2 IE length

BT_A2DP_MPEG_2_4_IE_LENGTH
MPEG2,4 IE length

BT_A2DP_SOURCE_SBC_CODEC_BUFFER_SIZE

BT_A2DP_SOURCE_SBC_CODEC_BUFFER_NOCACHED_SIZE

BT_A2DP_SINK_SBC_CODEC_BUFFER_SIZE

BT_A2DP_SINK_SBC_CODEC_BUFFER_NOCACHED_SIZE

BT_A2DP_EP_CONTENT_PROTECTION_INIT

BT_A2DP_EP_RECOVERY_SERVICE_INIT

BT_A2DP_EP_REPORTING_SERVICE_INIT

BT_A2DP_EP_DELAY_REPORTING_INIT

BT_A2DP_EP_HEADER_COMPRESSION_INIT

BT_A2DP_EP_MULTIPLEXING_INIT

BT_A2DP_ENDPOINT_INIT(*_role*, *_codec*, *_capability*, *_config*, *_codec_buffer*, *_codec_buffer_nocahced*)
define the audio endpoint

Parameters

- `_role`: BT_A2DP_SOURCE or BT_A2DP_SINK.
- `_codec`: value of enum bt_a2dp_codec_id.
- `_capability`: the codec capability.
- `config`: the default config to configure the peer same codec type endpoint.
- `_codec_buffer`: the codec function used buffer.
- `_codec_buffer_nocahced`: the codec function used nocached buffer.

BT_A2DP_SINK_ENDPOINT_INIT(*_codec*, *_capability*, *_codec_buffer*, *_codec_buffer_nocahced*)
define the audio sink endpoint

Parameters

- `_codec`: value of enum bt_a2dp_codec_id.
- `_capability`: the codec capability.
- `_codec_buffer`: the codec function used buffer.
- `_codec_buffer_nocahced`: the codec function used nocached buffer.

BT_A2DP_SOURCE_ENDPOINT_INIT(*_codec*, *_capability*, *_config*, *_codec_buffer*, *_codec_buffer_nocahced*)

define the audio source endpoint

Parameters

- `_codec`: value of enum bt_a2dp_codec_id.
- `_capability`: the codec capability.
- `_config`: the default config to configure the peer same codec type endpoint.
- `_codec_buffer`: the codec function used buffer.
- `_codec_buffer_nocahced`: the codec function used nocached buffer.

BT_A2DP_SBC_SINK_ENDPOINT(*_name*)

define the default SBC sink endpoint that can be used as bt_a2dp_register_endpoint's parameter.

SBC is mandatory as a2dp specification, BT_A2DP_SBC_SINK_ENDPOINT is more convenient for user to register SBC endpoint.

Parameters

- `_name`: the endpoint variable name.

BT_A2DP_SBC_SOURCE_ENDPOINT(*_name*, *_config_freq*)

define the default SBC source endpoint that can be used as bt_a2dp_register_endpoint's parameter.

SBC is mandatory as a2dp specification, BT_A2DP_SBC_SOURCE_ENDPOINT is more convenient for user to register SBC endpoint.

Parameters

- `_name`: the endpoint variable name.
- `_config_freq`: the frequency to configure the peer same codec type endpoint.

Typedefs

typedef uint8_t (***bt_a2dp_discover_peer_endpoint_cb_t**)(**struct** bt_a2dp *a2dp, **struct** *bt_a2dp_endpoint* *endpoint, int err)

Get peer's endpoints callback.

Enums

enum bt_a2dp_codec_id

Codec ID.

Values:

enumerator BT_A2DP_SBC

Codec SBC

enumerator BT_A2DP_MPEG1

Codec MPEG-1

enumerator BT_A2DP_MPEG2
Codec MPEG-2

enumerator BT_A2DP_ATRAC
Codec ATRAC

enumerator BT_A2DP_VENDOR
Codec Non-A2DP

enum MEDIA_TYPE
Stream End Point Media Type.

Values:

enumerator BT_A2DP_AUDIO
Audio Media Type

enumerator BT_A2DP_VIDEO
Video Media Type

enumerator BT_A2DP_MULTIMEDIA
Multimedia Media Type

enum ROLE_TYPE
Stream End Point Role.

Values:

enumerator BT_A2DP_SOURCE
Source Role

enumerator BT_A2DP_SINK
Sink Role

enum [anonymous]
Helper enum to be used as return value of bt_a2dp_discover_peer_endpoint_cb_t. The value informs the caller to perform further pending actions or stop them.

Values:

enumerator BT_A2DP_DISCOVER_ENDPOINT_STOP

enumerator BT_A2DP_DISCOVER_ENDPOINT_CONTINUE

Functions

struct bt_a2dp ***bt_a2dp_connect**(**struct** bt_conn **conn*)
A2DP Connect.

This function is to be called after the conn parameter is obtained by performing a GAP procedure. The API is to be used to establish A2DP connection between devices. This funciton only establish AVDTP L2CAP connection. After connection success, the callback that is registered by bt_a2dp_register_connect_callback is called.

Return pointer to struct bt_a2dp in case of success or NULL in case of error.

Parameters

- `conn`: Pointer to bt_conn structure.

int **bt_a2dp_disconnect**(**struct** bt_a2dp **a2dp*)
disconnect l2cap a2dp

Return 0 in case of success and error code in case of error.

Parameters

- `a2dp`: The a2dp instance.

int **bt_a2dp_register_endpoint**(**struct** *bt_a2dp_endpoint* **endpoint*, uint8_t *media_type*, uint8_t *role*)
Endpoint Registration.

This function is used for registering the stream end points. The user has to take care of allocating the memory of the endpoint pointer and then pass the required arguments. Also, only one sep can be registered at a time. Multiple stream end points can be registered by calling multiple times. The endpoint registered first has a higher priority than the endpoint registered later. The priority is used in bt_a2dp_configure.

Return 0 in case of success and error code in case of error.

Parameters

- `endpoint`: Pointer to *bt_a2dp_endpoint* structure.
- `media_type`: Media type that the Endpoint is.
- `role`: Role of Endpoint.

int **bt_a2dp_register_connect_callback**(**struct** *bt_a2dp_connect_cb* **cb*)
register connecting callback.

The cb is called when bt_a2dp_connect is called or it is connected by peer device.

Return 0 in case of success and error code in case of error.

Parameters

- `cb`: The callback function.

int **bt_a2dp_configure**(**struct** bt_a2dp **a2dp*, void (**result_cb*)) int err
configure control callback.

This function will get peer's all endpoints and select one endpoint based on the priority of registered endpoints, then configure the endpoint based on the "config" of endpoint. Note: (1) priority is described in bt_a2dp_register_endpoint; (2) "config" is the config field of struct *bt_a2dp_endpoint* that is registered by bt_a2dp_register_endpoint.

Return 0 in case of success and error code in case of error.

Parameters

- `a2dp`: The a2dp instance.
- `result_cb`: The result callback function.

int **bt_a2dp_discover_peer_endpoints**(**struct** bt_a2dp **a2dp*, *bt_a2dp_discover_peer_endpoint_cb_t* *cb*)
get peer's endpoints.

bt_a2dp_configure can be called to configure a2dp. bt_a2dp_discover_peer_endpoints and bt_a2dp_configure_endpoint can be used too. In bt_a2dp_configure, the endpoint is selected automatically based on the prioriy. If bt_a2dp_configure fails, it means the default config of endpoint is not reasonal. bt_a2dp_discover_peer_endpoints and bt_a2dp_configure_endpoint can be used. bt_a2dp_discover_peer_endpoints is used to get peer endpoints. the peer endpoint is returned in the cb. then endpoint can be selected and configufed by bt_a2dp_configure_endpoint. If user stops to discover more peer endpoins, return BT_A2DP_DISCOVER_ENDPOINT_STOP in the cb; if user wants to discover more peer endpoints, return BT_A2DP_DISCOVER_ENDPOINT_CONTINUE in the cb.

Return 0 in case of success and error code in case of error.

Parameters

- `a2dp`: The a2dp instance.
- `cb`: notify the result.

int **`bt_a2dp_configure_endpoint`**(**`struct`** bt_a2dp **a2dp*, **`struct`** *bt_a2dp_endpoint* **endpoint*, **`struct`** *bt_a2dp_endpoint* **peer_endpoint*, **`struct`** *bt_a2dp_endpoint_config* **config*)

configure endpoint.

If the bt_a2dp_configure is failed or user want to change configured endpoint, user can call bt_a2dp_discover_peer_endpoints and this function to configure the selected endpoint.

Return 0 in case of success and error code in case of error.

Parameters

- `a2dp`: The a2dp instance.
- `endpoint`: The configured endpoint that is registered.
- `config`: The config to configure the endpoint.

int **`bt_a2dp_deconfigure`**(**`struct`** *bt_a2dp_endpoint* **endpoint*)

revert the configuration, then it can be configured again.

Release the endpoint based on the endpoint's state. After this, the endpoint can be re-configured again.

Return 0 in case of success and error code in case of error.

Parameters

- `endpoint`: the registered endpoint.

int **`bt_a2dp_start`**(**`struct`** *bt_a2dp_endpoint* **endpoint*)

start a2dp streamer, it is source only.

Return 0 in case of success and error code in case of error.

Parameters

- `endpoint`: The endpoint.

int **`bt_a2dp_stop`**(**`struct`** *bt_a2dp_endpoint* **endpoint*)

stop a2dp streamer, it is source only.

Return 0 in case of success and error code in case of error.

Parameters

- `endpoint`: The registered endpoint.

int **`bt_a2dp_reconfigure`**(**`struct`** *bt_a2dp_endpoint* **endpoint*, **`struct`** *bt_a2dp_endpoint_config* **config*)

re-configure a2dp streamer

This function send the AVDTP_RECONFIGURE command

Return 0 in case of success and error code in case of error.

Parameters

- `a2dp`: The a2dp instance.
- `endpoint`: the endpoint.
- `config`: The config to configure the endpoint.

`struct bt_a2dp_codec_ie`

#include <a2dp.h> codec information elements for the endpoint

Public Members

uint8_t **`len`**

Length of capabilities

uint8_t **`codec_ie`**[0]

codec information element

`struct bt_a2dp_endpoint_config`

#include <a2dp.h> The endpoint configuration.

Public Members

`struct` *bt_a2dp_codec_ie* ***`media_config`**

The media configuration content

`struct bt_a2dp_endpoint_configure_result`

#include <a2dp.h> The configuration result.

Public Members

int **`err`**

0 - success; other values - fail code

`struct` bt_a2dp ***`a2dp`**

which a2dp connection the endpoint is configured

`struct` bt_conn ***`conn`**

which conn the endpoint is configured

`struct` *bt_a2dp_endpoint_config* **`config`**

The configuration content

struct bt_a2dp_control_cb

#include <a2dp.h> The callback that is controlled by peer.

Public Members

void (***configured**) (**struct** *bt_a2dp_endpoint_configure_result* *config)

a2dp is configured by peer.

Parameters
- `err`: a2dp configuration result.

void (***deconfigured**) (int err)

a2dp is de-configured by peer.

Parameters
- `err`: a2dp configuration result.

void (***start_play**) (int err)

The result of starting media streamer.

void (***stop_play**) (int err)

the result of stopping media streaming.

void (***sink_streamer_data**) (uint8_t *data, uint32_t length)

the media streaming data, only for sink.

Parameters
- `data`: the data buffer pointer.
- `length`: the data length.

struct bt_a2dp_connect_cb

#include <a2dp.h> The connecting callback.

Public Members

void (***connected**) (**struct** bt_a2dp *a2dp, int err)

A a2dp connection has been established.

This callback notifies the application of a a2dp connection. It means the AVDTP L2CAP connection. In case the err parameter is non-zero it means that the connection establishment failed.

Parameters
- `a2dp`: a2dp connection object.
- `err`: error code.

void (***disconnected**) (**struct** bt_a2dp *a2dp)

A a2dp connection has been disconnected.

This callback notifies the application that a a2dp connection has been disconnected.

Parameters
- `a2dp`: a2dp connection object.

struct bt_a2dp_endpoint

#include <a2dp.h> Stream End Point.

Public Members

uint8_t **`codec_id`**
Code ID

`struct` bt_avdtp_seid_lsep **`info`**
Stream End Point Information

`struct` *bt_a2dp_codec_ie* ***`config`**
Pointer to codec default config

`struct` *bt_a2dp_codec_ie* ***`capabilities`**
Capabilities

`struct` *bt_a2dp_control_cb* **`control_cbs`**
endpoint control callbacks

uint8_t ***`codec_buffer`**
reserved codec related buffer (can be cacaheable ram)

uint8_t ***`codec_buffer_nocached`**
reserved codec related buffer (nocached)

1.10 Serial Port Profile (SPP)

1.10.1 API Reference

group **`bt_spp`**
Serial Port Profile (SPP)

Typedefs

`typedef enum` *bt_spp_role* **`bt_spp_role_t`**
SPP Role Value.

`typedef struct` *_bt_spp_callback* **`bt_spp_callback`**
spp application callback function

(defined(CONFIG_BT_SPP_ENABLE_CONTROL_CMD) && (CONFIG_BT_SPP_ENABLE_CONTROL_CMD > 0))

`typedef` int (***`bt_spp_discover_callback`**)(**`struct`** bt_conn *conn, uint8_t count, uint16_t *channel)
spp sdp discover callback function

Enums

`enum bt_spp_role`
SPP Role Value.

Values:

`enumerator BT_SPP_ROLE_SERVER`

`enumerator BT_SPP_ROLE_CLIENT`

Functions

int **`bt_spp_server_register`**(uint8_t *channel*, *bt_spp_callback* **cb*)

Register a SPP server.

Register a SPP server channel, wait for spp connection from SPP client. Once it's connected by spp client, will notify application by calling cb->connected.

Return 0 in case of success or negative value in case of error.

Parameters

- `channel`: Registered server channel.
- `cb`: Application callback.

int **`bt_spp_discover`**(**`struct`** bt_conn **conn*, *discover_cb_t* **cb*)

Discover SPP server channel.

Discover peer SPP server channel after basic BR connection is created. Will notify application discover results by calling cb->cb.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: BR connection handle.
- `cb`: Discover callback.

int **`bt_spp_client_connect`**(**`struct`** bt_conn **conn*, uint8_t *channel*, *bt_spp_callback* **cb*, **`struct`** bt_spp ***spp*)

Connect SPP server channel.

Create SPP connection with remote SPP server channel. Once connection is created successfully, will notify application by calling cb->connected.

Return 0 in case of success or negative value in case of error.

Parameters

- `conn`: Conn handle created with remote device.
- `channel`: Remote server channel to be connected, if it's 0, will connect remote BT_RFCOMM_CHAN_SPP channel.
- `cb`: Application callback.
- `spp`: SPP handle.

int **`bt_spp_data_send`**(**`struct`** bt_spp **spp*, uint8_t **data*, uint16_t *len*)

Send data to peer SPP device.

Send data to connected peer spp. Once data is sent, will notify application by calling cb->data_sent, which is provided by bt_spp_server_register or bt_spp_client_connect. If peer spp receives data, will notify application by calling cb->data_received.

Return 0 in case of success or negative value in case of error.

Parameters

- `spp`: SPP handle.
- `data`: Data buffer.
- `len`: Data length.

int **`bt_spp_disconnect`**(**`struct`** bt_spp **spp*)

Disconnect SPP connection.

Disconnect SPP connection. Once connection is disconnected, will notify application by calling cb->disconnected, which is provided by bt_spp_server_register or bt_spp_client_connect.

Return 0 in case of success or negative value in case of error.

Parameters

- `spp`: SPP handle.

int **`bt_spp_get_channel`**(**`struct`** bt_spp **spp*, uint8_t **channel*)

Get channel of SPP handle.

Return 0 in case of success or negative value in case of error.

Parameters

- `spp`: SPP handle.
- `channel`: Pointer to channel of spp handle.

int **`bt_spp_get_role`**(**`struct`** bt_spp **spp*, *bt_spp_role_t* **role*)

Get role of SPP handle.

Return 0 in case of success or negative value in case of error.

Parameters

- `spp`: SPP handle.
- `role`: Pointer to role of spp handle.

int **`bt_spp_get_conn`**(**`struct`** bt_spp **spp*, **`struct`** bt_conn ***conn*)

Get conn handle of SPP handle.

Return 0 in case of success or negative value in case of error.

Parameters

- `spp`: SPP handle.
- `conn`: Pointer to conn handle of spp handle.

`struct _bt_spp_callback`

#include <spp.h> spp application callback function

(defined(CONFIG_BT_SPP_ENABLE_CONTROL_CMD) && (CONFIG_BT_SPP_ENABLE_CONTROL_CMD > 0))

`struct discover_cb_t`

#include <spp.h> bt_spp_discover callback parameter

1.11 Universal Unique Identifiers (UUIDs)

1.11.1 API Reference

group **bt_uuid**

UUIDs.

Defines

BT_UUID_SIZE_16

Size in octets of a 16-bit UUID

BT_UUID_SIZE_32

Size in octets of a 32-bit UUID

BT_UUID_SIZE_128

Size in octets of a 128-bit UUID

BT_UUID_INIT_16(*value*)

Initialize a 16-bit UUID.

Parameters

- `value`: 16-bit UUID value in host endianness.

BT_UUID_INIT_32(*value*)

Initialize a 32-bit UUID.

Parameters

- `value`: 32-bit UUID value in host endianness.

BT_UUID_INIT_128(*value...*)

Initialize a 128-bit UUID.

Parameters

- `value`: 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to initialize a UUID from the readable form of UUIDs.

BT_UUID_DECLARE_16(*value*)

Helper to declare a 16-bit UUID inline.

Return Pointer to a generic UUID.

Parameters

- `value`: 16-bit UUID value in host endianness.

BT_UUID_DECLARE_32(*value*)

Helper to declare a 32-bit UUID inline.

Return Pointer to a generic UUID.

Parameters

- `value`: 32-bit UUID value in host endianness.

`BT_UUID_DECLARE_128`(*value...*)
Helper to declare a 128-bit UUID inline.

Return Pointer to a generic UUID.

Parameters

- `value`: 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to declare a UUID from the readable form of UUIDs.

`BT_UUID_16`(*__u*)
Helper macro to access the 16-bit UUID from a generic UUID.

`BT_UUID_32`(*__u*)
Helper macro to access the 32-bit UUID from a generic UUID.

`BT_UUID_128`(*__u*)
Helper macro to access the 128-bit UUID from a generic UUID.

`BT_UUID_128_ENCODE`(*w32*, *w1*, *w2*, *w3*, *w48*)
Encode 128 bit UUID into array values in little-endian format.

Helper macro to initialize a 128-bit UUID array value from the readable form of UUIDs, or encode 128-bit UUID values into advertising data Can be combined with BT_UUID_DECLARE_128 to declare a 128-bit UUID.

Example of how to declare the UUID `6E400001-B5A3-F393-E0A9-E50E24DCCA9E`

```
*  BT_UUID_DECLARE_128(
*       BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9,␣
↪0xE50E24DCCA9E))
*
```

Example of how to encode the UUID `6E400001-B5A3-F393-E0A9-E50E24DCCA9E` into advertising data.

```
*  BT_DATA_BYTES(BT_DATA_UUID128_ALL,
*       BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9,␣
↪0xE50E24DCCA9E))
*
```

Just replace the hyphen by the comma and add `0x` prefixes.

Return The comma separated values for UUID 128 initializer that may be used directly as an argument for *BT_UUID_INIT_128* or *BT_UUID_DECLARE_128*

Parameters

- `w32`: First part of the UUID (32 bits)
- `w1`: Second part of the UUID (16 bits)
- `w2`: Third part of the UUID (16 bits)

- `w3`: Fourth part of the UUID (16 bits)
- `w48`: Fifth part of the UUID (48 bits)

BT_UUID_16_ENCODE(*w16*)

Encode 16-bit UUID into array values in little-endian format.

Helper macro to encode 16-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a` into advertising data.

```
*  BT_DATA_BYTES(BT_DATA_UUID16_ALL, BT_UUID_16_ENCODE(0x180a))
*
```

Return The comma separated values for UUID 16 value that may be used directly as an argument for *BT_DATA_BYTES*.

Parameters

- `w16`: UUID value (16-bits)

BT_UUID_32_ENCODE(*w32*)

Encode 32-bit UUID into array values in little-endian format.

Helper macro to encode 32-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a01af` into advertising data.

```
*  BT_DATA_BYTES(BT_DATA_UUID32_ALL, BT_UUID_32_ENCODE(0x180a01af))
*
```

Return The comma separated values for UUID 32 value that may be used directly as an argument for *BT_DATA_BYTES*.

Parameters

- `w32`: UUID value (32-bits)

BT_UUID_STR_LEN

Recommended length of user string buffer for Bluetooth UUID.

The recommended length guarantee the output of UUID conversion will not lose valuable information about the UUID being processed. If the length of the UUID is known the string can be shorter.

BT_UUID_GAP_VAL

Generic Access UUID value.

BT_UUID_GAP

Generic Access.

BT_UUID_GATT_VAL

Generic attribute UUID value.

BT_UUID_GATT

Generic Attribute.

BT_UUID_IAS_VAL

Immediate Alert Service UUID value.

BT_UUID_IAS
Immediate Alert Service.

BT_UUID_LLS_VAL
Link Loss Service UUID value.

BT_UUID_LLS
Link Loss Service.

BT_UUID_TPS_VAL
Tx Power Service UUID value.

BT_UUID_TPS
Tx Power Service.

BT_UUID_CTS_VAL
Current Time Service UUID value.

BT_UUID_CTS
Current Time Service.

BT_UUID_HTS_VAL
Health Thermometer Service UUID value.

BT_UUID_HTS
Health Thermometer Service.

BT_UUID_DIS_VAL
Device Information Service UUID value.

BT_UUID_DIS
Device Information Service.

BT_UUID_HRS_VAL
Heart Rate Service UUID value.

BT_UUID_HRS
Heart Rate Service.

BT_UUID_BAS_VAL
Battery Service UUID value.

BT_UUID_BAS
Battery Service.

BT_UUID_HIDS_VAL
HID Service UUID value.

BT_UUID_HIDS
HID Service.

BT_UUID_CSC_VAL
Cycling Speed and Cadence Service UUID value.

BT_UUID_CSC
Cycling Speed and Cadence Service.

BT_UUID_ESS_VAL
Environmental Sensing Service UUID value.

BT_UUID_ESS
Environmental Sensing Service.

BT_UUID_BMS_VAL
Bond Management Service UUID value.

BT_UUID_BMS
Bond Management Service.

BT_UUID_IPSS_VAL
IP Support Service UUID value.

BT_UUID_IPSS
IP Support Service.

BT_UUID_HPS_VAL
HTTP Proxy Service UUID value.

BT_UUID_HPS
HTTP Proxy Service.

BT_UUID_OTS_VAL
Object Transfer Service UUID value.

BT_UUID_OTS
Object Transfer Service.

BT_UUID_MESH_PROV_VAL
Mesh Provisioning Service UUID value.

BT_UUID_MESH_PROV
Mesh Provisioning Service.

BT_UUID_MESH_PROXY_VAL
Mesh Proxy Service UUID value.

BT_UUID_MESH_PROXY
Mesh Proxy Service.

BT_UUID_GATT_PRIMARY_VAL
GATT Primary Service UUID value.

BT_UUID_GATT_PRIMARY
GATT Primary Service.

BT_UUID_GATT_SECONDARY_VAL
GATT Secondary Service UUID value.

BT_UUID_GATT_SECONDARY
GATT Secondary Service.

BT_UUID_GATT_INCLUDE_VAL
GATT Include Service UUID value.

BT_UUID_GATT_INCLUDE
GATT Include Service.

BT_UUID_GATT_CHRC_VAL
GATT Characteristic UUID value.

BT_UUID_GATT_CHRC
GATT Characteristic.

BT_UUID_GATT_CEP_VAL
GATT Characteristic Extended Properties UUID value.

BT_UUID_GATT_CEP
GATT Characteristic Extended Properties.

BT_UUID_GATT_CUD_VAL
GATT Characteristic User Description UUID value.

BT_UUID_GATT_CUD
GATT Characteristic User Description.

BT_UUID_GATT_CCC_VAL
GATT Client Characteristic Configuration UUID value.

BT_UUID_GATT_CCC
GATT Client Characteristic Configuration.

BT_UUID_GATT_SCC_VAL
GATT Server Characteristic Configuration UUID value.

BT_UUID_GATT_SCC
GATT Server Characteristic Configuration.

BT_UUID_GATT_CPF_VAL
GATT Characteristic Presentation Format UUID value.

BT_UUID_GATT_CPF
GATT Characteristic Presentation Format.

BT_UUID_VALID_RANGE_VAL
Valid Range Descriptor UUID value.

BT_UUID_VALID_RANGE
Valid Range Descriptor.

BT_UUID_HIDS_EXT_REPORT_VAL
HID External Report Descriptor UUID value.

BT_UUID_HIDS_EXT_REPORT
HID External Report Descriptor.

BT_UUID_HIDS_REPORT_REF_VAL
HID Report Reference Descriptor UUID value.

BT_UUID_HIDS_REPORT_REF
HID Report Reference Descriptor.

BT_UUID_ES_CONFIGURATION_VAL
Environmental Sensing Configuration Descriptor UUID value.

BT_UUID_ES_CONFIGURATION
Environmental Sensing Configuration Descriptor.

BT_UUID_ES_MEASUREMENT_VAL
Environmental Sensing Measurement Descriptor UUID value.

BT_UUID_ES_MEASUREMENT
Environmental Sensing Measurement Descriptor.

BT_UUID_ES_TRIGGER_SETTING_VAL
Environmental Sensing Trigger Setting Descriptor UUID value.

BT_UUID_ES_TRIGGER_SETTING
Environmental Sensing Trigger Setting Descriptor.

BT_UUID_GAP_DEVICE_NAME_VAL
GAP Characteristic Device Name UUID value.

BT_UUID_GAP_DEVICE_NAME
GAP Characteristic Device Name.

BT_UUID_GAP_APPEARANCE_VAL
GAP Characteristic Appearance UUID value.

BT_UUID_GAP_APPEARANCE
GAP Characteristic Appearance.

BT_UUID_GAP_PPCP_VAL
GAP Characteristic Peripheral Preferred Connection Parameters UUID value.

BT_UUID_GAP_PPCP
GAP Characteristic Peripheral Preferred Connection Parameters.

BT_UUID_GATT_SC_VAL
GATT Characteristic Service Changed UUID value.

BT_UUID_GATT_SC
GATT Characteristic Service Changed.

BT_UUID_ALERT_LEVEL_VAL
Alert Level UUID value.

BT_UUID_ALERT_LEVEL
Alert Level.

BT_UUID_TPS_TX_POWER_LEVEL_VAL
TPS Characteristic Tx Power Level UUID value.

BT_UUID_TPS_TX_POWER_LEVEL
TPS Characteristic Tx Power Level.

BT_UUID_BAS_BATTERY_LEVEL_VAL
BAS Characteristic Battery Level UUID value.

BT_UUID_BAS_BATTERY_LEVEL
BAS Characteristic Battery Level.

BT_UUID_HTS_MEASUREMENT_VAL
HTS Characteristic Measurement Value UUID value.

BT_UUID_HTS_MEASUREMENT
HTS Characteristic Measurement Value.

BT_UUID_HIDS_BOOT_KB_IN_REPORT_VAL
HID Characteristic Boot Keyboard Input Report UUID value.

BT_UUID_HIDS_BOOT_KB_IN_REPORT
HID Characteristic Boot Keyboard Input Report.

BT_UUID_DIS_SYSTEM_ID_VAL
DIS Characteristic System ID UUID value.

BT_UUID_DIS_SYSTEM_ID
DIS Characteristic System ID.

BT_UUID_DIS_MODEL_NUMBER_VAL
DIS Characteristic Model Number String UUID value.

BT_UUID_DIS_MODEL_NUMBER
DIS Characteristic Model Number String.

BT_UUID_DIS_SERIAL_NUMBER_VAL
DIS Characteristic Serial Number String UUID value.

BT_UUID_DIS_SERIAL_NUMBER
DIS Characteristic Serial Number String.

BT_UUID_DIS_FIRMWARE_REVISION_VAL
DIS Characteristic Firmware Revision String UUID value.

BT_UUID_DIS_FIRMWARE_REVISION
DIS Characteristic Firmware Revision String.

BT_UUID_DIS_HARDWARE_REVISION_VAL
DIS Characteristic Hardware Revision String UUID value.

BT_UUID_DIS_HARDWARE_REVISION
DIS Characteristic Hardware Revision String.

BT_UUID_DIS_SOFTWARE_REVISION_VAL
DIS Characteristic Software Revision String UUID value.

BT_UUID_DIS_SOFTWARE_REVISION
DIS Characteristic Software Revision String.

BT_UUID_DIS_MANUFACTURER_NAME_VAL
DIS Characteristic Manufacturer Name String UUID Value.

BT_UUID_DIS_MANUFACTURER_NAME
DIS Characteristic Manufacturer Name String.

BT_UUID_DIS_PNP_ID_VAL
DIS Characteristic PnP ID UUID value.

BT_UUID_DIS_PNP_ID
DIS Characteristic PnP ID.

BT_UUID_CTS_CURRENT_TIME_VAL
CTS Characteristic Current Time UUID value.

BT_UUID_CTS_CURRENT_TIME
CTS Characteristic Current Time.

BT_UUID_MAGN_DECLINATION_VAL
Magnetic Declination Characteristic UUID value.

BT_UUID_MAGN_DECLINATION
Magnetic Declination Characteristic.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT_VAL
HID Boot Keyboard Output Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT
HID Boot Keyboard Output Report Characteristic.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT_VAL
HID Boot Mouse Input Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT
HID Boot Mouse Input Report Characteristic.

BT_UUID_HRS_MEASUREMENT_VAL
HRS Characteristic Measurement Interval UUID value.

BT_UUID_HRS_MEASUREMENT
HRS Characteristic Measurement Interval.

BT_UUID_HRS_BODY_SENSOR
HRS Characteristic Body Sensor Location.

BT_UUID_HRS_BODY_SENSOR_VAL

BT_UUID_HRS_CONTROL_POINT
HRS Characteristic Control Point.

BT_UUID_HRS_CONTROL_POINT_VAL
HRS Characteristic Control Point UUID value.

BT_UUID_HIDS_INFO_VAL
HID Information Characteristic UUID value.

BT_UUID_HIDS_INFO
HID Information Characteristic.

BT_UUID_HIDS_REPORT_MAP_VAL
HID Report Map Characteristic UUID value.

BT_UUID_HIDS_REPORT_MAP
HID Report Map Characteristic.

BT_UUID_HIDS_CTRL_POINT_VAL
HID Control Point Characteristic UUID value.

BT_UUID_HIDS_CTRL_POINT
HID Control Point Characteristic.

BT_UUID_HIDS_REPORT_VAL
HID Report Characteristic UUID value.

BT_UUID_HIDS_REPORT
HID Report Characteristic.

BT_UUID_HIDS_PROTOCOL_MODE_VAL
HID Protocol Mode Characteristic UUID value.

BT_UUID_HIDS_PROTOCOL_MODE
HID Protocol Mode Characteristic.

BT_UUID_CSC_MEASUREMENT_VAL
CSC Measurement Characteristic UUID value.

BT_UUID_CSC_MEASUREMENT
CSC Measurement Characteristic.

BT_UUID_CSC_FEATURE_VAL
CSC Feature Characteristic UUID value.

BT_UUID_CSC_FEATURE
CSC Feature Characteristic.

BT_UUID_SENSOR_LOCATION_VAL
Sensor Location Characteristic UUID value.

BT_UUID_SENSOR_LOCATION
Sensor Location Characteristic.

BT_UUID_SC_CONTROL_POINT_VAL
SC Control Point Characteristic UUID value.

BT_UUID_SC_CONTROL_POINT
SC Control Point Characteristic.

BT_UUID_ELEVATION_VAL
Elevation Characteristic UUID value.

BT_UUID_ELEVATION
Elevation Characteristic.

BT_UUID_PRESSURE_VAL
Pressure Characteristic UUID value.

BT_UUID_PRESSURE
Pressure Characteristic.

BT_UUID_TEMPERATURE_VAL
Temperature Characteristic UUID value.

BT_UUID_TEMPERATURE
Temperature Characteristic.

BT_UUID_HUMIDITY_VAL
Humidity Characteristic UUID value.

BT_UUID_HUMIDITY
Humidity Characteristic.

BT_UUID_TRUE_WIND_SPEED_VAL
True Wind Speed Characteristic UUID value.

BT_UUID_TRUE_WIND_SPEED
True Wind Speed Characteristic.

BT_UUID_TRUE_WIND_DIR_VAL
True Wind Direction Characteristic UUID value.

BT_UUID_TRUE_WIND_DIR
True Wind Direction Characteristic.

BT_UUID_APPARENT_WIND_SPEED_VAL
Apparent Wind Speed Characteristic UUID value.

BT_UUID_APPARENT_WIND_SPEED
Apparent Wind Speed Characteristic.

BT_UUID_APPARENT_WIND_DIR_VAL
Apparent Wind Direction Characteristic UUID value.

BT_UUID_APPARENT_WIND_DIR
Apparent Wind Direction Characteristic.

BT_UUID_GUST_FACTOR_VAL
Gust Factor Characteristic UUID value.

BT_UUID_GUST_FACTOR
Gust Factor Characteristic.

BT_UUID_POLLEN_CONCENTRATION_VAL
Pollen Concentration Characteristic UUID value.

BT_UUID_POLLEN_CONCENTRATION
Pollen Concentration Characteristic.

BT_UUID_UV_INDEX_VAL
UV Index Characteristic UUID value.

BT_UUID_UV_INDEX
UV Index Characteristic.

BT_UUID_IRRADIANCE_VAL
Irradiance Characteristic UUID value.

BT_UUID_IRRADIANCE
Irradiance Characteristic.

BT_UUID_RAINFALL_VAL
Rainfall Characteristic UUID value.

BT_UUID_RAINFALL
Rainfall Characteristic.

BT_UUID_WIND_CHILL_VAL
Wind Chill Characteristic UUID value.

BT_UUID_WIND_CHILL
Wind Chill Characteristic.

BT_UUID_HEAT_INDEX_VAL
Heat Index Characteristic UUID value.

BT_UUID_HEAT_INDEX
Heat Index Characteristic.

BT_UUID_DEW_POINT_VAL
Dew Point Characteristic UUID value.

BT_UUID_DEW_POINT
Dew Point Characteristic.

BT_UUID_DESC_VALUE_CHANGED_VAL
Descriptor Value Changed Characteristic UUID value.

BT_UUID_DESC_VALUE_CHANGED
Descriptor Value Changed Characteristic.

BT_UUID_MAGN_FLUX_DENSITY_2D_VAL
Magnetic Flux Density - 2D Characteristic UUID value.

BT_UUID_MAGN_FLUX_DENSITY_2D
Magnetic Flux Density - 2D Characteristic.

BT_UUID_MAGN_FLUX_DENSITY_3D_VAL
Magnetic Flux Density - 3D Characteristic UUID value.

BT_UUID_MAGN_FLUX_DENSITY_3D
Magnetic Flux Density - 3D Characteristic.

BT_UUID_BAR_PRESSURE_TREND_VAL
Barometric Pressure Trend Characteristic UUID value.

BT_UUID_BAR_PRESSURE_TREND
Barometric Pressure Trend Characteristic.

BT_UUID_BMS_CONTROL_POINT_VAL
Bond Management Control Point UUID value.

BT_UUID_BMS_CONTROL_POINT
Bond Management Control Point.

BT_UUID_BMS_FEATURE_VAL
Bond Management Feature UUID value.

BT_UUID_BMS_FEATURE
Bond Management Feature.

BT_UUID_CENTRAL_ADDR_RES_VAL
Central Address Resolution Characteristic UUID value.

BT_UUID_CENTRAL_ADDR_RES
Central Address Resolution Characteristic.

BT_UUID_URI_VAL
URI UUID value.

BT_UUID_URI
URI.

BT_UUID_HTTP_HEADERS_VAL
HTTP Headers UUID value.

BT_UUID_HTTP_HEADERS
HTTP Headers.

BT_UUID_HTTP_STATUS_CODE_VAL
HTTP Status Code UUID value.

BT_UUID_HTTP_STATUS_CODE
HTTP Status Code.

BT_UUID_HTTP_ENTITY_BODY_VAL
HTTP Entity Body UUID value.

BT_UUID_HTTP_ENTITY_BODY
HTTP Entity Body.

BT_UUID_HTTP_CONTROL_POINT_VAL
HTTP Control Point UUID value.

BT_UUID_HTTP_CONTROL_POINT
HTTP Control Point.

BT_UUID_HTTPS_SECURITY_VAL
HTTPS Security UUID value.

BT_UUID_HTTPS_SECURITY
HTTPS Security.

BT_UUID_OTS_FEATURE_VAL
OTS Feature Characteristic UUID value.

BT_UUID_OTS_FEATURE
OTS Feature Characteristic.

BT_UUID_OTS_NAME_VAL
OTS Object Name Characteristic UUID value.

BT_UUID_OTS_NAME
OTS Object Name Characteristic.

BT_UUID_OTS_TYPE_VAL
OTS Object Type Characteristic UUID value.

BT_UUID_OTS_TYPE
OTS Object Type Characteristic.

BT_UUID_OTS_SIZE_VAL
OTS Object Size Characteristic UUID value.

BT_UUID_OTS_SIZE
OTS Object Size Characteristic.

BT_UUID_OTS_FIRST_CREATED_VAL
OTS Object First-Created Characteristic UUID value.

BT_UUID_OTS_FIRST_CREATED
OTS Object First-Created Characteristic.

BT_UUID_OTS_LAST_MODIFIED_VAL
OTS Object Last-Modified Characteristic UUI value.

BT_UUID_OTS_LAST_MODIFIED
OTS Object Last-Modified Characteristic.

BT_UUID_OTS_ID_VAL
OTS Object ID Characteristic UUID value.

BT_UUID_OTS_ID
OTS Object ID Characteristic.

BT_UUID_OTS_PROPERTIES_VAL
OTS Object Properties Characteristic UUID value.

BT_UUID_OTS_PROPERTIES
OTS Object Properties Characteristic.

BT_UUID_OTS_ACTION_CP_VAL
OTS Object Action Control Point Characteristic UUID value.

BT_UUID_OTS_ACTION_CP
OTS Object Action Control Point Characteristic.

BT_UUID_OTS_LIST_CP_VAL
OTS Object List Control Point Characteristic UUID value.

BT_UUID_OTS_LIST_CP
OTS Object List Control Point Characteristic.

BT_UUID_OTS_LIST_FILTER_VAL
OTS Object List Filter Characteristic UUID value.

BT_UUID_OTS_LIST_FILTER
OTS Object List Filter Characteristic.

BT_UUID_OTS_CHANGED_VAL
OTS Object Changed Characteristic UUID value.

BT_UUID_OTS_CHANGED
OTS Object Changed Characteristic.

BT_UUID_OTS_TYPE_UNSPECIFIED_VAL
OTS Unspecified Object Type UUID value.

BT_UUID_OTS_TYPE_UNSPECIFIED
OTS Unspecified Object Type.

BT_UUID_OTS_DIRECTORY_LISTING_VAL
OTS Directory Listing UUID value.

BT_UUID_OTS_DIRECTORY_LISTING
OTS Directory Listing.

BT_UUID_MESH_PROV_DATA_IN_VAL
Mesh Provisioning Data In UUID value.

BT_UUID_MESH_PROV_DATA_IN
Mesh Provisioning Data In.

BT_UUID_MESH_PROV_DATA_OUT_VAL
Mesh Provisioning Data Out UUID value.

BT_UUID_MESH_PROV_DATA_OUT
Mesh Provisioning Data Out.

BT_UUID_MESH_PROXY_DATA_IN_VAL
Mesh Proxy Data In UUID value.

BT_UUID_MESH_PROXY_DATA_IN
Mesh Proxy Data In.

BT_UUID_MESH_PROXY_DATA_OUT_VAL
Mesh Proxy Data Out UUID value.

BT_UUID_MESH_PROXY_DATA_OUT
Mesh Proxy Data Out.

BT_UUID_GATT_CLIENT_FEATURES_VAL
Client Supported Features UUID value.

BT_UUID_GATT_CLIENT_FEATURES
Client Supported Features.

BT_UUID_GATT_DB_HASH_VAL
Database Hash UUID value.

BT_UUID_GATT_DB_HASH
Database Hash.

BT_UUID_GATT_SERVER_FEATURES_VAL
Server Supported Features UUID value.

BT_UUID_GATT_SERVER_FEATURES
Server Supported Features.

BT_UUID_SDP_VAL

BT_UUID_SDP

BT_UUID_UDP_VAL

BT_UUID_UDP

BT_UUID_RFCOMM_VAL

BT_UUID_RFCOMM

BT_UUID_TCP_VAL

BT_UUID_TCP

BT_UUID_TCS_BIN_VAL

BT_UUID_TCS_BIN

BT_UUID_TCS_AT_VAL

BT_UUID_TCS_AT

BT_UUID_ATT_VAL

BT_UUID_ATT

BT_UUID_OBEX_VAL

BT_UUID_OBEX

BT_UUID_IP_VAL

BT_UUID_IP

BT_UUID_FTP_VAL

BT_UUID_FTP

BT_UUID_HTTP_VAL

BT_UUID_HTTP

BT_UUID_BNEP_VAL

BT_UUID_BNEP

BT_UUID_UPNP_VAL

BT_UUID_UPNP

BT_UUID_HIDP_VAL

BT_UUID_HIDP

BT_UUID_HCRP_CTRL_VAL

BT_UUID_HCRP_CTRL

BT_UUID_HCRP_DATA_VAL

BT_UUID_HCRP_DATA

BT_UUID_HCRP_NOTE_VAL

BT_UUID_HCRP_NOTE

BT_UUID_AVCTP_VAL

BT_UUID_AVCTP

BT_UUID_AVDTP_VAL

BT_UUID_AVDTP

BT_UUID_CMTP_VAL

BT_UUID_CMTP

BT_UUID_UDI_VAL

BT_UUID_UDI

BT_UUID_MCAP_CTRL_VAL

BT_UUID_MCAP_CTRL

BT_UUID_MCAP_DATA_VAL

BT_UUID_MCAP_DATA

BT_UUID_L2CAP_VAL

BT_UUID_L2CAP

Enums

enum [anonymous]

Bluetooth UUID types.

Values:

enumerator BT_UUID_TYPE_16

UUID type 16-bit.

enumerator BT_UUID_TYPE_32

UUID type 32-bit.

enumerator BT_UUID_TYPE_128

UUID type 128-bit.

Functions

int **bt_uuid_cmp**(**const struct** *bt_uuid* **u1*, **const struct** *bt_uuid* **u2*)

Compare Bluetooth UUIDs.

Compares 2 Bluetooth UUIDs, if the types are different both UUIDs are first converted to 128 bits format before comparing.

Return negative value if *u1* < *u2*, 0 if *u1* == *u2*, else positive

Parameters

- `u1`: First Bluetooth UUID to compare
- `u2`: Second Bluetooth UUID to compare

bool **bt_uuid_create**(**struct** *bt_uuid* **uuid*, **const** uint8_t **data*, uint8_t *data_len*)

Create a *bt_uuid* from a little-endian data buffer.

Create a *bt_uuid* from a little-endian data buffer. The data_len parameter is used to determine whether the UUID is in 16, 32 or 128 bit format (length 2, 4 or 16). Note: 32 bit format is not allowed over the air.

Return true if the data was valid and the UUID was successfully created.

Parameters

- `uuid`: Pointer to the *bt_uuid* variable
- `data`: pointer to UUID stored in little-endian data buffer
- `data_len`: length of the UUID in the data buffer

void **bt_uuid_to_str**(**const struct** *bt_uuid* **uuid*, char **str*, size_t *len*)

Convert Bluetooth UUID to string.

Converts Bluetooth UUID to string. UUID can be in any format, 16-bit, 32-bit or 128-bit.

Return N/A

Parameters

- `uuid`: Bluetooth UUID
- `str`: pointer where to put converted string
- `len`: length of str

struct bt_uuid

#include <uuid.h> This is a 'tentative' type and should be used as a pointer only.

struct bt_uuid_16

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint16_t **val**

UUID value, 16-bit in host endianness.

struct bt_uuid_32

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint32_t **val**

UUID value, 32-bit in host endianness.

struct bt_uuid_128

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint8_t **val**[16]

UUID value, 128-bit in little-endian format.

1.12 services

1.12.1 HTTP Proxy Service (HPS)

1.12.1.1 API Reference

group **bt_hps**

HTTP Proxy Service (HPS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Defines

MAX_URI_LEN

MAX_HEADERS_LEN

MAX_BODY_LEN

Typedefs

typedef uint8_t **hps_data_status_t**

typedef uint8_t **hps_http_command_t**

typedef uint8_t **hps_state_t**

typedef uint8_t **hps_flags_t**

Enums

enum [anonymous]

Values:

enumerator HPS_HEADERS_RECEIVED

enumerator HPS_HEADERS_TRUNCATED

enumerator HPS_BODY_RECEIVED

enumerator HPS_BODY_TRUNCATED

enum [anonymous]

Values:

enumerator HTTP_GET_REQ

enumerator HTTP_HEAD_REQ

enumerator HTTP_POST_REQ

enumerator HTTP_PUT_REQ

enumerator HTTP_DELETE_REQ

enumerator HTTPS_GET_REQ

enumerator HTTPS_HEAD_REQ

enumerator HTTPS_POST_REQ

enumerator HTTPS_PUT_REQ

enumerator HTTPS_DELETE_REQ

enumerator HTTP_REQ_CANCEL

enum [anonymous]
Values:

enumerator IDLE_STATE

enumerator BUSY_STATE

enum [anonymous]
Values:

enumerator URI_SET

enumerator HEADERS_SET

enumerator BODY_SET

enum [anonymous]
Values:

enumerator HPS_ERR_INVALID_REQUEST

enumerator HPS_ERR_CCCD_IMPROPERLY_CONFIGURED

enumerator HPS_ERR_PROC_ALREADY_IN_PROGRESS

enum [anonymous]
Values:

enumerator HTTPS_CERTIFICATE_INVALID

enumerator HTTPS_CERTIFICATE_VALID

Functions

ssize_t **write_http_headers**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, **const** void **buf*, uint16_t *len*, uint16_t *offset*, uint8_t *flags*)

HTTP Headers GATT write callback.

If called with conn == NULL, it is a local write.

Return Number of bytes written.

ssize_t **write_http_entity_body**(**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, **const** void **buf*, uint16_t *len*, uint16_t *offset*, uint8_t *flags*)

HTTP Entity Body GATT write callback.

If called with conn == NULL, it is a local write.

Return Number of bytes written.

int **bt_hps_init**(osa_msgq_handle_t *queue*)

HTTP Proxy Server initialization.

Return Zero in case of success and error code in case of error.

void **`bt_hps_set_status_code`** (uint16_t *http_status*)
Sets Status Code after HTTP request was fulfilled.

int **`bt_hps_notify`** (void)
Notify HTTP Status after Request was fulfilled.

This will send a GATT notification to the subscriber.

Return Zero in case of success and error code in case of error.

`struct hps_status_t`
#include <hps.h>

`struct hps_config_t`
#include <hps.h>

1.12.2 Health Thermometer Service (HTS)

1.12.2.1 API Reference

group **`bt_hts`**
Health Thermometer Service (HTS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Defines

`hts_unit_celsius_c`

`hts_unit_fahrenheit_c`

`hts_include_temp_type`

Enums

`enum` **[anonymous]**
Values:

`enumerator hts_no_temp_type`

`enumerator hts_armpit`

`enumerator hts_body`

`enumerator hts_ear`

`enumerator hts_finger`

`enumerator hts_gastroInt`

`enumerator hts_mouth`

`enumerator hts_rectum`

`enumerator hts_toe`

`enumerator hts_tympanum`

Functions

void **bt_hts_indicate** (void)

Notify indicate a temperature measurement.

This will send a GATT indication to all current subscribers. Awaits an indication response from peer.

Return Zero in case of success and error code in case of error.

Parameters

- `none.`:

struct temp_measurement

#include <hts.h>

1.12.3 Internet Protocol Support Profile (IPSP)

1.12.3.1 API Reference

group **bt_ipsp**

Internet Protocol Support Profile (IPSP)

Defines

USER_DATA_MIN

Typedefs

typedef int (***ipsp_rx_cb_t**) (**struct** net_buf *buf)

Functions

int **ipsp_init** (*ipsp_rx_cb_t pf_rx_cb*)

Initialize the service.

This will setup the data receive callback.

Return Zero in case of success and error code in case of error.

Parameters

- `pf_rx_cb`: Pointer to the callback used for receiiving data.

int **ipsp_connect** (**struct** bt_conn **conn*)

Start a connection to an IPSP Node using this connection.

This will try to connect to the Node present.

Return Zero in case of success and error code in case of error.

Parameters

- `conn`: Pointer to the connection to be used.

int **`ipsp_send`** (**`struct`** net_buf **buf*)
Send data to the peer IPSP Node/Router.

Return Zero in case of success and error code in case of error.

Parameters

- `conn`: Pointer to the buffer containing data.

int **`ipsp_listen`** (void)
Setup an IPSP Server.

Return Zero in case of success and error code in case of error.

1.12.4 Proximity Reporter (PXR)

1.12.4.1 API Reference

group **`bt_pxr`**
Proximity Reporter (PXR)

Typedefs

`typedef` void (***`alert_ui_cb`**) (uint8_t param)

Enums

`enum` **[anonymous]**
Values:

`enumerator NO_ALERT`

`enumerator MILD_ALERT`

`enumerator HIGH_ALERT`

Functions

ssize_t **`write_ias_alert_level`** (**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr* **attr*, **`const`** void **buf*, uint16_t *len*, uint16_t *offset*, uint8_t *flags*)
IAS Alert Level GATT write callback.

If called with conn == NULL, it is a local write.

Return Number of bytes written.

ssize_t **`read_lls_alert_level`** (**`struct`** bt_conn **conn*, **`const struct`** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)
IAS Alert Level GATT read callback.

Return Number of bytes read.

ssize_t **write_lls_alert_level** (**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, **const** void **buf*, uint16_t *len*, uint16_t *offset*, uint8_t *flags*)

LLS Alert Level GATT write callback.

If called with conn == NULL, it is a local write.

Return Number of bytes written.

ssize_t **read_tps_power_level** (**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

TPS Power Level GATT read callback.

Return Number of bytes read.

ssize_t **read_tps_power_level_desc** (**struct** bt_conn **conn*, **const struct** *bt_gatt_attr* **attr*, void **buf*, uint16_t *len*, uint16_t *offset*)

TPS Power Level Descriptor GATT read callback.

Return Number of bytes read.

uint8_t **pxr_lls_get_alert_level** (void)

Read LLS Alert Level locally.

Return Number of bytes written.

uint8_t **pxr_ias_get_alert_level** (void)

Read IAS Alert Level locally.

Return Number of bytes written.

int8_t **pxr_tps_get_power_level** (void)

Read TPS Power Level locally.

Return Number of bytes written.

void **pxr_tps_set_power_level** (int8_t *power_level*)

Write TPS Power Level locally.

Return Number of bytes written.

int **pxr_init** (*alert_ui_cb* *cb*)

Initialize PXR Service.

Return Success or error.

int **pxr_deinit** (void)

Deinitialize PXR Service.

Return Success or error.

INDEX

Symbols

_BT_GATT_ATTRS_ARRAY_DEFINE (*C macro*), 70
_BT_GATT_SERVICE_ARRAY_ITEM (*C macro*), 70
_bt_gatt_ccc (*C struct*), 81
_bt_gatt_ccc.cfg (*C var*), 81
_bt_gatt_ccc.cfg_changed (*C var*), 81
_bt_gatt_ccc.cfg_match (*C var*), 81
_bt_gatt_ccc.cfg_write (*C var*), 81
_bt_gatt_ccc.value (*C var*), 81
_bt_security (*C enum*), 5
_bt_security.BT_SECURITY_FORCE_PAIR (*C enumerator*), 5
_bt_security.BT_SECURITY_L0 (*C enumerator*), 5
_bt_security.BT_SECURITY_L1 (*C enumerator*), 5
_bt_security.BT_SECURITY_L2 (*C enumerator*), 5
_bt_security.BT_SECURITY_L3 (*C enumerator*), 5
_bt_security.BT_SECURITY_L4 (*C enumerator*), 5
_bt_spp_callback (*C struct*), 138
_hf_multiparty_call_option_t (*C enum*), 93
_hf_multiparty_call_option_t.hf_multiparty_call_option_five (*C enumerator*), 93
_hf_multiparty_call_option_t.hf_multiparty_call_option_four (*C enumerator*), 93
_hf_multiparty_call_option_t.hf_multiparty_call_option_one (*C enumerator*), 93
_hf_multiparty_call_option_t.hf_multiparty_call_option_three (*C enumerator*), 93
_hf_multiparty_call_option_t.hf_multiparty_call_option_two (*C enumerator*), 93
_hf_volume_type_t (*C enum*), 92, 93
_hf_volume_type_t.hf_volume_type_mic (*C enumerator*), 92, 93
_hf_volume_type_t.hf_volume_type_speaker (*C enumerator*), 92, 93
_hf_waiting_call_state_t (*C struct*), 104
_hfp_ag_call_status_t (*C enum*), 92
_hfp_ag_call_status_t.hfp_ag_call_call_active (*C enumerator*), 92
_hfp_ag_call_status_t.hfp_ag_call_call_end (*C enumerator*), 92
_hfp_ag_call_status_t.hfp_ag_call_call_incoming (*C enumerator*), 92
_hfp_ag_call_status_t.hfp_ag_call_call_outgoing (*C enumerator*), 92
_hfp_ag_cind_t (*C struct*), 102
_hfp_ag_get_config (*C struct*), 102
[anonymous] (*C enum*), 4, 5, 31, 33, 34, 62, 63, 66, 74, 84, 85, 115, 126, 131, 154, 156–158, 160
[anonymous].BODY_SET (*C enumerator*), 157
[anonymous].BT_A2DP_DISCOVER_ENDPOINT_CONTINUE (*C enumerator*), 131
[anonymous].BT_A2DP_DISCOVER_ENDPOINT_STOP (*C enumerator*), 131
[anonymous].BT_CONN_LE_OPT_CODED (*C enumerator*), 5
[anonymous].BT_CONN_LE_OPT_NONE (*C enumerator*), 5
[anonymous].BT_CONN_LE_OPT_NO_1M (*C enumerator*), 5
[anonymous].BT_CONN_LE_PHY_OPT_CODED_S2 (*C enumerator*), 4
[anonymous].BT_CONN_LE_PHY_OPT_CODED_S8 (*C enumerator*), 4
[anonymous].BT_CONN_LE_PHY_OPT_NONE (*C enumerator*), 4
[anonymous].BT_CONN_ROLE_MASTER (*C enumerator*), 4
[anonymous].BT_CONN_ROLE_SLAVE (*C enumerator*), 4
[anonymous].BT_CONN_TYPE_ALL (*C enumerator*), 4
[anonymous].BT_CONN_TYPE_BR (*C enumerator*), 4
[anonymous].BT_CONN_TYPE_ISO (*C enumerator*), 4
[anonymous].BT_CONN_TYPE_LE (*C enumerator*), 4
[anonymous].BT_CONN_TYPE_SCO (*C enumerator*), 4

[anonymous].BT_GAP_ADV_PROP_CONNECTABLE (*C enumerator*), 63
[anonymous].BT_GAP_ADV_PROP_DIRECTED (*C enumerator*), 63
[anonymous].BT_GAP_ADV_PROP_EXT_ADV (*C enumerator*), 63
[anonymous].BT_GAP_ADV_PROP_SCANNABLE (*C enumerator*), 63
[anonymous].BT_GAP_ADV_PROP_SCAN_RESPONSE (*C enumerator*), 63
[anonymous].BT_GAP_ADV_TYPE_ADV_DIRECT_IND (*C enumerator*), 62
[anonymous].BT_GAP_ADV_TYPE_ADV_IND (*C enumerator*), 62
[anonymous].BT_GAP_ADV_TYPE_ADV_NONCONN_IND (*C enumerator*), 63
[anonymous].BT_GAP_ADV_TYPE_ADV_SCAN_IND (*C enumerator*), 62
[anonymous].BT_GAP_ADV_TYPE_EXT_ADV (*C enumerator*), 63
[anonymous].BT_GAP_ADV_TYPE_SCAN_RSP (*C enumerator*), 63
[anonymous].BT_GAP_CTE_AOA (*C enumerator*), 63
[anonymous].BT_GAP_CTE_AOD_1US (*C enumerator*), 63
[anonymous].BT_GAP_CTE_AOD_2US (*C enumerator*), 63
[anonymous].BT_GAP_CTE_NONE (*C enumerator*), 63
[anonymous].BT_GAP_LE_PHY_1M (*C enumerator*), 62
[anonymous].BT_GAP_LE_PHY_2M (*C enumerator*), 62
[anonymous].BT_GAP_LE_PHY_CODED (*C enumerator*), 62
[anonymous].BT_GAP_LE_PHY_NONE (*C enumerator*), 62
[anonymous].BT_GATT_DISCOVER_ATTRIBUTE (*C enumerator*), 85
[anonymous].BT_GATT_DISCOVER_CHARACTERISTIC (*C enumerator*), 84
[anonymous].BT_GATT_DISCOVER_DESCRIPTOR (*C enumerator*), 84
[anonymous].BT_GATT_DISCOVER_INCLUDE (*C enumerator*), 84
[anonymous].BT_GATT_DISCOVER_PRIMARY (*C enumerator*), 84
[anonymous].BT_GATT_DISCOVER_SECONDARY (*C enumerator*), 84
[anonymous].BT_GATT_ITER_CONTINUE (*C enumerator*), 74
[anonymous].BT_GATT_ITER_STOP (*C enumerator*), 74
[anonymous].BT_GATT_PERM_NONE (*C enumerator*), 66
[anonymous].BT_GATT_PERM_PREPARE_WRITE (*C enumerator*), 66
[anonymous].BT_GATT_PERM_READ (*C enumerator*), 66
[anonymous].BT_GATT_PERM_READ_AUTHEN (*C enumerator*), 66
[anonymous].BT_GATT_PERM_READ_ENCRYPT (*C enumerator*), 66
[anonymous].BT_GATT_PERM_WRITE (*C enumerator*), 66
[anonymous].BT_GATT_PERM_WRITE_AUTHEN (*C enumerator*), 66
[anonymous].BT_GATT_PERM_WRITE_ENCRYPT (*C enumerator*), 66
[anonymous].BT_GATT_SUBSCRIBE_FLAG_NO_RESUB (*C enumerator*), 85
[anonymous].BT_GATT_SUBSCRIBE_FLAG_VOLATILE (*C enumerator*), 85
[anonymous].BT_GATT_SUBSCRIBE_FLAG_WRITE_PENDING (*C enumerator*), 85
[anonymous].BT_GATT_SUBSCRIBE_NUM_FLAGS (*C enumerator*), 85
[anonymous].BT_GATT_WRITE_FLAG_CMD (*C enumerator*), 66
[anonymous].BT_GATT_WRITE_FLAG_PREPARE (*C enumerator*), 66
[anonymous].BT_LE_ADV_OPT_ANONYMOUS (*C enumerator*), 33
[anonymous].BT_LE_ADV_OPT_CODED (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_CONNECTABLE (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_DIR_ADDR_RPA (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_DISABLE_CHAN_37 (*C enumerator*), 33
[anonymous].BT_LE_ADV_OPT_DISABLE_CHAN_38 (*C enumerator*), 33
[anonymous].BT_LE_ADV_OPT_DISABLE_CHAN_39 (*C enumerator*), 33
[anonymous].BT_LE_ADV_OPT_EXT_ADV (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_FILTER_CONN (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_FILTER_SCAN_REQ (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_NONE (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_NOTIFY_SCAN_REQ (*C enumerator*), 32

[anonymous].BT_LE_ADV_OPT_NO_2M (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_ONE_TIME (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_SCANNABLE (*C enumerator*), 32
[anonymous].BT_LE_ADV_OPT_USE_IDENTITY (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_USE_NAME (*C enumerator*), 31
[anonymous].BT_LE_ADV_OPT_USE_TX_POWER (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_OPT_NONE (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_OPT_USE_TX_POWER (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOA (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_1US (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_2US (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_NONE (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_REPORTING_INITIALLY_DISABLED (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_SYNC_ONLY_CONST_TONE_EXT (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST (*C enumerator*), 33
[anonymous].BT_LE_PER_ADV_SYNC_TRANSFER_OPT_NONE (*C enumerator*), 34
[anonymous].BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOA (*C enumerator*), 34
[anonymous].BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_1US (*C enumerator*), 34
[anonymous].BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_2US (*C enumerator*), 34
[anonymous].BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_ONLY_CTE (*C enumerator*), 34
[anonymous].BT_LE_SCAN_OPT_CODED (*C enumerator*), 34
[anonymous].BT_LE_SCAN_OPT_FILTER_DUPLICATE (*C enumerator*), 34
[anonymous].BT_LE_SCAN_OPT_FILTER_WHITELIST (*C enumerator*), 34
[anonymous].BT_LE_SCAN_OPT_NONE (*C enumerator*), 34
[anonymous].BT_LE_SCAN_OPT_NO_1M (*C enumerator*), 34
[anonymous].BT_LE_SCAN_TYPE_ACTIVE (*C enumerator*), 34
[anonymous].BT_LE_SCAN_TYPE_PASSIVE (*C enumerator*), 34
[anonymous].BT_RFCOMM_CHAN_HFP_AG (*C enumerator*), 115
[anonymous].BT_RFCOMM_CHAN_HFP_HF (*C enumerator*), 115
[anonymous].BT_RFCOMM_CHAN_HSP_AG (*C enumerator*), 115
[anonymous].BT_RFCOMM_CHAN_HSP_HS (*C enumerator*), 115
[anonymous].BT_RFCOMM_CHAN_SPP (*C enumerator*), 115
[anonymous].BT_SDP_DISCOVER_UUID_CONTINUE (*C enumerator*), 126
[anonymous].BT_SDP_DISCOVER_UUID_STOP (*C enumerator*), 126
[anonymous].BT_UUID_TYPE_128 (*C enumerator*), 154
[anonymous].BT_UUID_TYPE_16 (*C enumerator*), 154
[anonymous].BT_UUID_TYPE_32 (*C enumerator*), 154
[anonymous].BUSY_STATE (*C enumerator*), 157
[anonymous].HEADERS_SET (*C enumerator*), 157
[anonymous].HIGH_ALERT (*C enumerator*), 160
[anonymous].HPS_BODY_RECEIVED (*C enumerator*), 156
[anonymous].HPS_BODY_TRUNCATED (*C enumerator*), 156
[anonymous].HPS_ERR_CCCD_IMPROPERLY_CONFIGURED (*C enumerator*), 157
[anonymous].HPS_ERR_INVALID_REQUEST (*C enumerator*), 157
[anonymous].HPS_ERR_PROC_ALREADY_IN_PROGRESS (*C enumerator*), 157
[anonymous].HPS_HEADERS_RECEIVED (*C enumerator*), 156
[anonymous].HPS_HEADERS_TRUNCATED (*C enumerator*), 156
[anonymous].HTTPS_CERTIFICATE_INVALID (*C enumerator*), 157
[anonymous].HTTPS_CERTIFICATE_VALID (*C enumerator*), 157
[anonymous].HTTPS_DELETE_REQ (*C enumerator*), 157
[anonymous].HTTPS_GET_REQ (*C enumerator*), 156
[anonymous].HTTPS_HEAD_REQ (*C enumerator*), 156
[anonymous].HTTPS_POST_REQ (*C enumerator*), 156
[anonymous].HTTPS_PUT_REQ (*C enumerator*), 156
[anonymous].HTTP_DELETE_REQ (*C enumerator*), 156
[anonymous].HTTP_GET_REQ (*C enumerator*), 156

[anonymous].HTTP_HEAD_REQ (*C enumerator*), 156
[anonymous].HTTP_POST_REQ (*C enumerator*), 156
[anonymous].HTTP_PUT_REQ (*C enumerator*), 156
[anonymous].HTTP_REQ_CANCEL (*C enumerator*), 157
[anonymous].IDLE_STATE (*C enumerator*), 157
[anonymous].MILD_ALERT (*C enumerator*), 160
[anonymous].NO_ALERT (*C enumerator*), 160
[anonymous].URI_SET (*C enumerator*), 157
[anonymous].hts_armpit (*C enumerator*), 158
[anonymous].hts_body (*C enumerator*), 158
[anonymous].hts_ear (*C enumerator*), 158
[anonymous].hts_finger (*C enumerator*), 158
[anonymous].hts_gastroInt (*C enumerator*), 158
[anonymous].hts_mouth (*C enumerator*), 158
[anonymous].hts_no_temp_type (*C enumerator*), 158
[anonymous].hts_rectum (*C enumerator*), 158
[anonymous].hts_toe (*C enumerator*), 158
[anonymous].hts_tympanum (*C enumerator*), 158

A

alert_ui_cb (*C type*), 160

B

bt_a2dp_codec_id (*C enum*), 130
bt_a2dp_codec_id.BT_A2DP_ATRAC (*C enumerator*), 131
bt_a2dp_codec_id.BT_A2DP_MPEG1 (*C enumerator*), 130
bt_a2dp_codec_id.BT_A2DP_MPEG2 (*C enumerator*), 130
bt_a2dp_codec_id.BT_A2DP_SBC (*C enumerator*), 130
bt_a2dp_codec_id.BT_A2DP_VENDOR (*C enumerator*), 131
bt_a2dp_codec_ie (*C struct*), 134
bt_a2dp_codec_ie.codec_ie (*C var*), 134
bt_a2dp_codec_ie.len (*C var*), 134
bt_a2dp_configure (*C function*), 132
bt_a2dp_configure_endpoint (*C function*), 133
bt_a2dp_connect (*C function*), 131
bt_a2dp_connect_cb (*C struct*), 135
bt_a2dp_connect_cb.connected (*C var*), 135
bt_a2dp_connect_cb.disconnected (*C var*), 135
bt_a2dp_control_cb (*C struct*), 134
bt_a2dp_control_cb.configured (*C var*), 135
bt_a2dp_control_cb.deconfigured (*C var*), 135
bt_a2dp_control_cb.sink_streamer_data (*C var*), 135
bt_a2dp_control_cb.start_play (*C var*), 135
bt_a2dp_control_cb.stop_play (*C var*), 135
bt_a2dp_deconfigure (*C function*), 133
bt_a2dp_disconnect (*C function*), 131
bt_a2dp_discover_peer_endpoint_cb_t (*C type*), 130
bt_a2dp_discover_peer_endpoints (*C function*), 132
bt_a2dp_endpoint (*C struct*), 135
bt_a2dp_endpoint.capabilities (*C var*), 136
bt_a2dp_endpoint.codec_buffer (*C var*), 136
bt_a2dp_endpoint.codec_buffer_nocached (*C var*), 136
bt_a2dp_endpoint.codec_id (*C var*), 136
bt_a2dp_endpoint.config (*C var*), 136
bt_a2dp_endpoint.control_cbs (*C var*), 136
bt_a2dp_endpoint.info (*C var*), 136
bt_a2dp_endpoint_config (*C struct*), 134
bt_a2dp_endpoint_config.media_config (*C var*), 134
bt_a2dp_endpoint_configure_result (*C struct*), 134
bt_a2dp_endpoint_configure_result.a2dp (*C var*), 134
bt_a2dp_endpoint_configure_result.config (*C var*), 134
bt_a2dp_endpoint_configure_result.conn (*C var*), 134
bt_a2dp_endpoint_configure_result.err (*C var*), 134
BT_A2DP_ENDPOINT_INIT (*C macro*), 129
BT_A2DP_EP_CONTENT_PROTECTION_INIT (*C macro*), 129
BT_A2DP_EP_DELAY_REPORTING_INIT (*C macro*), 129
BT_A2DP_EP_HEADER_COMPRESSION_INIT (*C macro*), 129
BT_A2DP_EP_MULTIPLEXING_INIT (*C macro*), 129
BT_A2DP_EP_RECOVERY_SERVICE_INIT (*C macro*), 129
BT_A2DP_EP_REPORTING_SERVICE_INIT (*C macro*), 129
BT_A2DP_MPEG_1_2_IE_LENGTH (*C macro*), 129
BT_A2DP_MPEG_2_4_IE_LENGTH (*C macro*), 129
bt_a2dp_reconfigure (*C function*), 134
bt_a2dp_register_connect_callback (*C function*), 132
bt_a2dp_register_endpoint (*C function*), 132
BT_A2DP_SBC_IE_LENGTH (*C macro*), 129
BT_A2DP_SBC_SINK_ENDPOINT (*C macro*), 130
BT_A2DP_SBC_SOURCE_ENDPOINT (*C macro*), 130
BT_A2DP_SINK_ENDPOINT_INIT (*C macro*), 129

BT_A2DP_SINK_SBC_CODEC_BUFFER_NOCACHED_SIZE (*C macro*), 129
BT_A2DP_SINK_SBC_CODEC_BUFFER_SIZE (*C macro*), 129
BT_A2DP_SOURCE_ENDPOINT_INIT (*C macro*), 129
BT_A2DP_SOURCE_SBC_CODEC_BUFFER_NOCACHED_SIZE (*C macro*), 129
BT_A2DP_SOURCE_SBC_CODEC_BUFFER_SIZE (*C macro*), 129
bt_a2dp_start (*C function*), 133
bt_a2dp_stop (*C function*), 133
BT_ADDR_ANY (*C macro*), 57
bt_addr_cmp (*C function*), 58
bt_addr_copy (*C function*), 58
bt_addr_from_str (*C function*), 59
BT_ADDR_IS_NRPA (*C macro*), 57
BT_ADDR_IS_RPA (*C macro*), 57
BT_ADDR_IS_STATIC (*C macro*), 57
BT_ADDR_LE_ANY (*C macro*), 57
bt_addr_le_cmp (*C function*), 58
bt_addr_le_copy (*C function*), 58
bt_addr_le_create_nrpa (*C function*), 59
bt_addr_le_create_static (*C function*), 59
bt_addr_le_from_str (*C function*), 60
bt_addr_le_is_identity (*C function*), 59
bt_addr_le_is_rpa (*C function*), 59
BT_ADDR_LE_NONE (*C macro*), 57
BT_ADDR_LE_PUBLIC (*C macro*), 57
BT_ADDR_LE_PUBLIC_ID (*C macro*), 57
BT_ADDR_LE_RANDOM (*C macro*), 57
BT_ADDR_LE_RANDOM_ID (*C macro*), 57
BT_ADDR_LE_STR_LEN (*C macro*), 58
bt_addr_le_t (*C struct*), 60
bt_addr_le_to_str (*C function*), 59
BT_ADDR_NONE (*C macro*), 57
BT_ADDR_SET_NRPA (*C macro*), 58
BT_ADDR_SET_RPA (*C macro*), 57
BT_ADDR_SET_STATIC (*C macro*), 58
BT_ADDR_STR_LEN (*C macro*), 58
bt_addr_t (*C struct*), 60
bt_addr_to_str (*C function*), 59
bt_bond_info (*C struct*), 57
bt_bond_info.addr (*C var*), 57
BT_BR_CONN_PARAM (*C macro*), 3
bt_br_conn_param (*C struct*), 25
BT_BR_CONN_PARAM_DEFAULT (*C macro*), 3
BT_BR_CONN_PARAM_INIT (*C macro*), 3
bt_br_discovery_cb_t (*C type*), 31
bt_br_discovery_param (*C struct*), 56
bt_br_discovery_param.length (*C var*), 57
bt_br_discovery_param.limited (*C var*), 57
bt_br_discovery_result (*C struct*), 56
bt_br_discovery_result._priv (*C var*), 56
bt_br_discovery_result.addr (*C var*), 56
bt_br_discovery_result.cod (*C var*), 56
bt_br_discovery_result.eir (*C var*), 56
bt_br_discovery_result.rssi (*C var*), 56
bt_br_discovery_start (*C function*), 47
bt_br_discovery_stop (*C function*), 47
bt_br_oob (*C struct*), 57
bt_br_oob.addr (*C var*), 57
bt_br_oob_get_local (*C function*), 47
bt_br_set_connectable (*C function*), 48
bt_br_set_discoverable (*C function*), 47
bt_buf_data (*C struct*), 27
bt_buf_get_cmd_complete (*C function*), 26
bt_buf_get_evt (*C function*), 26
bt_buf_get_rx (*C function*), 26
bt_buf_get_tx (*C function*), 26
bt_buf_get_type (*C function*), 27
BT_BUF_RESERVE (*C macro*), 25
BT_BUF_RX_SIZE (*C macro*), 25
bt_buf_set_type (*C function*), 27
BT_BUF_SIZE (*C macro*), 25
bt_buf_type (*C enum*), 25
bt_buf_type.BT_BUF_ACL_IN (*C enumerator*), 25
bt_buf_type.BT_BUF_ACL_OUT (*C enumerator*), 25
bt_buf_type.BT_BUF_CMD (*C enumerator*), 25
bt_buf_type.BT_BUF_EVT (*C enumerator*), 25
bt_buf_type.BT_BUF_H4 (*C enumerator*), 25
bt_buf_type.BT_BUF_ISO_IN (*C enumerator*), 25
bt_buf_type.BT_BUF_ISO_OUT (*C enumerator*), 25
BT_COMP_ID_LF (*C macro*), 60
bt_conn_auth_cancel (*C function*), 13
bt_conn_auth_cb (*C struct*), 22
bt_conn_auth_cb.bond_deleted (*C var*), 24
bt_conn_auth_cb.cancel (*C var*), 23
bt_conn_auth_cb.oob_data_request (*C var*), 23
bt_conn_auth_cb.pairing_accept (*C var*), 22
bt_conn_auth_cb.pairing_complete (*C var*), 24
bt_conn_auth_cb.pairing_confirm (*C var*), 24
bt_conn_auth_cb.pairing_failed (*C var*), 24
bt_conn_auth_cb.passkey_confirm (*C var*), 23
bt_conn_auth_cb.passkey_display (*C var*), 22
bt_conn_auth_cb.passkey_entry (*C var*), 23
bt_conn_auth_cb.pincode_entry (*C var*), 24
bt_conn_auth_cb_register (*C function*), 13
bt_conn_auth_pairing_confirm (*C function*), 13
bt_conn_auth_passkey_confirm (*C function*), 13

bt_conn_auth_passkey_entry (*C function*), 13
bt_conn_auth_pincode_entry (*C function*), 14
bt_conn_br_info (*C struct*), 16
bt_conn_br_remote_info (*C struct*), 17
bt_conn_br_remote_info.features (*C var*), 17
bt_conn_br_remote_info.num_pages (*C var*), 17
bt_conn_cb (*C struct*), 18
bt_conn_cb.connected (*C var*), 19
bt_conn_cb.disconnected (*C var*), 19
bt_conn_cb.identity_resolved (*C var*), 20
bt_conn_cb.le_data_len_updated (*C var*), 20
bt_conn_cb.le_param_req (*C var*), 19
bt_conn_cb.le_param_updated (*C var*), 19
bt_conn_cb.le_phy_updated (*C var*), 20
bt_conn_cb.remote_info_available (*C var*), 20
bt_conn_cb.security_changed (*C var*), 20
bt_conn_cb_register (*C function*), 11
bt_conn_create_auto_stop (*C function*), 10
bt_conn_create_br (*C function*), 14
bt_conn_create_sco (*C function*), 14
bt_conn_disconnect (*C function*), 8
bt_conn_enc_key_size (*C function*), 11
bt_conn_foreach (*C function*), 6
bt_conn_get_dst (*C function*), 6
bt_conn_get_dst_br (*C function*), 7
bt_conn_get_info (*C function*), 7
bt_conn_get_remote_info (*C function*), 7
bt_conn_get_security (*C function*), 11
bt_conn_index (*C function*), 7
bt_conn_info (*C struct*), 16
bt_conn_info.__unnamed__ (*C union*), 16
bt_conn_info.__unnamed__.br (*C var*), 16
bt_conn_info.__unnamed__.le (*C var*), 16
bt_conn_info.id (*C var*), 16
bt_conn_info.role (*C var*), 16
bt_conn_info.type (*C var*), 16
bt_conn_info.[anonymous] (*C var*), 16
bt_conn_le_create (*C function*), 9
bt_conn_le_create_auto (*C function*), 9
BT_CONN_LE_CREATE_CONN (*C macro*), 3
BT_CONN_LE_CREATE_CONN_AUTO (*C macro*), 3
BT_CONN_LE_CREATE_PARAM (*C macro*), 3
bt_conn_le_create_param (*C struct*), 18
bt_conn_le_create_param.interval (*C var*), 18
bt_conn_le_create_param.interval_coded (*C var*), 18
bt_conn_le_create_param.options (*C var*), 18
bt_conn_le_create_param.timeout (*C var*), 18
bt_conn_le_create_param.window (*C var*), 18
bt_conn_le_create_param.window_coded (*C var*), 18
BT_CONN_LE_CREATE_PARAM_INIT (*C macro*), 3
bt_conn_le_data_len_info (*C struct*), 15
bt_conn_le_data_len_info.rx_max_len (*C var*), 15
bt_conn_le_data_len_info.rx_max_time (*C var*), 15
bt_conn_le_data_len_info.tx_max_len (*C var*), 15
bt_conn_le_data_len_info.tx_max_time (*C var*), 15
BT_CONN_LE_DATA_LEN_PARAM (*C macro*), 2
bt_conn_le_data_len_param (*C struct*), 15
bt_conn_le_data_len_param.tx_max_len (*C var*), 15
bt_conn_le_data_len_param.tx_max_time (*C var*), 15
BT_CONN_LE_DATA_LEN_PARAM_INIT (*C macro*), 2
bt_conn_le_data_len_update (*C function*), 8
bt_conn_le_get_tx_power_level (*C function*), 8
bt_conn_le_info (*C struct*), 15
bt_conn_le_info.dst (*C var*), 16
bt_conn_le_info.latency (*C var*), 16
bt_conn_le_info.local (*C var*), 16
bt_conn_le_info.phy (*C var*), 16
bt_conn_le_info.remote (*C var*), 16
bt_conn_le_info.src (*C var*), 16
bt_conn_le_info.timeout (*C var*), 16
bt_conn_le_param_update (*C function*), 8
bt_conn_le_phy_info (*C struct*), 14
bt_conn_le_phy_info.rx_phy (*C var*), 15
BT_CONN_LE_PHY_PARAM (*C macro*), 2
bt_conn_le_phy_param (*C struct*), 15
bt_conn_le_phy_param.pref_rx_phy (*C var*), 15
bt_conn_le_phy_param.pref_tx_phy (*C var*), 15
BT_CONN_LE_PHY_PARAM_1M (*C macro*), 2
BT_CONN_LE_PHY_PARAM_2M (*C macro*), 2
BT_CONN_LE_PHY_PARAM_ALL (*C macro*), 2
BT_CONN_LE_PHY_PARAM_CODED (*C macro*), 2
BT_CONN_LE_PHY_PARAM_INIT (*C macro*), 2
bt_conn_le_phy_update (*C function*), 8
bt_conn_le_remote_info (*C struct*), 16
bt_conn_le_remote_info.features (*C var*), 17
bt_conn_le_tx_power (*C struct*), 17
bt_conn_le_tx_power.current_level (*C var*), 18
bt_conn_le_tx_power.max_level (*C var*), 18

bt_conn_le_tx_power.phy (*C var*), 18
bt_conn_le_tx_power_phy (*C enum*), 4
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_1M (*C enumerator*), 4
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_2M (*C enumerator*), 4
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_CODED_S2 (*C enumerator*), 4
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_CODED_S8 (*C enumerator*), 4
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_NONE (*C enumerator*), 4
bt_conn_lookup_addr_le (*C function*), 6
bt_conn_oob_info (*C struct*), 21
bt_conn_oob_info.__unnamed__ (*C union*), 21
bt_conn_oob_info.__unnamed__.lesc (*C struct*), 21
bt_conn_oob_info.__unnamed__.lesc (*C var*), 21
bt_conn_oob_info.__unnamed__.lesc.oob_config (*C var*), 21
bt_conn_oob_info.type (*C var*), 21
bt_conn_oob_info.[anonymous] (*C enum*), 21
bt_conn_oob_info.[anonymous].BT_CONN_OOB_LE_LEGACY (*C enumerator*), 21
bt_conn_oob_info.[anonymous].BT_CONN_OOB_LE_SC (*C enumerator*), 21
bt_conn_pairing_feat (*C struct*), 21
bt_conn_pairing_feat.auth_req (*C var*), 22
bt_conn_pairing_feat.init_key_dist (*C var*), 22
bt_conn_pairing_feat.io_capability (*C var*), 22
bt_conn_pairing_feat.max_enc_key_size (*C var*), 22
bt_conn_pairing_feat.oob_data_flag (*C var*), 22
bt_conn_pairing_feat.resp_key_dist (*C var*), 22
bt_conn_ref (*C function*), 6
bt_conn_remote_info (*C struct*), 17
bt_conn_remote_info.__unnamed__ (*C union*), 17
bt_conn_remote_info.__unnamed__.br (*C var*), 17
bt_conn_remote_info.__unnamed__.le (*C var*), 17
bt_conn_remote_info.manufacturer (*C var*), 17
bt_conn_remote_info.subversion (*C var*), 17
bt_conn_remote_info.type (*C var*), 17
bt_conn_remote_info.version (*C var*), 17
bt_conn_set_security (*C function*), 10
bt_conn_unref (*C function*), 6
BT_DATA (*C macro*), 27
bt_data (*C struct*), 50
BT_DATA_BIG_INFO (*C macro*), 61
BT_DATA_BROADCAST_CODE (*C macro*), 61
BT_DATA_BYTES (*C macro*), 27
BT_DATA_FLAGS (*C macro*), 60
BT_DATA_GAP_APPEARANCE (*C macro*), 61
BT_DATA_LE_BT_DEVICE_ADDRESS (*C macro*), 61
BT_DATA_LE_ROLE (*C macro*), 61
BT_DATA_LE_SC_CONFIRM_VALUE (*C macro*), 61
BT_DATA_LE_SC_RANDOM_VALUE (*C macro*), 61
BT_DATA_MANUFACTURER_DATA (*C macro*), 61
BT_DATA_MESH_BEACON (*C macro*), 61
BT_DATA_MESH_MESSAGE (*C macro*), 61
BT_DATA_MESH_PROV (*C macro*), 61
BT_DATA_NAME_COMPLETE (*C macro*), 60
BT_DATA_NAME_SHORTENED (*C macro*), 60
bt_data_parse (*C function*), 46
BT_DATA_SM_OOB_FLAGS (*C macro*), 60
BT_DATA_SM_TK_VALUE (*C macro*), 60
BT_DATA_SOLICIT128 (*C macro*), 61
BT_DATA_SOLICIT16 (*C macro*), 60
BT_DATA_SOLICIT32 (*C macro*), 61
BT_DATA_SVC_DATA128 (*C macro*), 61
BT_DATA_SVC_DATA16 (*C macro*), 61
BT_DATA_SVC_DATA32 (*C macro*), 61
BT_DATA_TX_POWER (*C macro*), 60
BT_DATA_URI (*C macro*), 61
BT_DATA_UUID128_ALL (*C macro*), 60
BT_DATA_UUID128_SOME (*C macro*), 60
BT_DATA_UUID16_ALL (*C macro*), 60
BT_DATA_UUID16_SOME (*C macro*), 60
BT_DATA_UUID32_ALL (*C macro*), 60
BT_DATA_UUID32_SOME (*C macro*), 60
bt_enable (*C function*), 35
bt_foreach_bond (*C function*), 48
BT_GAP_ADV_FAST_INT_MAX_1 (*C macro*), 61
BT_GAP_ADV_FAST_INT_MAX_2 (*C macro*), 61
BT_GAP_ADV_FAST_INT_MIN_1 (*C macro*), 61
BT_GAP_ADV_FAST_INT_MIN_2 (*C macro*), 61
BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT (*C macro*), 62
BT_GAP_ADV_MAX_ADV_DATA_LEN (*C macro*), 61
BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN (*C macro*), 62
BT_GAP_ADV_SLOW_INT_MAX (*C macro*), 61
BT_GAP_ADV_SLOW_INT_MIN (*C macro*), 61
BT_GAP_DATA_LEN_DEFAULT (*C macro*), 62
BT_GAP_DATA_LEN_MAX (*C macro*), 62
BT_GAP_DATA_TIME_DEFAULT (*C macro*), 62
BT_GAP_DATA_TIME_MAX (*C macro*), 62
BT_GAP_INIT_CONN_INT_MAX (*C macro*), 61
BT_GAP_INIT_CONN_INT_MIN (*C macro*), 61
BT_GAP_NO_TIMEOUT (*C macro*), 62

BT_GAP_PER_ADV_MAX_MAX_SKIP (*C macro*), 62
BT_GAP_PER_ADV_MAX_MAX_TIMEOUT (*C macro*), 62
BT_GAP_RSSI_INVALID (*C macro*), 62
BT_GAP_SCAN_FAST_INTERVAL (*C macro*), 61
BT_GAP_SCAN_FAST_WINDOW (*C macro*), 61
BT_GAP_SCAN_SLOW_INTERVAL_1 (*C macro*), 61
BT_GAP_SCAN_SLOW_INTERVAL_2 (*C macro*), 61
BT_GAP_SCAN_SLOW_WINDOW_1 (*C macro*), 61
BT_GAP_SCAN_SLOW_WINDOW_2 (*C macro*), 61
BT_GAP_SID_INVALID (*C macro*), 62
BT_GAP_SID_MAX (*C macro*), 62
BT_GAP_TX_POWER_INVALID (*C macro*), 62
bt_gatt_attr (*C struct*), 67
bt_gatt_attr.handle (*C var*), 67
bt_gatt_attr.perm (*C var*), 67
bt_gatt_attr.read (*C var*), 67
bt_gatt_attr.user_data (*C var*), 67
bt_gatt_attr.uuid (*C var*), 67
bt_gatt_attr.write (*C var*), 67
bt_gatt_attr_func_t (*C type*), 73
bt_gatt_attr_get_handle (*C function*), 75
bt_gatt_attr_next (*C function*), 75
bt_gatt_attr_read (*C function*), 75
bt_gatt_attr_read_ccc (*C function*), 77
bt_gatt_attr_read_cep (*C function*), 77
bt_gatt_attr_read_chrc (*C function*), 76
bt_gatt_attr_read_cpf (*C function*), 78
bt_gatt_attr_read_cud (*C function*), 78
bt_gatt_attr_read_included (*C function*), 76
bt_gatt_attr_read_service (*C function*), 76
bt_gatt_attr_value_handle (*C function*), 75
bt_gatt_attr_write_ccc (*C function*), 77
BT_GATT_ATTRIBUTE (*C macro*), 72
bt_gatt_cancel (*C function*), 88
BT_GATT_CCC (*C macro*), 71
bt_gatt_ccc (*C struct*), 69
bt_gatt_ccc.flags (*C var*), 69
bt_gatt_ccc_cfg (*C struct*), 81
bt_gatt_ccc_cfg.id (*C var*), 81
bt_gatt_ccc_cfg.peer (*C var*), 81
bt_gatt_ccc_cfg.value (*C var*), 81
BT_GATT_CCC_INDICATE (*C macro*), 66
BT_GATT_CCC_INITIALIZER (*C macro*), 71
BT_GATT_CCC_MANAGED (*C macro*), 71
BT_GATT_CCC_MAX (*C macro*), 71
BT_GATT_CCC_NOTIFY (*C macro*), 65
BT_GATT_CEP (*C macro*), 72
bt_gatt_cep (*C struct*), 69
bt_gatt_cep.properties (*C var*), 69
BT_GATT_CEP_RELIABLE_WRITE (*C macro*), 65
BT_GATT_CEP_WRITABLE_AUX (*C macro*), 65
BT_GATT_CHARACTERISTIC (*C macro*), 71
bt_gatt_chrc (*C struct*), 68
bt_gatt_chrc.properties (*C var*), 69
bt_gatt_chrc.uuid (*C var*), 69
bt_gatt_chrc.value_handle (*C var*), 69
BT_GATT_CHRC_AUTH (*C macro*), 65
BT_GATT_CHRC_BROADCAST (*C macro*), 65
BT_GATT_CHRC_EXT_PROP (*C macro*), 65
BT_GATT_CHRC_INDICATE (*C macro*), 65
BT_GATT_CHRC_INIT (*C macro*), 71
BT_GATT_CHRC_NOTIFY (*C macro*), 65
BT_GATT_CHRC_READ (*C macro*), 65
BT_GATT_CHRC_WRITE (*C macro*), 65
BT_GATT_CHRC_WRITE_WITHOUT_RESP (*C macro*), 65
bt_gatt_complete_func_t (*C type*), 73
BT_GATT_CPF (*C macro*), 72
bt_gatt_cpf (*C struct*), 69
bt_gatt_cpf.description (*C var*), 69
bt_gatt_cpf.exponent (*C var*), 69
bt_gatt_cpf.format (*C var*), 69
bt_gatt_cpf.name_space (*C var*), 69
bt_gatt_cpf.unit (*C var*), 69
BT_GATT_CUD (*C macro*), 72
BT_GATT_DESCRIPTOR (*C macro*), 72
bt_gatt_discover (*C function*), 85
bt_gatt_discover_func_t (*C type*), 83
bt_gatt_discover_params (*C struct*), 89
bt_gatt_discover_params.__unnamed__ (*C union*), 89
bt_gatt_discover_params.__unnamed__._included (*C struct*), 89
bt_gatt_discover_params.__unnamed__._included (*C var*), 89
bt_gatt_discover_params.__unnamed__._included.attr_ (*C var*), 89
bt_gatt_discover_params.__unnamed__._included.end_ (*C var*), 89
bt_gatt_discover_params.__unnamed__._included.start (*C var*), 89
bt_gatt_discover_params.__unnamed__.start_handle (*C var*), 89
bt_gatt_discover_params.end_handle (*C var*), 89
bt_gatt_discover_params.func (*C var*), 89
bt_gatt_discover_params.type (*C var*), 89
bt_gatt_discover_params.uuid (*C var*), 89
BT_GATT_ERR (*C macro*), 65
bt_gatt_exchange_mtu (*C function*), 85
bt_gatt_exchange_params (*C struct*), 88
bt_gatt_exchange_params.func (*C var*), 89
bt_gatt_foreach_attr (*C function*), 75
bt_gatt_foreach_attr_type (*C function*), 74
bt_gatt_get_mtu (*C function*), 81
bt_gatt_include (*C struct*), 68
bt_gatt_include.end_handle (*C var*), 68

bt_gatt_include.start_handle (*C var*), 68
bt_gatt_include.uuid (*C var*), 68
BT_GATT_INCLUDE_SERVICE (*C macro*), 71
bt_gatt_indicate (*C function*), 80
bt_gatt_indicate_func_t (*C type*), 73
bt_gatt_indicate_params (*C struct*), 82
bt_gatt_indicate_params._ref (*C var*), 82
bt_gatt_indicate_params.attr (*C var*), 82
bt_gatt_indicate_params.data (*C var*), 82
bt_gatt_indicate_params.destroy (*C var*), 82
bt_gatt_indicate_params.func (*C var*), 82
bt_gatt_indicate_params.len (*C var*), 82
bt_gatt_indicate_params.uuid (*C var*), 82
bt_gatt_indicate_params_destroy_t (*C type*), 73
bt_gatt_is_subscribed (*C function*), 80
bt_gatt_notify (*C function*), 79
bt_gatt_notify_cb (*C function*), 78
bt_gatt_notify_func_t (*C type*), 84
bt_gatt_notify_multiple (*C function*), 79
bt_gatt_notify_params (*C struct*), 82
bt_gatt_notify_params.attr (*C var*), 82
bt_gatt_notify_params.data (*C var*), 82
bt_gatt_notify_params.func (*C var*), 82
bt_gatt_notify_params.len (*C var*), 82
bt_gatt_notify_params.user_data (*C var*), 82
bt_gatt_notify_params.uuid (*C var*), 82
bt_gatt_notify_uuid (*C function*), 79
BT_GATT_PRIMARY_SERVICE (*C macro*), 70
bt_gatt_read (*C function*), 86
bt_gatt_read_func_t (*C type*), 83
bt_gatt_read_params (*C struct*), 89
bt_gatt_read_params.__unnamed__ (*C union*), 90
bt_gatt_read_params.__unnamed__.by_uuid (*C struct*), 90
bt_gatt_read_params.__unnamed__.by_uuid (*C var*), 90
bt_gatt_read_params.__unnamed__.by_uuid.end_handle (*C var*), 90
bt_gatt_read_params.__unnamed__.by_uuid.start_handle (*C var*), 90
bt_gatt_read_params.__unnamed__.by_uuid.uuid (*C var*), 90
bt_gatt_read_params.__unnamed__.handles (*C var*), 90
bt_gatt_read_params.__unnamed__.single (*C struct*), 90
bt_gatt_read_params.__unnamed__.single (*C var*), 90
bt_gatt_read_params.__unnamed__.single.handle (*C var*), 90
bt_gatt_read_params.__unnamed__.single.offset (*C var*), 90
bt_gatt_read_params.func (*C var*), 90
bt_gatt_read_params.handle_count (*C var*), 90
bt_gatt_resubscribe (*C function*), 88
BT_GATT_SECONDARY_SERVICE (*C macro*), 70
BT_GATT_SERVICE (*C macro*), 70
bt_gatt_service (*C struct*), 68
bt_gatt_service.attr_count (*C var*), 68
bt_gatt_service.attrs (*C var*), 68
BT_GATT_SERVICE_DEFINE (*C macro*), 70
BT_GATT_SERVICE_INSTANCE_DEFINE (*C macro*), 70
bt_gatt_service_register (*C function*), 74
bt_gatt_service_static (*C struct*), 67
bt_gatt_service_static.attr_count (*C var*), 68
bt_gatt_service_static.attrs (*C var*), 68
bt_gatt_service_unregister (*C function*), 74
bt_gatt_service_val (*C struct*), 68
bt_gatt_service_val.end_handle (*C var*), 68
bt_gatt_service_val.uuid (*C var*), 68
bt_gatt_subscribe (*C function*), 87
bt_gatt_subscribe_params (*C struct*), 91
bt_gatt_subscribe_params.ccc_handle (*C var*), 91
bt_gatt_subscribe_params.notify (*C var*), 91
bt_gatt_subscribe_params.value (*C var*), 91
bt_gatt_subscribe_params.value_handle (*C var*), 91
bt_gatt_subscribe_params.write (*C var*), 91
bt_gatt_unsubscribe (*C function*), 88
bt_gatt_write (*C function*), 86
bt_gatt_write_func_t (*C type*), 84
bt_gatt_write_params (*C struct*), 90
bt_gatt_write_params.data (*C var*), 90
bt_gatt_write_params.func (*C var*), 90
bt_gatt_write_params.handle (*C var*), 90
bt_gatt_write_params.length (*C var*), 91
bt_gatt_write_params.offset (*C var*), 90
bt_gatt_write_without_response (*C function*), 87
bt_gatt_write_without_response_cb (*C function*), 86
bt_get_name (*C function*), 35
bt_hfp_ag_call_status_pl (*C function*), 97
bt_hfp_ag_cb (*C struct*), 102
bt_hfp_ag_cb.ata_response (*C var*), 103
bt_hfp_ag_cb.brva (*C var*), 104
bt_hfp_ag_cb.chld (*C var*), 104
bt_hfp_ag_cb.chup_response (*C var*), 103
bt_hfp_ag_cb.codec_negotiate (*C var*), 104

bt_hfp_ag_cb.connected (*C var*), 103
bt_hfp_ag_cb.dial (*C var*), 103
bt_hfp_ag_cb.disconnected (*C var*), 103
bt_hfp_ag_cb.hfu_brsf (*C var*), 103
bt_hfp_ag_cb.nrec (*C var*), 104
bt_hfp_ag_cb.unkown_at (*C var*), 104
bt_hfp_ag_cb.volume_control (*C var*), 103
bt_hfp_ag_close_audio (*C function*), 94
bt_hfp_ag_codec_selector (*C function*), 99
bt_hfp_ag_connect (*C function*), 93
bt_hfp_ag_deinit (*C function*), 93
bt_hfp_ag_disconnect (*C function*), 94
bt_hfp_ag_discover (*C function*), 94
bt_hfp_ag_discover_callback (*C type*), 92
bt_hfp_ag_get_peer_supp_features (*C function*), 95
bt_hfp_ag_handle_btrh (*C function*), 97
bt_hfp_ag_handle_indicator_enable (*C function*), 97
bt_hfp_ag_init (*C function*), 93
bt_hfp_ag_open_audio (*C function*), 94
bt_hfp_ag_register_cind_features (*C function*), 95
bt_hfp_ag_register_supp_features (*C function*), 95
bt_hfp_ag_send_battery_indicator (*C function*), 99
bt_hfp_ag_send_call_indicator (*C function*), 98
bt_hfp_ag_send_callring (*C function*), 98
bt_hfp_ag_send_callsetup_indicator (*C function*), 98
bt_hfp_ag_send_ccwa_indicator (*C function*), 99
bt_hfp_ag_send_disable_voice_ecnr (*C function*), 96
bt_hfp_ag_send_disable_voice_recognition (*C function*), 95
bt_hfp_ag_send_enable_voice_ecnr (*C function*), 96
bt_hfp_ag_send_enable_voice_recognition (*C function*), 95
bt_hfp_ag_send_roaming_indicator (*C function*), 99
bt_hfp_ag_send_service_indicator (*C function*), 98
bt_hfp_ag_send_signal_indicator (*C function*), 98
bt_hfp_ag_set_cops (*C function*), 96
bt_hfp_ag_set_inband_ring_tone (*C function*), 97
bt_hfp_ag_set_phnum_tag (*C function*), 97
bt_hfp_ag_set_volume_control (*C function*), 96
bt_hfp_ag_unknown_at_response (*C function*), 99
bt_hfp_hf_at_cmd (*C enum*), 93
bt_hfp_hf_at_cmd.BT_HFP_HF_AT_CHUP (*C enumerator*), 93
bt_hfp_hf_at_cmd.BT_HFP_HF_ATA (*C enumerator*), 93
bt_hfp_hf_cb (*C struct*), 105
bt_hfp_hf_cb.battery (*C var*), 106
bt_hfp_hf_cb.call (*C var*), 105
bt_hfp_hf_cb.call_held (*C var*), 105
bt_hfp_hf_cb.call_phnum (*C var*), 106
bt_hfp_hf_cb.call_setup (*C var*), 105
bt_hfp_hf_cb.cmd_complete_cb (*C var*), 107
bt_hfp_hf_cb.connected (*C var*), 105
bt_hfp_hf_cb.disconnected (*C var*), 105
bt_hfp_hf_cb.ring_indication (*C var*), 107
bt_hfp_hf_cb.roam (*C var*), 106
bt_hfp_hf_cb.service (*C var*), 105
bt_hfp_hf_cb.signal (*C var*), 106
bt_hfp_hf_cb.voicetag_phnum (*C var*), 106
bt_hfp_hf_cb.waiting_call (*C var*), 106
bt_hfp_hf_cmd_complete (*C struct*), 104
bt_hfp_hf_dial (*C function*), 101
bt_hfp_hf_dial_memory (*C function*), 101
bt_hfp_hf_disable_call_waiting_notification (*C function*), 102
bt_hfp_hf_disable_clip_notification (*C function*), 102
bt_hfp_hf_enable_call_waiting_notification (*C function*), 102
bt_hfp_hf_enable_clip_notification (*C function*), 101
bt_hfp_hf_get_last_voice_tag_number (*C function*), 102
bt_hfp_hf_last_dial (*C function*), 101
bt_hfp_hf_multiparty_call_option (*C function*), 101
bt_hfp_hf_register (*C function*), 100
bt_hfp_hf_send_cmd (*C function*), 100
bt_hfp_hf_start_voice_recognition (*C function*), 100
bt_hfp_hf_stop_voice_recognition (*C function*), 100
bt_hfp_hf_volume_update (*C function*), 100
bt_hps_init (*C function*), 157
bt_hps_notify (*C function*), 158
bt_hps_set_status_code (*C function*), 157
bt_hts_indicate (*C function*), 159
bt_id_create (*C function*), 36
BT_ID_DEFAULT (*C macro*), 27
bt_id_delete (*C function*), 37
bt_id_get (*C function*), 35
bt_id_reset (*C function*), 36

bt_l2cap_br_chan (*C struct*), 113
bt_l2cap_br_chan.chan (*C var*), 113
bt_l2cap_br_chan.rx (*C var*), 113
bt_l2cap_br_chan.tx (*C var*), 113
bt_l2cap_br_endpoint (*C struct*), 112
bt_l2cap_br_endpoint.cid (*C var*), 113
bt_l2cap_br_endpoint.mtu (*C var*), 113
bt_l2cap_br_server_register (*C function*), 110
BT_L2CAP_BUF_SIZE (*C macro*), 108
BT_L2CAP_CFG_OPT_EXT_FLOW_SPEC (*C macro*), 108
BT_L2CAP_CFG_OPT_EXT_WIN_SIZE (*C macro*), 108
BT_L2CAP_CFG_OPT_FCS (*C macro*), 108
BT_L2CAP_CFG_OPT_FUSH_TIMEOUT (*C macro*), 108
BT_L2CAP_CFG_OPT_MTU (*C macro*), 108
BT_L2CAP_CFG_OPT_QOS (*C macro*), 108
BT_L2CAP_CFG_OPT_RETRANS_FC (*C macro*), 108
bt_l2cap_cfg_options (*C struct*), 113
bt_l2cap_chan (*C struct*), 111
bt_l2cap_chan.conn (*C var*), 112
bt_l2cap_chan.ops (*C var*), 112
bt_l2cap_chan_connect (*C function*), 110
bt_l2cap_chan_destroy_t (*C type*), 109
bt_l2cap_chan_disconnect (*C function*), 111
bt_l2cap_chan_ops (*C struct*), 113
bt_l2cap_chan_ops.alloc_buf (*C var*), 114
bt_l2cap_chan_ops.connected (*C var*), 113
bt_l2cap_chan_ops.disconnected (*C var*), 113
bt_l2cap_chan_ops.encrypt_change (*C var*), 113
bt_l2cap_chan_ops.recv (*C var*), 114
bt_l2cap_chan_ops.sent (*C var*), 114
bt_l2cap_chan_ops.status (*C var*), 114
bt_l2cap_chan_recv_complete (*C function*), 111
bt_l2cap_chan_send (*C function*), 111
BT_L2CAP_CHAN_SEND_RESERVE (*C macro*), 109
bt_l2cap_chan_state (*C enum*), 109
bt_l2cap_chan_state.BT_L2CAP_CONFIG (*C enumerator*), 109
bt_l2cap_chan_state.BT_L2CAP_CONNECT (*C enumerator*), 109
bt_l2cap_chan_state.BT_L2CAP_CONNECTED (*C enumerator*), 109
bt_l2cap_chan_state.BT_L2CAP_DISCONNECT (*C enumerator*), 109
bt_l2cap_chan_state.BT_L2CAP_DISCONNECTED (*C enumerator*), 109
bt_l2cap_chan_state_t (*C type*), 109
bt_l2cap_chan_status (*C enum*), 109
bt_l2cap_chan_status.BT_L2CAP_NUM_STATUS (*C enumerator*), 110
bt_l2cap_chan_status.BT_L2CAP_STATUS_ENCRYPT_PENDIN (*C enumerator*), 110
bt_l2cap_chan_status.BT_L2CAP_STATUS_OUT (*C enumerator*), 109
bt_l2cap_chan_status.BT_L2CAP_STATUS_SHUTDOWN (*C enumerator*), 109
bt_l2cap_chan_status_t (*C type*), 109
bt_l2cap_ecred_chan_connect (*C function*), 110
bt_l2cap_ext_flow_spec (*C struct*), 113
BT_L2CAP_FEATURE_EFS_BR_EDR (*C macro*), 109
BT_L2CAP_FEATURE_ERTM (*C macro*), 108
BT_L2CAP_FEATURE_EXTENDED_WINDOW_SIZE (*C macro*), 109
BT_L2CAP_FEATURE_FC (*C macro*), 108
BT_L2CAP_FEATURE_FCS (*C macro*), 109
BT_L2CAP_FEATURE_FIXED_CHANNELS (*C macro*), 109
BT_L2CAP_FEATURE_QOS (*C macro*), 108
BT_L2CAP_FEATURE_RTM (*C macro*), 108
BT_L2CAP_FEATURE_SM (*C macro*), 109
BT_L2CAP_FEATURE_UCD (*C macro*), 109
BT_L2CAP_HDR_SIZE (*C macro*), 108
BT_L2CAP_LE_CHAN (*C macro*), 108
bt_l2cap_le_chan (*C struct*), 112
bt_l2cap_le_chan._sdu (*C var*), 112
bt_l2cap_le_chan.chan (*C var*), 112
bt_l2cap_le_chan.rx (*C var*), 112
bt_l2cap_le_chan.tx (*C var*), 112
bt_l2cap_le_chan.tx_buf (*C var*), 112
bt_l2cap_le_chan.tx_queue (*C var*), 112
bt_l2cap_le_chan.tx_work (*C var*), 112
bt_l2cap_le_endpoint (*C struct*), 112
bt_l2cap_le_endpoint.cid (*C var*), 112
bt_l2cap_le_endpoint.credits (*C var*), 112
bt_l2cap_le_endpoint.init_credits (*C var*), 112
bt_l2cap_le_endpoint.mps (*C var*), 112
bt_l2cap_le_endpoint.mtu (*C var*), 112
BT_L2CAP_MODE_BASIC (*C macro*), 108
BT_L2CAP_MODE_ERTM (*C macro*), 108
BT_L2CAP_MODE_FC (*C macro*), 108
BT_L2CAP_MODE_RTM (*C macro*), 108
BT_L2CAP_MODE_SM (*C macro*), 108
bt_l2cap_qos (*C struct*), 113
bt_l2cap_retrans_fc (*C struct*), 113
bt_l2cap_server (*C struct*), 114
bt_l2cap_server.accept (*C var*), 115
bt_l2cap_server.psm (*C var*), 115
bt_l2cap_server.sec_level (*C var*), 115
bt_l2cap_server_register (*C function*), 110
BT_LE_AD_GENERAL (*C macro*), 61

BT_LE_AD_LIMITED (*C macro*), 61
BT_LE_AD_NO_BREDR (*C macro*), 61
BT_LE_ADV_CONN (*C macro*), 28
BT_LE_ADV_CONN_DIR (*C macro*), 28
BT_LE_ADV_CONN_DIR_LOW_DUTY (*C macro*), 28
BT_LE_ADV_CONN_NAME (*C macro*), 28
BT_LE_ADV_NCONN (*C macro*), 28
BT_LE_ADV_NCONN_IDENTITY (*C macro*), 28
BT_LE_ADV_NCONN_NAME (*C macro*), 28
BT_LE_ADV_PARAM (*C macro*), 28
bt_le_adv_param (*C struct*), 50
bt_le_adv_param.id (*C var*), 50
bt_le_adv_param.interval_max (*C var*), 50
bt_le_adv_param.interval_min (*C var*), 50
bt_le_adv_param.options (*C var*), 50
bt_le_adv_param.peer (*C var*), 50
bt_le_adv_param.secondary_max_skip (*C var*), 50
bt_le_adv_param.sid (*C var*), 50
BT_LE_ADV_PARAM_INIT (*C macro*), 28
bt_le_adv_start (*C function*), 37
bt_le_adv_stop (*C function*), 38
bt_le_adv_update_data (*C function*), 37
BT_LE_CONN_PARAM (*C macro*), 1
bt_le_conn_param (*C struct*), 14
BT_LE_CONN_PARAM_DEFAULT (*C macro*), 2
BT_LE_CONN_PARAM_INIT (*C macro*), 1
BT_LE_DATA_LEN_PARAM_DEFAULT (*C macro*), 2
BT_LE_DATA_LEN_PARAM_MAX (*C macro*), 3
bt_le_ext_adv_cb (*C struct*), 49
bt_le_ext_adv_cb.connected (*C var*), 49
bt_le_ext_adv_cb.scanned (*C var*), 49
bt_le_ext_adv_cb.sent (*C var*), 49
BT_LE_EXT_ADV_CODED_NCONN (*C macro*), 29
BT_LE_EXT_ADV_CODED_NCONN_IDENTITY (*C macro*), 29
BT_LE_EXT_ADV_CODED_NCONN_NAME (*C macro*), 29
bt_le_ext_adv_connected_info (*C struct*), 49
bt_le_ext_adv_connected_info.conn (*C var*), 49
bt_le_ext_adv_create (*C function*), 38
bt_le_ext_adv_delete (*C function*), 39
bt_le_ext_adv_get_index (*C function*), 40
bt_le_ext_adv_get_info (*C function*), 40
bt_le_ext_adv_info (*C struct*), 51
bt_le_ext_adv_info.tx_power (*C var*), 51
BT_LE_EXT_ADV_NCONN (*C macro*), 28
BT_LE_EXT_ADV_NCONN_IDENTITY (*C macro*), 28
BT_LE_EXT_ADV_NCONN_NAME (*C macro*), 28
bt_le_ext_adv_oob_get_local (*C function*), 46
bt_le_ext_adv_scanned_info (*C struct*), 49
bt_le_ext_adv_scanned_info.addr (*C var*), 49
bt_le_ext_adv_sent_info (*C struct*), 48
bt_le_ext_adv_sent_info.num_sent (*C var*), 49
bt_le_ext_adv_set_data (*C function*), 39
bt_le_ext_adv_start (*C function*), 38
BT_LE_EXT_ADV_START_DEFAULT (*C macro*), 29
BT_LE_EXT_ADV_START_PARAM (*C macro*), 29
bt_le_ext_adv_start_param (*C struct*), 51
bt_le_ext_adv_start_param.num_events (*C var*), 51
bt_le_ext_adv_start_param.timeout (*C var*), 51
BT_LE_EXT_ADV_START_PARAM_INIT (*C macro*), 29
bt_le_ext_adv_stop (*C function*), 38
bt_le_ext_adv_update_param (*C function*), 39
bt_le_oob (*C struct*), 56
bt_le_oob.addr (*C var*), 56
bt_le_oob.le_sc_data (*C var*), 56
bt_le_oob_get_local (*C function*), 46
bt_le_oob_get_sc_data (*C function*), 12
bt_le_oob_sc_data (*C struct*), 56
bt_le_oob_sc_data.c (*C var*), 56
bt_le_oob_sc_data.r (*C var*), 56
bt_le_oob_set_legacy_tk (*C function*), 11
bt_le_oob_set_sc_data (*C function*), 12
BT_LE_PER_ADV_DEFAULT (*C macro*), 29
bt_le_per_adv_list_add (*C function*), 43
bt_le_per_adv_list_clear (*C function*), 44
bt_le_per_adv_list_remove (*C function*), 44
BT_LE_PER_ADV_PARAM (*C macro*), 29
bt_le_per_adv_param (*C struct*), 51
bt_le_per_adv_param.interval_max (*C var*), 51
bt_le_per_adv_param.interval_min (*C var*), 51
bt_le_per_adv_param.options (*C var*), 51
BT_LE_PER_ADV_PARAM_INIT (*C macro*), 29
bt_le_per_adv_set_data (*C function*), 40
bt_le_per_adv_set_info_transfer (*C function*), 43
bt_le_per_adv_set_param (*C function*), 40
bt_le_per_adv_start (*C function*), 40
bt_le_per_adv_stop (*C function*), 41
bt_le_per_adv_sync_cb (*C struct*), 53
bt_le_per_adv_sync_cb.recv (*C var*), 53
bt_le_per_adv_sync_cb.state_changed (*C var*), 53
bt_le_per_adv_sync_cb.synced (*C var*), 53
bt_le_per_adv_sync_cb.term (*C var*), 53
bt_le_per_adv_sync_cb_register (*C function*), 42
bt_le_per_adv_sync_create (*C function*), 41
bt_le_per_adv_sync_delete (*C function*), 41

bt_le_per_adv_sync_get_index (*C function*), 41
bt_le_per_adv_sync_param (*C struct*), 53
bt_le_per_adv_sync_param.addr (*C var*), 54
bt_le_per_adv_sync_param.options (*C var*), 54
bt_le_per_adv_sync_param.sid (*C var*), 54
bt_le_per_adv_sync_param.skip (*C var*), 54
bt_le_per_adv_sync_param.timeout (*C var*), 54
bt_le_per_adv_sync_recv_disable (*C function*), 42
bt_le_per_adv_sync_recv_enable (*C function*), 42
bt_le_per_adv_sync_recv_info (*C struct*), 52
bt_le_per_adv_sync_recv_info.addr (*C var*), 52
bt_le_per_adv_sync_recv_info.cte_type (*C var*), 52
bt_le_per_adv_sync_recv_info.rssi (*C var*), 52
bt_le_per_adv_sync_recv_info.sid (*C var*), 52
bt_le_per_adv_sync_recv_info.tx_power (*C var*), 52
bt_le_per_adv_sync_state_info (*C struct*), 52
bt_le_per_adv_sync_state_info.recv_enabled (*C var*), 53
bt_le_per_adv_sync_synced_info (*C struct*), 51
bt_le_per_adv_sync_synced_info.addr (*C var*), 52
bt_le_per_adv_sync_synced_info.conn (*C var*), 52
bt_le_per_adv_sync_synced_info.interval (*C var*), 52
bt_le_per_adv_sync_synced_info.phy (*C var*), 52
bt_le_per_adv_sync_synced_info.recv_enabled (*C var*), 52
bt_le_per_adv_sync_synced_info.service_data (*C var*), 52
bt_le_per_adv_sync_synced_info.sid (*C var*), 52
bt_le_per_adv_sync_term_info (*C struct*), 52
bt_le_per_adv_sync_term_info.addr (*C var*), 52
bt_le_per_adv_sync_term_info.sid (*C var*), 52
bt_le_per_adv_sync_transfer (*C function*), 42
bt_le_per_adv_sync_transfer_param (*C struct*), 54
bt_le_per_adv_sync_transfer_param.options (*C var*), 54
bt_le_per_adv_sync_transfer_param.skip (*C var*), 54
bt_le_per_adv_sync_transfer_param.timeout (*C var*), 54
bt_le_per_adv_sync_transfer_subscribe (*C function*), 43
bt_le_per_adv_sync_transfer_unsubscribe (*C function*), 43
BT_LE_SCAN_ACTIVE (*C macro*), 30
bt_le_scan_cb (*C struct*), 55
bt_le_scan_cb.recv (*C var*), 56
bt_le_scan_cb.timeout (*C var*), 56
bt_le_scan_cb_register (*C function*), 44
bt_le_scan_cb_t (*C type*), 30
bt_le_scan_cb_unregister (*C function*), 45
BT_LE_SCAN_CODED_ACTIVE (*C macro*), 30
BT_LE_SCAN_CODED_PASSIVE (*C macro*), 30
BT_LE_SCAN_PARAM (*C macro*), 30
bt_le_scan_param (*C struct*), 54
bt_le_scan_param.interval (*C var*), 54
bt_le_scan_param.interval_coded (*C var*), 55
bt_le_scan_param.options (*C var*), 54
bt_le_scan_param.timeout (*C var*), 54
bt_le_scan_param.type (*C var*), 54
bt_le_scan_param.window (*C var*), 54
bt_le_scan_param.window_coded (*C var*), 55
BT_LE_SCAN_PARAM_INIT (*C macro*), 29
BT_LE_SCAN_PASSIVE (*C macro*), 30
bt_le_scan_recv_info (*C struct*), 55
bt_le_scan_recv_info.addr (*C var*), 55
bt_le_scan_recv_info.adv_props (*C var*), 55
bt_le_scan_recv_info.adv_type (*C var*), 55
bt_le_scan_recv_info.interval (*C var*), 55
bt_le_scan_recv_info.primary_phy (*C var*), 55
bt_le_scan_recv_info.rssi (*C var*), 55
bt_le_scan_recv_info.secondary_phy (*C var*), 55
bt_le_scan_recv_info.sid (*C var*), 55
bt_le_scan_recv_info.tx_power (*C var*), 55
bt_le_scan_start (*C function*), 44
bt_le_scan_stop (*C function*), 44
bt_le_set_auto_conn (*C function*), 10
bt_le_set_chan_map (*C function*), 46
bt_le_whitelist_add (*C function*), 45
bt_le_whitelist_clear (*C function*), 45
bt_le_whitelist_rem (*C function*), 45
BT_PASSKEY_INVALID (*C macro*), 3
bt_passkey_set (*C function*), 12
bt_ready_cb_t (*C type*), 30
bt_rfcomm_create_pdu (*C function*), 117
bt_rfcomm_dlc (*C struct*), 117
bt_rfcomm_dlc_connect (*C function*), 116

bt_rfcomm_dlc_disconnect (*C function*), 116
bt_rfcomm_dlc_ops (*C struct*), 117
bt_rfcomm_dlc_ops.connected (*C var*), 117
bt_rfcomm_dlc_ops.disconnected (*C var*), 117
bt_rfcomm_dlc_ops.recv (*C var*), 117
bt_rfcomm_dlc_ops.sent (*C var*), 117
bt_rfcomm_dlc_send (*C function*), 116
bt_rfcomm_role (*C enum*), 115
bt_rfcomm_role.BT_RFCOMM_ROLE_ACCEPTOR (*C enumerator*), 116
bt_rfcomm_role.BT_RFCOMM_ROLE_INITIATOR (*C enumerator*), 116
bt_rfcomm_role_t (*C type*), 115
bt_rfcomm_server (*C struct*), 117
bt_rfcomm_server.accept (*C var*), 118
bt_rfcomm_server.channel (*C var*), 118
bt_rfcomm_server_register (*C function*), 116
BT_SDP_ADVANCED_AUDIO_SVCLASS (*C macro*), 118
BT_SDP_ALT16 (*C macro*), 123
BT_SDP_ALT32 (*C macro*), 123
BT_SDP_ALT8 (*C macro*), 123
BT_SDP_ALT_UNSPEC (*C macro*), 123
BT_SDP_APPLE_AGENT_SVCLASS (*C macro*), 120
BT_SDP_ARRAY_16 (*C macro*), 123
BT_SDP_ARRAY_32 (*C macro*), 123
BT_SDP_ARRAY_8 (*C macro*), 123
BT_SDP_ATTR_ADD_PROTO_DESC_LIST (*C macro*), 121
BT_SDP_ATTR_AUDIO_FEEDBACK_SUPPORT (*C macro*), 121
BT_SDP_ATTR_BROWSE_GRP_LIST (*C macro*), 120
BT_SDP_ATTR_CLNT_EXEC_URL (*C macro*), 121
BT_SDP_ATTR_DATA_EXCHANGE_SPEC (*C macro*), 121
BT_SDP_ATTR_DOC_URL (*C macro*), 120
BT_SDP_ATTR_EXTERNAL_NETWORK (*C macro*), 121
BT_SDP_ATTR_FAX_CLASS1_SUPPORT (*C macro*), 121
BT_SDP_ATTR_FAX_CLASS20_SUPPORT (*C macro*), 121
BT_SDP_ATTR_FAX_CLASS2_SUPPORT (*C macro*), 121
BT_SDP_ATTR_GOEP_L2CAP_PSM (*C macro*), 121
BT_SDP_ATTR_GROUP_ID (*C macro*), 121
BT_SDP_ATTR_HID_BATTERY_POWER (*C macro*), 122
BT_SDP_ATTR_HID_BOOT_DEVICE (*C macro*), 122
BT_SDP_ATTR_HID_COUNTRY_CODE (*C macro*), 122
BT_SDP_ATTR_HID_DESCRIPTOR_LIST (*C macro*), 122
BT_SDP_ATTR_HID_DEVICE_RELEASE_NUMBER (*C macro*), 122
BT_SDP_ATTR_HID_DEVICE_SUBCLASS (*C macro*), 122
BT_SDP_ATTR_HID_LANG_ID_BASE_LIST (*C macro*), 122
BT_SDP_ATTR_HID_NORMALLY_CONNECTABLE (*C macro*), 122
BT_SDP_ATTR_HID_PARSER_VERSION (*C macro*), 122
BT_SDP_ATTR_HID_PROFILE_VERSION (*C macro*), 122
BT_SDP_ATTR_HID_RECONNECT_INITIATE (*C macro*), 122
BT_SDP_ATTR_HID_REMOTE_WAKEUP (*C macro*), 122
BT_SDP_ATTR_HID_SDP_DISABLE (*C macro*), 122
BT_SDP_ATTR_HID_SUPERVISION_TIMEOUT (*C macro*), 122
BT_SDP_ATTR_HID_VIRTUAL_CABLE (*C macro*), 122
BT_SDP_ATTR_HOMEPAGE_URL (*C macro*), 121
BT_SDP_ATTR_ICON_URL (*C macro*), 121
BT_SDP_ATTR_IP4_SUBNET (*C macro*), 121
BT_SDP_ATTR_IP6_SUBNET (*C macro*), 121
BT_SDP_ATTR_IP_SUBNET (*C macro*), 121
BT_SDP_ATTR_LANG_BASE_ATTR_ID_LIST (*C macro*), 120
BT_SDP_ATTR_MAP_SUPPORTED_FEATURES (*C macro*), 122
BT_SDP_ATTR_MAS_INSTANCE_ID (*C macro*), 122
BT_SDP_ATTR_MAX_NET_ACCESSRATE (*C macro*), 121
BT_SDP_ATTR_MCAP_SUPPORTED_PROCEDURES (*C macro*), 121
BT_SDP_ATTR_MPMD_SCENARIOS (*C macro*), 121
BT_SDP_ATTR_MPS_DEPENDENCIES (*C macro*), 121
BT_SDP_ATTR_MPSD_SCENARIOS (*C macro*), 121
BT_SDP_ATTR_NET_ACCESS_TYPE (*C macro*), 121
BT_SDP_ATTR_NETWORK (*C macro*), 121
BT_SDP_ATTR_NETWORK_ADDRESS (*C macro*), 121
BT_SDP_ATTR_PBAP_SUPPORTED_FEATURES (*C macro*), 122
BT_SDP_ATTR_PRIMARY_RECORD (*C macro*), 122
BT_SDP_ATTR_PRODUCT_ID (*C macro*), 122
BT_SDP_ATTR_PROFILE_DESC_LIST (*C macro*), 120
BT_SDP_ATTR_PROTO_DESC_LIST (*C macro*), 120
BT_SDP_ATTR_PROVNAME_PRIMARY (*C macro*), 122
BT_SDP_ATTR_RECORD_HANDLE (*C macro*), 120
BT_SDP_ATTR_RECORD_STATE (*C macro*), 120
BT_SDP_ATTR_REMOTE_AUDIO_VOLUME_CONTROL (*C macro*), 121
BT_SDP_ATTR_SECURITY_DESC (*C macro*), 121
BT_SDP_ATTR_SERVICE_AVAILABILITY (*C macro*), 120

BT_SDP_ATTR_SERVICE_ID (*C macro*), 120
BT_SDP_ATTR_SERVICE_VERSION (*C macro*), 121
BT_SDP_ATTR_SPECIFICATION_ID (*C macro*), 122
BT_SDP_ATTR_SUPPORTED_CAPABILITIES (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_DATA_STORES_LIST (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_FEATURES (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_FEATURES_LIST (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_FORMATS_LIST (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_FUNCTIONS (*C macro*), 121
BT_SDP_ATTR_SUPPORTED_MESSAGE_TYPES (*C macro*), 122
BT_SDP_ATTR_SUPPORTED_REPOSITORIES (*C macro*), 122
BT_SDP_ATTR_SVCDB_STATE (*C macro*), 121
BT_SDP_ATTR_SVCDESC_PRIMARY (*C macro*), 122
BT_SDP_ATTR_SVCINFO_TTL (*C macro*), 120
BT_SDP_ATTR_SVCLASS_ID_LIST (*C macro*), 120
BT_SDP_ATTR_SVCNAME_PRIMARY (*C macro*), 122
BT_SDP_ATTR_TOTAL_IMAGING_DATA_CAPACITY (*C macro*), 122
BT_SDP_ATTR_VENDOR_ID (*C macro*), 122
BT_SDP_ATTR_VENDOR_ID_SOURCE (*C macro*), 122
BT_SDP_ATTR_VERSION (*C macro*), 122
BT_SDP_ATTR_VERSION_NUM_LIST (*C macro*), 121
BT_SDP_ATTR_WAP_GATEWAY (*C macro*), 121
BT_SDP_ATTR_WAP_STACK_TYPE (*C macro*), 121
bt_sdp_attribute (*C struct*), 128
BT_SDP_AUDIO_SINK_SVCLASS (*C macro*), 118
BT_SDP_AUDIO_SOURCE_SVCLASS (*C macro*), 118
BT_SDP_AV_REMOTE_CONTROLLER_SVCLASS (*C macro*), 119
BT_SDP_AV_REMOTE_SVCLASS (*C macro*), 118
BT_SDP_AV_REMOTE_TARGET_SVCLASS (*C macro*), 118
BT_SDP_AV_SVCLASS (*C macro*), 119
BT_SDP_BASIC_PRINTING_SVCLASS (*C macro*), 119
BT_SDP_BOOL (*C macro*), 123
BT_SDP_BROWSE_GRP_DESC_SVCLASS (*C macro*), 118
BT_SDP_CIP_SVCLASS (*C macro*), 119
bt_sdp_client_result (*C struct*), 128
BT_SDP_CORDLESS_TELEPHONY_SVCLASS (*C macro*), 118
bt_sdp_data_elem (*C struct*), 128
BT_SDP_DATA_ELEM_LIST (*C macro*), 124
BT_SDP_DATA_NIL (*C macro*), 122
BT_SDP_DIALUP_NET_SVCLASS (*C macro*), 118
BT_SDP_DIRECT_PRINTING_SVCLASS (*C macro*), 119
BT_SDP_DIRECT_PRT_REFOBJS_SVCLASS (*C macro*), 119
bt_sdp_discover (*C function*), 126
bt_sdp_discover_cancel (*C function*), 126
bt_sdp_discover_func_t (*C type*), 125
bt_sdp_discover_params (*C struct*), 128
bt_sdp_discover_params.func (*C var*), 128
bt_sdp_discover_params.pool (*C var*), 128
bt_sdp_discover_params.uuid (*C var*), 128
BT_SDP_FAX_SVCLASS (*C macro*), 119
BT_SDP_GENERIC_ACCESS_SVCLASS (*C macro*), 120
BT_SDP_GENERIC_ATTRIB_SVCLASS (*C macro*), 120
BT_SDP_GENERIC_AUDIO_SVCLASS (*C macro*), 120
BT_SDP_GENERIC_FILETRANS_SVCLASS (*C macro*), 120
BT_SDP_GENERIC_NETWORKING_SVCLASS (*C macro*), 120
BT_SDP_GENERIC_TELEPHONY_SVCLASS (*C macro*), 120
bt_sdp_get_addl_proto_param (*C function*), 127
bt_sdp_get_features (*C function*), 128
bt_sdp_get_profile_version (*C function*), 127
bt_sdp_get_proto_param (*C function*), 127
BT_SDP_GN_SVCLASS (*C macro*), 119
BT_SDP_GNSS_SERVER_SVCLASS (*C macro*), 120
BT_SDP_GNSS_SVCLASS (*C macro*), 120
BT_SDP_HANDSFREE_AGW_SVCLASS (*C macro*), 119
BT_SDP_HANDSFREE_SVCLASS (*C macro*), 119
BT_SDP_HCR_PRINT_SVCLASS (*C macro*), 119
BT_SDP_HCR_SCAN_SVCLASS (*C macro*), 119
BT_SDP_HCR_SVCLASS (*C macro*), 119
BT_SDP_HDP_SINK_SVCLASS (*C macro*), 120
BT_SDP_HDP_SOURCE_SVCLASS (*C macro*), 120
BT_SDP_HDP_SVCLASS (*C macro*), 120
BT_SDP_HEADSET_AGW_SVCLASS (*C macro*), 119
BT_SDP_HEADSET_SVCLASS (*C macro*), 118
BT_SDP_HID_SVCLASS (*C macro*), 119
BT_SDP_IMAGING_ARCHIVE_SVCLASS (*C macro*), 119
BT_SDP_IMAGING_REFOBJS_SVCLASS (*C macro*), 119
BT_SDP_IMAGING_RESPONDER_SVCLASS (*C macro*), 119
BT_SDP_IMAGING_SVCLASS (*C macro*), 119
BT_SDP_INT128 (*C macro*), 123
BT_SDP_INT16 (*C macro*), 123
BT_SDP_INT32 (*C macro*), 123
BT_SDP_INT64 (*C macro*), 123
BT_SDP_INT8 (*C macro*), 123

BT_SDP_INTERCOM_SVCLASS (*C macro*), 119
BT_SDP_IRMC_SYNC_CMD_SVCLASS (*C macro*), 118
BT_SDP_IRMC_SYNC_SVCLASS (*C macro*), 118
BT_SDP_LAN_ACCESS_SVCLASS (*C macro*), 118
BT_SDP_LIST (*C macro*), 124
BT_SDP_MAP_MCE_SVCLASS (*C macro*), 119
BT_SDP_MAP_MSE_SVCLASS (*C macro*), 119
BT_SDP_MAP_SVCLASS (*C macro*), 120
BT_SDP_MPS_SC_SVCLASS (*C macro*), 120
BT_SDP_MPS_SVCLASS (*C macro*), 120
BT_SDP_NAP_SVCLASS (*C macro*), 119
BT_SDP_NEW_SERVICE (*C macro*), 124
BT_SDP_OBEX_FILETRANS_SVCLASS (*C macro*), 118
BT_SDP_OBEX_OBJPUSH_SVCLASS (*C macro*), 118
BT_SDP_PANU_SVCLASS (*C macro*), 119
BT_SDP_PBAP_PCE_SVCLASS (*C macro*), 119
BT_SDP_PBAP_PSE_SVCLASS (*C macro*), 119
BT_SDP_PBAP_SVCLASS (*C macro*), 119
BT_SDP_PNP_INFO_SVCLASS (*C macro*), 120
BT_SDP_PRIMARY_LANG_BASE (*C macro*), 122
BT_SDP_PRINTING_STATUS_SVCLASS (*C macro*), 119
bt_sdp_proto (*C enum*), 126
bt_sdp_proto.BT_SDP_PROTO_L2CAP (*C enumerator*), 126
bt_sdp_proto.BT_SDP_PROTO_RFCOMM (*C enumerator*), 126
BT_SDP_PUBLIC_BROWSE_GROUP (*C macro*), 118
BT_SDP_RECORD (*C macro*), 125
bt_sdp_record (*C struct*), 128
BT_SDP_REFERENCE_PRINTING_SVCLASS (*C macro*), 119
BT_SDP_REFLECTED_UI_SVCLASS (*C macro*), 119
bt_sdp_register_service (*C function*), 126
BT_SDP_SAP_SVCLASS (*C macro*), 119
BT_SDP_SDP_SERVER_SVCLASS (*C macro*), 118
BT_SDP_SEQ16 (*C macro*), 123
BT_SDP_SEQ32 (*C macro*), 123
BT_SDP_SEQ8 (*C macro*), 123
BT_SDP_SEQ_UNSPEC (*C macro*), 123
BT_SDP_SERIAL_PORT_SVCLASS (*C macro*), 118
BT_SDP_SERVER_RECORD_HANDLE (*C macro*), 120
BT_SDP_SERVICE_ID (*C macro*), 124
BT_SDP_SERVICE_NAME (*C macro*), 124
BT_SDP_SIZE_DESC_MASK (*C macro*), 123
BT_SDP_SIZE_INDEX_OFFSET (*C macro*), 123
BT_SDP_SUPPORTED_FEATURES (*C macro*), 124
BT_SDP_TEXT_STR16 (*C macro*), 123
BT_SDP_TEXT_STR32 (*C macro*), 123
BT_SDP_TEXT_STR8 (*C macro*), 123
BT_SDP_TEXT_STR_UNSPEC (*C macro*), 123
BT_SDP_TYPE_DESC_MASK (*C macro*), 123
BT_SDP_TYPE_SIZE (*C macro*), 123
BT_SDP_TYPE_SIZE_VAR (*C macro*), 124
BT_SDP_UDI_MT_SVCLASS (*C macro*), 119
BT_SDP_UDI_TA_SVCLASS (*C macro*), 119
BT_SDP_UINT128 (*C macro*), 123
BT_SDP_UINT16 (*C macro*), 122
BT_SDP_UINT32 (*C macro*), 122
BT_SDP_UINT64 (*C macro*), 122
BT_SDP_UINT8 (*C macro*), 122
BT_SDP_UPNP_IP_SVCLASS (*C macro*), 120
BT_SDP_UPNP_L2CAP_SVCLASS (*C macro*), 120
BT_SDP_UPNP_LAP_SVCLASS (*C macro*), 120
BT_SDP_UPNP_PAN_SVCLASS (*C macro*), 120
BT_SDP_UPNP_SVCLASS (*C macro*), 120
BT_SDP_URL_STR16 (*C macro*), 123
BT_SDP_URL_STR32 (*C macro*), 123
BT_SDP_URL_STR8 (*C macro*), 123
BT_SDP_URL_STR_UNSPEC (*C macro*), 123
BT_SDP_UUID128 (*C macro*), 123
BT_SDP_UUID16 (*C macro*), 123
BT_SDP_UUID32 (*C macro*), 123
BT_SDP_UUID_UNSPEC (*C macro*), 123
BT_SDP_VIDEO_CONF_GW_SVCLASS (*C macro*), 119
BT_SDP_VIDEO_DISTRIBUTION_SVCLASS (*C macro*), 120
BT_SDP_VIDEO_SINK_SVCLASS (*C macro*), 120
BT_SDP_VIDEO_SOURCE_SVCLASS (*C macro*), 120
BT_SDP_WAP_CLIENT_SVCLASS (*C macro*), 119
BT_SDP_WAP_SVCLASS (*C macro*), 119
bt_security_err (*C enum*), 5
bt_security_err.BT_SECURITY_ERR_AUTH_FAIL (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_AUTH_REQUIREMENT (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_INVALID_PARAM (*C enumerator*), 6
bt_security_err.BT_SECURITY_ERR_OOB_NOT_AVAILABLE (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_PAIR_NOT_ALLOWED (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_PAIR_NOT_SUPPORTED (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_PIN_OR_KEY_MISSING (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_SUCCESS (*C enumerator*), 5
bt_security_err.BT_SECURITY_ERR_UNSPECIFIED (*C enumerator*), 6
bt_security_t (*C type*), 4
bt_set_bondable (*C function*), 11
bt_set_id_addr (*C function*), 35
bt_set_name (*C function*), 35
bt_set_oob_data_flag (*C function*), 11
bt_spp_callback (*C type*), 136
bt_spp_client_connect (*C function*), 137

bt_spp_data_send (*C function*), 137
bt_spp_disconnect (*C function*), 138
bt_spp_discover (*C function*), 137
bt_spp_discover_callback (*C type*), 136
bt_spp_get_channel (*C function*), 138
bt_spp_get_conn (*C function*), 138
bt_spp_get_role (*C function*), 138
bt_spp_role (*C enum*), 136
bt_spp_role.BT_SPP_ROLE_CLIENT (*C enumerator*), 136
bt_spp_role.BT_SPP_ROLE_SERVER (*C enumerator*), 136
bt_spp_role_t (*C type*), 136
bt_spp_server_register (*C function*), 137
bt_unpair (*C function*), 48
bt_uuid (*C struct*), 155
BT_UUID_128 (*C macro*), 140
bt_uuid_128 (*C struct*), 155
bt_uuid_128.uuid (*C var*), 155
bt_uuid_128.val (*C var*), 155
BT_UUID_128_ENCODE (*C macro*), 140
BT_UUID_16 (*C macro*), 140
bt_uuid_16 (*C struct*), 155
bt_uuid_16.uuid (*C var*), 155
bt_uuid_16.val (*C var*), 155
BT_UUID_16_ENCODE (*C macro*), 141
BT_UUID_32 (*C macro*), 140
bt_uuid_32 (*C struct*), 155
bt_uuid_32.uuid (*C var*), 155
bt_uuid_32.val (*C var*), 155
BT_UUID_32_ENCODE (*C macro*), 141
BT_UUID_ALERT_LEVEL (*C macro*), 145
BT_UUID_ALERT_LEVEL_VAL (*C macro*), 145
BT_UUID_APPARENT_WIND_DIR (*C macro*), 148
BT_UUID_APPARENT_WIND_DIR_VAL (*C macro*), 148
BT_UUID_APPARENT_WIND_SPEED (*C macro*), 148
BT_UUID_APPARENT_WIND_SPEED_VAL (*C macro*), 148
BT_UUID_ATT (*C macro*), 153
BT_UUID_ATT_VAL (*C macro*), 153
BT_UUID_AVCTP (*C macro*), 153
BT_UUID_AVCTP_VAL (*C macro*), 153
BT_UUID_AVDTP (*C macro*), 153
BT_UUID_AVDTP_VAL (*C macro*), 153
BT_UUID_BAR_PRESSURE_TREND (*C macro*), 149
BT_UUID_BAR_PRESSURE_TREND_VAL (*C macro*), 149
BT_UUID_BAS (*C macro*), 142
BT_UUID_BAS_BATTERY_LEVEL (*C macro*), 145
BT_UUID_BAS_BATTERY_LEVEL_VAL (*C macro*), 145
BT_UUID_BAS_VAL (*C macro*), 142
BT_UUID_BMS (*C macro*), 143
BT_UUID_BMS_CONTROL_POINT (*C macro*), 150
BT_UUID_BMS_CONTROL_POINT_VAL (*C macro*), 149
BT_UUID_BMS_FEATURE (*C macro*), 150
BT_UUID_BMS_FEATURE_VAL (*C macro*), 150
BT_UUID_BMS_VAL (*C macro*), 142
BT_UUID_BNEP (*C macro*), 153
BT_UUID_BNEP_VAL (*C macro*), 153
BT_UUID_CENTRAL_ADDR_RES (*C macro*), 150
BT_UUID_CENTRAL_ADDR_RES_VAL (*C macro*), 150
bt_uuid_cmp (*C function*), 154
BT_UUID_CMTP (*C macro*), 153
BT_UUID_CMTP_VAL (*C macro*), 153
bt_uuid_create (*C function*), 154
BT_UUID_CSC (*C macro*), 142
BT_UUID_CSC_FEATURE (*C macro*), 147
BT_UUID_CSC_FEATURE_VAL (*C macro*), 147
BT_UUID_CSC_MEASUREMENT (*C macro*), 147
BT_UUID_CSC_MEASUREMENT_VAL (*C macro*), 147
BT_UUID_CSC_VAL (*C macro*), 142
BT_UUID_CTS (*C macro*), 142
BT_UUID_CTS_CURRENT_TIME (*C macro*), 146
BT_UUID_CTS_CURRENT_TIME_VAL (*C macro*), 146
BT_UUID_CTS_VAL (*C macro*), 142
BT_UUID_DECLARE_128 (*C macro*), 140
BT_UUID_DECLARE_16 (*C macro*), 139
BT_UUID_DECLARE_32 (*C macro*), 139
BT_UUID_DESC_VALUE_CHANGED (*C macro*), 149
BT_UUID_DESC_VALUE_CHANGED_VAL (*C macro*), 149
BT_UUID_DEW_POINT (*C macro*), 149
BT_UUID_DEW_POINT_VAL (*C macro*), 149
BT_UUID_DIS (*C macro*), 142
BT_UUID_DIS_FIRMWARE_REVISION (*C macro*), 146
BT_UUID_DIS_FIRMWARE_REVISION_VAL (*C macro*), 146
BT_UUID_DIS_HARDWARE_REVISION (*C macro*), 146
BT_UUID_DIS_HARDWARE_REVISION_VAL (*C macro*), 146
BT_UUID_DIS_MANUFACTURER_NAME (*C macro*), 146
BT_UUID_DIS_MANUFACTURER_NAME_VAL (*C macro*), 146
BT_UUID_DIS_MODEL_NUMBER (*C macro*), 145
BT_UUID_DIS_MODEL_NUMBER_VAL (*C macro*), 145
BT_UUID_DIS_PNP_ID (*C macro*), 146
BT_UUID_DIS_PNP_ID_VAL (*C macro*), 146
BT_UUID_DIS_SERIAL_NUMBER (*C macro*), 146
BT_UUID_DIS_SERIAL_NUMBER_VAL (*C macro*), 146
BT_UUID_DIS_SOFTWARE_REVISION (*C macro*), 146

BT_UUID_DIS_SOFTWARE_REVISION_VAL (*C macro*), 146
BT_UUID_DIS_SYSTEM_ID (*C macro*), 145
BT_UUID_DIS_SYSTEM_ID_VAL (*C macro*), 145
BT_UUID_DIS_VAL (*C macro*), 142
BT_UUID_ELEVATION (*C macro*), 148
BT_UUID_ELEVATION_VAL (*C macro*), 148
BT_UUID_ES_CONFIGURATION (*C macro*), 144
BT_UUID_ES_CONFIGURATION_VAL (*C macro*), 144
BT_UUID_ES_MEASUREMENT (*C macro*), 144
BT_UUID_ES_MEASUREMENT_VAL (*C macro*), 144
BT_UUID_ES_TRIGGER_SETTING (*C macro*), 144
BT_UUID_ES_TRIGGER_SETTING_VAL (*C macro*), 144
BT_UUID_ESS (*C macro*), 142
BT_UUID_ESS_VAL (*C macro*), 142
BT_UUID_FTP (*C macro*), 153
BT_UUID_FTP_VAL (*C macro*), 153
BT_UUID_GAP (*C macro*), 141
BT_UUID_GAP_APPEARANCE (*C macro*), 145
BT_UUID_GAP_APPEARANCE_VAL (*C macro*), 145
BT_UUID_GAP_DEVICE_NAME (*C macro*), 145
BT_UUID_GAP_DEVICE_NAME_VAL (*C macro*), 144
BT_UUID_GAP_PPCP (*C macro*), 145
BT_UUID_GAP_PPCP_VAL (*C macro*), 145
BT_UUID_GAP_VAL (*C macro*), 141
BT_UUID_GATT (*C macro*), 141
BT_UUID_GATT_CCC (*C macro*), 144
BT_UUID_GATT_CCC_VAL (*C macro*), 144
BT_UUID_GATT_CEP (*C macro*), 143
BT_UUID_GATT_CEP_VAL (*C macro*), 143
BT_UUID_GATT_CHRC (*C macro*), 143
BT_UUID_GATT_CHRC_VAL (*C macro*), 143
BT_UUID_GATT_CLIENT_FEATURES (*C macro*), 152
BT_UUID_GATT_CLIENT_FEATURES_VAL (*C macro*), 152
BT_UUID_GATT_CPF (*C macro*), 144
BT_UUID_GATT_CPF_VAL (*C macro*), 144
BT_UUID_GATT_CUD (*C macro*), 144
BT_UUID_GATT_CUD_VAL (*C macro*), 144
BT_UUID_GATT_DB_HASH (*C macro*), 152
BT_UUID_GATT_DB_HASH_VAL (*C macro*), 152
BT_UUID_GATT_INCLUDE (*C macro*), 143
BT_UUID_GATT_INCLUDE_VAL (*C macro*), 143
BT_UUID_GATT_PRIMARY (*C macro*), 143
BT_UUID_GATT_PRIMARY_VAL (*C macro*), 143
BT_UUID_GATT_SC (*C macro*), 145
BT_UUID_GATT_SC_VAL (*C macro*), 145
BT_UUID_GATT_SCC (*C macro*), 144
BT_UUID_GATT_SCC_VAL (*C macro*), 144
BT_UUID_GATT_SECONDARY (*C macro*), 143
BT_UUID_GATT_SECONDARY_VAL (*C macro*), 143
BT_UUID_GATT_SERVER_FEATURES (*C macro*), 152
BT_UUID_GATT_SERVER_FEATURES_VAL (*C macro*), 152
BT_UUID_GATT_VAL (*C macro*), 141
BT_UUID_GUST_FACTOR (*C macro*), 148
BT_UUID_GUST_FACTOR_VAL (*C macro*), 148
BT_UUID_HCRP_CTRL (*C macro*), 153
BT_UUID_HCRP_CTRL_VAL (*C macro*), 153
BT_UUID_HCRP_DATA (*C macro*), 153
BT_UUID_HCRP_DATA_VAL (*C macro*), 153
BT_UUID_HCRP_NOTE (*C macro*), 153
BT_UUID_HCRP_NOTE_VAL (*C macro*), 153
BT_UUID_HEAT_INDEX (*C macro*), 149
BT_UUID_HEAT_INDEX_VAL (*C macro*), 149
BT_UUID_HIDP (*C macro*), 153
BT_UUID_HIDP_VAL (*C macro*), 153
BT_UUID_HIDS (*C macro*), 142
BT_UUID_HIDS_BOOT_KB_IN_REPORT (*C macro*), 145
BT_UUID_HIDS_BOOT_KB_IN_REPORT_VAL (*C macro*), 145
BT_UUID_HIDS_BOOT_KB_OUT_REPORT (*C macro*), 146
BT_UUID_HIDS_BOOT_KB_OUT_REPORT_VAL (*C macro*), 146
BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT (*C macro*), 146
BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT_VAL (*C macro*), 146
BT_UUID_HIDS_CTRL_POINT (*C macro*), 147
BT_UUID_HIDS_CTRL_POINT_VAL (*C macro*), 147
BT_UUID_HIDS_EXT_REPORT (*C macro*), 144
BT_UUID_HIDS_EXT_REPORT_VAL (*C macro*), 144
BT_UUID_HIDS_INFO (*C macro*), 147
BT_UUID_HIDS_INFO_VAL (*C macro*), 147
BT_UUID_HIDS_PROTOCOL_MODE (*C macro*), 147
BT_UUID_HIDS_PROTOCOL_MODE_VAL (*C macro*), 147
BT_UUID_HIDS_REPORT (*C macro*), 147
BT_UUID_HIDS_REPORT_MAP (*C macro*), 147
BT_UUID_HIDS_REPORT_MAP_VAL (*C macro*), 147
BT_UUID_HIDS_REPORT_REF (*C macro*), 144
BT_UUID_HIDS_REPORT_REF_VAL (*C macro*), 144
BT_UUID_HIDS_REPORT_VAL (*C macro*), 147
BT_UUID_HIDS_VAL (*C macro*), 142
BT_UUID_HPS (*C macro*), 143
BT_UUID_HPS_VAL (*C macro*), 143
BT_UUID_HRS (*C macro*), 142
BT_UUID_HRS_BODY_SENSOR (*C macro*), 147
BT_UUID_HRS_BODY_SENSOR_VAL (*C macro*), 147
BT_UUID_HRS_CONTROL_POINT (*C macro*), 147
BT_UUID_HRS_CONTROL_POINT_VAL (*C macro*), 147
BT_UUID_HRS_MEASUREMENT (*C macro*), 147
BT_UUID_HRS_MEASUREMENT_VAL (*C macro*), 146

BT_UUID_HRS_VAL (*C macro*), 142
BT_UUID_HTS (*C macro*), 142
BT_UUID_HTS_MEASUREMENT (*C macro*), 145
BT_UUID_HTS_MEASUREMENT_VAL (*C macro*), 145
BT_UUID_HTS_VAL (*C macro*), 142
BT_UUID_HTTP (*C macro*), 153
BT_UUID_HTTP_CONTROL_POINT (*C macro*), 150
BT_UUID_HTTP_CONTROL_POINT_VAL (*C macro*), 150
BT_UUID_HTTP_ENTITY_BODY (*C macro*), 150
BT_UUID_HTTP_ENTITY_BODY_VAL (*C macro*), 150
BT_UUID_HTTP_HEADERS (*C macro*), 150
BT_UUID_HTTP_HEADERS_VAL (*C macro*), 150
BT_UUID_HTTP_STATUS_CODE (*C macro*), 150
BT_UUID_HTTP_STATUS_CODE_VAL (*C macro*), 150
BT_UUID_HTTP_VAL (*C macro*), 153
BT_UUID_HTTPS_SECURITY (*C macro*), 150
BT_UUID_HTTPS_SECURITY_VAL (*C macro*), 150
BT_UUID_HUMIDITY (*C macro*), 148
BT_UUID_HUMIDITY_VAL (*C macro*), 148
BT_UUID_IAS (*C macro*), 141
BT_UUID_IAS_VAL (*C macro*), 141
BT_UUID_INIT_128 (*C macro*), 139
BT_UUID_INIT_16 (*C macro*), 139
BT_UUID_INIT_32 (*C macro*), 139
BT_UUID_IP (*C macro*), 153
BT_UUID_IP_VAL (*C macro*), 153
BT_UUID_IPSS (*C macro*), 143
BT_UUID_IPSS_VAL (*C macro*), 143
BT_UUID_IRRADIANCE (*C macro*), 149
BT_UUID_IRRADIANCE_VAL (*C macro*), 149
BT_UUID_L2CAP (*C macro*), 154
BT_UUID_L2CAP_VAL (*C macro*), 154
BT_UUID_LLS (*C macro*), 142
BT_UUID_LLS_VAL (*C macro*), 142
BT_UUID_MAGN_DECLINATION (*C macro*), 146
BT_UUID_MAGN_DECLINATION_VAL (*C macro*), 146
BT_UUID_MAGN_FLUX_DENSITY_2D (*C macro*), 149
BT_UUID_MAGN_FLUX_DENSITY_2D_VAL (*C macro*), 149
BT_UUID_MAGN_FLUX_DENSITY_3D (*C macro*), 149
BT_UUID_MAGN_FLUX_DENSITY_3D_VAL (*C macro*), 149
BT_UUID_MCAP_CTRL (*C macro*), 154
BT_UUID_MCAP_CTRL_VAL (*C macro*), 153
BT_UUID_MCAP_DATA (*C macro*), 154
BT_UUID_MCAP_DATA_VAL (*C macro*), 154
BT_UUID_MESH_PROV (*C macro*), 143
BT_UUID_MESH_PROV_DATA_IN (*C macro*), 152
BT_UUID_MESH_PROV_DATA_IN_VAL (*C macro*), 152
BT_UUID_MESH_PROV_DATA_OUT (*C macro*), 152
BT_UUID_MESH_PROV_DATA_OUT_VAL (*C macro*), 152
BT_UUID_MESH_PROV_VAL (*C macro*), 143
BT_UUID_MESH_PROXY (*C macro*), 143
BT_UUID_MESH_PROXY_DATA_IN (*C macro*), 152
BT_UUID_MESH_PROXY_DATA_IN_VAL (*C macro*), 152
BT_UUID_MESH_PROXY_DATA_OUT (*C macro*), 152
BT_UUID_MESH_PROXY_DATA_OUT_VAL (*C macro*), 152
BT_UUID_MESH_PROXY_VAL (*C macro*), 143
BT_UUID_OBEX (*C macro*), 153
BT_UUID_OBEX_VAL (*C macro*), 153
BT_UUID_OTS (*C macro*), 143
BT_UUID_OTS_ACTION_CP (*C macro*), 151
BT_UUID_OTS_ACTION_CP_VAL (*C macro*), 151
BT_UUID_OTS_CHANGED (*C macro*), 151
BT_UUID_OTS_CHANGED_VAL (*C macro*), 151
BT_UUID_OTS_DIRECTORY_LISTING (*C macro*), 152
BT_UUID_OTS_DIRECTORY_LISTING_VAL (*C macro*), 152
BT_UUID_OTS_FEATURE (*C macro*), 150
BT_UUID_OTS_FEATURE_VAL (*C macro*), 150
BT_UUID_OTS_FIRST_CREATED (*C macro*), 151
BT_UUID_OTS_FIRST_CREATED_VAL (*C macro*), 151
BT_UUID_OTS_ID (*C macro*), 151
BT_UUID_OTS_ID_VAL (*C macro*), 151
BT_UUID_OTS_LAST_MODIFIED (*C macro*), 151
BT_UUID_OTS_LAST_MODIFIED_VAL (*C macro*), 151
BT_UUID_OTS_LIST_CP (*C macro*), 151
BT_UUID_OTS_LIST_CP_VAL (*C macro*), 151
BT_UUID_OTS_LIST_FILTER (*C macro*), 151
BT_UUID_OTS_LIST_FILTER_VAL (*C macro*), 151
BT_UUID_OTS_NAME (*C macro*), 150
BT_UUID_OTS_NAME_VAL (*C macro*), 150
BT_UUID_OTS_PROPERTIES (*C macro*), 151
BT_UUID_OTS_PROPERTIES_VAL (*C macro*), 151
BT_UUID_OTS_SIZE (*C macro*), 151
BT_UUID_OTS_SIZE_VAL (*C macro*), 151
BT_UUID_OTS_TYPE (*C macro*), 151
BT_UUID_OTS_TYPE_UNSPECIFIED (*C macro*), 152
BT_UUID_OTS_TYPE_UNSPECIFIED_VAL (*C macro*), 151
BT_UUID_OTS_TYPE_VAL (*C macro*), 151
BT_UUID_OTS_VAL (*C macro*), 143
BT_UUID_POLLEN_CONCENTRATION (*C macro*), 148
BT_UUID_POLLEN_CONCENTRATION_VAL (*C macro*), 148
BT_UUID_PRESSURE (*C macro*), 148
BT_UUID_PRESSURE_VAL (*C macro*), 148
BT_UUID_RAINFALL (*C macro*), 149
BT_UUID_RAINFALL_VAL (*C macro*), 149
BT_UUID_RFCOMM (*C macro*), 152

BT_UUID_RFCOMM_VAL (*C macro*), 152
BT_UUID_SC_CONTROL_POINT (*C macro*), 148
BT_UUID_SC_CONTROL_POINT_VAL (*C macro*), 147
BT_UUID_SDP (*C macro*), 152
BT_UUID_SDP_VAL (*C macro*), 152
BT_UUID_SENSOR_LOCATION (*C macro*), 147
BT_UUID_SENSOR_LOCATION_VAL (*C macro*), 147
BT_UUID_SIZE_128 (*C macro*), 139
BT_UUID_SIZE_16 (*C macro*), 139
BT_UUID_SIZE_32 (*C macro*), 139
BT_UUID_STR_LEN (*C macro*), 141
BT_UUID_TCP (*C macro*), 153
BT_UUID_TCP_VAL (*C macro*), 152
BT_UUID_TCS_AT (*C macro*), 153
BT_UUID_TCS_AT_VAL (*C macro*), 153
BT_UUID_TCS_BIN (*C macro*), 153
BT_UUID_TCS_BIN_VAL (*C macro*), 153
BT_UUID_TEMPERATURE (*C macro*), 148
BT_UUID_TEMPERATURE_VAL (*C macro*), 148
bt_uuid_to_str (*C function*), 154
BT_UUID_TPS (*C macro*), 142
BT_UUID_TPS_TX_POWER_LEVEL (*C macro*), 145
BT_UUID_TPS_TX_POWER_LEVEL_VAL (*C macro*), 145
BT_UUID_TPS_VAL (*C macro*), 142
BT_UUID_TRUE_WIND_DIR (*C macro*), 148
BT_UUID_TRUE_WIND_DIR_VAL (*C macro*), 148
BT_UUID_TRUE_WIND_SPEED (*C macro*), 148
BT_UUID_TRUE_WIND_SPEED_VAL (*C macro*), 148
BT_UUID_UDI (*C macro*), 153
BT_UUID_UDI_VAL (*C macro*), 153
BT_UUID_UDP (*C macro*), 152
BT_UUID_UDP_VAL (*C macro*), 152
BT_UUID_UPNP (*C macro*), 153
BT_UUID_UPNP_VAL (*C macro*), 153
BT_UUID_URI (*C macro*), 150
BT_UUID_URI_VAL (*C macro*), 150
BT_UUID_UV_INDEX (*C macro*), 149
BT_UUID_UV_INDEX_VAL (*C macro*), 149
BT_UUID_VALID_RANGE (*C macro*), 144
BT_UUID_VALID_RANGE_VAL (*C macro*), 144
BT_UUID_WIND_CHILL (*C macro*), 149
BT_UUID_WIND_CHILL_VAL (*C macro*), 149

D

discover_cb_t (*C struct*), 138

H

hf_multiparty_call_option_t (*C type*), 92
hf_volume_type_t (*C type*), 92
hf_waiting_call_state_t (*C type*), 92
hfp_ag_call_setup_status_t (*C enum*), 92
hfp_ag_call_setup_status_t.HFP_AG_CALL_SETUP_STATUS_IDLE (*C enumerator*), 92
hfp_ag_call_setup_status_t.HFP_AG_CALL_SETUP_STATUS (*C enumerator*), 93
hfp_ag_call_setup_status_t.HFP_AG_CALL_SETUP_STATUS (*C enumerator*), 93
hfp_ag_call_setup_status_t.HFP_AG_CALL_SETUP_STATUS (*C enumerator*), 93
hfp_ag_call_status_t (*C type*), 92
hfp_ag_cind_t (*C type*), 92
hfp_ag_get_config (*C type*), 92
HFP_HF_CMD_CME_ERROR (*C macro*), 91
HFP_HF_CMD_ERROR (*C macro*), 91
HFP_HF_CMD_OK (*C macro*), 91
HFP_HF_CMD_UNKNOWN_ERROR (*C macro*), 91
HFP_HF_DIGIT_ARRAY_SIZE (*C macro*), 91
HFP_HF_MAX_OPERATOR_NAME_LEN (*C macro*), 91
hps_config_t (*C struct*), 158
hps_data_status_t (*C type*), 156
hps_flags_t (*C type*), 156
hps_http_command_t (*C type*), 156
hps_state_t (*C type*), 156
hps_status_t (*C struct*), 158
hts_include_temp_type (*C macro*), 158
hts_unit_celsius_c (*C macro*), 158
hts_unit_fahrenheit_c (*C macro*), 158

I

ipsp_connect (*C function*), 159
ipsp_init (*C function*), 159
ipsp_listen (*C function*), 160
ipsp_rx_cb_t (*C type*), 159
ipsp_send (*C function*), 160

M

MAX_BODY_LEN (*C macro*), 156
MAX_HEADERS_LEN (*C macro*), 156
MAX_URI_LEN (*C macro*), 156
MEDIA_TYPE (*C enum*), 131
MEDIA_TYPE.BT_A2DP_AUDIO (*C enumerator*), 131
MEDIA_TYPE.BT_A2DP_MULTIMEDIA (*C enumerator*), 131
MEDIA_TYPE.BT_A2DP_VIDEO (*C enumerator*), 131

P

pxr_deinit (*C function*), 161
pxr_ias_get_alert_level (*C function*), 161
pxr_init (*C function*), 161
pxr_lls_get_alert_level (*C function*), 161
pxr_tps_get_power_level (*C function*), 161
pxr_tps_set_power_level (*C function*), 161

R

read_lls_alert_level (*C function*), 160
read_tps_power_level (*C function*), 161

read_tps_power_level_desc (*C function*), 161
ROLE_TYPE (*C enum*), 131
ROLE_TYPE.BT_A2DP_SINK (*C enumerator*), 131
ROLE_TYPE.BT_A2DP_SOURCE (*C enumerator*), 131

T

temp_measurement (*C struct*), 159

U

USER_DATA_MIN (*C macro*), 159

W

write_http_entity_body (*C function*), 157
write_http_headers (*C function*), 157
write_ias_alert_level (*C function*), 160
write_lls_alert_level (*C function*), 161

Date of release: 23 March 2021

Document identifier: EFBTPALAPIRM