VERSION 1.0
August 29, 2017

BTstack Manual

Including Quickstart Guide

Dr. sc. Milanka Ringwald
Dr. sc. Matthias Ringwald
contact@bluekitchen-gmbh.com

Contents

1. Welcome

Thanks for checking out BTstack!

In this manual, we first provide a 'quick starter guide' for common platforms before highlighting BTstack's main design choices and go over all implemented protocols and profiles.

A series of examples show how BTstack can be used to implement common use cases.

Finally, we outline the basic steps when integrating BTstack into existing single-threaded or even multi-threaded environments.

2. Quick Start

2.1. General Tools.

Most ports use a regular Makefile to build the examples.

On Unix-based systems, git, make, and Python are usually installed. If not, use the system's packet manager to install them.

On Windows, there is no packet manager, but it's easy to download and install all requires development packets quickly by hand. You'll need:

- Python for Windows. When using the official installer, please confirm adding Python to the Windows Path.
- MSYS2 is used to provide the bash shell and most standard POSIX command line tools.
- MinGW64 GCC for Windows 64 & 32 bits incl. make. To install with MSYS2: pacman -S mingw-w64-x86_64-gcc
- git is used to download BTstack source code. To install with MSYS2: pacman -S git
- winpty a wrapper to allow for console input when running in MSYS2: To install with MSYS2: pacman -S winpty

2.2. Getting BTstack from GitHub.

Use git to clone the latest version:

```
git clone https://github.com/bluekitchen/btstack.git
```

Alternatively, you can download it as a ZIP archive from BTstack's page on GitHub.

2.3. Compiling the examples and loading firmware.

This step is platform specific. To compile and run the examples, you need to download and install the platform specific toolchain and a flash tool. For TI's CC256x chipsets, you also need the correct init script, or "Service Pack" in TI nomenclature. Assuming that these are provided, go to folder `btstack/port/$PLATFORM$` in command prompt and run make. If all the paths are correct, it will generate several firmware files. These firmware files can be loaded onto the device using platform specific flash programmer. For the PIC32-Harmony platform, a project file for the MPLAB X IDE is provided, too.

2.4. **Run the Example.** As a first test, we recommend the SPP Counter example. During the startup, for TI chipsets, the init script is transferred, and the Bluetooth stack brought up. After that, the development board is discoverable as "BTstack SPP Counter" and provides a single virtual serial port. When you connect to it, you'll receive a counter value as text every second.

2.5. **Platform specifics.** In the following, we provide more information on specific platform setups, toolchains, programmers, and init scripts.

2.5.1. *libusb.* The quickest way to try BTstack is on a Linux or OS X system with an additional USB Bluetooth module. The Makefile `port/libusb` in requires pkg-config and libusb-1.0 or higher to be installed.

On Linux, it's usually necessary to run the examples as root as the kernel needs to detach from the USB module.

On OS X, it's necessary to tell the OS to only use the internal Bluetooth. For this, execute:

```
sudo nvram bluetoothHostControllerSwitchBehavior=never
```

2.6. **Windows-WinUSB.** Although libusb basically works with the POSIX Run Loop on Windows, we recommend to use the Windows-WinUSB port that uses a native run loop and the native WinUSB API to access a USB Bluetooth dongle.

For libusb or WinUSB, you need to install a special device driver to make the USB dongle accessible to user space. It works like this:

- Start Zadig
- Select Options -> "List all devices"
- Select USB Bluetooth dongle in the big pull down list
- Select WinUSB (libusb) in the right pull pull down list
- Select "Replace Driver"

When running the examples in the MSYS2 shell, the console input (via btstack_stdin_support) doesn't work. It works in the older MSYS and also the regular CMD.exe environment. Another option is to install WinPTY and then start the example via WinPTY like this:

```
$ winpty ./hfp_hf_demo.exe
```

2.6.1. *Texas Instruments MSP430-based boards.* **Compiler Setup.** The MSP430 port of BTstack is developed using the Long Term Support (LTS) version of mspgcc. General information about it and installation instructions are provided on the MSPGCC Wiki. On Windows, you need to download and extract mspgcc to `C:\mspgcc`. Add `C:\mspgcc\bin` folder to the Windows Path in Environment variable as explained here.

Loading Firmware. To load firmware files onto the MSP430 MCU for the MSP-EXP430F5438 Experimenter board, you need a programmer like the MSP430 MSP-FET430UIF debugger or something similar. The eZ430-RF2560 and MSP430F5529LP contain a basic debugger. Now, you can use one of following software tools:

- MSP430Flasher (windows-only):
  - Use the following command, where you need to replace the `BINARY\_FILE\_NAME.hex` with the name of your application:

```
MSP430Flasher.exe -n MSP430F5438A -w "BINARY_FILE_NAME.hex" -v -g -z
    [VCC]
```

- MSPDebug: An example session with the MSP-FET430UIF connected on OS X is given in following listing:

```
mspdebug -j -d /dev/tty.FET430UIFfd130 uif
...
prog blink.hex
run
```

2.6.2. *Texas Instruments CC256x-based chipsets.* **CC256x Init Scripts.** In order to use the CC256x chipset on the PAN13xx modules and others, an initialization script must be obtained. Due to licensing restrictions, this initialization script must be obtained separately as follows:

- Download the BTS file for your CC256x-based module.
- Copy the included .bts file into
- In `chipset/cc256x`, run the Python script:

```
./convert_bts_init_scripts.py
```

The common code for all CC256x chipsets is provided by *btstack_chipset_cc256x.c*. During the setup, *btstack_chipset_cc256x_instance* function is used to get a *btstack_control_t* instance and passed to *hci_init* function.

Note: Depending on the CC256x-based module you're using, you'll need to update the reference in the Makefile to match the downloaded file.

Update: For the latest revision of the CC256x chipsets, the CC2560B and CC2564B, TI decided to split the init script into a main part and the BLE part. The conversion script has been updated to detect *bluetooth_init_cc256x_1.2.bts* and adds *BLE_init_cc256x_1.2.bts* if present and merges them into a single .c file.

Update 2: In May 2015, TI renamed the init scripts to match the naming scheme previously used on Linux systems. The conversion script has been updated to also detect *initscripts_TIInit_6.7.16_bt_spec_4.1.bts* and integrates *initscripts_TIInit_6.7.16_ble_add-on.bts* if present.

2.6.3. *MSP-EXP430F5438 + CC256x Platform.* **Hardware Setup.** We assume that a PAN1315, PAN1317, or PAN1323 module is plugged into RF1 and RF2 of the MSP-EXP430F5438 board and the "RF3 Adapter board" is used or at least simulated. See User Guide.

2.6.4. *STM32F103RB Nucleo + CC256x Platform.* To try BTstack on this platform, you'll need a simple adaptor board. For details, please read the documentation in `platforms/stm32-f103rb-nucleo/README.md`.

2.6.5. *PIC32 Bluetooth Audio Development Kit.* The PIC32 Bluetooth Audio Development Kit comes with the CSR8811-based BTM805 Bluetooth module. In the port, the UART on the DAC daughter board was used for the debug output. Please remove the DAC board and connect a 3.3V USB-2-UART converter to GND and TX to get the debug output.

In `platforms/pic32-harmony`, a project file for the MPLAB X IDE is provided as well as a regular Makefile. Both assume that the MPLAB XC32 compiler is installed. The project is set to use -Os optimization which will cause warnings if you only have the Free version. It will still compile a working example. For this platform, we only provide the SPP and LE Counter example directly. Other examples can be run by replacing the *spp_and_le_counter.c* file with one of the other example files.

3. BTstack Architecture

As well as any other communication stack, BTstack is a collection of state machines that interact with each other. There is one or more state machines for each protocol and service that it implements. The rest of the architecture follows these fundamental design guidelines:

- *Single threaded design* - BTstack does not use or require multi-threading to handle data sources and timers. Instead, it uses a single run loop.
- *No blocking anywhere* - If Bluetooth processing is required, its result will be delivered as an event via registered packet handlers.
- *No artificially limited buffers/pools* - Incoming and outgoing data packets are not queued.
- *Statically bounded memory (optionally)* - The number of maximum connections/channels/services can be configured.

Figure 1 shows the general architecture of a BTstack-based single-threaded application that includes the BTstack run loop. The Main Application contains the application logic, e.g., reading a sensor value and providing it via the Communication Logic as a SPP Server. The Communication Logic is often modeled as a finite state machine with events and data coming from either the Main Application or from BTstack via registered packet handlers (PH). BTstack's Run Loop is responsible for providing timers and processing incoming data.

3.1. Single threaded design.

BTstack does not use or require multi-threading. It uses a single run loop to handle data sources and timers. Data sources represent communication interfaces like an UART or an USB driver. Timers are used by BTstack to implement various Bluetooth-related timeouts. For example, to disconnect a Bluetooth baseband channel without an active L2CAP channel after

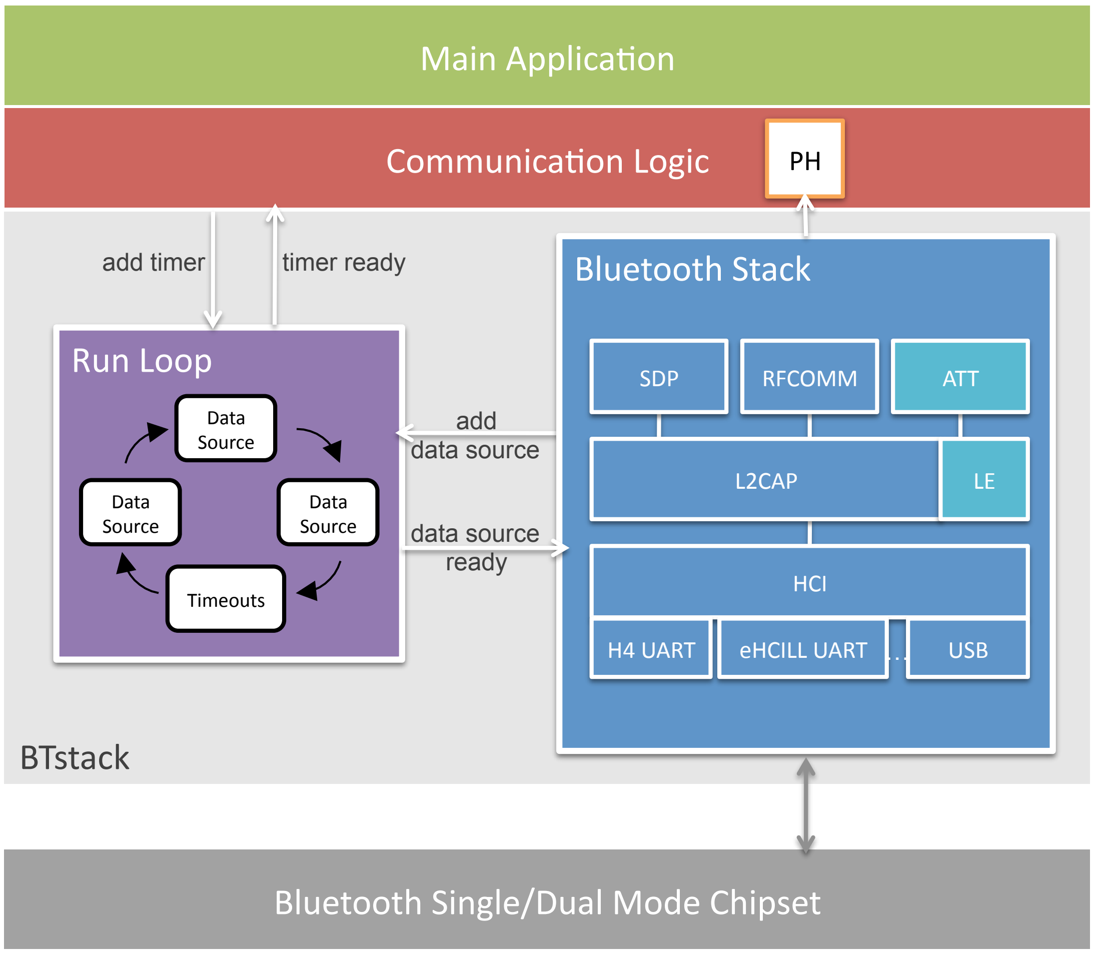

FIGURE 1. Architecture of a BTstack-based application.

20 seconds. They can also be used to handle periodic events. During a run loop cycle, the callback functions of all registered data sources are called. Then, the callback functions of timers that are ready are executed.

For adapting BTstack to multi-threaded environments check here.

3.2. **No blocking anywhere.** Bluetooth logic is event-driven. Therefore, all BTstack functions are non-blocking, i.e., all functions that cannot return immediately implement an asynchronous pattern. If the arguments of a function are valid, the necessary commands are sent to the Bluetooth chipset and the function returns with a success value. The actual result is delivered later as an asynchronous event via registered packet handlers.

If a Bluetooth event triggers longer processing by the application, the processing should be split into smaller chunks. The packet handler could then schedule a timer that manages the sequential execution of the chunks.

3.3. **No artificially limited buffers/pools.** Incoming and outgoing data packets are not queued. BTstack delivers an incoming data packet to the application before it receives the next one from the Bluetooth chipset. Therefore, it relies on the link layer of the Bluetooth chipset to slow down the remote sender when needed.

Similarly, the application has to adapt its packet generation to the remote receiver for outgoing data. L2CAP relies on ACL flow control between sender and receiver. If there are no free ACL buffers in the Bluetooth module, the application cannot send. For RFCOMM, the mandatory credit-based flow-control limits the data sending rate additionally. The application can only send an RFCOMM packet if it has RFCOMM credits.

3.4. **Statically bounded memory.** BTstack has to keep track of services and active connections on the various protocol layers. The number of maximum connections/channels/services can be configured. In addition, the non-persistent database for remote device names and link keys needs memory and can be be configured, too. These numbers determine the amount of static memory allocation.

4. How to configure BTstack

BTstack implements a set of Bluetooth protocols and profiles. To connect to other Bluetooth devices or to provide a Bluetooth services, BTstack has to be properly configured.

The configuration of BTstack is done both at compile time as well as at run time:

- compile time configuration:
  - adjust *btstack_config.h* - this file describes the system configuration, used functionality, and also the memory configuration
  - add necessary source code files to your project
- run time configuration of:
  - Bluetooth chipset
  - run loop
  - HCI transport layer
  - provided services
  - packet handlers

In the following, we provide an overview of the configuration that is necessary to setup BTstack. From the point when the run loop is executed, the application runs as a finite state machine, which processes events received from BTstack. BTstack groups events logically and provides them via packet handlers. We provide their overview here. For the case that there is a need to inspect the data exchanged between BTstack and the Bluetooth chipset, we describe how to configure packet logging mechanism. Finally, we provide an overview on power management in Bluetooth in general and how to save energy in BTstack.

4.1. **Configuration in btstack_config.h.** The file *btstack_config.h* contains three parts:

- #define HAVE_* directives listed here. These directives describe available system properties, similar to config.h in a autoconf setup.

- #define ENABLE_* directives listed here. These directives list enabled properties, most importantly ENABLE_CLASSIC and ENABLE_BLE.
- other #define directives for BTstack configuration, most notably static memory, see next section and NVM configuration.

4.1.1. *HAVE_* directives.* System properties:

5. define | Description

HAVE_MALLOC	Use dynamic memory
HAVE_AES128	Use platform AES128 engine - not needed usually
HAVE_BTSTACK_STDIN	STDIN is available for CLI interface

Embedded platform properties:

6. define | Description

HAVE_EMBEDDED_TIME_MS	System provides time in milliseconds
HAVE_EMBEDDED_TICK	System provides tick interrupt

POSIX platform properties:

7. define | Description

————————————————|————————————————— HAVE_POSIX_B300_MAPPED_TO_2000
| Workaround to use serial port with 2 mbps HAVE_POSIX_B600_MAPPED_TO_3000000
| Workaround to use serial port with 3 mpbs HAVE_POSIX_FILE_IO | POSIX File i/o used for hci dump HAVE_POSIX_TIME | System provides time function LINK_KEY_PATH | Path to stored link keys LE_DEVICE_DB_PATH | Path to stored LE device information

7.0.2. *ENABLE_* directives.* BTstack properties:

8. define | Description

ENABLE_CLASSIC	Enable C
ENABLE_BLE	Enable B
ENABLE_EHCILL	Enable e
ENABLE_LOG_DEBUG	Enable l
ENABLE_LOG_ERROR	Enable l
ENABLE_LOG_INFO	Enable l
ENABLE_SCO_OVER_HCI	Enable S
ENABLE_HFP_WIDE_BAND_SPEECH	Enable s

ENBALE_LE_PERIPHERAL	Enable s
ENBALE_LE_CENTRAL	Enable s
ENABLE_LE_SECURE_CONNECTIONS	Enable L
ENABLE_MICRO_ECC_FOR_LE_SECURE_CONNECTIONS	Use micr
ENABLE_LE_DATA_CHANNELS	Enable L
ENABLE_LE_DATA_LENGTH_EXTENSION	Enable L
ENABLE_LE_SIGNED_WRITE	Enable L
ENABLE_L2CAP_ENHANCED_RETRANSMISSION_MODE	Enable L
ENABLE_HCI_CONTROLLER_TO_HOST_FLOW_CONTROL	Enable H
ENABLE_CC256X_BAUDRATE_CHANGE_FLOWCONTROL_BUG_WORKAROUND	Enable w

Notes: - ENABLE_MICRO_ECC_FOR_LE_SECURE_CONNECTIONS: Only some Bluetooth 4.2+ controllers (e.g., EM9304, ESP32) support the necessary HCI commands. Others reasons to enable the ECC software implementations are if the Host is much faster or if the micro-ecc library is already provided (e.g., ESP32, WICED)

8.0.3. *HCI Controller to Host Flow Control.* In general, BTstack relies on flow control of the HCI transport, either via Hardware CTS/RTS flow control for UART or regular USB flow control. If this is not possible, e.g on an SoC, BTstack can use HCI Controller to Host Flow Control by defining ENABLE_HCI_CONTROLLER_TO_HOST If enabled, the HCI Transport implementation must be able to buffer the specified packets. In addition, it also need to be able to buffer a few HCI Events. Using a low number of host buffers might result in less throughput.

Host buffer configuration for HCI Controller to Host Flow Control:

9. define | Description

HCI_HOST_ACL_PACKET_NUM	Max number of ACL packets
HCI_HOST_ACL_PACKET_LEN	Max size of HCI Host ACL packets
HCI_HOST_SCO_PACKET_NUM	Max number of ACL packets
HCI_HOST_SCO_PACKET_LEN	Max size of HCI Host SCO packets

9.0.4. *Memory configuration directives.* The structs for services, active connections and remote devices can be allocated in two different manners:

- statically from an individual memory pool, whose maximal number of elements is defined in the btstack_config.h file. To initialize the static pools, you need to call at runtime *btstack_memory_init* function. An example of memory configuration for a single SPP service with a minimal L2CAP MTU is shown in Listing {@lst:memoryConfigurationSPP}.

- dynamically using the *malloc/free* functions, if HAVE_MALLOC is defined in btstack_config.h file.

For each HCI connection, a buffer of size HCI_ACL_PAYLOAD_SIZE is reserved. For fast data transfer, however, a large ACL buffer of 1021 bytes is recommend. The large ACL buffer is required for 3-DH5 packets to be used.

10. DEFINE | DESCRIPTION

HCI_ACL_PAYLOAD_SIZE	Max size of HCI ACL payloads
MAX_NR_BNEP_CHANNELS	Max number of BNEP channels
MAX_NR_BNEP_SERVICES	Max number of BNEP services
MAX_NR_BTSTACK_LINK_KEY_DB_MEMORY_ENTRIES	Max number of link key entries cac
MAX_NR_GATT_CLIENTS	Max number of GATT clients
MAX_NR_HCI_CONNECTIONS	Max number of HCI connections
MAX_NR_HFP_CONNECTIONS	Max number of HFP connections
MAX_NR_L2CAP_CHANNELS	Max number of L2CAP connection
MAX_NR_L2CAP_SERVICES	Max number of L2CAP services
MAX_NR_RFCOMM_CHANNELS	Max number of RFOMMM connect
MAX_NR_RFCOMM_MULTIPLEXERS	Max number of RFCOMM multiple
MAX_NR_RFCOMM_SERVICES	Max number of RFCOMM services
MAX_NR_SERVICE_RECORD_ITEMS	Max number of SDP service record
MAX_NR_SM_LOOKUP_ENTRIES	Max number of items in Security M
MAX_NR_WHITELIST_ENTRIES	Max number of items in GAP LE V
MAX_NR_LE_DEVICE_DB_ENTRIES	Max number of items in LE Device

The memory is set up by calling *btstack_memory_init* function:

```
btstack_memory_init();
```

Here's the memory configuration for a basic SPP server.

```
#define HCI_ACL_PAYLOAD_SIZE 52
#define MAX_NR_HCI_CONNECTIONS 1
#define MAX_NR_L2CAP_SERVICES   2
#define MAX_NR_L2CAP_CHANNELS   2
#define MAX_NR_RFCOMM_MULTIPLEXERS 1
#define MAX_NR_RFCOMM_SERVICES 1
#define MAX_NR_RFCOMM_CHANNELS 1
#define MAX_NR_BTSTACK_LINK_KEY_DB_MEMORY_ENTRIES   3
```

Listing: Memory configuration for a basic SPP server. {#lst:memoryConfigurationSPP}

In this example, the size of ACL packets is limited to the minimum of 52 bytes, resulting in an L2CAP MTU of 48 bytes. Only a singleHCI connection can be established at any time. On it, two L2CAP services are provided, which can be active at the same time. Here, these two can be RFCOMM and SDP. Then, memory for one RFCOMM multiplexer is reserved over which one connection can be active. Finally, up to three link keys can be cached in RAM.

10.0.5. *Non-volatile memory (NVM) directives.* If implemented, bonding information is stored in Non-volatile memory. For Classic, a single link keys and its type is stored. For LE, the bonding information contains various values (long term key, random number, EDIV, signing counter, identity, . . .)Often, this implemented using Flash memory. Then, the number of stored entries are limited by:

11. define | Description

NVM_NUM_LINK_KEYS	Max number of Classic Link Keys that can be stored
NVM_NUM_DEVICE_DB_ENTRIES	Max number of LE Device DB entries that can be stored

11.1. Source tree structure.

The source tree has been organized to easily setup new projects.

Path	Description
chipset	Support for individual Bluetooth chipsets
doc	Sources for BTstack documentation
example	Example applications available for all ports
platform	Support for special OSs and/or MCU architectures
port	Complete port for a MCU + Chipset combinations
src	Bluetooth stack implementation
test	Unit and PTS tests
tool	Helper tools for BTstack

The core of BTstack, including all protocol and profiles, is in *src/*.

Support for a particular platform is provided by the *platform/* subfolder. For most embedded ports, *platform/embedded/* provides *btstack_run_loop_embedded* and the *hci_transport_h4_embedded* implementation that require *hal_cpu.h*, *hal_led.h*, and *hal_uart_dma.h* plus *hal_tick.h* or *hal_time_ms* to be implemented by the user.

To accommodate a particular Bluetooth chipset, the *chipset/* subfolders provide various btstack_chipset_* implementations. Please have a look at the existing ports in *port/*.

11.2. **Run loop configuration.** To initialize BTstack you need to initialize the memory and the run loop respectively, then setup HCI and all needed higher level protocols.

BTstack uses the concept of a run loop to handle incoming data and to schedule work. The run loop handles events from two different types of sources: data sources and timers. Data sources represent communication interfaces like an UART or an USB driver. Timers are used by BTstack to implement various Bluetooth-related timeouts. They can also be used to handle periodic events.

Data sources and timers are represented by the *btstack_data_source_t* and *btstack_timer_source_t* structs respectively. Each of these structs contain at least a linked list node and a pointer to a callback function. All active timers and data sources are kept in link lists. While the list of data sources is unsorted, the timers are sorted by expiration timeout for efficient processing.

Timers are single shot: a timer will be removed from the timer list before its event handler callback is executed. If you need a periodic timer, you can re-register the same timer source in the callback function, as shown in Listing [PeriodicTimerHandler]. Note that BTstack expects to get called periodically to keep its time, see Section on time abstraction for more on the tick hardware abstraction.

BTstack provides different run loop implementations that implement the *btstack_run_loop_t* interface:

- Embedded: the main implementation for embedded systems, especially without an RTOS.
- POSIX: implementation for POSIX systems based on the select() call.
- CoreFoundation: implementation for iOS and OS X applications
- WICED: implementation for the Broadcom WICED SDK RTOS abstraction that wraps FreeRTOS or ThreadX.
- Windows: implementation for Windows based on Event objects and WaitForMultipleObjects() call.

Depending on the platform, data sources are either polled (embedded), or the platform provides a way to wait for a data source to become ready for read or write (POSIX, CoreFoundation, Windows), or, are not used as the HCI transport driver and the run loop is implemented in a different way (WICED). In any case, the callbacks must be to explicitly enabled with the *btstack_run_loop_enable_data_source_callbacks(..)* function.

In your code, you'll have to configure the run loop before you start it as shown in Listing [listing:btstackInit]. The application can register data sources as well as timers, e.g., for periodical sampling of sensors, or for communication over the UART.

The run loop is set up by calling *btstack_run_loop_init* function and providing an instance of the actual run loop. E.g. for the embedded platform, it is:

```
btstack_run_loop_init(btstack_run_loop_embedded_get_instance());
```

The complete Run loop API is provided here.

11.2.1. *Run loop embedded.* In the embedded run loop implementation, data sources are constantly polled and the system is put to sleep if no IRQ happens during the poll of all data sources.

The complete run loop cycle looks like this: first, the callback function of all registered data sources are called in a round robin way. Then, the callback functions of timers that are ready are executed. Finally, it will be checked if another run loop iteration has been requested by an interrupt handler. If not, the run loop will put the MCU into sleep mode.

Incoming data over the UART, USB, or timer ticks will generate an interrupt and wake up the microcontroller. In order to avoid the situation where a data source becomes ready just before the run loop enters sleep mode, an interrupt-driven data source has to call the *btstack_run_loop_embedded_trigger* function. The call to *btstack_run_loop_embedded_trigger* sets an internal flag that is checked in the critical section just before entering sleep mode causing another run loop cycle.

To enable the use of timers, make sure that you defined HAVE_EMBEDDED_TICK or HAVE_EMBEDDED_TIME_MS in the config file.

11.2.2. *Run loop POSIX.* The data sources are standard File Descriptors. In the run loop execute implementation, select() call is used to wait for file descriptors to become ready to read or write, while waiting for the next timeout.

To enable the use of timers, make sure that you defined HAVE_POSIX_TIME in the config file.

11.2.3. *Run loop CoreFoundation (OS X/iOS).* This run loop directly maps BT-stack's data source and timer source with CoreFoundation objects. It supports ready to read and write similar to the POSIX implementation. The call to *btstack_run_loop_execute()* then just calls *CFRunLoopRun().*

To enable the use of timers, make sure that you defined HAVE_POSIX_TIME in the config file.

11.2.4. *Run loop Windows.* The data sources are Event objects. In the run loop implementation WaitForMultipleObjects() call is all is used to wait for the Event object to become ready while waiting for the next timeout.

11.2.5. *Run loop WICED.* WICED SDK API does not provide asynchronous read and write to the UART and no direct way to wait for one or more peripherals to become ready. Therefore, BTstack does not provide direct support for data sources. Instead, the run loop provides a message queue that allows to schedule functions calls on its thread via *btstack_run_loop_wiced_execute_code_on_main_thread().*

The HCI transport H4 implementation then uses two lightweight threads to do the blocking read and write operations. When a read or write is complete on the helper threads, a callback to BTstack is scheduled.

11.3. **HCI Transport configuration.** The HCI initialization has to adapt BT-stack to the used platform. The first call is to *hci_init()* and requires information about the HCI Transport to use. The arguments are:

- *HCI Transport implementation*: On embedded systems, a Bluetooth module can be connected via USB or an UART port. On embedded, BTstack implements HCI UART Transport Layer (H4) and H4 with eHCILL support, a lightweight low-power variant by Texas Instruments. For POSIX, there is an implementation for HCI H4, HCI H5 and H2 libUSB, and for WICED HCI H4 WICED. These are accessed by linking the appropriate file, e.g., `platform/embedded/hci\_transport\_h4\_embedded.c` and then getting a pointer to HCI Transport implementation. For more information on adapting HCI Transport to different environments, see here.

```
hci_transport_t * transport = hci_transport_h4_instance();
```

- *HCI Transport configuration*: As the configuration of the UART used in the H4 transport interface are not standardized, it has to be provided by the main application to BTstack. In addition to the initial UART baud rate, the main baud rate can be specified. The HCI layer of BTstack will change the init baud rate to the main one after the basic setup of the Bluetooth module. A baud rate change has to be done in a coordinated way at both HCI and hardware level. For example, on the CC256x, the HCI command to change the baud rate is sent first, then it is necessary to wait for the confirmation event from the Bluetooth module. Only now, can the UART baud rate changed.

```
hci_uart_config_t* config = &hci_uart_config;
```

After these are ready, HCI is initialized like this:

```
hci_init(transport, config);
```

In addition to these, most UART-based Bluetooth chipset require some special logic for correct initialization that is not covered by the Bluetooth specification. In particular, this covers:

- setting the baudrate
- setting the BD ADDR for devices without an internal persistent storage
- upload of some firmware patches.

This is provided by the various *btstack_chipset_t* implementation in the *chipset/* subfolders. As an example, the *bstack_chipset_cc256x_instance* function returns a pointer to a chipset struct suitable for the CC256x chipset.

```
btstack_chipset_t * chipset = btstack_chipset_cc256x_instance();
hci_set_chipset(chipset);
```

In some setups, the hardware setup provides explicit control of Bluetooth power and sleep modes. In this case, a *btstack_control_t* struct can be set with *hci_set_control*.

Finally, the HCI implementation requires some form of persistent storage for link keys generated during either legacy pairing or the Secure Simple Pairing (SSP). This commonly requires platform specific code to access the MCU's EEPROM of Flash storage. For the first steps, BTstack provides a (non) persistent store in memory. For more see here.

```
btstack_link_key_db_t * link_key_db = &
    btstack_link_key_db_memory_instance();
btstack_set_link_key_db(link_key_db);
```

The higher layers only rely on BTstack and are initialized by calling the respective **_init* function. These init functions register themselves with the underlying layer. In addition, the application can register packet handlers to get events and data as explained in the following section.

11.4. **Services.** One important construct of BTstack is *service*. A service represents a server side component that handles incoming connections. So far, BTstack provides L2CAP, BNEP, and RFCOMM services. An L2CAP service handles incoming connections for an L2CAP channel and is registered with its protocol service multiplexer ID (PSM). Similarly, an RFCOMM service handles incoming RFCOMM connections and is registered with the RFCOMM channel ID. Outgoing connections require no special registration, they are created by the application when needed.

11.5. **Packet handlers configuration.** After the hardware and BTstack are set up, the run loop is entered. From now on everything is event driven. The application calls BTstack functions, which in turn may send commands to the Bluetooth module. The resulting events are delivered back to the application. Instead of writing a single callback handler for each possible event (as it is done in some other Bluetooth stacks), BTstack groups events logically and provides them over a single generic interface. Appendix Events and Errors summarizes the parameters and event codes of L2CAP and RFCOMM events, as well as possible errors and the corresponding error codes.

Here is summarized list of packet handlers that an application might use:

- HCI event handler - allows to observer HCI, GAP, and general BTstack events.
- L2CAP packet handler - handles LE Connection parameter requeset updates
- L2CAP service packet handler - handles incoming L2CAP connections, i.e., channels initiated by the remote.
- L2CAP channel packet handler - handles outgoing L2CAP connections, i.e., channels initiated internally.
- RFCOMM service packet handler - handles incoming RFCOMM connections, i.e., channels initiated by the remote.

- RFCOMM channel packet handler - handles outgoing RFCOMM connections, i.e., channels initiated internally.

These handlers are registered with the functions listed in Table 8.

Packet Handler	Registering Function
HCI packet handler	hci_add_event_handler
L2CAP packet handler	l2cap_register_packet_handler
L2CAP service packet handler	l2cap_register_service
L2CAP channel packet handler	l2cap_create_channel
RFCOMM service packet handler	rfcomm_register_service and rfcomm_register_service_with_initi
RFCOMM channel packet handler	rfcomm_create_channel and rfcomm_create_channel_with_initia

Table 8: Functions for registering packet handlers.

HCI, GAP, and general BTstack events are delivered to the packet handler specified by *hci_add_event_handler* function. In L2CAP, BTstack discriminates incoming and outgoing connections, i.e., event and data packets are delivered to different packet handlers. Outgoing connections are used access remote services, incoming connections are used to provide services. For incoming connections, the packet handler specified by *l2cap_register_service* is used. For outgoing connections, the handler provided by *l2cap_create_channel* is used. RFCOMM and BNEP are similar.

The application can register a single shared packet handler for all protocols and services, or use separate packet handlers for each protocol layer and service. A shared packet handler is often used for stack initialization and connection management.

Separate packet handlers can be used for each L2CAP service and outgoing connection. For example, to connect with a Bluetooth HID keyboard, your application could use three packet handlers: one to handle HCI events during discovery of a keyboard registered by *l2cap_register_packet_handler*; one that will be registered to an outgoing L2CAP channel to connect to keyboard and to receive keyboard data registered by *l2cap_create_channel*; after that keyboard can reconnect by itself. For this, you need to register L2CAP services for the HID Control and HID Interrupt PSMs using *l2cap_register_service*. In this call, you'll also specify a packet handler to accept and receive keyboard data.

All events names have the form MODULE_EVENT_NAME now, e.g., *gap_event_advertising_repor*
To facilitate working with events and get rid of manually calculating offsets into packets, BTstack provides auto-generated getters for all fields of all events in *src/hci_event.h*. All functions are defined as static inline, so they are not wasting any program memory if not used. If used, the memory footprint should be identical to accessing the field directly via offsets into the packet. For example, to access fields address_type and address from the *gap_event_advertising_report* event use following getters:

uint8_t address type = gap_event_advertising_report_get_address_type(event); bd_addr_t address; gap_event_advertising_report_get_address(event, address);

11.6. Bluetooth HCI Packet Logs. If things don't work as expected, having a look at the data exchanged between BTstack and the Bluetooth chipset often helps.

For this, BTstack provides a configurable packet logging mechanism via hci_dump.h:

```
// formats: HCI_DUMP_BLUEZ, HCI_DUMP_PACKETLOGGER, HCI_DUMP_STDOUT
void hci_dump_open(const char *filename, hci_dump_format_t format);
```

On POSIX systems, you can call *hci_dump_open* with a path and *HCI_DUMP_BLUEZ* or *HCI_DUMP_PACKETLOGGER* in the setup, i.e., before entering the run loop. The resulting file can be analyzed with Wireshark or the Apple's PacketLogger tool.

On embedded systems without a file system, you still can call *hci_dump_open(NULL, HCI_DUMP_STDOUT)*. It will log all HCI packets to the console via printf. If you capture the console output, incl. your own debug messages, you can use the create_packet_log.py tool in the tools folder to convert a text output into a PacketLogger file.

In addition to the HCI packets, you can also enable BTstack's debug information by adding

```
#define ENABLE_LOG_INFO
#define ENABLE_LOG_ERROR
```

to the btstack_config.h and recompiling your application.

11.7. Bluetooth Power Control. In most BTstack examples, the device is set to be discoverable and connectable. In this mode, even when there's no active connection, the Bluetooth Controller will periodically activate its receiver in order to listen for inquiries or connecting requests from another device. The ability to be discoverable requires more energy than the ability to be connected. Being discoverable also announces the device to anybody in the area. Therefore, it is a good idea to pause listening for inquiries when not needed. Other devices that have your Bluetooth address can still connect to your device.

To enable/disable discoverability, you can call:

```
/**
 * @brief Allows to control if device is discoverable. OFF by
    default.
 */
void gap_discoverable_control(uint8_t enable);
```

If you don't need to become connected from other devices for a longer period of time, you can also disable the listening to connection requests.

To enable/disable connectability, you can call:

```
/**
 * @brief Override page scan mode. Page scan mode enabled by l2cap
    when services are registered
 * @note Might be used to reduce power consumption while Bluetooth
    module stays powered but no (new)
 *       connections are expected
 */
void gap_connectable_control(uint8_t enable);
```

For Bluetooth Low Energy, the radio is periodically used to broadcast advertisements that are used for both discovery and connection establishment.

To enable/disable advertisements, you can call:

```
/**
 * @brief Enable/Disable Advertisements. OFF by default.
 * @param enabled
 */
void gap_advertisements_enable(int enabled);
```

If a Bluetooth Controller is neither discoverable nor connectable, it does not need to periodically turn on its radio and it only needs to respond to commands from the Host. In this case, the Bluetooth Controller is free to enter some kind of deep sleep where the power consumption is minimal.

Finally, if that's not sufficient for your application, you could request BTstack to shutdown the Bluetooth Controller. For this, the "on" and "off" functions in the btstack_control_t struct must be implemented. To shutdown the Bluetooth Controller, you can call:

```
/**
 * @brief Requests the change of BTstack power mode.
 */
int  hci_power_control(HCI_POWER_MODE mode);
```

with mode set to *HCI_POWER_OFF*. When needed later, Bluetooth can be started again via by calling it with mode *HCI_POWER_ON*, as seen in all examples.

12. Protocols

BTstack is a modular dual-mode Bluetooth stack, supporting both Bluetooth Basic Rate/Enhanced Date Rate (BR/EDR) as well as Bluetooth Low Energy (LE). The BR/EDR technology, also known as Classic Bluetooth, provides a robust wireless connection between devices designed for high data rates. In contrast, the LE technology has a lower throughput but also lower energy consumption, faster connection setup, and the ability to connect to more devices in parallel.

Whether Classic or LE, a Bluetooth device implements one or more Bluetooth profiles. A Bluetooth profile specifies how one or more Bluetooth protocols are used to achieve its goals. For example, every Bluetooth device must implement the Generic Access Profile (GAP), which defines how devices find each other and how they establish a connection. This profile mainly make use of the Host Controller Interface (HCI) protocol, the lowest protocol in the stack hierarchy which implements a command interface to the Bluetooth chipset.

In addition to GAP, a popular Classic Bluetooth example would be a peripheral devices that can be connected via the Serial Port Profile (SPP). SPP basically specifies that a compatible device should provide a Service Discovery Protocol (SDP) record containing an RFCOMM channel number, which will be used for the actual communication.

Similarly, for every LE device, the Generic Attribute Profile (GATT) profile must be implemented in addition to GAP. GATT is built on top of the Attribute Protocol (ATT), and defines how one device can interact with GATT Services on a remote device.

So far, the most popular use of BTstack is in peripheral devices that can be connected via SPP (Android 2.0 or higher) and GATT (Android 4.3 or higher, and iOS 5 or higher). If higher data rates are required between a peripheral and iOS device, the iAP1 and iAP2 protocols of the Made for iPhone program can be used instead of GATT. Please contact us directly for information on BTstack and MFi.

Figure 2 depicts Bluetooth protocols and profiles that are currently implemented by BTstack. In the following, we first explain how the various Bluetooth protocols are used in BTstack. In the next chapter, we go over the profiles.

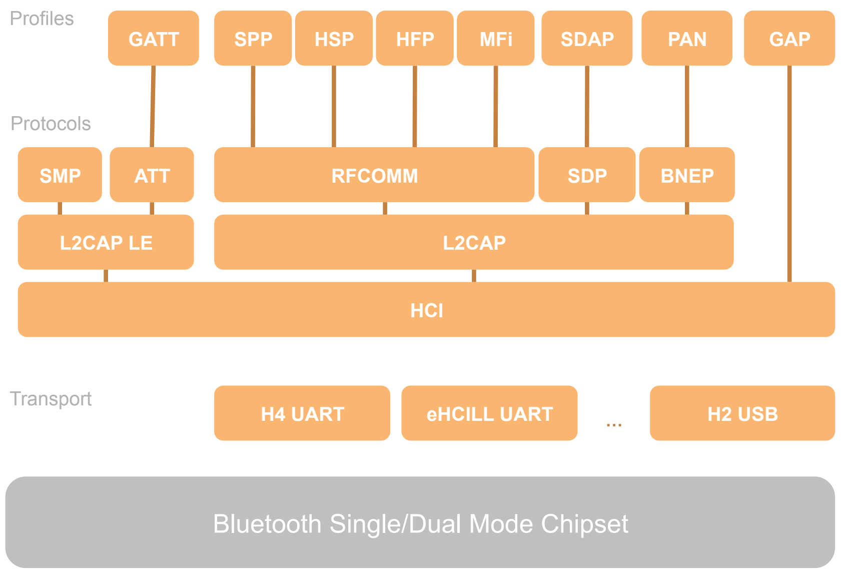

Figure 2. Architecture of a BTstack-based application.

12.1. **HCI - Host Controller Interface.** The HCI protocol provides a command interface to the Bluetooth chipset. In BTstack, the HCI implementation also keeps track of all active connections and handles the fragmentation and re-assembly of higher layer (L2CAP) packets.

Please note, that an application rarely has to send HCI commands on its own. Instead, BTstack provides convenience functions in GAP and higher level protocols that use HCI automatically. E.g. to set the name, you call *gap_set_local_name()* before powering up. The main use of HCI commands in application is during the startup phase to configure special features that are not available via the GAP API yet. How to send a custom HCI command is explained in the following section.

12.1.1. *Defining custom HCI command templates.* Each HCI command is assigned a 2-byte OpCode used to uniquely identify different types of commands. The OpCode parameter is divided into two fields, called the OpCode Group Field (OGF) and OpCode Command Field (OCF), see Bluetooth Specification - Core Version 4.0, Volume 2, Part E, Chapter 5.4.

Listing 1 shows the OGFs provided by BTstackin file `src/hci.h`:

```
#define OGF_LINK_CONTROL    0x01
#define OGF_LINK_POLICY    0x02
#define OGF_CONTROLLER_BASEBAND    0x03
#define OGF_INFORMATIONAL_PARAMETERS 0x04
#define OGF_LE_CONTROLLER    0x08
#define OGF_BTSTACK    0x3d
#define OGF_VENDOR    0x3f
```

Listing 1. HCI OGFs provided by BTstack.

For all existing Bluetooth commands and their OCFs see Bluetooth Specification - Core Version 4.0, Volume 2, Part E, Chapter 7.

In a HCI command packet, the OpCode is followed by parameter total length, and the actual parameters. The OpCode of a command can be calculated using the OPCODE macro. BTstack provides the *hci_cmd_t* struct as a compact format to define HCI command packets, see Listing 2 , and`include/btstack/hci\_cmd.h` file in the source code.

```
// Calculate combined ogf/ocf value.
#define OPCODE(ogf, ocf) (ocf | ogf << 10)

// Compact HCI Command packet description.
typedef struct {
    uint16_t    opcode;
    const char *format;
} hci_cmd_t;
```

Listing 2. HCI command struct.

Listing 3 illustrates the*hci_write_local_name* HCI command template from library:

```
// Sets local Bluetooth name
const hci_cmd_t hci_write_local_name = {
    OPCODE(OGF_CONTROLLER_BASEBAND, 0x13), "N"
    // Local name (UTF-8, Null Terminated, max 248 octets)
};
```

Listing 3. HCI command example.

It uses OGF_CONTROLLER_BASEBAND as OGF, 0x13 as OCF, and has one parameter with format "N" indicating a null terminated UTF-8 string. Table 9 lists the format specifiers supported by BTstack. Check for other predefined HCI commands and info on their parameters.

Format Specifier	Description
1,2,3,4	one to four byte value
A	31 bytes advertising data
B	Bluetooth Baseband Address
D	8 byte data block
E	Extended Inquiry Information 240 octets
H	HCI connection handle
N	Name up to 248 chars, UTF8 string, null terminated
P	16 byte Pairing code, e.g. PIN code or link key
S	Service Record (Data Element Sequence)

Table 9: Supported Format Specifiers of HCI Command Parameter.

12.1.2. *Sending HCI command based on a template.* You can use the *hci_send_cmd* function to send HCI command based on a template and a list of parameters. However, it is necessary to check that the outgoing packet buffer is empty and that the Bluetooth module is ready to receive the next command - most modern Bluetooth modules only allow to send a single HCI command. This can be done by calling *hci_can_send_command_packet_now()* function, which returns true, if it is ok to send.

Listing 4 illustrates how tomanually set the device name with the HCI Write Local Name command.

```
    if (hci_can_send_packet_now(HCI_COMMAND_DATA_PACKET)){
        hci_send_cmd(&hci_write_local_name, "BTstack Demo");
    }
```

Listing 4. Sending HCI command example.

Please note, that an application rarely has to send HCI commands on its own. Instead, BTstack provides convenience functions in GAP and higher level protocols that use HCI automatically.

12.2. **L2CAP - Logical Link Control and Adaptation Protocol.** The L2CAP protocol supports higher level protocol multiplexing and packet fragmentation. It provides the base for the RFCOMM and BNEP protocols. For all profiles that are officially supported by BTstack, L2CAP does not need to be used directly. For testing or the development of custom protocols, it's helpful to be able to access and provide L2CAP services however.

12.2.1. *Access an L2CAP service on a remote device.* L2CAP is based around the concept of channels. A channel is a logical connection on top of a baseband connection. Each channel is bound to a single protocol in a many-to-one fashion. Multiple channels can be bound to the same protocol, but a channel cannot be bound to multiple protocols. Multiple channels can share the same baseband connection.

To communicate with an L2CAP service on a remote device, the application on a local Bluetooth device initiates the L2CAP layer using the *l2cap_init* function, and then creates an outgoing L2CAP channel to the PSM of a remote device using the *l2cap_create_channel* function. The *l2cap_create_channel* function will initiate a new baseband connection if it does not already exist. The packet handler that is given as an input parameter of the L2CAP create channel function will be assigned to the new outgoing L2CAP channel. This handler receives the L2CAP_EVENT_CHANNEL_OPENED and L2CAP_EVENT_CHANNEL_CLOSED events and L2CAP data packets, as shown in Listing 5 .

```

    btstack_packet_handler_t l2cap_packet_handler;

    void l2cap_packet_handler(uint8_t packet_type, uint16_t channel,
         uint8_t *packet, uint16_t size){
        bd_addr_t event_addr;
        switch (packet_type){
            case HCI_EVENT_PACKET:
                switch (hci_event_packet_get_type(packet)){
                    case L2CAP_EVENT_CHANNEL_OPENED:
                        l2cap_event_channel_opened_get_address(
                            packet, &event_addr);
                        psm        =
                            l2cap_event_channel_opened_get_psm(
                            packet);
                        local_cid  =
                            l2cap_event_channel_opened_get_local_cid
                            (packet);
                        handle    =
                            l2cap_event_channel_opened_get_handle(
                            packet);
                        if (l2cap_event_channel_opened_get_status(
                            packet)) {
                             printf("Connection failed\n\r");
                        } else
                             printf("Connected\n\r");
                        }
                        break;
                    case L2CAP_EVENT_CHANNEL_CLOSED:
                        break;
                        ...
                }
            case L2CAP_DATA_PACKET:
                // handle L2CAP data packet
                break;
            ...
        }
    }

    void create_outgoing_l2cap_channel(bd_addr_t address, uint16_t
        psm, uint16_t mtu){
          l2cap_create_channel(NULL, l2cap_packet_handler,
              remote_bd_addr, psm, mtu);
    }

    void btstack_setup(){
         ...
         l2cap_init();
    }
```

Listing 5. Accessing an L2CAP service on a remote device.

12.2.2. *Provide an L2CAP service.* To provide an L2CAP service, the application on a local Bluetooth device must init the L2CAP layer and register the service with *l2cap_register_service*. From there on, it can wait for incoming L2CAP connections. The application can accept or deny an incoming connection by calling the *l2cap_accept_connection* and *l2cap_deny_connection* functions respectively.

If a connection is accepted and the incoming L2CAP channel gets successfully opened, the L2CAP service can send and receive L2CAP data packets to the connected device with *l2cap_send*.

Listing 6 provides L2CAP serviceexample code.

```
    void packet_handler (uint8_t packet_type, uint16_t channel,
        uint8_t *packet, uint16_t size){
         bd_addr_t event_addr;
         switch (packet_type){
             case HCI_EVENT_PACKET:
                switch (hci_event_packet_get_type(packet)){
                    case L2CAP_EVENT_INCOMING_CONNECTION:
                        local_cid =
                            l2cap_event_incoming_connection_get_local_cid
                            (packet);
                        l2cap_accept_connection(local_cid);
                        break;
                    case L2CAP_EVENT_CHANNEL_OPENED:
                        l2cap_event_channel_opened_get_address(
                            packet, &event_addr);
                        psm       =
                            l2cap_event_channel_opened_get_psm(
                            packet);
                        local_cid =
                            l2cap_event_channel_opened_get_local_cid
                            (packet);
                        handle    =
                            l2cap_event_channel_opened_get_handle(
                            packet);
                        if (l2cap_event_channel_opened_get_status(
                            packet)) {
                             printf("Connection failed\n\r");
                        } else
                             printf("Connected\n\r");
                        }
                        break;
                    case L2CAP_EVENT_CHANNEL_CLOSED:
                        break;
                        ...
                }
            case L2CAP_DATA_PACKET:
                // handle L2CAP data packet
                break;
            ...
        }
    }

    void btstack_setup(){
        ...
        l2cap_init();
        l2cap_register_service(NULL, packet_handler, 0x11,100);
    }
```

Listing 6. Providing an L2CAP service.

12.2.3. *Sending L2CAP Data.* Sending of L2CAP data packets may fail due to a full internal BTstack outgoing packet buffer, or if the ACL buffers in the Bluetooth module become full, i.e., if the application is sending faster than the packets can be transferred over the air.

Instead of directly calling *l2cap_send*, it is recommended to call *l2cap_request_can_send_now_event(*
which will trigger an L2CAP_EVENT_CAN_SEND_NOW as soon as possible. It might happen that the event is received via packet handler before the *l2cap_request_can_send_now_eve*

function returns. The L2CAP_EVENT_CAN_SEND_NOW indicates a channel ID on which sending is possible.

12.2.4. *LE Data Channels.* The full title for LE Data Channels is actually LE Connection-Oriented Channels with LE Credit-Based Flow-Control Mode. In this mode, data is sent as Service Data Units (SDUs) that can be larger than an individual HCI LE ACL packet.

LE Data Channels are similar to Classic L2CAP Channels but also provide a credit-based flow control similar to RFCOMM Channels. Unless the LE Data Packet Extension of Bluetooth Core 4.2 specification is used, the maximum packet size for LE ACL packets is 27 bytes. In order to send larger packets, each packet will be split into multiple ACL LE packets and recombined on the receiving side.

Since multiple SDUs can be transmitted at the same time and the individual ACL LE packets can be sent interleaved, BTstack requires a dedicated receive buffer per channel that has to be passed when creating the channel or accepting it. Similarly, when sending SDUs, the data provided to the *l2cap_le_send_data* must stay valid until the *L2CAP_EVENT_LE_PACKET_SENT* is received.

When creating an outgoing connection of accepting an incoming, the *initial_credits* allows to provide a fixed number of credits to the remote side. Further credits can be provided anytime with *l2cap_le_provide_credits*. If *L2CAP_LE_AUTOMATIC_CREDI* is used, BTstack automatically provides credits as needed - effectively trading in the flow-control functionality for convenience.

The remainder of the API is similar to the one of L2CAP:

- *l2cap_le_register_service* and *l2cap_le_unregister_service* are used to manage local services.
- *l2cap_le_accept_connection* and *l2cap_le_decline_connection* are used to accept or deny an incoming connection request.
- *l2cap_le_create_channel* creates an outgoing connections.
- *l2cap_le_can_send_now* checks if a packet can be scheduled for transmission now.
- *l2cap_le_request_can_send_now_event* requests an *L2CAP_EVENT_LE_CAN_SEND_NOW* event as soon as possible.
- *l2cap_le_disconnect* closes the connection.

12.3. **RFCOMM - Radio Frequency Communication Protocol.** The Radio frequency communication (RFCOMM) protocol provides emulation of serial ports over the L2CAP protocol and reassembly. It is the base for the Serial Port Profile and other profiles used for telecommunication like Head-Set Profile, Hands-Free Profile, Object Exchange (OBEX) etc.

12.3.1. *RFCOMM flow control.* RFCOMM has a mandatory credit-based flow-control. This means that two devices that established RFCOMM connection, use credits to keep track of how many more RFCOMM data packets can be sent to each. If a device has no (outgoing) credits left, it cannot send another RFCOMM packet, the transmission must be paused. During the connection establishment, initial credits are provided. BTstack tracks the number of credits in both directions. If no outgoing credits are available, the RFCOMM send function will return an error, and you can try later. For incoming data, BTstack provides channels and services with and without automatic credit management via different functions to create/register them respectively. If the management of credits is automatic, the new credits are provided when needed relying on ACL flow control - this is only useful if there is not much data transmitted and/or only one physical connection is used. If the management of credits is manual, credits are provided by the application such that it can manage its receive buffers explicitly.

12.3.2. *Access an RFCOMM service on a remote device.* To communicate with an RFCOMM service on a remote device, the application on a local Bluetooth device initiates the RFCOMM layer using the *rfcomm_init* function, and then creates an outgoing RFCOMM channel to a given server channel on a remote device using the *rfcomm_create_channel* function. The *rfcomm_create_channel* function will initiate a new L2CAP connection for the RFCOMM multiplexer, if it does not already exist. The channel will automatically provide enough credits to the remote side. To provide credits manually, you have to create the RFCOMM connection by calling *rfcomm_create_channel_with_initial_credits* - see Section on manual credit assignement.

The packet handler that is given as an input parameter of the RFCOMM create channel function will be assigned to the new outgoing channel. This handler receives the RFCOMM_EVENT_CHANNEL_OPENED and RFCOMM_EVENT_CHANNEL_CLOSE events, and RFCOMM data packets, as shown in Listing 7 .

```
void rfcomm_packet_handler(uint8_t packet_type, uint16_t channel
    , uint8_t *packet, uint16_t size){
    switch (packet_type){
        case HCI_EVENT_PACKET:
            switch (hci_event_packet_get_type(packet)){
                case RFCOMM_EVENT_CHANNEL_OPENED:
                    if (
                        rfcomm_event_open_channel_complete_get_status
                        (packet)) {
                         printf("Connection failed\n\r");
                    } else {
                         printf("Connected\n\r");
                    }
                    break;
                case RFCOMM_EVENT_CHANNEL_CLOSED:
                    break;
                ...
            }
            break;
        case RFCOMM_DATA_PACKET:
            // handle RFCOMM data packets
            return;
    }
}
```

```
    void create_rfcomm_channel(uint8_t packet_type, uint8_t *packet,
        uint16_t size){
        rfcomm_create_channel(rfcomm_packet_handler, addr,
            rfcomm_channel);
    }

    void btstack_setup(){
        ...
        l2cap_init();
        rfcomm_init();
    }
```

Listing 7. RFCOMM handler for outgoing RFCOMM channel.

12.3.3. *Provide an RFCOMM service.* To provide an RFCOMM service, the application on a local Bluetooth device must first init the L2CAP and RFCOMM layers and then register the service with *rfcomm_register_service*. From there on, it can wait for incoming RFCOMM connections. The application can accept or deny an incoming connection by calling the *rfcomm_accept_connection* and *rfcomm_deny_connection* functions respectively. If a connection is accepted and the incoming RFCOMM channel gets successfully opened, the RFCOMM service can send RFCOMM data packets to the connected device with *rfcomm_send* and receive data packets by the packet handler provided by the *rfcomm_register_service* call.

Listing 8 provides the RFCOMM serviceexample code.

```
    void packet_handler(uint8_t packet_type, uint16_t channel,
       uint8_t *packet, uint16_t size){
        switch (packet_type){
            case HCI_EVENT_PACKET:
                switch (hci_event_packet_get_type(packet)){
                    case RFCOMM_EVENT_INCOMING_CONNECTION:
                        rfcomm_channel_id =
                            rfcomm_event_incoming_connection_get_rfcomm_cid
                            (packet);
                        rfcomm_accept_connection(rfcomm_channel_id);
                        break;
                    case RFCOMM_EVENT_CHANNEL_OPENED:
                        if (
                            rfcomm_event_open_channel_complete_get_status
                            (packet)){
                             printf("RFCOMM channel open failed.");
                             break;
                        }
                        rfcomm_channel_id =
                            rfcomm_event_open_channel_complete_get_rfcomm_cid
                            (packet);
                        mtu =
                            rfcomm_event_open_channel_complete_get_max_frame_size
                            (packet);
                        printf("RFCOMM channel open succeeded, max
                           frame size %u.", mtu);
                        break;
                    case RFCOMM_EVENT_CHANNEL_CLOSED:
                        printf("Channel closed.");
                        break;
                    ...
                }
                break;
        case RFCOMM_DATA_PACKET:
                // handle RFCOMM data packets
                return;
            ...
        }
        ...
    }

    void btstack_setup(){
        ...
        l2cap_init();
        rfcomm_init();
        rfcomm_register_service(packet_handler, rfcomm_channel_nr,
           mtu);
    }
```

Listing 8. Providing an RFCOMM service.

12.3.4. *Slowing down RFCOMM data reception.* RFCOMM's credit-based flow-control can be used to adapt, i.e., slow down the RFCOMM data to your processing speed. For incoming data, BTstack provides channels and services with and without automatic credit management. If the management of credits is automatic, new credits are provided when needed relying on ACL flow control. This is only useful if there is not much data transmitted and/or only one physical connection is used. See Listing 9 .

```
    void btstack_setup(void){
        ...
        // init RFCOMM
        rfcomm_init();
        rfcomm_register_service(packet_handler, rfcomm_channel_nr,
           100);
    }
```

Listing 9. RFCOMM service with automatic credit management.

If the management of credits is manual, credits are provided by the application such that it can manage its receive buffers explicitly, see Listing 10 . Manual credit management is recommended when received RFCOMM data cannot be processed immediately. In the SPP flow control example, delayed processing of received data is simulated with the help of a periodic timer. To provide new credits, you call the *rfcomm_grant_credits* function with the RFCOMM channel ID and the number of credits as shown in Listing 11 .

```
void btstack_setup(void){
    ...
    // init RFCOMM
    rfcomm_init();
    // reserved channel, mtu=100, 1 credit
    rfcomm_register_service_with_initial_credits(packet_handler,
        rfcomm_channel_nr, 100, 1);
}
```

Listing 10. RFCOMM service with manual credit management.

```
void processing(){
    // process incoming data packet
    ...
    // provide new credit
    rfcomm_grant_credits(rfcomm_channel_id, 1);
}
```

Listing 11. Granting RFCOMM credits.

Please note that providing single credits effectively reduces the credit-based (sliding window) flow control to a stop-and-wait flow-control that limits the data throughput substantially. On the plus side, it allows for a minimal memory footprint. If possible, multiple RFCOMM buffers should be used to avoid pauses while the sender has to wait for a new credit.

12.3.5. *Sending RFCOMM data.* Outgoing packets, both commands and data, are not queued in BTstack. This section explains the consequences of this design decision for sending data and why it is not as bad as it sounds.

Independent from the number of output buffers, packet generation has to be adapted to the remote receiver and/or maximal link speed. Therefore, a packet can only be generated when it can get sent. With this assumption, the single output buffer design does not impose additional restrictions. In the following, we show how this is used for adapting the RFCOMM send rate.

When there is a need to send a packet, call *rcomm_request_can_send_now* and wait for the reception of the RFCOMM_EVENT_CAN_SEND_NOW event to send the packet, as shown in Listing 12 .

```
void prepare_data(uint16_t rfcomm_channel_id){
    ...
    // prepare data in data_buffer
    rfcomm_request_can_send_now_event(rfcomm_channel_id);
}

void send_data(uint16_t rfcomm_channel_id){
    rfcomm_send(rfcomm_channel_id, data_buffer, data_len);
    // packet is handed over to BTstack, we can prepare the next
        one
    prepare_data(rfcomm_channel_id);
}
```

```
void packet_handler(uint8_t packet_type, uint16_t channel,
    uint8_t *packet, uint16_t size){
    switch (packet_type){
        case HCI_EVENT_PACKET:
            switch (hci_event_packet_get_type(packet)){
                ...
                case RFCOMM_EVENT_CAN_SEND_NOW:
                    rfcomm_channel_id =
                        rfcomm_event_can_send_now_get_rfcomm_cid
                        (packet);
                    send_data(rfcomm_channel_id);
                    break;
                ...
            }
            ...
        }
    }
}
```

LISTING 12. Preparing and sending data.

12.3.6. *Optimized sending of RFCOMM data.* When sending RFCOMM data via *rfcomm_send*, BTstack needs to copy the data from the user provided buffer into the outgoing buffer. This requires both an additional buffer for the user data as well requires a copy operation.

To avoid this, it is possible to directly write the user data into the outgoing buffer.

When get the RFCOMM_CAN_SEND_NOW event, you call *rfcomm_reserve_packet_buffer* to lock the buffer for your send operation. Then, you can ask how many bytes you can send with *rfcomm_get_max_frame_size* and get a pointer to BTstack's buffer with *rfcomm_get_outgoing_buffer*. Now, you can fill that buffer and finally send the data with *rfcomm_send_prepared*.

12.4. **SDP - Service Discovery Protocol.** The SDP protocol allows to announce services and discover services provided by a remote Bluetooth device.

12.4.1. *Create and announce SDP records.* BTstack contains a complete SDP server and allows to register SDP records. An SDP record is a list of SDP Attribute {*ID, Value*} pairs that are stored in a Data Element Sequence (DES). The Attribute ID is a 16-bit number, the value can be of other simple types like integers or strings or can itself contain other DES.

To create an SDP record for an SPP service, you can call *spp_create_sdp_record* from with a pointer to a buffer to store the record, the server channel number, and a record name.

For other types of records, you can use the other functions in, using the data element *de_* functions. Listing [sdpCreate] shows how an SDP record containing two SDP attributes can be created. First, a DES is created and then the Service Record Handle and Service Class ID List attributes are added to it. The Service Record Handle attribute is added by calling the *de_add_number* function twice: the first time to add 0x0000 as attribute ID, and the second time to add the actual record handle (here 0x1000) as attribute value. The Service Class ID List attribute has ID 0x0001, and it requires a list of UUIDs as attribute value. To create the list, *de_push_sequence* is called, which "opens" a sub-DES. The returned pointer is used to add elements to this sub-DES. After adding all UUIDs, the sub-DES is "closed" with *de_pop_sequence*.

To register an SDP record, you call *sdp_register_service* with a pointer to it. The SDP record can be stored in FLASH since BTstack only stores the pointer. Please note that the buffer needs to persist (e.g. global storage, dynamically allocated from the heap or in FLASH) and cannot be used to create another SDP record.

12.4.2. *Query remote SDP service.* BTstack provides an SDP client to query SDP services of a remote device. The SDP Client API is shown in here. The *sdp_client_query* function initiates an L2CAP connection to the remote SDP server. Upon connect, a *Service Search Attribute* request with a *Service Search Pattern* and a *Attribute ID List* is sent. The result of the *Service Search Attribute* query contains a list of *Service Records*, and each of them contains the requested attributes. These records are handled by the SDP parser. The parser delivers SDP_PARSER_ATTRIBUTE_VALUE and SDP_PARSER_COMPLETE events via a registered callback. The SDP_PARSER_ATTRIBUTE_VALUE event delivers the attribute value byte by byte.

On top of this, you can implement specific SDP queries. For example, BTstack provides a query for RFCOMM service name and channel number. This information is needed, e.g., if you want to connect to a remote SPP service. The query delivers all matching RFCOMM services, including its name and the channel number, as well as a query complete event via a registered callback, as shown in Listing 13 .

```
bd_addr_t remote = {0x04,0x0C,0xCE,0xE4,0x85,0xD3};

void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
    if (packet_type != HCI_EVENT_PACKET) return;

    uint8_t event = packet[0];
    switch (event) {
        case BTSTACK_EVENT_STATE:
            // bt stack activated, get started
            if (btstack_event_state_get_state(packet) ==
                HCI_STATE_WORKING){
                    sdp_client_query_rfcomm_channel_and_name_for_uuid
                        (remote, 0x0003);
            }
            break;
        default:
            break;
    }
}
```

```
static void btstack_setup(){
    ...
    // init L2CAP
    l2cap_init();
    l2cap_register_packet_handler(packet_handler);
}

void handle_query_rfcomm_event(sdp_query_event_t * event, void *
     context){
    sdp_client_query_rfcomm_service_event_t * ve;

    switch (event->type){
        case SDP_EVENT_QUERY_RFCOMM_SERVICE:
            ve = (sdp_client_query_rfcomm_service_event_t*)
                event;
            printf("Service name: '%s', RFCOMM port %u\n", ve->
                service_name, ve->channel_nr);
            break;
        case SDP_EVENT_QUERY_COMPLETE:
            report_found_services();
            printf("Client query response done with status %d. \
                n", ce->status);
            break;
    }
}

int main(void){
    hw_setup();
    btstack_setup();

    // register callback to receive matching RFCOMM Services and
    // query complete event
    sdp_client_query_rfcomm_register_callback(
        handle_query_rfcomm_event, NULL);

    // turn on!
    hci_power_control(HCI_POWER_ON);
    // go!
    btstack_run_loop_execute();
    return 0;
}
```

Listing 13. Searching RFCOMM services on a remote device.

12.5. **BNEP - Bluetooth Network Encapsulation Protocol.** The BNEP protocol is used to transport control and data packets over standard network protocols such as TCP, IPv4 or IPv6. It is built on top of L2CAP, and it specifies a minimum L2CAP MTU of 1691 bytes.

12.5.1. *Receive BNEP events.* To receive BNEP events, please register a packet handler with *bnep_register_packet_handler*.

12.5.2. *Access a BNEP service on a remote device.* To connect to a remote BNEP service, you need to know its UUID. The set of available UUIDs can be queried by a SDP query for the PAN profile. Please see section on PAN profile for details. With the remote UUID, you can create a connection using the *bnep_connect* function. You'll receive a *BNEP_EVENT_CHANNEL_OPENED* on success or failure.

After the connection was opened successfully, you can send and receive Ethernet packets. Before sending an Ethernet frame with *bnep_send*, *bnep_can_send_packet_now* needs to return true. Ethernet frames are received via the registered packet handler with packet type *BNEP_DATA_PACKET*.

BTstack BNEP implementation supports both network protocol filter and multicast filters with *bnep_set_net_type_filter* and *bnep_set_multicast_filter* respectively.

Finally, to close a BNEP connection, you can call *bnep_disconnect*.

12.5.3. *Provide BNEP service.* To provide a BNEP service, call *bnep_register_service* with the provided service UUID and a max frame size.

A *BNEP_EVENT_INCOMING_CONNECTION* event will mark that an incoming connection is established. At this point you can start sending and receiving Ethernet packets as described in the previous section.

12.5.4. *Sending Ethernet packets.* Similar to L2CAP and RFOMM, directly sending an Ethernet packet via BNEP might fail, if the outgoing packet buffer or the ACL buffers in the Bluetooth module are full.

When there's a need to send an Ethernet packet, call *bnep_request_can_send_now* and send the packet when the BNEP_EVENT_CAN_SEND_NOW event gets received.

12.6. **ATT - Attribute Protocol.** The ATT protocol is used by an ATT client to read and write attribute values stored on an ATT server. In addition, the ATT server can notify the client about attribute value changes. An attribute has a handle, a type, and a set of properties.

The Generic Attribute (GATT) profile is built upon ATT and provides higher level organization of the ATT attributes into GATT Services and GATT Characteristics. In BTstack, the complete ATT client functionality is included within the GATT Client. See GATT client for more.

On the server side, one ore more GATT profiles are converted ahead of time into the corresponding ATT attribute database and provided by the *att_server* implementation. The constant data are automatically served by the ATT server upon client request. To receive the dynamic data, such is characteristic value, the application needs to register read and/or write callback. In addition, notifications and indications can be sent. Please see Section on GATT server for more.

12.7. **SMP - Security Manager Protocol.** The SMP protocol allows to setup authenticated and encrypted LE connection. After initialization and configuration, SMP handles security related functions on its own but emits events when feedback from the main app or the user is required. The two main tasks of the SMP protocol are: bonding and identity resolving.

12.7.1. *LE Legacy Pairing and LE Secure Connections.* The original pairing algorithm introduced in Bluetooth Core V4.0 does not provide security in case of an attacker present during the initial pairing. To fix this, the Bluetooth Core V4.2 specification introduced the new *LE Secure Connections* method, while referring to the original method as *LE Legacy Pairing*.

BTstack supports both pairing methods. To enable the more secure LE Secure Connections method, *ENABLE_LE_SECURE_CONNECTIONS* needs to be defined in *btstack_config.h*.

LE Secure Connections are based on Elliptic Curve Diffie-Hellman (ECDH) algorithm for the key exchange. On start, a new public/private key pair is generated. During pairing, the Long Term Key (LTK) is generated based on the local keypair and the remote public key. To facilitate the creation of such a keypairs and the calculation of the LTK, the Bluetooth Core V4.2 specification introduced appropriate commands for the Bluetooth controller.

As an alternative for controllers that don't provide these primitives, BTstack provides the relevant cryptographic functions in software via the BSD-2-Clause licensed micro-ecc library.

When using using LE Secure Connections, the Peripheral must store LTK in non-volatile memory.

12.7.2. *Initialization.* To activate the security manager, call *sm_init()*.

If you're creating a product, you should also call *sm_set_ir()* and *sm_set_er()* with a fixed random 16 byte number to create the IR and ER key seeds. If possible use a unique random number per device instead of deriving it from the product serial number or something similar. The encryption key generated by the BLE peripheral will be ultimately derived from the ER key seed. See Bluetooth Specification - Bluetooth Core V4.0, Vol 3, Part G, 5.2.2 for more details on deriving the different keys. The IR key is used to identify a device if private, resolvable Bluetooth addresses are used.

12.7.3. *Configuration.* To receive events from the Security Manager, a callback is necessary. How to register this packet handler depends on your application configuration.

When *att_server* is used to provide a GATT/ATT service, *att_server* registers itself as the Security Manager packet handler. Security Manager events are then received by the application via the *att_server* packet handler.

If *att_server* is not used, you can directly register your packet handler with the security manager by calling *sm_register_packet_handler*.

The default SMP configuration in BTstack is to be as open as possible:

- accept all Short Term Key (STK) Generation methods,
- accept encryption key size from 7..16 bytes,
- expect no authentication requirements,
- don't support LE Secure Connections, and
- IO Capabilities set to *IO_CAPABILITY_NO_INPUT_NO_OUTPUT*.

You can configure these items by calling following functions respectively:

- *sm_set_accepted_stk_generation_methods*
- *sm_set_encryption_key_size_range*

- *sm_set_authentication_requirements* : add SM_AUTHREQ_SECURE_CONNECTION flag to enable LE Secure Connections
- *sm_set_io_capabilities*

12.7.4. *Identity Resolving.* Identity resolving is the process of matching a private, resolvable Bluetooth address to a previously paired device using its Identity Resolving (IR) key. After an LE connection gets established, BTstack automatically tries to resolve the address of this device. During this lookup, BTstack will emit the following events:

- *SM_EVENT_IDENTITY_RESOLVING_STARTED* to mark the start of a lookup,

and later:

- *SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED* on lookup success, or
- *SM_EVENT_IDENTITY_RESOLVING_FAILED* on lookup failure.

12.7.5. *User interaction.* Depending on the authentication requirements, IO capabilities, available OOB data, and the enabled STK generation methods, BTstack will request feedback from the app in the form of an event:

- *SM_EVENT_JUST_WORKS_REQUEST*: request a user to accept a Just Works pairing
- *SM_EVENT_PASSKEY_INPUT_NUMBER*: request user to input a passkey
- *SM_EVENT_PASSKEY_DISPLAY_NUMBER*: show a passkey to the user
- *SM_EVENT_NUMERIC_COMPARISON_REQUEST*: show a passkey to the user and request confirmation

To stop the bonding process, *sm_bonding_decline* should be called. Otherwise, *sm_just_works_confirm* or *sm_passkey_input* can be called.

After the bonding process, *SM_EVENT_JUST_WORKS_CANCEL*, *SM_EVENT_PASSKEY_DIS* or *SM_EVENT_NUMERIC_COMPARISON_CANCEL* is emitted to update the user interface if an Just Works request or a passkey has been shown before.

12.7.6. *Keypress Notifications.* As part of Bluetooth Core V4.2 specification, a device with a keyboard but no display can send keypress notifications to provide better user feedback. In BTstack, the *sm_keypress_notification()* function is used for sending notifications. Notifications are received by BTstack via the *SM_EVENT_KEYPRESS_NOTIFICATION* event.

12.7.7. *Cross-transport Key Derivation for LE Secure Connections.* In a dual-mode configuration, BTstack automatically generates an BR/EDR Link Key from the LE LTK via the Link Key Conversion function *h6*. It is then stored in the link key db.

To derive an LE LTK from a BR/EDR link key, the Bluetooth controller needs to support Secure Connections via NIST P-256 elliptic curves and the LE Secure Connections needs to get established via the LE Transport. BTstack does not support LE Secure Connections via LE Transport currently.

12.7.8. *Out-of-Band Data with LE Legacy Pairing.* LE Legacy Pairing can be made secure by providing a way for both devices to acquire a pre-shared secret 16 byte key by some fancy method. In most cases, this is not an option, especially since popular OS like iOS don't provide a way to specify it. In some applications, where both sides of a Bluetooth link are developed together, this could provide a viable option.

To provide OOB data, you can register an OOB data callback with *sm_register_oob_data_callback*.

13. Profiles

In the following, we explain how the various Bluetooth profiles are used in BTstack.

13.1. GAP - Generic Access Profile: Classic.

The GAP profile defines how devices find each other and establish a secure connection for other profiles. As mentioned before, the GAP functionality is split between and . Please check both.

13.1.1. *Become discoverable.* A remote unconnected Bluetooth device must be set as "discoverable" in order to be seen by a device performing the inquiry scan. To become discoverable, an application can call *gap_discoverable_control* with input parameter 1. If you want to provide a helpful name for your device, the application can set its local name by calling $gap_set_local_name$. To save energy, you may set the device as undiscoverable again, once a connection is established. See Listing 14 for an example.

```
int main(void){
    ...
    // make discoverable
    gap_discoverable_control(1);
    btstack_run_loop_execute();
    return 0;
}
void packet_handler (uint8_t packet_type, uint8_t *packet,
   uint16_t size){
     ...
     switch(state){
          case W4_CHANNEL_COMPLETE:
              // if connection is successful, make device
                 undiscoverable
              gap_discoverable_control(0);
          ...
     }
 }
```

Listing 14. Setting discoverable mode.

13.1.2. *Discover remote devices.* To scan for remote devices, the *hci_inquiry* command is used. Found remote devices are reported as a part of:

- HCI_EVENT_INQUIRY_RESULT,

- HCI_EVENT-*INQUIRY* RESULT_WITH_RSSI, or
- HCI_EVENT_EXTENDED_INQUIRY_RESPONSE events.

Each response contains at least the Bluetooth address, the class of device, the page scan repetition mode, and the clock offset of found device. The latter events add information about the received signal strength or provide the Extended Inquiry Result (EIR). A code snippet is shown in Listing 15 .

```
void print_inquiry_results(uint8_t *packet){
    int event = packet[0];
    int numResponses =
        hci_event_inquiry_result_get_num_responses(packet);
    uint16_t classOfDevice, clockOffset;
    uint8_t  rssi, pageScanRepetitionMode;
    for (i=0; i<numResponses; i++){
        bt_flip_addr(addr, &packet[3+i*6]);
        pageScanRepetitionMode = packet [3 + numResponses*6 + i
            ];
        if (event == HCI_EVENT_INQUIRY_RESULT){
            classOfDevice = little_endian_read_24(packet, 3 +
                numResponses*(6+1+1+1) + i*3);
            clockOffset =  little_endian_read_16(packet, 3 +
                numResponses*(6+1+1+1+3) + i*2) & 0x7fff;
            rssi  = 0;
        } else {
            classOfDevice = little_endian_read_24(packet, 3 +
                numResponses*(6+1+1)     + i*3);
            clockOffset =  little_endian_read_16(packet, 3 +
                numResponses*(6+1+1+3)   + i*2) & 0x7fff;
            rssi  = packet [3 + numResponses*(6+1+1+3+2) + i*1];
        }
        printf("Device found: %s with COD: 0x%06x, pageScan %u,
            clock offset 0x%04x, rssi 0x%02x\n", bd_addr_to_str(
            addr), classOfDevice, pageScanRepetitionMode,
            clockOffset, rssi);
    }
}

void packet_handler (uint8_t packet_type, uint8_t *packet,
    uint16_t size){
    ...
    switch (event) {
         case HCI_STATE_WORKING:
            hci_send_cmd(&hci_write_inquiry_mode, 0x01); // with
                RSSI
            break;
        case HCI_EVENT_COMMAND_COMPLETE:
            if (COMMAND_COMPLETE_EVENT(packet,
                hci_write_inquiry_mode) ) {
                 start_scan();
            }
        case HCI_EVENT_COMMAND_STATUS:
            if (COMMAND_STATUS_EVENT(packet,
                hci_write_inquiry_mode) ) {
                 printf("Ignoring error (0x%x) from
                     hci_write_inquiry_mode.\n", packet[2]);
                 hci_send_cmd(&hci_inquiry, HCI_INQUIRY_LAP,
                     INQUIRY_INTERVAL, 0);
            }
            break;
        case HCI_EVENT_INQUIRY_RESULT:
        case HCI_EVENT_INQUIRY_RESULT_WITH_RSSI:
            print_inquiry_results(packet);
            break;
        ...
    }
}
```

Listing 15. Discover remote devices.

By default, neither RSSI values nor EIR are reported. If the Bluetooth device implements Bluetooth Specification 2.1 or higher, the *hci_write_inquiry_mode* command enables reporting of this advanced features (0 for standard results, 1 for RSSI, 2 for RSSI and EIR).

A complete GAP inquiry example is provided here.

13.1.3. *Pairing of Devices.* By default, Bluetooth communication is not authenticated, and any device can talk to any other device. A Bluetooth device (for example, cellular phone) may choose to require authentication to provide a particular service (for example, a Dial-Up service). The process of establishing authentication is called pairing. Bluetooth provides two mechanism for this.

On Bluetooth devices that conform to the Bluetooth v2.0 or older specification, a PIN code (up to 16 bytes ASCII) has to be entered on both sides. This isn't optimal for embedded systems that do not have full I/O capabilities. To support pairing with older devices using a PIN, see Listing 16 .

```
void packet_handler (uint8_t packet_type, uint8_t *packet,
    uint16_t size){
    ...
    switch (event) {
        case HCI_EVENT_PIN_CODE_REQUEST:
            // inform about pin code request
            printf("Pin code request - using '0000'\n\r");
            hci_event_pin_code_request_get_bd_addr(packet,
                bd_addr);

            // baseband address, pin length, PIN: c-string
            hci_send_cmd(&hci_pin_code_request_reply, &bd_addr,
                4, "0000");
            break;
        ...
    }
}
```

Listing 16. PIN code request.

The Bluetooth v2.1 specification introduces Secure Simple Pairing (SSP), which is a better approach as it both improves security and is better adapted to embedded systems. With SSP, the devices first exchange their IO Capabilities and then settle on one of several ways to verify that the pairing is legitimate. If the Bluetooth device supports SSP, BTstack enables it by default and even automatically accepts SSP pairing requests. Depending on the product in which BTstack is used, this may not be desired and should be replaced with code to interact with the user.

Regardless of the authentication mechanism (PIN/SSP), on success, both devices will generate a link key. The link key can be stored either in the Bluetooth module itself or in a persistent storage, see here. The next time the device connects and requests an authenticated connection, both devices can use the previously generated link key. Please note that the pairing must be repeated if the link key is lost by one device.

13.1.4. *Dedicated Bonding.* Aside from the regular bonding, Bluetooth also provides the concept of "dedicated bonding", where a connection is established for the sole purpose of bonding the device. After the bonding process is over, the connection will be automatically terminated. BTstack supports dedicated bonding via the *gap_dedicated_bonding* function.

13.2. SPP - Serial Port Profile.

The SPP profile defines how to set up virtual serial ports and connect two Bluetooth enabled devices.

13.2.1. *Accessing an SPP Server on a remote device.* To access a remote SPP server, you first need to query the remote device for its SPP services. Section on querying remote SDP service shows how to query for all RFCOMM channels. For SPP, you can do the same but use the SPP UUID 0x1101 for the query. After you have identified the correct RFCOMM channel, you can create an RFCOMM connection as shown here.

13.2.2. *Providing an SPP Server.* To provide an SPP Server, you need to provide an RFCOMM service with a specific RFCOMM channel number as explained in section on RFCOMM service. Then, you need to create an SDP record for it and publish it with the SDP server by calling *sdp_register_service*. BTstack provides the *spp_create_sdp_record* function in that requires an empty buffer of approximately 200 bytes, the service channel number, and a service name. Have a look at the [SPP Counter example](examples/generated/#sec:sppcounterExample].

13.3. PAN - Personal Area Networking Profile.

The PAN profile uses BNEP to provide on-demand networking capabilities between Bluetooth devices. The PAN profile defines the following roles:

- PAN User (PANU)
- Network Access Point (NAP)
- Group Ad-hoc Network (GN)

PANU is a Bluetooth device that communicates as a client with GN, or NAP, or with another PANU Bluetooth device, through a point-to-point connection. Either the PANU or the other Bluetooth device may terminate the connection at anytime.

NAP is a Bluetooth device that provides the service of routing network packets between PANU by using BNEP and the IP routing mechanism. A NAP can also act as a bridge between Bluetooth networks and other network technologies by using the Ethernet packets.

The GN role enables two or more PANUs to interact with each other through a wireless network without using additional networking hardware. The devices are connected in a piconet where the GN acts as a master and communicates either point-to-point or a point-to-multipoint with a maximum of seven PANU slaves by using BNEP.

Currently, BTstack supports only PANU.

13.3.1. *Accessing a remote PANU service.* To access a remote PANU service, you first need perform an SDP query to get the L2CAP PSM for the requested PANU UUID. With these two pieces of information, you can connect BNEP to the remote PANU service with the *bnep_connect* function. The Section on PANU Demo example shows how this is accomplished.

13.3.2. *Providing a PANU service.* To provide a PANU service, you need to provide a BNEP service with the service UUID, e.g. the PANU UUID, and a maximal ethernet frame size, as explained in Section on BNEP service. Then, you need to create an SDP record for it and publish it with the SDP server by calling *sdp_register_service*. BTstack provides the *pan_create_panu_sdp_record* function in *src/pan.c* that requires an empty buffer of approximately 200 bytes, a description, and a security description.

13.4. **HSP - Headset Profile.** The HSP profile defines how a Bluetooth-enabled headset should communicate with another Bluetooth enabled device. It relies on SCO for audio encoded in 64 kbit/s CVSD and a subset of AT commands from GSM 07.07 for minimal controls including the ability to ring, answer a call, hang up and adjust the volume.

The HSP defines two roles: - Audio Gateway (AG) - a device that acts as the gateway of the audio, typically a mobile phone or PC. - Headset (HS) - a device that acts as the AG's remote audio input and output control.

There are following restrictions: - The CVSD is used for audio transmission. - Between headset and audio gateway, only one audio connection at a time is supported. - The profile offers only basic interoperability – for example, handling of multiple calls at the audio gateway is not included. - The only assumption on the headset's user interface is the possibility to detect a user initiated action (e.g. pressing a button).

%TODO: audio paths

13.5. **HFP - Hands-Free Profile.** The HFP profile defines how a Bluetooth-enabled device, e.g. a car kit or a headset, can be used to place and receive calls via a audio gateway device, typically a mobile phone. It relies on SCO for audio encoded in 64 kbit/s CVSD and a bigger subset of AT commands from

GSM 07.07 then HSP for controls including the ability to ring, to place and receive calls, join a conference call, to answer, hold or reject a call, and adjust the volume.

The HFP defines two roles: - Audio Gateway (AG) – a device that acts as the gateway of the audio,, typically a mobile phone. - Hands-Free Unit (HF) – a device that acts as the AG's remote audio input and output control.

%TODO: audio paths

13.6. GAP LE - Generic Access Profile for Low Energy.

As with GAP for Classic, the GAP LE profile defines how to discover and how to connect to a Bluetooth Low Energy device. There are several GAP roles that a Bluetooth device can take, but the most important ones are the Central and the Peripheral role. Peripheral devices are those that provide information or can be controlled. Central devices are those that consume information or control the peripherals. Before the connection can be established, devices are first going through an advertising process.

13.6.1. *Private addresses.*

To better protect privacy, an LE device can choose to use a private i.e. random Bluetooth address. This address changes at a user-specified rate. To allow for later reconnection, the central and peripheral devices will exchange their Identity Resolving Keys (IRKs) during bonding. The IRK is used to verify if a new address belongs to a previously bonded device.

To toggle privacy mode using private addresses, call the *gap_random_address_set_mode* function. The update period can be set with *gap_random_address_set_update_period*.

After a connection is established, the Security Manager will try to resolve the peer Bluetooth address as explained in Section on SMP.

13.6.2. *Advertising and Discovery.*

An LE device is discoverable and connectable, only if it periodically sends out Advertisements. An advertisement contains up to 31 bytes of data. To configure and enable advertisement broadcast, the following GAP functions can be used:

- *gap_advertisements_set_data*
- *gap_advertisements_set_params*
- *gap_advertisements_enable*

In addition to the Advertisement data, a device in the peripheral role can also provide Scan Response data, which has to be explicitly queried by the central device. It can be set with *gap_scan_response_set_data*.

Please have a look at the SPP and LE Counter example.

The scan parameters can be set with *gap_set_scan_parameters*. The scan can be started/stopped with *gap_start_scan*/*gap_stop_scan*.

Finally, if a suitable device is found, a connection can be initiated by calling *gap_connect*. In contrast to Bluetooth classic, there is no timeout for an LE connection establishment. To cancel such an attempt, *gap_connect_cancel* has be be called.

By default, a Bluetooth device stops sending Advertisements when it gets into the Connected state. However, it does not start broadcasting advertisements on disconnect again. To re-enable it, please send the *hci_le_set_advertise_enable* again .

13.7. **GATT - Generic Attribute Profile.** The GATT profile uses ATT Attributes to represent a hierarchical structure of GATT Services and GATT Characteristics. Each Service has one or more Characteristics. Each Characteristic has meta data attached like its type or its properties. This hierarchy of Characteristics and Services are queried and modified via ATT operations.

GATT defines both a server and a client role. A device can implement one or both GATT roles.

13.7.1. *GATT Client.* The GATT Client is used to discover services, and their characteristics and descriptors on a peer device. It can also subscribe for notifications or indications that the characteristic on the GATT server has changed its value.

To perform GATT queries, provides a rich interface. Before calling queries, the GATT client must be initialized with *gatt_client_init* once.

To allow for modular profile implementations, GATT client can be used independently by multiple entities.

To use it by a GATT client, you register a packet handler with *gatt_client_register_packet_handler*. The return value of that is a GATT client ID which has to be provided in all queries.

After an LE connection was created using the GAP LE API, you can query for the connection MTU with *gatt_client_get_mtu*.

GATT queries cannot be interleaved. Therefore, you can check if you can perform a GATT query on a particular connection using *gatt_client_is_ready*. As a result to a GATT query, zero to many *le_events* are returned before a *GATT_EVENT_QUERY_COMPLETE* event completes the query.

For more details on the available GATT queries, please consult GATT Client API.

13.7.2. *GATT Server.* The GATT server stores data and accepts GATT client requests, commands and confirmations. The GATT server sends responses to requests and when configured, sends indication and notifications asynchronously to the GATT client.

To save on both code space and memory, BTstack does not provide a GATT Server implementation. Instead, a textual description of the GATT profile is directly converted into a compact internal ATT Attribute database by a GATT profile compiler. The ATT protocol server - implemented by and - answers incoming ATT requests based on information provided in the compiled database and provides read- and write-callbacks for dynamic attributes.

GATT profiles are defined by a simple textual comma separated value (.csv) representation. While the description is easy to read and edit, it is compact and can be placed in ROM.

The current format is shown in Listing 17 .

```
// import service_name
#import <service_name.gatt>

PRIMARY_SERVICE, {SERVICE_UUID}
CHARACTERISTIC, {ATTRIBUTE_TYPE_UUID}, {PROPERTIES}, {VALUE}
```

```
    CHARACTERISTIC, {ATTRIBUTE_TYPE_UUID}, {PROPERTIES}, {VALUE}
    ...
    PRIMARY_SERVICE, {SERVICE_UUID}
    CHARACTERISTIC, {ATTRIBUTE_TYPE_UUID}, {PROPERTIES}, {VALUE}
    ...
```

Listing 17. GATT profile.

Properties can be a list of READ | WRITE | WRITE_WITHOUT_RESPONSE | NOTIFY | INDICATE | DYNAMIC.

Value can either be a string ("this is a string"), or, a sequence of hex bytes (e.g. 01 02 03).

UUIDs are either 16 bit (1800) or 128 bit (00001234-0000-1000-8000-00805F9B34FB).

Reads/writes to a Characteristic that is defined with the DYNAMIC flag, are forwarded to the application via callback. Otherwise, the Characteristics cannot be written and it will return the specified constant value.

Adding NOTIFY and/or INDICATE automatically creates an addition Client Configuration Characteristic.

To require encryption or authentication before a Characteristic can be accessed, you can add ENCRYPTION_KEY_SIZE_X - with $X \in [7..16]$ - or AUTHENTICATION_REQUIRED.

To use already implemented GATT Services, you can import it using the *#import* command. See list of provided services.

BTstack only provides an ATT Server, while the GATT Server logic is mainly provided by the GATT compiler. While GATT identifies Characteristics by UUIDs, ATT uses Handles (16 bit values). To allow to identify a Characteristic without hard-coding the attribute ID, the GATT compiler creates a list of defines in the generated *.h file.

Similar to other protocols, it might be not possible to send any time. To send a Notification, you can call *att_server_request_can_send_now* to receive a ATT_EVENT_CAN_SEND_NOW event.

13.7.3. *Implementing Standard GATT Services.*

Implementation of a standard GATT Service consists of the following 4 steps:

1. Identify full Service Name
2. Use Service Name to fetch XML definition at Bluetooth SIG site and convert into generic .gatt file
3. Edit .gatt file to set constant values and exclude unwanted Characteristics
4. Implement Service server, e.g., battery_service_server.c

Step 1:

To facilitate the creation of .gatt files for standard profiles defined by the Bluetooth SIG, the *tool/convert_gatt_service.py* script can be used. When run without a parameter, it queries the Bluetooth SIG website and lists the available Services by their Specification Name, e.g., *org.bluetooth.service.battery_service*.

```
$ tool/convert_gatt_service.py
```

```
Fetching list of services from https://www.bluetooth.com/
    specifications/gatt/services

Specification Type                                            |
    Specification Name               | UUID
--------------------------------------------------------------+-----------------------------

org.bluetooth.service.alert_notification                      | Alert
    Notification Service      | 0x1811
org.bluetooth.service.automation_io                           | Automation
    IO                       | 0x1815
org.bluetooth.service.battery_service                         | Battery
    Service                  | 0x180F
...
org.bluetooth.service.weight_scale                            | Weight
    Scale                    | 0x181D

To convert a service into a .gatt file template, please call the
    script again with the requested Specification Type and the
    output file name
Usage: tool/convert_gatt_service.py SPECIFICATION_TYPE [service_name
    .gatt]
```

Step 2:

To convert service into .gatt file, call *tool/convert_gatt_service.py with the requested Specification Type and the output file name.

```
$ tool/convert_gatt_service.py org.bluetooth.service.battery_service
     battery_service.gatt
Fetching org.bluetooth.service.battery_service from
https://www.bluetooth.com/api/gatt/xmlfile?xmlFileName=org.bluetooth
    .service.battery_service.xml

Service Battery Service
- Characteristic Battery Level - properties ['Read', 'Notify']
-- Descriptor Characteristic Presentation Format      - TODO:
    Please set values
-- Descriptor Client Characteristic Configuration

Service successfully converted into battery_service.gatt
Please check for TODOs in the .gatt file
```

Step 3:

In most cases, you will need to customize the .gatt file. Please pay attention to the tool output and have a look at the generated .gatt file.

E.g. in the generated .gatt file for the Battery Service

```
// Specification Type org.bluetooth.service.battery_service
// https://www.bluetooth.com/api/gatt/xmlfile?xmlFileName=org.
    bluetooth.service.battery_service.xml
```

```
// Battery Service 180F
PRIMARY_SERVICE, ORG_BLUETOOTH_SERVICE_BATTERY_SERVICE
CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_BATTERY_LEVEL, DYNAMIC
    | READ | NOTIFY,
// TODO: Characteristic Presentation Format: please set values
#TODO CHARACTERISTIC_FORMAT, READ, _format_, _exponent_, _unit_,
    _name_space_, _description_
CLIENT_CHARACTERISTIC_CONFIGURATION, READ | WRITE,
```

you could delete the line regarding the CHARACTERISTIC_FORMAT, since it's not required if there is a single instance of the service. Please compare the .gatt file against the Adopted Specifications.

Step 4:

As described above all read/write requests are handled by the application. To implement the new services as a reusable module, it's neccessary to get access to all read/write requests related to this service.

For this, the ATT DB allows to register read/write callbacks for a specific handle range with *att_server_register_can_send_now_callback()*.

Since the handle range depends on the application's .gatt file, the handle range for Primary and Secondary Services can be queried with *gatt_server_get_get_handle_range_for_service*

Similarly, you will need to know the attribute handle for particular Characteristics to handle Characteristic read/writes requests. You can get the attribute value handle for a Characteristics *gatt_server_get_value_handle_for_characteristic_with_uuid16()*.

In addition to the attribute value handle, the handle for the Client Characteristic Configuration is needed to support Indications/Notifications. You can get this attribute handle with *gatt_server_get_client_configuration_handle_for_characteristic_with_uuid16()*

Finally, in order to send Notifications and Indications independently from the main application, *att_server_register_can_send_now_callback* can be used to request a callback when it's possible to send a Notification or Indication.

To see how this works together, please check out the Battery Service Server in *src/ble/battery_service_server.c*.

14. Implemented GATT Services

BTstack allows to implement and use GATT Services in a modular way.

To use an already implemented GATT Service, you only have to add it to your application's GATT file with:

- *#import* for .gatt files located in *src/ble/gatt-service*
- *#import "service_name.gatt"* for .gatt files located in the same folder as your application's .gatt file.

Each service will have an API at *src/ble/gatt-service/service_name_server.h. To activate it, you need to call* service_name_init(..)*. Please see the .h file for details.

14.0.4. *Battery Service {#sec:BatteryService}*. The Battery Service allows to query your device's battery level in a standardized way.

After adding it to your .gatt file, you call *battery_service_server_init(value)* with the current value of your battery. The valid range for the battery level is 0-100.

If the battery level changes, you can call *battery_service_server_set_battery_value(value)*. The service supports sending Notifications if the client enables them.

15. Embedded Examples

In this section, we will describe a number of examples from the *example* folder. To allow code-reuse with different platforms as well as with new ports, the low-level initialization of BTstack and the hardware configuration has been extracted to the various *platforms/PLATFORM/main.c* files. The examples only contain the platform-independent Bluetooth logic. But let's have a look at the common init code.

Listing 18 shows a minimal platform setupfor an embedded system with a Bluetooth chipset connected via UART.

```
int main(){
  // ... hardware init: watchdoch, IOs, timers, etc...

  // setup BTstack memory pools
  btstack_memory_init();

  // select embedded run loop
  btstack_run_loop_init(btstack_run_loop_embedded_get_instance()
      );

  // use logger: format HCI_DUMP_PACKETLOGGER, HCI_DUMP_BLUEZ or
      HCI_DUMP_STDOUT
  hci_dump_open(NULL, HCI_DUMP_STDOUT);

  // init HCI
  hci_transport_t     * transport = hci_transport_h4_instance();
  hci_init(transport, NULL);

  // setup example
  btstack_main(argc, argv);

  // go
  btstack_run_loop_execute();
}
```

Listing 18. Minimal platform setup for an embedded system

First, BTstack's memory pools are setup up. Then, the standard run loop implementation for embedded systems is selected.

The call to *hci_dump_open* configures BTstack to output all Bluetooth packets and it's own debug and error message via printf. The Python script *tools/create_packet_log.py* can be used to convert the console output into a Bluetooth PacketLogger format that can be opened by the OS X PacketLogger tool as well as by Wireshark for further inspection. When asking for help, please always include a log created with HCI dump.

The *hci_init* function sets up HCI to use the HCI H4 Transport implementation. It doesn't provide a special transport configuration nor a special implementation for a particular Bluetooth chipset. It makes use of the *remote_device_db_memory* implementation that allows for re-connects without a new pairing but doesn't persist the bonding information.

Finally, it calls *btstack_main()* of the actual example before executing the run loop.

- Hello World example:
  - led_counter: Hello World: blinking LED without Bluetooth.
- GAP example:
  - gap_inquiry: GAP Inquiry Example.
- SDP Queries examples:
  - sdp_general_query: Dump remote SDP Records.
  - sdp_bnep_query: Dump remote BNEP PAN protocol UUID and L2CAP PSM.
- SPP Server examples:
  - spp_counter: SPP Server - Heartbeat Counter over RFCOMM.
  - spp_flowcontrol: SPP Server - Flow Control.
- BNEP/PAN example:
  - panu_demo: PANU Demo.
- HSP examples:
  - hsp_hs_demo: HSP Headset Demo.
  - hsp_ag_demo: HSP Audio Gateway Demo.
- HFP examples:
  - hfp_hf_demo: HFP Hands-Free (HF) Demo.
  - hfp_ag_demo: HFP Audio Gateway (AG) Demo.
- Low Energy examples:
  - gap_le_advertisements: GAP LE Advertisements Dumper.
  - gatt_browser: GATT Client - Discovering primary services and their characteristics.
  - le_counter: LE Peripheral - Heartbeat Counter over GATT.
  - le_streamer: LE Peripheral - Stream data over GATT.
- Dual Mode example:
  - spp_and_le_counter: Dual mode example.

15.1. **led_counter: Hello World: blinking LED without Bluetooth.** The example demonstrates how to provide a periodic timer to toggle an LED and send debug messages to the console as a minimal BTstack test.

15.1.1. *Periodic Timer Setup.* As timers in BTstack are single shot, the periodic counter is implemented by re-registering the timer source in the heartbeat handler callback function. Listing 19 shows heartbeat handler adapted to periodically toggle an LED and print number of toggles.

```
static void heartbeat_handler(btstack_timer_source_t *ts){
  UNUSED(ts);

  // increment counter
  char lineBuffer[30];
  sprintf(lineBuffer, "BTstack counter %04u\n\r", ++counter);
  puts(lineBuffer);

  // toggle LED
  hal_led_toggle();

  // re-register timer
  btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(&heartbeat);
}
```

Listing 19. Periodic counter

15.1.2. *Main Application Setup.* Listing 20 shows mainapplication code. It configures the heartbeat tier and adds it to the run loop.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]){
  (void)argc;
  (void)argv;

  // set one-shot timer
  heartbeat.process = &heartbeat_handler;
  btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(&heartbeat);

  printf("Running...\n\r");
  return 0;
}
```

Listing 20. Setup heartbeat timer

15.2. gap_inquiry: GAP Inquiry Example.

The Generic Access Profile (GAP) defines how Bluetooth devices discover and establish a connection with each other. In this example, the application discovers surrounding Bluetooth devices and collects their Class of Device (CoD), page scan mode, clock offset, and RSSI. After that, the remote name of each device is requested. In the following section we outline the Bluetooth logic part, i.e., how the packet handler handles the asynchronous events and data packets.

15.2.1. *Bluetooth Logic.* The Bluetooth logic is implemented as a state machine within the packet handler. In this example, the following states are passed sequentially: INIT, and ACTIVE.

In INIT, an inquiry scan is started, and the application transits to ACTIVE state.

In ACTIVE, the following events are processed:

- GAP Inquiry result event: BTstack provides a unified inquiry result that contain Class of Device (CoD), page scan mode, clock offset. RSSI and name (from EIR) are optional.
- Inquiry complete event: the remote name is requested for devices without a fetched name. The state of a remote name can be one of the following: REMOTE_NAME_REQUEST, REMOTE_NAME_INQUIRED, or REMOTE_NAME_FETCHED.
- Remote name request complete event: the remote name is stored in the table and the state is updated to REMOTE_NAME_FETCHED. The query of remote names is continued.

For more details on discovering remote devices, please see Section on GAP.

15.2.2. *Main Application Setup.* Listing 21 shows mainapplication code. It registers the HCI packet handler and starts the Bluetooth stack.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]) {
  (void)argc;
  (void)argv;

  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  // enabled EIR
  hci_set_inquiry_mode(INQUIRY_MODE_RSSI_AND_EIR);

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

Listing 21. Setup packet handler for GAP inquiry

15.3. sdp_general_query: Dump remote SDP Records.

The example shows how the SDP Client is used to get a list of service records on a remote device.

15.3.1. *SDP Client Setup.* SDP is based on L2CAP. To receive SDP query events you must register a callback, i.e. query handler, with the SPD parser, as shown in Listing 22 . Via this handler, theSDP client will receive the following events:

- SDP_EVENT_QUERY_ATTRIBUTE_VALUE containing the results of the query in chunks,
- SDP_EVENT_QUERY_COMPLETE indicating the end of the query and the status

```
static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size);
static void handle_sdp_client_query_result(uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size);

static void sdp_client_init(void){

  // register for HCI events
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  // init L2CAP
  l2cap_init();
}
```

Listing 22. SDP client setup

15.3.2. *SDP Client Query.* To trigger an SDP query to get the a list of service records on a remote device, you need to call sdp_client_query_uuid16() with the remote address and the UUID of the public browse group, as shown in Listing 24 . In this example weused fixed address of the remote device shown in Listing 23 . Please update it with theaddress of a device in your vicinity, e.g., one reported by the GAP Inquiry example in the previous section.

```
static bd_addr_t remote = {0x04,0x0C,0xCE,0xE4,0x85,0xD3};
```

Listing 23. Address of remote device in big-endian order

```
static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size){
  UNUSED(channel);
  UNUSED(size);

  if (packet_type != HCI_EVENT_PACKET) return;
  uint8_t event = hci_event_packet_get_type(packet);

  switch (event) {
    case BTSTACK_EVENT_STATE:
      // BTstack activated, get started
      if (btstack_event_state_get_state(packet) ==
          HCI_STATE_WORKING){
        sdp_client_query_uuid16(&handle_sdp_client_query_result,
            remote, BLUETOOTH_ATTRIBUTE_PUBLIC_BROWSE_ROOT);
      }
      break;
    default:
      break;
  }
}
```

Listing 24. Querying a list of service records on a remote device.

15.3.3. *Handling SDP Client Query Results.* The SDP Client returns the results of the query in chunks. Each result packet contains the record ID, the Attribute ID, and a chunk of the Attribute value. In this example, we append new chunks for the same Attribute ID in a large buffer, see Listing 25 . To save memory, it's also possible to process these chunks directly by a custom stream parser, similar to the way XML files are parsed by a SAX parser. Have a look at *src/sdp_client_rfcomm.c* which retrieves the RFCOMM channel number and the service name.

```
    static void handle_sdp_client_query_result(uint8_t packet_type,
        uint16_t channel, uint8_t *packet, uint16_t size){
      UNUSED(packet_type);
      UNUSED(channel);
      UNUSED(size);

      switch (packet[0]){
        case SDP_EVENT_QUERY_ATTRIBUTE_VALUE:
          // handle new record
          if (sdp_event_query_attribute_byte_get_record_id(packet)
              != record_id){
            record_id = sdp_event_query_attribute_byte_get_record_id
                (packet);
            printf("\n---\nRecord nr. %u\n", record_id);
          }

          assertBuffer(
              sdp_event_query_attribute_byte_get_attribute_length(
              packet));

          attribute_value[
              sdp_event_query_attribute_byte_get_data_offset(packet)
              ] = sdp_event_query_attribute_byte_get_data(packet);
          if ((uint16_t)(
              sdp_event_query_attribute_byte_get_data_offset(packet)
              +1) ==
              sdp_event_query_attribute_byte_get_attribute_length(
              packet)){
              printf("Attribute 0x%04x: ",
                  sdp_event_query_attribute_byte_get_attribute_id(
                  packet));
              de_dump_data_element(attribute_value);
          }
          break;
        case SDP_EVENT_QUERY_COMPLETE:
          if (sdp_event_query_complete_get_status(packet)){
            printf("SDP query failed 0x%02x\n",
                sdp_event_query_complete_get_status(packet));
            break;
          }
          printf("SDP query done.\n");
          break;
      }
    }
```

Listing 25. Handling query result chunks.

15.4. **sdp_bnep_query: Dump remote BNEP PAN protocol UUID and L2CAP PSM.** The example shows how the SDP Client is used to get all BNEP service records from a remote device. It extracts the remote BNEP PAN protocol UUID and the L2CAP PSM, which are needed to connect to a remote BNEP service.

15.4.1. *SDP Client Setup.* As with the previous example, you must register a callback, i.e. query handler, with the SPD parser, as shown in Listing 26 . Via this handler, theSDP client will receive events:

- SDP_EVENT_QUERY_ATTRIBUTE_VALUE containing the results of the query in chunks,
- SDP_EVENT_QUERY_COMPLETE reporting the status and the end of the query.

```
    static void packet_handler (uint8_t packet_type, uint16_t
        channel, uint8_t *packet, uint16_t size);
    static void handle_sdp_client_query_result(uint8_t packet_type,
        uint16_t channel, uint8_t *packet, uint16_t size);

    static void sdp_client_init(void){

      // register for HCI events
      hci_event_callback_registration.callback = &packet_handler;
      hci_add_event_handler(&hci_event_callback_registration);

      // init L2CAP
      l2cap_init();
    }
```

Listing 26. SDP client setup

15.4.2. *SDP Client Query.*

```
    static void packet_handler (uint8_t packet_type, uint16_t
        channel, uint8_t *packet, uint16_t size){
      UNUSED(channel);
    UNUSED(size);

    if (packet_type != HCI_EVENT_PACKET) return;
    uint8_t event = hci_event_packet_get_type(packet);

    switch (event) {
      case BTSTACK_EVENT_STATE:
        // BTstack activated, get started
        if (btstack_event_state_get_state(packet) ==
            HCI_STATE_WORKING){
          printf("Start SDP BNEP query.\n");
          sdp_client_query_uuid16(&handle_sdp_client_query_result,
              remote, BLUETOOTH_PROTOCOL_BNEP);
        }
        break;
      default:
        break;
    }
  }
```

Listing 27. Querying the a list of service records on a remote device.

15.4.3. *Handling SDP Client Query Result.* The SDP Client returns the result of the query in chunks. Each result packet contains the record ID, the Attribute ID, and a chunk of the Attribute value, see Listing 28 . Here, we show howto parse the Service Class ID List and Protocol Descriptor List, as they contain the BNEP Protocol UUID and L2CAP PSM respectively.

```
  static void handle_sdp_client_query_result(uint8_t packet_type,
      uint16_t channel, uint8_t *packet, uint16_t size){
    UNUSED(packet_type);
    UNUSED(channel);
    UNUSED(size);

  ...

            switch(sdp_event_query_attribute_byte_get_attribute_id(
                packet)){
              // 0x0001 "Service Class ID List"
              case BLUETOOTH_ATTRIBUTE_SERVICE_CLASS_ID_LIST:
                if (de_get_element_type(attribute_value) != DE_DES)
                    break;
                for (des_iterator_init(&des_list_it, attribute_value
                    ); des_iterator_has_more(&des_list_it);
                    des_iterator_next(&des_list_it)){
                  uint8_t * element = des_iterator_get_element(&
                      des_list_it);
                  if (de_get_element_type(element) != DE_UUID)
                      continue;
                  uint32_t uuid = de_get_uuid32(element);
                  switch (uuid){
                    case BLUETOOTH_SERVICE_CLASS_PANU:
                    case BLUETOOTH_SERVICE_CLASS_NAP:
                    case BLUETOOTH_SERVICE_CLASS_GN:
                      printf(" ** Attribute 0x%04x: BNEP PAN
                          protocol UUID: %04x\n",
                          sdp_event_query_attribute_byte_get_attribute_id
                          (packet), uuid);
                      break;
                    default:
                      break;
                  }
                }
                break;
    ...
              case BLUETOOTH_ATTRIBUTE_PROTOCOL_DESCRIPTOR_LIST:{
                  printf(" ** Attribute 0x%04x: ",
                      sdp_event_query_attribute_byte_get_attribute_id
                      (packet));

                  uint16_t l2cap_psm = 0;
                  uint16_t bnep_version = 0;
                  for (des_iterator_init(&des_list_it,
                      attribute_value); des_iterator_has_more(&
                      des_list_it); des_iterator_next(&des_list_it))
                      {
                    if (des_iterator_get_type(&des_list_it) !=
                        DE_DES) continue;
                    uint8_t * des_element = des_iterator_get_element
                        (&des_list_it);
                    des_iterator_init(&prot_it, des_element);
                    uint8_t * element = des_iterator_get_element(&
                        prot_it);

                    if (de_get_element_type(element) != DE_UUID)
                        continue;
                    uint32_t uuid = de_get_uuid32(element);
                    switch (uuid){
                      case BLUETOOTH_PROTOCOL_L2CAP:
                        if (!des_iterator_has_more(&prot_it))
                            continue;
                        des_iterator_next(&prot_it);
                        de_element_get_uint16(
                            des_iterator_get_element(&prot_it), &
                            l2cap_psm);
                        break;
                      case BLUETOOTH_PROTOCOL_BNEP:
                        if (!des_iterator_has_more(&prot_it))
                            continue;
                        des_iterator_next(&prot_it);
                        de_element_get_uint16(
                            des_iterator_get_element(&prot_it), &
                            bnep_version);
                        break;
                      default:
                        break;
                    }
                }
                printf("l2cap_psm 0x%04x, bnep_version 0x%04x\n",
                    l2cap_psm, bnep_version);
            }
            break;
    ...
}
```

Listing 28. Extracting BNEP Protcol UUID and L2CAP PSM

The Service Class ID List is a Data Element Sequence (DES) of UUIDs. The BNEP PAN protocol UUID is within this list.

The Protocol Descriptor List is DES which contains one DES for each protocol. For PAN serivces, it contains a DES with the L2CAP Protocol UUID and a PSM, and another DES with the BNEP UUID and the the BNEP version.

15.5. spp_counter: SPP Server - Heartbeat Counter over RFCOMM.

The Serial port profile (SPP) is widely used as it provides a serial port over Bluetooth. The SPP counter example demonstrates how to setup an SPP service, and provide a periodic timer over RFCOMM.

15.5.1. *SPP Service Setup.*

To provide an SPP service, the L2CAP, RFCOMM, and SDP protocol layers are required. After setting up an RFCOMM service with channel nubmer RFCOMM_SERVER_CHANNEL, an SDP record is created and registered with the SDP server. Example code for SPP service setup is provided in Listing 29 . The SDP record createdby function spp_create_sdp_record consists of a basic SPP definition that uses the provided RFCOMM channel ID and service name. For more details, please have a look at it in `src/sdp_util.c`. The SDP record is created on the fly in RAM and is deterministic. To preserve valuable RAM, the result could be stored as constant data inside the ROM.

```
static void spp_service_setup(void){

    // register for HCI events
    hci_event_callback_registration.callback = &packet_handler;
    hci_add_event_handler(&hci_event_callback_registration);

    l2cap_init();

    rfcomm_init();
    rfcomm_register_service(packet_handler, RFCOMM_SERVER_CHANNEL,
        0xffff);  // reserved channel, mtu limited by l2cap

    // init SDP, create record for SPP and register with SDP
    sdp_init();
    memset(spp_service_buffer, 0, sizeof(spp_service_buffer));
    spp_create_sdp_record(spp_service_buffer, 0x10001,
       RFCOMM_SERVER_CHANNEL, "SPP Counter");
    sdp_register_service(spp_service_buffer);
    printf("SDP service record size: %u\n", de_get_len(
        spp_service_buffer));
}
```

Listing 29. SPP service setup

15.5.2. *Periodic Timer Setup.* The heartbeat handler increases the real counter every second, and sends a text string with the counter value, as shown in Listing 30 .

```
static btstack_timer_source_t heartbeat;
static char lineBuffer[30];
static void  heartbeat_handler(struct btstack_timer_source *ts){
    static int counter = 0;

    if (rfcomm_channel_id){
        sprintf(lineBuffer, "BTstack counter %04u\n", ++counter);
        printf("%s", lineBuffer);

        rfcomm_request_can_send_now_event(rfcomm_channel_id);
    }

    btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
    btstack_run_loop_add_timer(ts);
}

static void one_shot_timer_setup(void){
    // set one-shot timer
    heartbeat.process = &heartbeat_handler;
    btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
    btstack_run_loop_add_timer(&heartbeat);
}
```

Listing 30. Periodic Counter

15.5.3. *Bluetooth Logic.* The Bluetooth logic is implemented within the packet handler, see Listing 31 . In thisexample, the following events are passed sequentially:

- BTSTACK_EVENT_STATE,
- HCI_EVENT_PIN_CODE_REQUEST (Standard pairing) or
- HCI_EVENT_USER_CONFIRMATION_REQUEST (Secure Simple Pairing),
- RFCOMM_EVENT_INCOMING_CONNECTION,
- RFCOMM_EVENT_CHANNEL_OPENED,
- RFCOMM_EVENT_CHANNEL_CLOSED

Upon receiving HCI_EVENT_PIN_CODE_REQUEST event, we need to handle authentication. Here, we use a fixed PIN code "0000".

When HCI_EVENT_USER_CONFIRMATION_REQUEST is received, the user will be asked to accept the pairing request. If the IO capability is set to SSP_IO_CAPABILITY_DISPLAY_YES_NO, the request will be automatically accepted.

The RFCOMM_EVENT_INCOMING_CONNECTION event indicates an incoming connection. Here, the connection is accepted. More logic is need, if you want to handle connections from multiple clients. The incoming RFCOMM connection event contains the RFCOMM channel number used during the SPP setup phase and the newly assigned RFCOMM channel ID that is used by all BTstack commands and events.

If RFCOMM_EVENT_CHANNEL_OPENED event returns status greater then 0, then the channel establishment has failed (rare case, e.g., client crashes). On successful connection, the RFCOMM channel ID and MTU for this channel are made available to the heartbeat counter. After opening the RFCOMM channel, the communication between client and the application takes place. In this example, the timer handler increases the real counter every second.

RFCOMM_EVENT_CAN_SEND_NOW indicates that it's possible to send an RFCOMM packet on the rfcomm_cid that is include

```
static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size){
  UNUSED(channel);

...
         case HCI_EVENT_PIN_CODE_REQUEST:
           // inform about pin code request
           printf("Pin code request - using '0000'\n");
           hci_event_pin_code_request_get_bd_addr(packet,
               event_addr);
           gap_pin_code_response(event_addr, "0000");
           break;

         case HCI_EVENT_USER_CONFIRMATION_REQUEST:
           // ssp: inform about user confirmation request
           printf("SSP User Confirmation Request with numeric
               value '%06"PRIu32"'\n", little_endian_read_32(
               packet, 8));
           printf("SSP User Confirmation Auto accept\n");
           break;

         case RFCOMM_EVENT_INCOMING_CONNECTION:
           // data: event (8), len(8), address(48), channel (8),
               rfcomm_cid (16)
           rfcomm_event_incoming_connection_get_bd_addr(packet,
               event_addr);
           rfcomm_channel_nr =
               rfcomm_event_incoming_connection_get_server_channel
               (packet);
```

```
                rfcomm_channel_id =
                    rfcomm_event_incoming_connection_get_rfcomm_cid(
                    packet);
                printf("RFCOMM channel %u requested for %s\n",
                    rfcomm_channel_nr, bd_addr_to_str(event_addr));
                rfcomm_accept_connection(rfcomm_channel_id);
                break;

            case RFCOMM_EVENT_CHANNEL_OPENED:
                // data: event(8), len(8), status (8), address (48),
                    server channel(8), rfcomm_cid(16), max frame size
                    (16)
                if (rfcomm_event_channel_opened_get_status(packet)) {
                  printf("RFCOMM channel open failed, status %u\n",
                      rfcomm_event_channel_opened_get_status(packet));
                } else {
                  rfcomm_channel_id =
                      rfcomm_event_channel_opened_get_rfcomm_cid(
                      packet);
                  mtu = rfcomm_event_channel_opened_get_max_frame_size
                      (packet);
                  printf("RFCOMM channel open succeeded. New RFCOMM
                      Channel ID %u, max frame size %u\n",
                      rfcomm_channel_id, mtu);
                }
                break;
            case RFCOMM_EVENT_CAN_SEND_NOW:
                rfcomm_send(rfcomm_channel_id, (uint8_t*) lineBuffer,
                    strlen(lineBuffer));
                break;

    ...
    }
```

Listing 31. SPP Server - Heartbeat Counter over RFCOMM

15.6. **spp_flowcontrol: SPP Server - Flow Control.** This example adds explicit flow control for incoming RFCOMM data to the SPP heartbeat counter example. We will highlight the changes compared to the SPP counter example.

15.6.1. *SPP Service Setup.* Listing 32 shows howto provide one initial credit during RFCOMM service initialization. Please note that providing a single credit effectively reduces the credit-based (sliding window) flow control to a stop-and-wait flow control that limits the data throughput substantially.

```
    static void spp_service_setup(void){

      // register for HCI events
      hci_event_callback_registration.callback = &packet_handler;
      hci_add_event_handler(&hci_event_callback_registration);
```

```
        // init L2CAP
        l2cap_init();

        // init RFCOMM
        rfcomm_init();
        // reserved channel, mtu limited by l2cap, 1 credit
        rfcomm_register_service_with_initial_credits(&packet_handler,
            rfcomm_channel_nr, 0xffff, 1);

        // init SDP, create record for SPP and register with SDP
        sdp_init();
        memset(spp_service_buffer, 0, sizeof(spp_service_buffer));
        spp_create_sdp_record(spp_service_buffer, 0x10001, 1, "SPP
            Counter");
        sdp_register_service(spp_service_buffer);
        printf("SDP service buffer size: %u\n\r", (uint16_t)
            de_get_len(spp_service_buffer));
    }
```

Listing 32. Providing one initial credit during RFCOMM service initialization

15.6.2. *Periodic Timer Setup.* Explicit credit management is recommended when received RFCOMM data cannot be processed immediately. In this example, delayed processing of received data is simulated with the help of a periodic timer as follows. When the packet handler receives a data packet, it does not provide a new credit, it sets a flag instead, see Listing 34 . If the flag is set, a newcredit will be granted by the heartbeat handler, introducing a delay of up to 1 second. The heartbeat handler code is shown in Listing 33 .

```
    static void  heartbeat_handler(struct btstack_timer_source *ts){
      if (rfcomm_send_credit){
        rfcomm_grant_credits(rfcomm_channel_id, 1);
        rfcomm_send_credit = 0;
      }
      btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
      btstack_run_loop_add_timer(ts);
    }
```

Listing 33. Heartbeat handler with manual credit management

```
    // Bluetooth logic
    static void packet_handler (uint8_t packet_type, uint16_t
        channel, uint8_t *packet, uint16_t size){
    ...
        case RFCOMM_DATA_PACKET:
          for (i=0;i<size;i++){
            putchar(packet[i]);
          };
          putchar('\n');
```

```
            rfcomm_send_credit = 1;
            break;
    ...
    }
```

Listing 34. Packet handler with manual credit management

15.7. **panu_demo: PANU Demo.** This example implements both a PANU client and a server. In server mode, it sets up a BNEP server and registers a PANU SDP record and waits for incoming connections. In client mode, it connects to a remote device, does an SDP Query to identify the PANU service and initiates a BNEP connection.

15.7.1. *Main application configuration.* In the application configuration, L2CAP and BNEP are initialized and a BNEP service, for server mode, is registered, before the Bluetooth stack gets started, as shown in Listing 35 .

```
static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size);
static void handle_sdp_client_query_result(uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size);

static void panu_setup(void){

  // register for HCI events
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  // Initialize L2CAP
  l2cap_init();

  // Initialise BNEP
  bnep_init();
  // Minimum L2CAP MTU for bnep is 1691 bytes
  bnep_register_service(packet_handler,
      BLUETOOTH_SERVICE_CLASS_PANU, 1691);
}
```

Listing 35. Panu setup

15.7.2. *TUN / TAP interface routines.* This example requires a TUN/TAP interface to connect the Bluetooth network interface with the native system. It has been tested on Linux and OS X, but should work on any system that provides TUN/TAP with minor modifications.

On Linux, TUN/TAP is available by default. On OS X, tuntaposx from http://tuntaposx.sourceforge.net needs to be installed.

The *tap_alloc* function sets up a virtual network interface with the given Bluetooth Address. It is rather low-level as it sets up and configures a network interface.

Listing 36 shows how a packetis received from the TAP network interface and forwarded over the BNEP connection.

After successfully reading a network packet, the call to the *bnep_can_send_packet_now* function checks, if BTstack can forward a network packet now. If that's not possible, the received data stays in the network buffer and the data source elements is removed from the run loop. The *process_tap_dev_data* function will not be called until the data source is registered again. This provides a basic flow control.

```
static void process_tap_dev_data(btstack_data_source_t *ds,
    btstack_data_source_callback_type_t callback_type)
{
  UNUSED(ds);
  UNUSED(callback_type);

  ssize_t len;
  len = read(ds->fd, network_buffer, sizeof(network_buffer));
  if (len <= 0){
    fprintf(stderr, "TAP: Error while reading: %s\n", strerror(
        errno));
    return;
  }

  network_buffer_len = len;
  if (bnep_can_send_packet_now(bnep_cid)) {
    bnep_send(bnep_cid, network_buffer, network_buffer_len);
    network_buffer_len = 0;
  } else {
    // park the current network packet
    btstack_run_loop_remove_data_source(&tap_dev_ds);
  }
  return;
}
```

Listing 36. Process incoming network packets

15.7.3. *SDP parser callback.* The SDP parsers retrieves the BNEP PAN UUID as explained in Section [on SDP BNEP Query example](#sec:sdpbnepqueryExample}.

15.7.4. *Packet Handler.* The packet handler responds to various HCI Events.

```
static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size)
{
...
  switch (packet_type) {
        case HCI_EVENT_PACKET:
      event = hci_event_packet_get_type(packet);
      switch (event) {
        case BTSTACK_EVENT_STATE:
          if (btstack_event_state_get_state(packet) ==
             HCI_STATE_WORKING){
            printf("Start SDP BNEP query.\n");
            sdp_client_query_uuid16(&
               handle_sdp_client_query_result, remote,
               BLUETOOTH_PROTOCOL_BNEP);
          }
          break;

...

            case BNEP_EVENT_CHANNEL_OPENED:
          if (bnep_event_channel_opened_get_status(packet)) {
            printf("BNEP channel open failed, status %02x\n",
               bnep_event_channel_opened_get_status(packet));
          } else {
            bnep_cid  = bnep_event_channel_opened_get_bnep_cid(
               packet);
            uuid_source =
               bnep_event_channel_opened_get_source_uuid(packet
               );
            uuid_dest   =
               bnep_event_channel_opened_get_destination_uuid(
               packet);
            mtu       = bnep_event_channel_opened_get_mtu(packet);
            //bt_flip_addr(event_addr, &packet[9]);
            memcpy(&event_addr, &packet[11], sizeof(bd_addr_t));
            printf("BNEP connection open succeeded to %s source
               UUID 0x%04x dest UUID: 0x%04x, max frame size %u
               \n", bd_addr_to_str(event_addr), uuid_source,
               uuid_dest, mtu);
            gap_local_bd_addr(local_addr);
            tap_fd = tap_alloc(tap_dev_name, local_addr);
            if (tap_fd < 0) {
              printf("Creating BNEP tap device failed: %s\n",
                 strerror(errno));
            } else {
              printf("BNEP device \"%s\" allocated.\n",
                 tap_dev_name);
              btstack_run_loop_set_data_source_fd(&tap_dev_ds,
                 tap_fd);
              btstack_run_loop_set_data_source_handler(&
                 tap_dev_ds, &process_tap_dev_data);
              btstack_run_loop_add_data_source(&tap_dev_ds);
            }
          }
                    break;

        case BNEP_EVENT_CHANNEL_TIMEOUT:
          printf("BNEP channel timeout! Channel will be closed\n
             ");
          break;
        case BNEP_EVENT_CHANNEL_CLOSED:
          printf("BNEP channel closed\n");
          btstack_run_loop_remove_data_source(&tap_dev_ds);
          if (tap_fd > 0) {
            close(tap_fd);
            tap_fd = -1;
          }
          break;

        case BNEP_EVENT_CAN_SEND_NOW:
          // Check for parked network packets and send it out
              now
          if (network_buffer_len > 0) {
            bnep_send(bnep_cid, network_buffer,
               network_buffer_len);
            network_buffer_len = 0;
            // Re-add the tap device data source
            btstack_run_loop_add_data_source(&tap_dev_ds);
          }

          break;

        default:
          break;
      }
      break;

    case BNEP_DATA_PACKET:
      // Write out the ethernet frame to the tap device
      if (tap_fd > 0) {
        rc = write(tap_fd, packet, size);
        if (rc < 0) {
          fprintf(stderr, "TAP: Could not write to TAP device: %
              s\n", strerror(errno));
        } else
        if (rc != size) {
          fprintf(stderr, "TAP: Package written only partially %
              d of %d bytes\n", rc, size);
        }
      }
      break;

    default:
      break;
  }
}
```

Listing 37. Packet Handler

When BTSTACK_EVENT_STATE with state HCI_STATE_WORKING is received and the example is started in client mode, the remote SDP BNEP query is started.

BNEP_EVENT_CHANNEL_OPENED is received after a BNEP connection was established or or when the connection fails. The status field returns the error code.

The TAP network interface is then configured. A data source is set up and registered with the run loop to receive Ethernet packets from the TAP interface.

The event contains both the source and destination UUIDs, as well as the MTU for this connection and the BNEP Channel ID, which is used for sending Ethernet packets over BNEP.

If there is a timeout during the connection setup, BNEP_EVENT_CHANNEL_TIMEOUT will be received and the BNEP connection will be closed

BNEP_EVENT_CHANNEL_CLOSED is received when the connection gets closed.

BNEP_EVENT_CAN_SEND_NOW indicates that a new packet can be send. This triggers the retry of a parked network packet. If this succeeds, the data source element is added to the run loop again.

Ethernet packets from the remote device are received in the packet handler with type BNEP_DATA_PACKET. It is forwarded to the TAP interface.

15.8. **hsp_hs_demo: HSP Headset Demo.** This example implements a HSP Headset device that sends and receives audio signal over HCI SCO. It demonstrates how to receive an output from a remote audio gateway (AG), and, if HAVE_BTSTACK_STDIN is defined, how to control the AG.

15.8.1. *Audio Transfer Setup.* A pre-computed sine wave (160Hz) is used as the input audio signal. 160 Hz. To send and receive an audio signal, ENABLE_SCO_OVER_HCI has to be defined.

Tested working setups:

- Ubuntu 14 64-bit, CC2564B connected via FTDI USB-2-UART adapter, 921600 baud
- Ubuntu 14 64-bit, CSR USB dongle
- OS X 10.11, CSR USB dongle

15.8.2. *Main Application Setup.* Listing 38 shows mainapplication code. To run a HSP Headset service you need to initialize the SDP, and to create and register HSP HS record with it. In this example, the SCO over HCI is used to receive and send an audio signal.

Two packet handlers are registered:

- The HCI SCO packet handler receives audio data.
- The HSP HS packet handler is used to trigger sending of audio data and commands to the AG. It also receives the AG's answers.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]){
  (void)argc;
  (void)argv;
```

```
    sco_demo_init();

    // register for HCI events
    hci_event_callback_registration.callback = &packet_handler;
    hci_add_event_handler(&hci_event_callback_registration);
    hci_register_sco_packet_handler(&packet_handler);

    l2cap_init();

    sdp_init();
    memset(hsp_service_buffer, 0, sizeof(hsp_service_buffer));
    hsp_hs_create_sdp_record(hsp_service_buffer, 0x10001,
        rfcomm_channel_nr, hsp_hs_service_name, 0);
    sdp_register_service(hsp_service_buffer);

    rfcomm_init();

    hsp_hs_init(rfcomm_channel_nr);
    hsp_hs_register_packet_handler(packet_handler);

#ifdef HAVE_BTSTACK_STDIN
    btstack_stdin_setup(stdin_process);
#endif

    gap_set_local_name("HSP HS Demo 00:00:00:00:00:00");
    gap_discoverable_control(1);
    gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
    gap_set_class_of_device(0x240404);

    // turn on!
    hci_power_control(HCI_POWER_ON);
    return 0;
}
```

Listing 38. Setup HSP Headset

15.9. **hsp_ag_demo: HSP Audio Gateway Demo.** This example implements a HSP Audio Gateway device that sends and receives audio signal over HCI SCO. It demonstrates how to receive an output from a remote headset (HS), and, if HAVE_BTSTACK_STDIN is defined, how to control the HS.

15.9.1. *Audio Transfer Setup.* A pre-computed sine wave (160Hz) is used as the input audio signal. 160 Hz. To send and receive an audio signal, ENABLE_SCO_OVER_HCI has to be defined.

Tested working setups:

- Ubuntu 14 64-bit, CC2564B connected via FTDI USB-2-UART adapter, 921600 baud
- Ubuntu 14 64-bit, CSR USB dongle
- OS X 10.11, CSR USB dongle

15.9.2. *Main Application Setup.* Listing 39 shows mainapplication code. To run a HSP Audio Gateway service you need to initialize the SDP, and to create and register HSP AG record with it. In this example, the SCO over HCI is used to receive and send an audio signal.

Two packet handlers are registered:

- The HCI SCO packet handler receives audio data.
- The HSP AG packet handler is used to trigger sending of audio data and commands to the HS. It also receives the AG's answers.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]){
  (void)argc;
  (void)argv;

  sco_demo_init();

  l2cap_init();

  sdp_init();

  memset((uint8_t *)hsp_service_buffer, 0, sizeof(
     hsp_service_buffer));
  hsp_ag_create_sdp_record(hsp_service_buffer, 0x10001,
     rfcomm_channel_nr, hsp_ag_service_name);
  printf("SDP service record size: %u\n", de_get_len(
     hsp_service_buffer));
  sdp_register_service(hsp_service_buffer);

  rfcomm_init();

  hsp_ag_init(rfcomm_channel_nr);
  hsp_ag_register_packet_handler(&packet_handler);
  hci_register_sco_packet_handler(&packet_handler);

  // parse human readable Bluetooth address
  sscanf_bd_addr(device_addr_string, device_addr);

#ifdef HAVE_BTSTACK_STDIN
  btstack_stdin_setup(stdin_process);
#endif

  gap_set_local_name(device_name);
  gap_discoverable_control(1);
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_set_class_of_device(0x400204);

  // turn on!
  hci_power_control(HCI_POWER_ON);
  return 0;
}
```

Listing 39. Setup HSP Audio Gateway

15.10. **hfp_hs_demo: HFP Hands-Free (HF) Demo.** This HFP Hands-Free example demonstrates how to receive an output from a remote HFP audio gateway (AG), and, if HAVE_BTSTACK_STDIN is defined, how to control the HFP AG.

15.10.1. *Main Application Setup.* Listing 40 shows mainapplication code. To run a HFP HF service you need to initialize the SDP, and to create and register HFP HF record with it. The packet_handler is used for sending commands to the HFP AG. It also receives the HFP AG's answers. The stdin_process callback allows for sending commands to the HFP AG. At the end the Bluetooth stack is started.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]){
  (void)argc;
  (void)argv;

  sco_demo_init();

  // register for HCI events
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);
  hci_register_sco_packet_handler(&packet_handler);

  gap_discoverable_control(1);
  gap_set_class_of_device(0x200408);
  gap_set_local_name("HFP HF Demo 00:00:00:00:00:00");

  // init L2CAP
  l2cap_init();

  uint16_t hf_supported_features          =
    (1<<HFP_HFSF_ESCO_S4)                 |
    (1<<HFP_HFSF_CLI_PRESENTATION_CAPABILITY) |
    (1<<HFP_HFSF_HF_INDICATORS)           |
    (1<<HFP_HFSF_CODEC_NEGOTIATION)       |
    (1<<HFP_HFSF_ENHANCED_CALL_STATUS)    |
    (1<<HFP_HFSF_REMOTE_VOLUME_CONTROL);
  int wide_band_speech = 1;

  rfcomm_init();
  hfp_hf_init(rfcomm_channel_nr);
  hfp_hf_init_supported_features(hf_supported_features);
  hfp_hf_init_hf_indicators(sizeof(indicators)/sizeof(uint16_t),
      indicators);
  hfp_hf_init_codecs(sizeof(codecs), codecs);

  hfp_hf_register_packet_handler(packet_handler);
  hci_register_sco_packet_handler(&packet_handler);

  sdp_init();
```

```
    memset(hfp_service_buffer, 0, sizeof(hfp_service_buffer));
    hfp_hf_create_sdp_record(hfp_service_buffer, 0x10001,
        rfcomm_channel_nr, hfp_hf_service_name,
        hf_supported_features, wide_band_speech);
    printf("SDP service record size: %u\n", de_get_len(
        hfp_service_buffer));
    sdp_register_service(hfp_service_buffer);

#ifdef HAVE_BTSTACK_STDIN
    // parse human readable Bluetooth address
    sscanf_bd_addr(device_addr_string, device_addr);
    btstack_stdin_setup(stdin_process);
#endif
    // turn on!
    hci_power_control(HCI_POWER_ON);
    return 0;
}
```

Listing 40. Setup HFP Hands-Free unit

15.11. **hfp_ag_demo: HFP Audio Gateway (AG) Demo.** This HFP Audio Gateway example demonstrates how to receive an output from a remote HFP Hands-Free (HF) unit, and, if HAVE_BTSTACK_STDIN is defined, how to control the HFP HF.

15.11.1. *Main Application Setup.* Listing 41 shows mainapplication code. To run a HFP AG service you need to initialize the SDP, and to create and register HFP AG record with it. The packet_handler is used for sending commands to the HFP HF. It also receives the HFP HF's answers. The stdin_process callback allows for sending commands to the HFP HF. At the end the Bluetooth stack is started.

```
int btstack_main(int argc, const char * argv[]);
int btstack_main(int argc, const char * argv[]){
    (void)argc;
    (void)argv;

    sco_demo_init();

    // register for HCI events
    hci_event_callback_registration.callback = &packet_handler;
    hci_add_event_handler(&hci_event_callback_registration);
    hci_register_sco_packet_handler(&packet_handler);

    gap_set_local_name("HFP AG Demo 00:00:00:00:00:00");
    gap_discoverable_control(1);

    // L2CAP
    l2cap_init();
    uint16_t supported_features          =
      (1<<HFP_AGSF_ESCO_S4)              |
      (1<<HFP_AGSF_HF_INDICATORS)              |
      (1<<HFP_AGSF_CODEC_NEGOTIATION)          |
      (1<<HFP_AGSF_EXTENDED_ERROR_RESULT_CODES)  |
      (1<<HFP_AGSF_ENHANCED_CALL_CONTROL)        |
      (1<<HFP_AGSF_ENHANCED_CALL_STATUS)       |
      (1<<HFP_AGSF_ABILITY_TO_REJECT_A_CALL)   |
      (1<<HFP_AGSF_IN_BAND_RING_TONE)          |
      (1<<HFP_AGSF_VOICE_RECOGNITION_FUNCTION)   |
      (1<<HFP_AGSF_THREE_WAY_CALLING);
    int wide_band_speech = 1;

    // HFP
    rfcomm_init();
    hfp_ag_init(rfcomm_channel_nr);
    hfp_ag_init_supported_features(supported_features);
    hfp_ag_init_codecs(sizeof(codecs), codecs);
    hfp_ag_init_ag_indicators(ag_indicators_nr, ag_indicators);
    hfp_ag_init_hf_indicators(hf_indicators_nr, hf_indicators);
    hfp_ag_init_call_hold_services(call_hold_services_nr,
        call_hold_services);
    hfp_ag_set_subcriber_number_information(&subscriber_number, 1)
        ;
    hfp_ag_register_packet_handler(&packet_handler);
    hci_register_sco_packet_handler(&packet_handler);

    // SDP Server
    sdp_init();
    memset(hfp_service_buffer, 0, sizeof(hfp_service_buffer));
    hfp_ag_create_sdp_record( hfp_service_buffer, 0x10001,
        rfcomm_channel_nr, hfp_ag_service_name, 0,
        supported_features, wide_band_speech);
    printf("SDP service record size: %u\n", de_get_len(
        hfp_service_buffer));
    sdp_register_service(hfp_service_buffer);

    // parse humand readable Bluetooth address
    sscanf_bd_addr(device_addr_string, device_addr);

#ifdef HAVE_BTSTACK_STDIN
    btstack_stdin_setup(stdin_process);
#endif
    // turn on!
    hci_power_control(HCI_POWER_ON);
    return 0;
}
```

Listing 41. Setup HFP Audio Gateway

15.12. **gap_le_advertisements: GAP LE Advertisements Dumper.** This example shows how to scan and parse advertisements.

15.12.1. *GAP LE setup for receiving advertisements.* GAP LE advertisements are received as custom HCI events of the GAP_EVENT_ADVERTISING_REPORT type. To receive them, you'll need to register the HCI packet handler, as shown in Listing 42 .

```
static void packet_handler(uint8_t packet_type, uint16_t channel
    , uint8_t *packet, uint16_t size);

static void gap_le_advertisements_setup(void){
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);
  // Active scanning, 100% (scan interval = scan window)
  gap_set_scan_parameters(1,48,48);
}
```

Listing 42. Setting up GAP LE client for receiving advertisements

15.12.2. *GAP LE Advertising Data Dumper.* Here, we use the definition of advertising data types and flags as specified in Assigned Numbers GAP and Supplement to the Bluetooth Core Specification, v4.

```
static char * ad_types[] = {
  "",
  "Flags",
  "Incomplete List of 16-bit Service Class UUIDs",
  "Complete List of 16-bit Service Class UUIDs",
  "Incomplete List of 32-bit Service Class UUIDs",
  "Complete List of 32-bit Service Class UUIDs",
  "Incomplete List of 128-bit Service Class UUIDs",
  "Complete List of 128-bit Service Class UUIDs",
  "Shortened Local Name",
  "Complete Local Name",
  "Tx Power Level",
  "",
  "",
  "Class of Device",
  "Simple Pairing Hash C",
  "Simple Pairing Randomizer R",
  "Device ID",
  "Security Manager TK Value",
  "Slave Connection Interval Range",
  "",
  "List of 16-bit Service Solicitation UUIDs",
  "List of 128-bit Service Solicitation UUIDs",
  "Service Data",
  "Public Target Address",
  "Random Target Address",
  "Appearance",
  "Advertising Interval"
};
```

```
static char * flags[] = {
  "LE Limited Discoverable Mode",
  "LE General Discoverable Mode",
  "BR/EDR Not Supported",
  "Simultaneous LE and BR/EDR to Same Device Capable (Controller
     )",
  "Simultaneous LE and BR/EDR to Same Device Capable (Host)",
  "Reserved",
  "Reserved",
  "Reserved"
};
```

Listing 43. Advertising data types and flags

BTstack offers an iterator for parsing sequence of advertising data (AD) structures, see BLE advertisements parser API. After initializing the iterator, each AD structure is dumped according to its type.

```
static void dump_advertisement_data(const uint8_t * adv_data,
    uint8_t adv_size){
  ad_context_t context;
  bd_addr_t address;
  uint8_t uuid_128[16];
  for (ad_iterator_init(&context, adv_size, (uint8_t *)adv_data)
     ; ad_iterator_has_more(&context) ; ad_iterator_next(&
     context)){
    uint8_t data_type   = ad_iterator_get_data_type(&context);
    uint8_t size      = ad_iterator_get_data_len(&context);
    const uint8_t * data = ad_iterator_get_data(&context);

    if (data_type > 0 && data_type < 0x1B){
      printf("  %s: ", ad_types[data_type]);
    }
    int i;
    // Assigned Numbers GAP

    switch (data_type){
      case BLUETOOTH_DATA_TYPE_FLAGS:
        // show only first octet, ignore rest
        for (i=0; i<8;i++){
          if (data[0] & (1<<i)){
            printf("%s; ", flags[i]);
          }

        }
        break;
      case
          BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
          :
```

```
          case
              BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
              :
          case
              BLUETOOTH_DATA_TYPE_LIST_OF_16_BIT_SERVICE_SOLICITATION_UUIDS
              :
            for (i=0; i<size;i+=2){
              printf("%02X ", little_endian_read_16(data, i));
            }
            break;
          case
              BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_32_BIT_SERVICE_CLASS_UUIDS
              :
          case
              BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_32_BIT_SERVICE_CLASS_UUIDS
              :
          case
              BLUETOOTH_DATA_TYPE_LIST_OF_32_BIT_SERVICE_SOLICITATION_UUIDS
              :
            for (i=0; i<size;i+=4){
              printf("%04"PRIX32, little_endian_read_32(data, i));
            }
            break;
          case
              BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_128_BIT_SERVICE_CLASS_UUIDS
              :
          case
              BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_128_BIT_SERVICE_CLASS_UUIDS
              :
          case
              BLUETOOTH_DATA_TYPE_LIST_OF_128_BIT_SERVICE_SOLICITATION_UUIDS
              :
            reverse_128(data, uuid_128);
            printf("%s", uuid128_to_str(uuid_128));
            break;
          case BLUETOOTH_DATA_TYPE_SHORTENED_LOCAL_NAME:
          case BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME:
            for (i=0; i<size;i++){
              printf("%c", (char)(data[i]));
            }
            break;
          case BLUETOOTH_DATA_TYPE_TX_POWER_LEVEL:
            printf("%d dBm", *(int8_t*)data);
            break;
          case BLUETOOTH_DATA_TYPE_SLAVE_CONNECTION_INTERVAL_RANGE:
            printf("Connection Interval Min = %u ms, Max = %u ms",
                little_endian_read_16(data, 0) * 5/4,
                little_endian_read_16(data, 2) * 5/4);
            break;
          case BLUETOOTH_DATA_TYPE_SERVICE_DATA:
            printf_hexdump(data, size);
            break;
          case BLUETOOTH_DATA_TYPE_PUBLIC_TARGET_ADDRESS:
          case BLUETOOTH_DATA_TYPE_RANDOM_TARGET_ADDRESS:
            reverse_bd_addr(data, address);
            printf("%s", bd_addr_to_str(address));
            break;
          case BLUETOOTH_DATA_TYPE_APPEARANCE:
            // https://developer.bluetooth.org/gatt/characteristics/
                Pages/CharacteristicViewer.aspx?u=org.bluetooth.
                characteristic.gap.appearance.xml
            printf("%02X", little_endian_read_16(data, 0) );
            break;
          case BLUETOOTH_DATA_TYPE_ADVERTISING_INTERVAL:
            printf("%u ms", little_endian_read_16(data, 0) * 5/8 );
            break;
          case BLUETOOTH_DATA_TYPE_3D_INFORMATION_DATA:
            printf_hexdump(data, size);
            break;
          case BLUETOOTH_DATA_TYPE_MANUFACTURER_SPECIFIC_DATA: //
              Manufacturer Specific Data
            break;
          case BLUETOOTH_DATA_TYPE_CLASS_OF_DEVICE:
          case BLUETOOTH_DATA_TYPE_SIMPLE_PAIRING_HASH_C:
          case BLUETOOTH_DATA_TYPE_SIMPLE_PAIRING_RANDOMIZER_R:
          case BLUETOOTH_DATA_TYPE_DEVICE_ID:
          case
              BLUETOOTH_DATA_TYPE_SECURITY_MANAGER_OUT_OF_BAND_FLAGS
              :
          default:
            printf("Advertising Data Type 0x%2x not handled yet",
                data_type);
            break;
        }
        printf("\n");
      }
      printf("\n");
    }
```

Listing 44. Parsing advertising data

15.12.3. *HCI packet handler.* The HCI packet handler has to start the scanning, and to handle received advertisements. Advertisements are received as HCI event packets of the GAP_EVENT_ADVERTISING_REPORT type, see Listing 45 .

```
    static void packet_handler(uint8_t packet_type, uint16_t channel
        , uint8_t *packet, uint16_t size){
      UNUSED(channel);
      UNUSED(size);

      if (packet_type != HCI_EVENT_PACKET) return;

      switch (hci_event_packet_get_type(packet)) {
        case BTSTACK_EVENT_STATE:
          // BTstack activated, get started
          if (btstack_event_state_get_state(packet) ==
              HCI_STATE_WORKING) {
            printf("Start scaning!\n");
            gap_set_scan_parameters(0,0x0030, 0x0030);
            gap_start_scan();
          }
          break;
        case GAP_EVENT_ADVERTISING_REPORT:{
          bd_addr_t address;
          gap_event_advertising_report_get_address(packet, address);
          uint8_t event_type =
              gap_event_advertising_report_get_advertising_event_type
              (packet);
          uint8_t address_type =
              gap_event_advertising_report_get_address_type(packet);
          int8_t rssi = gap_event_advertising_report_get_rssi(packet
              );
          uint8_t length =
              gap_event_advertising_report_get_data_length(packet);
          const uint8_t * data =
              gap_event_advertising_report_get_data(packet);

          printf("Advertisement event: evt-type %u, addr-type %u,
              addr %s, rssi %d, data[%u] ", event_type,
             address_type, bd_addr_to_str(address), rssi, length);
          printf_hexdump(data, length);
          dump_advertisement_data(data, length);
          break;
        }
        default:
          break;
      }
    }
```

Listing 45. Scanning and receiving advertisements

15.13. gatt_browser: GATT Client - Discovering primary services and their characteristics.

This example shows how to use the GATT Client API to discover primary services and their characteristics of the first found device that is advertising its services.

The logic is divided between the HCI and GATT client packet handlers. The HCI packet handler is responsible for finding a remote device, connecting to it, and for starting the first GATT client query. Then, the GATT client packet handler receives all primary services and requests the characteristics of the last one to keep the example short.

15.13.1. *GATT client setup.*

In the setup phase, a GATT client must register the HCI and GATT client packet handlers, as shown in Listing 46 . Additionally, thesecurity manager can be setup, if signed writes, or encrypted, or authenticated connection are required, to access the characteristics, as explained in Section on SMP.

```
// Handles connect, disconnect, and advertising report events,
// starts the GATT client, and sends the first query.
static void handle_hci_event(uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size);

// Handles GATT client query results, sends queries and the
// GAP disconnect command when the querying is done.
static void handle_gatt_client_event(uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size);

static void gatt_client_setup(void){

  // register for HCI events
  hci_event_callback_registration.callback = &handle_hci_event;
  hci_add_event_handler(&hci_event_callback_registration);

  // Initialize L2CAP and register HCI event handler
  l2cap_init();

  // Initialize GATT client
  gatt_client_init();

  // Optinoally, Setup security manager
  sm_init();
  sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
}
```

Listing 46. Setting up GATT client

15.13.2. *HCI packet handler.* The HCI packet handler has to start the scanning, to find the first advertising device, to stop scanning, to connect to and later to disconnect from it, to start the GATT client upon the connection is completed, and to send the first query - in this case the gatt_client_discover_primary_services() is called, see Listing 47 .

```
static void handle_hci_event(uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size){
  UNUSED(channel);
  UNUSED(size);

  if (packet_type != HCI_EVENT_PACKET) return;
  advertising_report_t report;

  uint8_t event = hci_event_packet_get_type(packet);
  switch (event) {
    case BTSTACK_EVENT_STATE:
      // BTstack activated, get started
      if (btstack_event_state_get_state(packet) !=
          HCI_STATE_WORKING) break;
```

```
        if (cmdline_addr_found){
          printf("Trying to connect to %s\n", bd_addr_to_str(
              cmdline_addr));
          gap_connect(cmdline_addr, 0);
          break;
        }
        printf("BTstack activated, start scanning!\n");
        gap_set_scan_parameters(0,0x0030, 0x0030);
        gap_start_scan();
        break;
      case GAP_EVENT_ADVERTISING_REPORT:
        fill_advertising_report_from_packet(&report, packet);
        dump_advertising_report(&report);

        // stop scanning, and connect to the device
        gap_stop_scan();
        gap_connect(report.address,report.address_type);
        break;
      case HCI_EVENT_LE_META:
        // wait for connection complete
        if (hci_event_le_meta_get_subevent_code(packet) !=
            HCI_SUBEVENT_LE_CONNECTION_COMPLETE) break;
        connection_handler =
            hci_subevent_le_connection_complete_get_connection_handle
            (packet);
        // query primary services
        gatt_client_discover_primary_services(
            handle_gatt_client_event, connection_handler);
        break;
      case HCI_EVENT_DISCONNECTION_COMPLETE:
        printf("\nGATT browser - DISCONNECTED\n");
        break;
      default:
        break;
    }
  }
```

Listing 47. Connecting and disconnecting from the GATT client

15.13.3. *GATT Client event handler.* Query results and further queries are handled by the GATT client packet handler, as shown in Listing 48 . Here, uponreceiving the primary services, the gatt_client_discover_characteristics_for_service() query for the last received service is sent. After receiving the characteristics for the service, gap_disconnect is called to terminate the connection. Upon disconnect, the HCI packet handler receives the disconnect complete event.

```
  static int search_services = 1;

  static void handle_gatt_client_event(uint8_t packet_type,
      uint16_t channel, uint8_t *packet, uint16_t size){
    UNUSED(packet_type);
    UNUSED(channel);
    UNUSED(size);

    gatt_client_service_t service;
    gatt_client_characteristic_t characteristic;
    switch(hci_event_packet_get_type(packet)){
      case GATT_EVENT_SERVICE_QUERY_RESULT:\
        gatt_event_service_query_result_get_service(packet, &
            service);
        dump_service(&service);
        services[service_count++] = service;
        break;
      case GATT_EVENT_CHARACTERISTIC_QUERY_RESULT:
        gatt_event_characteristic_query_result_get_characteristic(
            packet, &characteristic);
        dump_characteristic(&characteristic);
        break;
      case GATT_EVENT_QUERY_COMPLETE:
        if (search_services){
          // GATT_EVENT_QUERY_COMPLETE of search services
          service_index = 0;
          printf("\nGATT browser - CHARACTERISTIC for SERVICE %s\n
              ", uuid128_to_str(service.uuid128));
          search_services = 0;
          gatt_client_discover_characteristics_for_service(
              handle_gatt_client_event, connection_handler, &
              services[service_index]);
        } else {
          // GATT_EVENT_QUERY_COMPLETE of search characteristics
          if (service_index < service_count) {
            service = services[service_index++];
            printf("\nGATT browser - CHARACTERISTIC for SERVICE %s
                , [0x%04x-0x%04x]\n",
              uuid128_to_str(service.uuid128), service.
                  start_group_handle, service.end_group_handle);
            gatt_client_discover_characteristics_for_service(
                handle_gatt_client_event, connection_handler, &
                service);
            break;
          }
          service_index = 0;
          gap_disconnect(connection_handler);
        }
        break;
      default:
        break;
    }
  }
```

Listing 48. Handling of the GATT client queries

15.14. **le_counter: LE Peripheral - Heartbeat Counter over GATT.** All newer operating systems provide GATT Client functionality. The LE Counter examples demonstrates how to specify a minimal GATT Database with a custom GATT Service and a custom Characteristic that sends periodic notifications.

15.14.1. *Main Application Setup.* Listing 49 shows mainapplication code. It initializes L2CAP, the Security Manager and configures the ATT Server with the pre-compiled ATT Database generated from $le_counter.gatt$. Additionally, it enables the Battery Service Server with the current battery level. Finally, it configures the advertisements and the heartbeat handler and boots the Bluetooth stack. In this example, the Advertisement contains the Flags attribute and the device name. The flag 0x06 indicates: LE General Discoverable Mode and BR/EDR not supported.

```
static int  le_notification_enabled;
static btstack_timer_source_t heartbeat;
static btstack_packet_callback_registration_t
    hci_event_callback_registration;
static hci_con_handle_t con_handle;
static uint8_t battery = 100;

static void packet_handler (uint8_t packet_type, uint16_t
    channel, uint8_t *packet, uint16_t size);
static uint16_t att_read_callback(hci_con_handle_t con_handle,
    uint16_t att_handle, uint16_t offset, uint8_t * buffer,
    uint16_t buffer_size);
static int att_write_callback(hci_con_handle_t con_handle,
    uint16_t att_handle, uint16_t transaction_mode, uint16_t
    offset, uint8_t *buffer, uint16_t buffer_size);
static void  heartbeat_handler(struct btstack_timer_source *ts);
static void beat(void);

const uint8_t adv_data[] = {
  // Flags general discoverable, BR/EDR not supported
  0x02, BLUETOOTH_DATA_TYPE_FLAGS, 0x06,
  // Name
  0x0b, BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME, 'L', 'E', ' ',
      'C', 'o', 'u', 'n', 't', 'e', 'r',
  // Incomplete List of 16-bit Service Class UUIDs -- FF10 -
      only valid for testing!
  0x03,
      BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
      , 0x10, 0xff,
};
const uint8_t adv_data_len = sizeof(adv_data);

static void le_counter_setup(void){

  // register for HCI events
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  l2cap_init();
```

```
      // setup le device db
      le_device_db_init();

      // setup SM: Display only
      sm_init();

      // setup ATT server
      att_server_init(profile_data, att_read_callback,
         att_write_callback);
      att_server_register_packet_handler(packet_handler);

      // setup battery service
      battery_service_server_init(battery);

      // setup advertisements
      uint16_t adv_int_min = 0x0030;
      uint16_t adv_int_max = 0x0030;
      uint8_t adv_type = 0;
      bd_addr_t null_addr;
      memset(null_addr, 0, 6);
      gap_advertisements_set_params(adv_int_min, adv_int_max,
         adv_type, 0, null_addr, 0x07, 0x00);
      gap_advertisements_set_data(adv_data_len, (uint8_t*) adv_data)
         ;
      gap_advertisements_enable(1);

      // set one-shot timer
      heartbeat.process = &heartbeat_handler;
      btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
      btstack_run_loop_add_timer(&heartbeat);

      // beat once
      beat();
   }
```

Listing 49. Init L2CAP SM ATT Server and start heartbeat timer

15.14.2. *Heartbeat Handler.* The heartbeat handler updates the value of the single Characteristic provided in this example, and request a ATT_EVENT_CAN_SEND_NOW to send a notification if enabled see Listing 50 .

```
   static int  counter = 0;
   static char counter_string[30];
   static int  counter_string_len;

   static void beat(void){
      counter++;
      counter_string_len = sprintf(counter_string, "BTstack counter
         %04u", counter);
      puts(counter_string);
   }
```

```
static void heartbeat_handler(struct btstack_timer_source *ts){
  if (le_notification_enabled) {
    beat();
    att_server_request_can_send_now_event(con_handle);
  }

  // simulate battery drain
  battery--;
  if (battery < 50) {
    battery = 100;
  }
  battery_service_server_set_battery_value(battery);

  btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(ts);
}
```

Listing 50. Hearbeat Handler

15.14.3. *Packet Handler.* The packet handler is used to:

- stop the counter after a disconnect
- send a notification when the requested ATT_EVENT_CAN_SEND_NOW is received

```
static void packet_handler (uint8_t packet_type, uint16_t
   channel, uint8_t *packet, uint16_t size){
  UNUSED(channel);
  UNUSED(size);

  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (hci_event_packet_get_type(packet)) {
        case HCI_EVENT_DISCONNECTION_COMPLETE:
          le_notification_enabled = 0;
          break;
        case ATT_EVENT_CAN_SEND_NOW:
          att_server_notify(con_handle,
             ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HAN
             , (uint8_t*) counter_string, counter_string_len);
          break;
      }
      break;
  }
}
```

Listing 51. Packet Handler

15.14.4. *ATT Read.* The ATT Server handles all reads to constant data. For dynamic data like the custom characteristic, the registered att_read_callback is called. To handle long characteristics and long reads, the att_read_callback is first called with buffer == NULL, to request the total value length. Then it will be called again requesting a chunk of the value. See Listing 52 .

```
// ATT Client Read Callback for Dynamic Data
// - if buffer == NULL, don't copy data, just return size of
    value
// - if buffer != NULL, copy data and return number bytes copied
// @param offset defines start of attribute value
static uint16_t att_read_callback(hci_con_handle_t
    connection_handle, uint16_t att_handle, uint16_t offset,
    uint8_t * buffer, uint16_t buffer_size){
  UNUSED(connection_handle);

  if (att_handle ==
      ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
      ){
    if (buffer){
      memcpy(buffer, &counter_string[offset], buffer_size);
      return buffer_size;
    } else {
      return counter_string_len;
    }
  }
  return 0;
}
```

Listing 52. ATT Read

15.14.5. *ATT Write.* The only valid ATT write in this example is to the Client Characteristic Configuration, which configures notification and indication. If the ATT handle matches the client configuration handle, the new configuration value is stored and used in the heartbeat handler to decide if a new value should be sent. See Listing 53 .

```
static int att_write_callback(hci_con_handle_t connection_handle
    , uint16_t att_handle, uint16_t transaction_mode, uint16_t
    offset, uint8_t *buffer, uint16_t buffer_size){
  UNUSED(transaction_mode);
  UNUSED(offset);
  UNUSED(buffer_size);

  if (att_handle !=
      ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_CONFIGURAT
      ) return 0;
  le_notification_enabled = little_endian_read_16(buffer, 0) ==
      GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION;
  con_handle = connection_handle;
  return 0;
}
```

Listing 53. ATT Write

15.15. le_streamer: LE Peripheral - Stream data over GATT. All newer operating systems provide GATT Client functionality. This example shows how to get a maximal throughput via BLE:

- send whenever possible
- use the max ATT MTU

15.15.1. *Main Application Setup.* Listing 54 shows mainapplication code. It initializes L2CAP, the Security Manager, and configures the ATT Server with the pre-compiled ATT Database generated from *$le_streamer.gatt$*. Finally, it configures the advertisements and boots the Bluetooth stack.

```
static void le_streamer_setup(void){

  // register for HCI events
  hci_event_callback_registration.callback = &packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  l2cap_init();

  // setup le device db
  le_device_db_init();

  // setup SM: Display only
  sm_init();

  // setup ATT server
  att_server_init(profile_data, NULL, att_write_callback);
  att_server_register_packet_handler(packet_handler);

  // setup advertisements
  uint16_t adv_int_min = 0x0030;
  uint16_t adv_int_max = 0x0030;
  uint8_t adv_type = 0;
  bd_addr_t null_addr;
  memset(null_addr, 0, 6);
  gap_advertisements_set_params(adv_int_min, adv_int_max,
      adv_type, 0, null_addr, 0x07, 0x00);
  gap_advertisements_set_data(adv_data_len, (uint8_t*) adv_data)
      ;
  gap_advertisements_enable(1);

  // init client state
  init_connections();
}
```

Listing 54. Init L2CAP, SM, ATT Server, and enable advertisements

15.15.2. *Track throughput.* We calculate the throughput by setting a start time and measuring the amount of data sent. After a configurable REPORT_INTERVAL_MS, we print the throughput in kB/s and reset the counter and start time.

```
static void test_reset(le_streamer_connection_t * context){
  context->test_data_start = btstack_run_loop_get_time_ms();
  context->test_data_sent = 0;
}

static void test_track_sent(le_streamer_connection_t * context,
   int bytes_sent){
  context->test_data_sent += bytes_sent;
  // evaluate
  uint32_t now = btstack_run_loop_get_time_ms();
  uint32_t time_passed = now - context->test_data_start;
  if (time_passed < REPORT_INTERVAL_MS) return;
  // print speed
  int bytes_per_second = context->test_data_sent * 1000 /
      time_passed;
  printf("%c: %u bytes sent-> %u.%03u kB/s\n", context->name,
      context->test_data_sent, bytes_per_second / 1000,
      bytes_per_second % 1000);

  // restart
  context->test_data_start = now;
  context->test_data_sent  = 0;
}
```

Listing 55. Tracking throughput

15.15.3. *Packet Handler.* The packet handler is used to stop the notifications and reset the MTU on connect It would also be a good place to request the connection parameter update as indicated in the commented code block.

```
static void packet_handler (uint8_t packet_type, uint16_t
   channel, uint8_t *packet, uint16_t size){
  UNUSED(channel);
  UNUSED(size);

  int mtu;
  uint16_t conn_interval;
  le_streamer_connection_t * context;
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (hci_event_packet_get_type(packet)) {
        case HCI_EVENT_DISCONNECTION_COMPLETE:
            context = connection_for_conn_handle(
                hci_event_disconnection_complete_get_connection_handle
                (packet));
            if (!context) break;
            // free connection
            printf("%c: Disconnect, reason %02x\n", context->name,
                 hci_event_disconnection_complete_get_reason(
                packet));
            context->le_notification_enabled = 0;
            context->connection_handle = HCI_CON_HANDLE_INVALID;
            break;
        case HCI_EVENT_LE_META:
            switch (hci_event_le_meta_get_subevent_code(packet)) {
                case HCI_SUBEVENT_LE_CONNECTION_COMPLETE:
                    // setup new
                    context = connection_for_conn_handle(
                        HCI_CON_HANDLE_INVALID);
                    if (!context) break;
                    context->counter = 'A';
                    context->test_data_len = ATT_DEFAULT_MTU - 3;
                    context->connection_handle =
                        hci_subevent_le_connection_complete_get_connection_handle
                        (packet);
                    // print connection parameters (without using
                        float operations)
                    conn_interval =
                        hci_subevent_le_connection_complete_get_conn_interval
                        (packet);
                    printf("%c: Connection Interval: %u.%02u ms\n",
                        context->name, conn_interval * 125 / 100, 25 *
                         (conn_interval & 3));
                    printf("%c: Connection Latency: %u\n", context->
                        name,
                        hci_subevent_le_connection_complete_get_conn_latency
                        (packet));
                    // min con interval 20 ms
                    // gap_request_connection_parameter_update(
                        connection_handle, 0x10, 0x18, 0, 0x0048);
                    // printf("Connected, requesting conn param update
                         for handle 0x%04x\n", connection_handle);
                    break;
            }
            break;
        case ATT_EVENT_MTU_EXCHANGE_COMPLETE:
            mtu = att_event_mtu_exchange_complete_get_MTU(packet)
                - 3;
            context = connection_for_conn_handle(
                att_event_mtu_exchange_complete_get_handle(packet)
                );
            if (!context) break;
            context->test_data_len = btstack_min(mtu - 3, sizeof(
                context->test_data));
            printf("%c: ATT MTU = %u => use test data of len %u\n"
                , context->name, mtu, context->test_data_len);
                break;
            case ATT_EVENT_CAN_SEND_NOW:
                streamer();
                break;
        }
    }
}
```

Listing 56. Packet Handler

15.15.4. *Streamer.* The streamer function checks if notifications are enabled and if a notification can be sent now. It creates some test data - a single letter that gets increased every time - and tracks the data sent.

```
static void streamer(void){

    // find next active streaming connection
    int old_connection_index = connection_index;
    while (1){
        // active found?
        if ((le_streamer_connections[connection_index].
            connection_handle != HCI_CON_HANDLE_INVALID) &&
          (le_streamer_connections[connection_index].
            le_notification_enabled)) break;

        // check next
        next_connection_index();

        // none found
        if (connection_index == old_connection_index) return;
    }

    le_streamer_connection_t * context = &le_streamer_connections[
        connection_index];

    // create test data
    context->counter++;
    if (context->counter > 'Z') context->counter = 'A';
    memset(context->test_data, context->counter, context->
        test_data_len);

    // send
    att_server_notify(context->connection_handle,
        ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
        , (uint8_t*) context->test_data, context->test_data_len);

    // track
    test_track_sent(context, context->test_data_len);

    // request next send event
    att_server_request_can_send_now_event(context->
        connection_handle);

    // check next
    next_connection_index();
}
```

Listing 57. Streaming code

15.15.5. *ATT Write.* The only valid ATT write in this example is to the Client Characteristic Configuration, which configures notification and indication. If the ATT handle matches the client configuration handle, the new configuration value is stored. If notifications get enabled, an ATT_EVENT_CAN_SEND_NOW is requested. See Listing 58 .

```
static int att_write_callback(hci_con_handle_t con_handle,
    uint16_t att_handle, uint16_t transaction_mode, uint16_t
    offset, uint8_t *buffer, uint16_t buffer_size){
  UNUSED(offset);

  // printf("att_write_callback att_handle %04x, transaction
      mode %u\n", att_handle, transaction_mode);
  if (transaction_mode != ATT_TRANSACTION_MODE_NONE) return 0;
  le_streamer_connection_t * context =
      connection_for_conn_handle(con_handle);
  switch(att_handle){
    case
        ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_CONFIGUR
        :
      context->le_notification_enabled = little_endian_read_16(
          buffer, 0) ==
          GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION
          ;
      printf("%c: Notifications enabled %u\n", context->name,
          context->le_notification_enabled);
      if (context->le_notification_enabled){
        att_server_request_can_send_now_event(context->
            connection_handle);
      }
      test_reset(context);
      break;
    case
        ATT_CHARACTERISTIC_0000FF12_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
        :
      printf("%c: Write to ...FF12... : ", context->name);
      printf_hexdump(buffer, buffer_size);
      break;
  }
  return 0;
}
```

Listing 58. ATT Write

15.16. **spp_and_le_counter: Dual mode example.** The SPP and LE Counter example combines the Bluetooth Classic SPP Counter and the Bluetooth LE Counter into a single application.

In this Section, we only point out the differences to the individual examples and how how the stack is configured.

15.16.1. *Advertisements.* The Flags attribute in the Advertisement Data indicates if a device is in dual-mode or not. Flag 0x06 indicates LE General Discoverable, BR/EDR not supported although we're actually using BR/EDR. In the past, there have been problems with Anrdoid devices when the flag was not set. Setting it should prevent the remote implementation to try to use GATT over LE/EDR, which is not implemented by BTstack. So, setting the flag seems like the safer choice (while it's technically incorrect).

```
const uint8_t adv_data[] = {
  // Flags general discoverable, BR/EDR not supported
  0x02, BLUETOOTH_DATA_TYPE_FLAGS, 0x06,
  // Name
  0x0b, BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME, 'L', 'E', ' ',
      'C', 'o', 'u', 'n', 't', 'e', 'r',
  // Incomplete List of 16-bit Service Class UUIDs -- FF10 -
      only valid for testing!
  0x03,
      BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
      , 0x10, 0xff,
};
```

Listing 59. Advertisement data: Flag 0x06 indicates LE-only device

15.16.2. *Packet Handler.* The packet handler of the combined example is just the combination of the individual packet handlers.

15.16.3. *Heartbeat Handler.* Similar to the packet handler, the heartbeat handler is the combination of the individual ones. After updating the counter, it requests an ATT_EVENT_CAN_SEND_NOW and/or RFCOMM_EVENT_CAN_SEND_NOW

```
static void heartbeat_handler(struct btstack_timer_source *ts){

  if (rfcomm_channel_id || le_notification_enabled) {
    beat();
  }

  if (rfcomm_channel_id){
    rfcomm_request_can_send_now_event(rfcomm_channel_id);
  }

  if (le_notification_enabled) {
    att_server_request_can_send_now_event(att_con_handle);
  }

  btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(ts);
}
```

Listing 60. Combined Heartbeat handler

15.16.4. *Main Application Setup.* As with the packet and the heartbeat handlers, the combined app setup contains the code from the individual example setups.

```
int btstack_main(void);
int btstack_main(void)
{

    // register for HCI events
    hci_event_callback_registration.callback = &packet_handler;
    hci_add_event_handler(&hci_event_callback_registration);

    l2cap_init();

    rfcomm_init();
    rfcomm_register_service(packet_handler, RFCOMM_SERVER_CHANNEL,
         0xffff);

    // init SDP, create record for SPP and register with SDP
    sdp_init();
    memset(spp_service_buffer, 0, sizeof(spp_service_buffer));
    spp_create_sdp_record(spp_service_buffer, 0x10001,
       RFCOMM_SERVER_CHANNEL, "SPP Counter");
    sdp_register_service(spp_service_buffer);
    printf("SDP service record size: %u\n", de_get_len(
       spp_service_buffer));

    gap_set_local_name("SPP and LE Counter 00:00:00:00:00:00");
    gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
    gap_discoverable_control(1);

    // setup le device db
    le_device_db_init();

    // setup SM: Display only
    sm_init();

    // setup ATT server
    att_server_init(profile_data, att_read_callback,
       att_write_callback);
    att_server_register_packet_handler(packet_handler);

    // set one-shot timer
    heartbeat.process = &heartbeat_handler;
    btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
    btstack_run_loop_add_timer(&heartbeat);

    // setup advertisements
    uint16_t adv_int_min = 0x0030;
    uint16_t adv_int_max = 0x0030;
    uint8_t adv_type = 0;
    bd_addr_t null_addr;
    memset(null_addr, 0, 6);
    gap_advertisements_set_params(adv_int_min, adv_int_max,
        adv_type, 0, null_addr, 0x07, 0x00);
    gap_advertisements_set_data(adv_data_len, (uint8_t*) adv_data)
        ;
    gap_advertisements_enable(1);

    // beat once
    beat();

    // turn on!
      hci_power_control(HCI_POWER_ON);

    return 0;
}
```

Listing 61. Init L2CAP RFCOMM SDO SM ATT Server and start heartbeat timer

16. Chipsets

In this chapter, we first explain how Bluetooth chipsets are connected physically and then provide information about popular Bluetooth chipset and their use with BTstack.

16.1. HCI Interface.

The communication between a Host (a computer or an MCU) and a Host Controller (the actual Bluetooth chipset) follows the Host Controller Interface (HCI), see 3. HCI defines how commands, events, asynchronous and synchronous data packets are exchanged. Asynchronous packets (ACL) are used for data transfer, while synchronous packets (SCO) are used for Voice with the Headset and the Hands-Free Profiles.

16.1.1. *HCI H2.*

On desktop-class computers incl. laptops, USB is mainly used as HCI transport layer. For USB Bluetooth chipsets, there is little variation: most USB dongles on the market currently contain a Broadcom BCM20702 or a CSR 851x chipset. It is also called H2.

On embedded systems, UART connections are used instead, although USB could be used as well.

For UART connections, different transport layer variants exist.

16.1.2. *HCI H4.*

The most common one is the official “UART Transport”, also called H4. It requires hardware flow control via the CTS/RTS lines and assumes no errors on the UART lines.

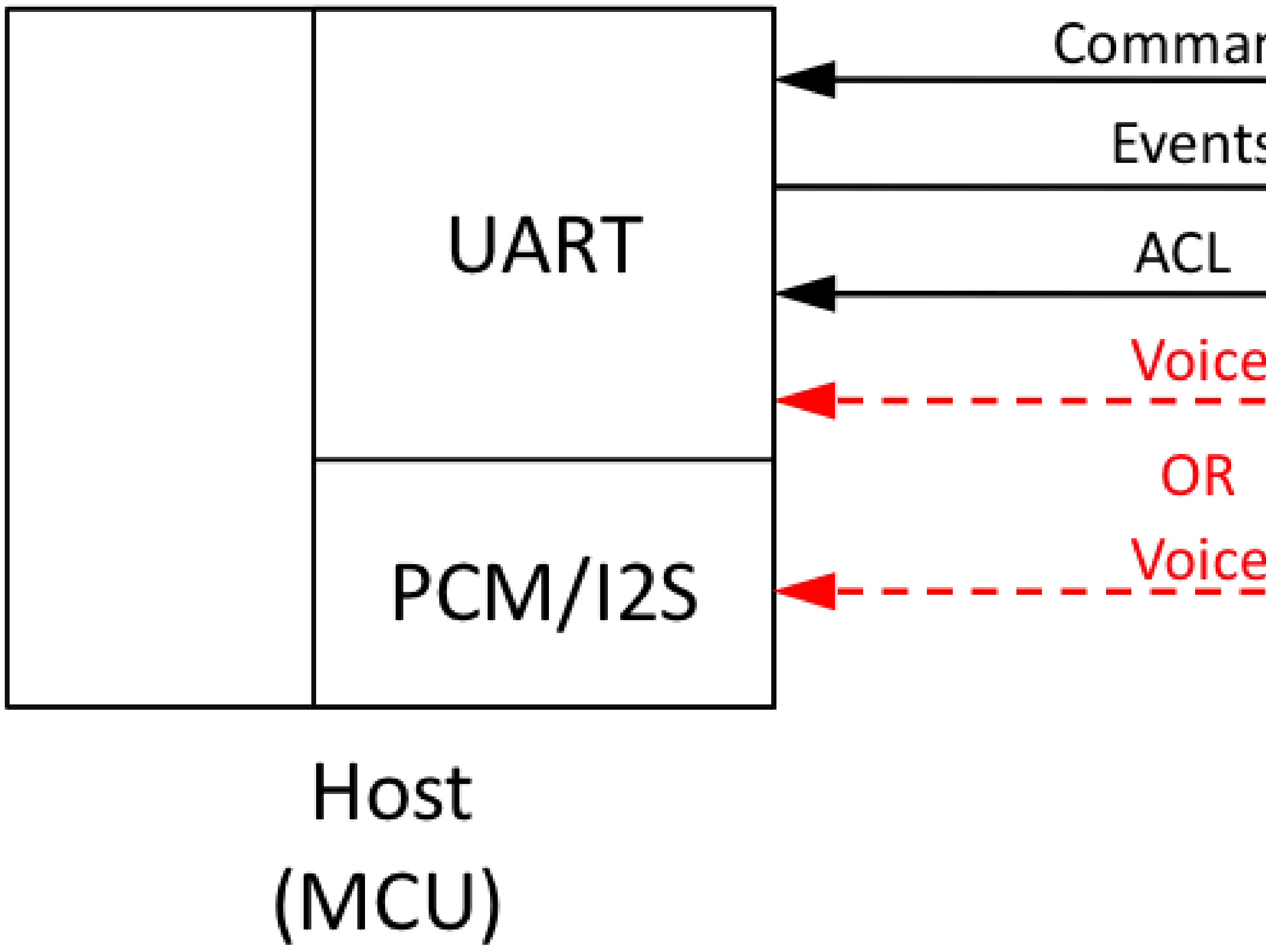

Figure 3. Host Controller to Host connection

16.1.3. *HCI H5.* The “Three-Wire UART Transport”, also called H5, makes use of the SLIP protocol to transmit a packet and can deal with packet loss and bit-errors by retransmission. While it is possible to use H5 really with “three wires” without hardware handshake, we recommend to use a full UART with hardware handshake. If your design lacks the hardware handshake, H5 is your only option.

16.1.4. *BCSP.* The predecessor of H5. The main difference to H5 is that Even Parity is used for BCSP. To use BCSP with BTstack, you use the H5 transport and can call *hci_transport_h5_enable_bcsp_mode*

16.1.5. *eHCILL.* Finally, Texas Instruments extended H4 to create the “eHCILL transport” layer that allows both sides to enter sleep mode without loosing synchronisation. While it is easier to implement than H5, it it is only supported by TI chipsets and cannot handle packet loss or bit-errors.

16.1.6. *H4 over SPI.* Chipsets from Dialog Semiconductor and EM Marin allow to send H4 formatted HCI packets via SPI. SPI has the benefit of a simpler implementation for both Host Controller and Host as it does not require an exact clock. The SPI Master, here the Host, provides the SPI Clock and the SPI Slave (Host Controller) only has to read and update it's data lines when the clock line changes. The EM9304 supports an SPI clock of up to 8 Mhz. However, an additional protocol is needed to let the Host know when the Host Controller has HCI packet for it. Often, an additional GPIO is used to signal this.

16.1.7. *HCI Shortcomings.* Unfortunately, the HCI standard misses a few relevant details:

- For UART based connections, the initial baud rate isn't defined but most Bluetooth chipsets use 115200 baud. For better throughput, a higher baud rate is necessary, but there's no standard HCI command to change it. Instead, each vendor had to come up with their own set of vendor-specific commands. Sometimes, additional steps, e.g. doing a warm reset, are necessary to activate the baud rate change as well.
- Some Bluetooth chipsets don't have a unique MAC address. On start, the MAC address needs to be set, but there's no standard HCI command to set it.
- SCO data for Voice can either be transmitted via the HCI interface or via an explicit PCM/I2S interface on the chipset. Most chipsets default to the PCM/I2S interface. To use it via USB or for Wide-Band Speech in the Hands-Free Profile, the data needs to be delivered to the host MCU. Newer Bluetooth standards define a HCI command to configure the SCO routing, but it is not implemented in the chipsets we've tested so far. Instead, this is configured in a vendor-specific way as well.
- In addition, most vendors allow to patch or configure their chipsets at run time by sending custom commands to the chipset. Obviously, this is also vendor dependent.

16.2. **Documentation and Support.** The level of developer documentation and support varies widely between the various Bluetooth chipset providers.

From our experience, only Texas Instruments and EM Microelectronics provide all relevant information directly on their website. Nordic Semiconductor does not officially have Bluetooth chipsets with HCI interface, but their the documentation on their nRF5 series is complete and very informative. TI and Nordic also provide excellent support via their respective web forum.

Broadcom, whose Bluetooth + Wifi division has been acquired by the Cypress Semiconductor Corporation, provides developer documentation only to large customers as far as we know. It's possible to join their Community forum and download the WICED SDK. The WICED SDK is targeted at Wifi + Bluetooth Combo chipsets and contains the necessary chipset patch files.

CSR, which has been acquired by Qualcomm, provides all relevant information on their Support website after signing an NDA.

Chipset	Type	HCI Transport	BD_ADDR (1)	SCO over HCI (2)	LE
Atmel ATWILC3000	Dual mode	H4	Yes	Don't know	No
Broadcom UART	Dual mode	H4, H5	Rarely	Probably (2)	No
Broadcom USB Dongles	Dual mode	USB	Yes	Yes	No
CSR UART	Dual mode	H4, H5, BCSP	Rarely	No (didn't work)	No
CSR USB Dongles	Dual mode	USB	Mostly	Yes	No
Dialog DA14581	LE	H4, SPI	No	n.a.	No
Espressif ESP32	Dual mode	VHCI	Yes	Probably	Yes
EM 9301	LE	SPI, H4	No	n.a.	No
EM 9304	LE	SPI, H4	No	n.a.	Yes
Nordic nRF	LE	H4	Fixed Random	n.a.	Yes
STM STLC2500D	Classic	H4	No	No (didn't try)	n.a
Toshiba TC35661	Dual mode	H4	No	No (didn't try)	No
TI CC256x, WL183x	Dual mode	H4, H5, eHCILL	Yes	Yes	No

16.3. **Chipset Overview.** esp32thing: 24:0A:C4:00:8B:C4 nina1: 18:FE:34:6D:1B:D2 nina2: 18:FE:34:6D:17:66

Notes:

1. BD_ADDR: Indciates if Bluetooth chipset compes with its own valid MAC Addess. Better Broadcom and CSR dongles usually come with a MAC address from the dongle manufacturer, but cheaper ones might come with identical addresses.
2. SCO over HCI: All Bluetooth Classic chipsets support SCO over HCI, for those that are marked with No, we either didn't try or didn't found enough information to configure it correctly.
3. Multiple LE Roles: Apple uses Broadcom Bluetooth+Wifi in their iOS devices and newer iOS versions support multiple concurrent LE roles, so at least some Broadcom models support multiple concurrent LE roles.

16.4. **Atmel/Microchip.** The ATILC3000 Bluetooth/Wifi combo controller has been used with Linux on embedded devices by Atmel/Microchip. Drivers and documentation are available from a GitHub repository. The ATWILC3000 has a basic HCI implementation stored in ROM and requires a firmware image to be uploaded before it can be used. Please note: the Bluetooth firmware is 270 kB.

BD Addr can be set with vendor-specific command although all chipsets have an official address stored. The BD_ADDR lookup results in "Newport Media Inc." which was acquired by Atmel in 2014.

Baud rate can be set with a custom command.

BTstack integration: *btstack_chipset_atwilc3000.c* contains the code to download the Bluetooth firmware image into the RAM of the ATWILC3000. After that, it can be normally used by BTstack.

16.5. **Broadcom.** Before the Broadcom Wifi+Bluetooth division was taken over by Cypress Semiconductor, it was not possible to buy Broadcom chipset in low quantities. Nevertheless, module manufacturers like Ampak created modules that contained Broadcom BCM chipsets (Bluetooth as well as Bluetooth+Wifi combos) that might already have been pre-tested for FCC and similar certifications. A popular example is the Ampak AP6212A module that contains an BCM 43438A1 and is used on the Raspberry Pi 3, the RedBear Duo, and the RedBear IoT pHAT for older Raspberry Pi models.

The best source for documentation on vendor specific commands so far has been the source code for blueZ and the Bluedroid Bluetooth stack from Android.

Broadcom USB dongles do not require special configuration, however SCO data is not routed over USB by default.

Init scripts: For UART connected chipsets, an init script has to be uploaded after power on. For Bluetooth chipsets that are used in Broadcom Wifi+Bluetooth combos, this file often can be found as a binary file in Linux distributions with the ending *'.hcd'* or as part of the WICED SDK as C source file that contains the init script as a data array for use without a file system.

To find the correct file, Broadcom chipsets return their model number when asked for their local name.

BTstack supports uploading of the init script in two variants: using .hcd files looked up by name in the posix-h4 port and by linking against the init script in the WICED port. While the init script is processed, the chipsets RTS line goes high, but only 2 ms after the command complete event for the last command from the init script was sent. BTstack waits for 10 ms after receiving the command complete event for the last command to avoid sending before RTS goes high and the command fails.

BD Addr can be set with a custom command. A fixed address is provided on some modules, e.g. the AP6212A, but not on others.

SCO data can be configured with a custom command found in the bluez sources. It works with USB chipsets. The chipsets don't implement the SCO Flow Control that is used by BTstack for UART connected devices. A forum suggests to send SCO packets as fast as they are received since both directions have the same constant speed.

Baud rate can be set with custom command. The baud rate resets during the warm start after uploading the init script. So, the overall scheme is this: start at default baud rate, get local version info, send custom Broadcom baud rate change command, wait for response, set local UART to high baud rate, and then send init script. After sending the last command from the init script, reset the local UART. Finally, send custom baud rate change command, wait for response, and set local UART to high baud rate.

BTstack integration: The common code for all Broadcom chipsets is provided by *btstack_chipset_bcm.c*. During the setup, *btstack_chipset_bcm_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function.

SCO Data can be routed over HCI for both USB dongles and UART connections, however BTstack does not provide any form of flow control for UART connections. HSP and HFP Narrow Band Speech is supported via I2C/PCM pins.

16.6. **CSR.** Similar to Broadcom, the best source for documentation is the source code for blueZ.

CSR USB dongles do not require special configuration and SCO data is routed over USB by default.

CSR chipset do not require an actual init script in general, but they allow to configure the chipset via so-called PSKEYs. After setting one or more PSKEYs, a warm reset activates the new setting.

BD Addr can be set via PSKEY. A fixed address can be provided if the chipset has some kind of persistent memory to store it. Most USB Bluetooth dongles have a fixed BD ADDR.

SCO data can be configured via a set of PSKEYs. We haven't been able to route SCO data over HCI for UART connections yet.

Baud rate can be set as part of the initial configuration and gets actived by the warm reset.

BTstack integration: The common code for all Broadcom chipsets is provided by *btstack_chipset_csr.c*. During the setup, *btstack_chipset_csr_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function. The baud rate is set during the general configuration.

SCO Data is routed over HCI for USB dongles, but not for UART connections. HSP and HFP Narrow Band Speech is supported via I2C/PCM pins.

16.7. **Dialog Semiconductor.** Dialog Semiconductor offers the DA14581, an LE-only SoC that can be programmed with an HCI firmware. The HCI firmware can be uploaded on boot into SRAM or stored in the OTP (One-time programmable) memory, or in an external SPI.

It does not implement the Data Length Extension or supports multiple concurrent roles.

BD Addr fixed to 80:EA:CA:00:00:01. No command in HCI firmware to set it differently. Random addresses could be used instead.

Baud rate: The baud rate is fixed at 115200 with the provided firmware. A higher baud rate could be achieved by re-compiling the HCI firmware using Dialog's HCI SDK.

BTstack integration: *btstack_chipset_da14581.c* contains the code to download the provided HCI firmware into the SRAM of the DA14581. After that, it can be used as any other HCI chipset.

16.8. **Espressif ESP32.** The ESP32 is a SoC with a built-in Dual mode Bluetooth and Wifi radio. The HCI Controller is implemented in software and accessed via a so called Virtual HCI (VHCI) interface. It supports both LE Data Length Extensions (DLE) as well as multiple LE roles. Flow control between the VHCI and BTstack is problematic as there's no way to stop the VHCI from delivering packets. BTstack impelemts the Host Controller to Host Flow Control to deal with this. Right now, this works for HCI Events and Classic ACL packets but not for LE ACL packets. Espressif is working on a solution for this: https://github.com/espressif/esp-idf/issues/644

16.9. **EM Microelectronic Marin.** For a long time, the EM9301 has been the only Bluetooth Single-Mode LE chipset with an HCI interface. The EM9301 can be connected via SPI or UART. The UART interface does not support hardware flow control and is not recommended for use with BTstack. The SPI mode uses a proprietary but documented extension to implement flow control and signal if the EM9301 has data to send.

In December 2016, EM released the new EM9304 that also features an HCI mode and adds support for optional Bluetooth 4.2. features. It supports the Data Length Extension and up to 8 LE roles. The EM9304 is a larger MCU that allows to run custom code on it. For this, an advanced mechanism to upload configuration and firmware to RAM or into an One-Time-Programmable area of 128 kB is supported. It supports a superset of the vendor specific commands of the EM9301.

EM9304 is used by the 'stm32-l053r8-em9304' port in BTstack. The port.c file also contains an IRQ+DMA-driven implementation of the SPI H4 protocol specified in the datasheet.

BD Addr must be set during startup since it does not have a stored fix address.

SCO data is not supported since it is LE only.

Baud rate could be set for UART mode. For SPI, the master controls the speed via the SPI Clock line. With 3.3V, 16 Mhz is supported.

Init scripts are not required although it is possible to upload small firmware patches to RAM or the OTP memory.

BTstack integration: The common code for the EM9304 is provided by *btstack_chipset_em9301.c.* During the setup, *btstack_chipset_em9301_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function. It enables to set the BD Addr during start.

16.10. **Nordic nRF5 series.** The Single-Mode LE chipsets from the Nordic nRF5 series chipsets do not have an HCI interface. Instead, they provide an LE Bluetooth Stack as a binary library, the so-called *SoftDevices*. Developer can write their Bluetooth application on top of this library usually. Since the chipset can be programmed, it can also be loaded with a firmware that provides a regular HCI H4 interface for a Host.

An interesting feature of the nRF5 chipsets is that they can support multiple LE roles at the same time, e.g. being Central in one connection and a Peripheral in another connection. Also, the nRF52 SoftDevice implementation supports the Bluetooth 4.2 Data Length Extension.

Both nRF5 series, the nRF51 and the nRF52, can be used with an HCI firmware. The HCI firmware does not support the Data Length Extension yet, but this will be supported soon. Also, the nRF51 does not support encrypted connections at the moment (November 18th, 2016) although this might become supported as well.

BD ADDR is not set automatically. However, during production, a 64-bit random number is stored in the each chip. Nordic uses this random number as a random static address in their SoftDevice implementation.

SCO data is not supported since it is LE only.

Baud rate is fixed to 115200 by the patch although the firmware could be extended to support a baud rate change.

Init script is not required.

BTstack integration: No special chipset driver is provided. In order to use the random static address, the provided patch stores this address as the (invalid) public address that is returned by the HCI Read BD Addr command. When BTstack detects that it is a Nordic chipset, it automatically uses this address as random static address - unless the app chooses to use private addresses.

To use these chipsets with BTstack, you need to install an arm-none-eabi gcc toolchain and the nRF5x Command Line Tools incl. the J-Link drivers, checkout the Zephyr project, apply a minimal patch to help with using a random static address, and flash it onto the chipset:

- Install J-Link Software and documentation pack.
- Get nrfjprog as part of the nRFx-Command-Line-Tools. Click on Downloads tab on the top and look for your OS.
- Checkout Zephyr and install toolchain. We recommend using the arm-non-eabi gcc binaries instead of compiling it yourself. At least on OS X, this failed for us.
- Download our patch into the Zephyr root folder and apply it there:

$ patch -p1 < hci_firmware.patch

- In *samples/bluetooth/hci_uart* compile the firmware for nRF52 Dev Kit

```
$ make BOARD=nrf52_pca10040
```

- Upload the firmware
  $ ./flash_nrf52_pca10040.sh
- For the nRF51 Dev Kit, use `make BOARD=nrf51_pca10028` and `./flash_nrf51_10028.sh` with the nRF51 kit.
- The nRF5 dev kit acts as an LE HCI Controller with H4 interface.

16.11. STMicroelectronics.

STMicroelectronics offers the Bluetooth V2.1 + EDR chipset STLC2500D that supports SPI and UART H4 connection.

BD Addr can be set with custom command although all chipsets have an official address stored.

SCO data might work. We didn't try.

Baud rate can be set with custom command. The baud rate change of the chipset happens within 0.5 seconds. At least on BTstack, knowning exactly when the command was fully sent over the UART is non-trivial, so BTstack switches to the new baud rate after 100 ms to expect the command response on the new speed.

Init scripts are not required although it is possible to upload firmware patches.

BTstack integration: Support for the STLC2500C is provided by *btstack_chipset_stlc.c*. During the setup, *btstack_chipset_stlc2500d_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function. It enables higher UART baud rate and to set the BD Addr during startup.

16.12. **Texas Instruments CC256x series.** The Texas Instruments CC256x series is currently in its fourth iteration and provides a Classic-only (CC2560), a Dual-mode (CC2564), and a Classic + ANT (CC2567) model. A variant of the Dual-mode chipset is also integrated into TI's WiLink 8 Wifi+Bluetooth combo modules of the WL183x, WL185x, WL187x, and WL189x series.

The CC256x chipset is connected via an UART connection and supports the H4, H5 (since third iteration), and eHCILL.

The latest generation CC256xC chipsets support multiple LE roles in parallel.

The different CC256x chipset can be identified by the LMP Subversion returned by the *hci_read_local_version_information* command. TI also uses a numeric way (AKA) to identify their chipsets. The table shows the LMP Subversion and AKA number for the CC256x and the WL18xx series.

Chipset	LMP Subversion	AKA
CC2560	0x191f	6.2.31
CC2560A, CC2564	0x1B0F	6.6.15
CC256xB	0x1B90	6.7.16
CC256xC	0x9a1a	6.12.26
WL18xx	0xac20	11.8.32

SCO data is routed to the I2S/PCM interface but can be configured with the HCI_VS_Write_SCO_Configuration command.

Baud rate can be set with HCI_VS_Update_UART_HCI_Baudrate. The chipset confirms the change with a command complete event after which the local UART is set to the new speed. Oddly enough, the CC256x chipsets ignore the incoming CTS line during this particular command complete response.

If you've implemented the hal_uart_dma.h without an additional ring buffer (as recommended!) and you have a bit of delay, e.g. because of thread switching on a RTOS, this could cause a UART overrun. If this happens, BTstack provides a workaround in the HCI H4 transport implementation by adding #define ENABLE_CC256X_BAUDRATE_CHANGE_FLOWCONTROL_BUG_WORKAROUND to your btstack_config.h. If this is enabled, the H4 transport layer will resort to "deep packet inspection" to first check if its a TI controller and then wait for the HCI_VS_Update_UART_HCI_Baudrate. When detected, it will tweak the next UART read to expect the HCI Command Complete event.

BD Addr can be set with HCI_VS_Write_BD_Addr although all chipsets have an official address stored.

Init Scripts. In order to use the CC256x chipset an initialization script must be obtained and converted into a C file for use with BTstack. For newer revisions, TI provides a main.bts and a ble_add_on.bts that need to be combined.

The Makefile at *chipset/cc256x/Makefile.inc* is able to automatically download and convert the requested file. It does this by:

- Downloading one or more BTS files for your chipset.
- Running the Python script:

```
./convert_bts_init_scripts.py main.bts [ble_add_on.bts] output_file.
  c
```

BTstack integration: The common code for all CC256x chipsets is provided by *btstack_chipset_cc256x.c*. During the setup, *btstack_chipset_cc256x_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function. *btstack_chipset_cc256x_lmp_subversion* provides the LMP Subversion for the selected init script.

SCO Data can be routed over HCI, so HFP Wide-Band Speech is supported.

16.13. Toshiba.

The Toshiba TC35661 Dual-Mode chipset is available in three variants: standalone incl. binary Bluetooth stack, as a module with embedded stack or with a regular HCI interface. The HCI variant has the model number TC35661–007.

We've tried their USB Evaluation Stick that contains an USB-to-UART adapter and the PAN1026 module that contains the TC35661 -501. We have been told by our distributor that the -501 variant also supports the HCI interface. However, while our tests have shown that Classic Bluetooth with SPP works fine with this variant, none of the LE commands work.

SCO data might work. We didn't try.

Baud rate can be set with custom command.

BD Addr must be set with custom command. It does not have a stored valid public BD Addr.

Init Script is not required. A patch file might be uploaded.

BTstack integration: Support for the TC35661 series is provided by *btstack_chipset_tc3566x.c*. During the setup, *btstack_chipset_tc3566x_instance* function is used to get a *btstack_chipset_t* instance and passed to *hci_init* function. It enables higher UART baud rate and sets the BD Addr during startup.

17. Porting to Other Platforms

In this section, we highlight the BTstack components that need to be adjusted for different hardware platforms.

17.1. Time Abstraction Layer.

BTstack requires a way to learn about passing time. *btstack_run_loop_embedded.c* supports two different modes: system ticks or a system clock with millisecond resolution. BTstack's timing requirements are quite low as only Bluetooth timeouts in the second range need to be handled.

17.1.1. *Tick Hardware Abstraction.* If your platform doesn't require a system clock or if you already have a system tick (as it is the default with CMSIS on ARM Cortex devices), you can use that to implement BTstack's time abstraction in *include/btstack/hal_tick.h>*.

For this, you need to define *HAVE_EMBEDDED_TICK* in *btstack_config.h*:

```
#define HAVE_EMBEDDED_TICK
```

Then, you need to implement the functions *hal_tick_init* and *hal_tick_set_handler*, which will be called during the initialization of the run loop.

```
void hal_tick_init(void);
void hal_tick_set_handler(void (*tick_handler)(void));
int  hal_tick_get_tick_period_in_ms(void);
```

After BTstack calls *hal_tick_init()* and *hal_tick_set_handler(tick_handler)*, it expects that the *tick_handler* gets called every *hal_tick_get_tick_period_in_ms()* ms.

17.1.2. *Time MS Hardware Abstraction.* If your platform already has a system clock or it is more convenient to provide such a clock, you can use the Time MS Hardware Abstraction in *include/btstack/hal_time_ms.h*.

For this, you need to define *HAVE_EMBEDDED_TIME_MS* in *btstack_config.h*:

```
#define HAVE_EMBEDDED_TIME_MS
```

Then, you need to implement the function *hal_time_ms()*, which will be called from BTstack's run loop and when setting a timer for the future. It has to return the time in milliseconds.

```
uint32_t hal_time_ms(void);
```

17.2. **Bluetooth Hardware Control API.** The Bluetooth hardware control API can provide the HCI layer with a custom initialization script, a vendor-specific baud rate change command, and system power notifications. It is also used to control the power mode of the Bluetooth module, i.e., turning it on/off and putting to sleep. In addition, it provides an error handler *hw_error* that is called when a Hardware Error is reported by the Bluetooth module. The callback allows for persistent logging or signaling of this failure.

Overall, the struct *btstack_control_t* encapsulates common functionality that is not covered by the Bluetooth specification. As an example, the *btstack_chipset_cc256x_instance* function returns a pointer to a control struct suitable for the CC256x chipset.

17.3. **HCI Transport Implementation.** On embedded systems, a Bluetooth module can be connected via USB or an UART port. BTstack implements three UART based protocols for carrying HCI commands, events and data between a host and a Bluetooth module: HCI UART Transport Layer (H4), H4 with eHCILL support, a lightweight low-power variant by Texas Instruments, and the Three-Wire UART Transport Layer (H5).

17.3.1. *HCI UART Transport Layer (H4).* Most embedded UART interfaces operate on the byte level and generate a processor interrupt when a byte was received. In the interrupt handler, common UART drivers then place the received data in a ring buffer and set a flag for further processing or notify the higher-level code, i.e., in our case the Bluetooth stack.

Bluetooth communication is packet-based and a single packet may contain up to 1021 bytes. Calling a data received handler of the Bluetooth stack for every byte creates an unnecessary overhead. To avoid that, a Bluetooth packet can be read as multiple blocks where the amount of bytes to read is known in advance. Even better would be the use of on-chip DMA modules for these block reads, if available.

The BTstack UART Hardware Abstraction Layer API reflects this design approach and the underlying UART driver has to implement the following API:

```
void hal_uart_dma_init(void);
void hal_uart_dma_set_block_received(void (*block_handler)(void));
void hal_uart_dma_set_block_sent(void (*block_handler)(void));
int  hal_uart_dma_set_baud(uint32_t baud);
void hal_uart_dma_send_block(const uint8_t *buffer, uint16_t len);
void hal_uart_dma_receive_block(uint8_t *buffer, uint16_t len);
```

The main HCI H4 implementations for embedded system is *hci_h4_transport- _dma* function. This function calls the following sequence: *hal_uart_dma_init*, *hal_uart_dma_set_block_received* and *hal_uart_dma_set_block_sent* functions. this sequence, the HCI layer will start packet processing by calling *hal_uart*-dma*receive_block* function. The HAL implementation is responsible for reading the requested amount of bytes, stopping incoming data via the RTS line when the requested amount of data was received and has to call the handler. By this, the HAL implementation can stay generic, while requiring only three callbacks per HCI packet.

17.3.2. *H4 with eHCILL support.* With the standard H4 protocol interface, it is not possible for either the host nor the baseband controller to enter a sleep mode. Besides the official H5 protocol, various chip vendors came up with proprietary solutions to this. The eHCILL support by Texas Instruments allows both the host and the baseband controller to independently enter sleep mode without loosing their synchronization with the HCI H4 Transport Layer. In addition to the IRQ-driven block-wise RX and TX, eHCILL requires a callback for CTS interrupts.

```
void hal_uart_dma_set_cts_irq_handler(void(*cts_irq_handler)(void));
void hal_uart_dma_set_sleep(uint8_t sleep);
```

17.3.3. *H5.* H5, makes use of the SLIP protocol to transmit a packet and can deal with packet loss and bit-errors by retransmission. Since it can recover from packet loss, it's also possible for either side to enter sleep mode without loosing synchronization.

The use of hardware flow control in H5 is optional, however, since BTstack uses hardware flow control to avoid packet buffers, it's recommended to only use H5 with RTS/CTS as well.

For porting, the implementation follows the regular H4 protocol described above.

17.4. Persistent Storage APIs.

On embedded systems there is no generic way to persist data like link keys or remote device names, as every type of a device has its own capabilities, particularities and limitations. The persistent storage APIs provides an interface to implement concrete drivers for a particular system.

17.4.1. *Link Key DB.* As an example and for testing purposes, BTstack provides the memory-only implementation *btstack_link_key_db_memory*. An implementation has to conform to the interface in Listing 62 .

```
    typedef struct {
        // management
        void (*open)();
        void (*close)();

        // link key
        int  (*get_link_key)(bd_addr_t bd_addr, link_key_t link_key)
            ;
        void (*put_link_key)(bd_addr_t bd_addr, link_key_t key);
        void (*delete_link_key)(bd_addr_t bd_addr);
    } btstack_link_key_db_t;
```

Listing 62. Persistent storage interface.

18. Integrating with Existing Systems

While the run loop provided by BTstack is sufficient for new designs, BTstack is often used with or added to existing projects. In this case, the run loop, data sources, and timers may need to be adapted. The following two sections provides a guideline for single and multi-threaded environments.

To simplify the discussion, we'll consider an application split into "Main", "Communication Logic", and "BTstack". The Communication Logic contains the packet handler (PH) that handles all asynchronous events and data packets from BTstack. The Main Application makes use of the Communication Logic for its Bluetooth communication.

18.1. **Adapting BTstack for Single-Threaded Environments.** In a single-threaded environment, all application components run on the same (single) thread and use direct function calls as shown in Figure 4.

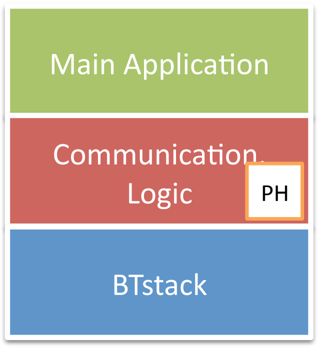

FIGURE 4. BTstack in single-threaded environment.

BTstack provides a basic run loop that supports the concept of data sources and timers, which can be registered centrally. This works well when working with a small MCU and without an operating system. To adapt to a basic operating system or a different scheduler, BTstack's run loop can be implemented based on the functions and mechanism of the existing system.

Currently, we have two examples for this:

- *btstack_run_loop_posix.c* is an implementation for POSIX compliant systems. The data sources are modeled as file descriptors and managed in a linked list. Then, the *select* function is used to wait for the next file descriptor to become ready or timer to expire.
- *btstack_run_loop_cocoa.c* is an implementation for the CoreFoundation Framework used in OS X and iOS. All run loop functions are implemented in terms of CoreFoundation calls, data sources and timers are modeled as CFSockets and CFRunLoopTimer respectively.
- *btstack_run_loop_windows* is an implementation for Windows environment. The data sources are modeled with Event objects and managed in a linked list. Then, the *WaitForMultipleObjects* is used to wait for the next Event to becomre ready or timer to expire.

18.2. **Adapting BTstack for Multi-Threaded Environments.** The basic execution model of BTstack is a general while loop. Aside from interrupt-driven UART and timers, everything happens in sequence. When using BTstack in a multi-threaded environment, this assumption has to stay valid - at least with respect to BTstack. For this, there are two common options:

- The Communication Logic is implemented on a dedicated BTstack thread, and the Main Application communicates with the BTstack thread via application-specific messages over an Interprocess Communication (IPC) as depicted in Figure 5. This option results in less code and quick adaption.
- BTstack must be extended to run standalone, i.e, as a Daemon, on a dedicated thread and the Main Application controls this daemon via BTstack

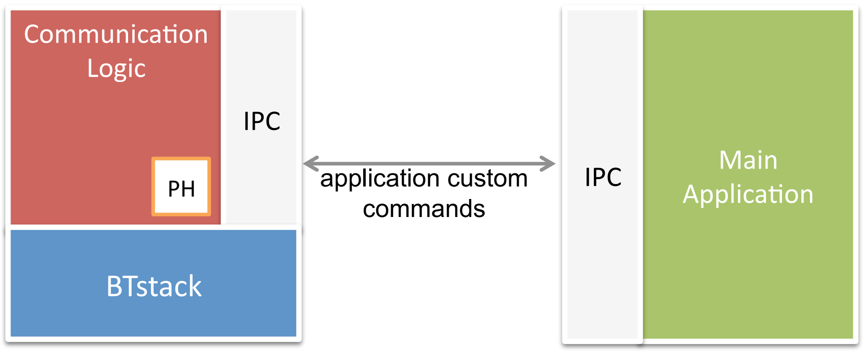

FIGURE 5. BTstack in multi-threaded environment - monolithic solution.

extended HCI command over IPC - this is used for the non-embedded version of BTstack e.g., on the iPhone and it is depicted in Figure 6. This option requires more code but provides more flexibility.

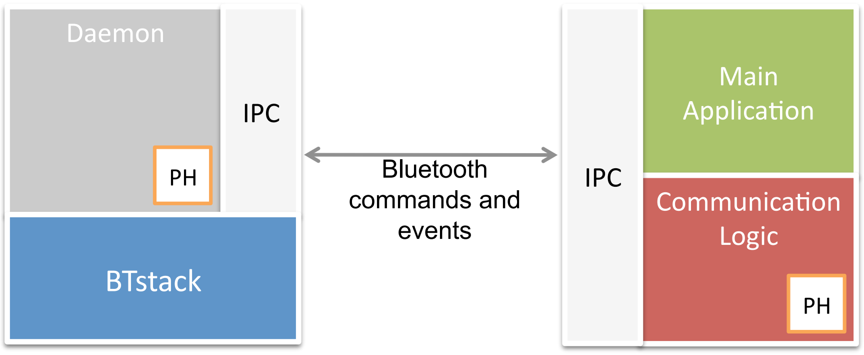

FIGURE 6. BTstack in multi-threaded environment - solution with daemon.

19. APIs

19.1. BLE ANCS Client API.

```
void ancs_client_init(void);
void ancs_client_register_callback(btstack_packet_handler_t callback
    );
const char * ancs_client_attribute_name_for_id(int id);
```

19.2. BLE ATT Database API.

```
/**
 * @brief Init ATT DB storage
 */
void att_db_util_init(void);

/**
 * @brief Add primary service for 16-bit UUID
 */
void att_db_util_add_service_uuid16(uint16_t udid16);

/**
 * @brief Add primary service for 128-bit UUID
 */
void att_db_util_add_service_uuid128(uint8_t * udid128);

/**
 * @brief Add Characteristic with 16-bit UUID, properties, and data
 * @returns attribute value handle
 * @see ATT_PROPERTY_* in ble/att_db.h
 */
uint16_t att_db_util_add_characteristic_uuid16(uint16_t  udid16,
    uint16_t properties, uint8_t * data, uint16_t data_len);

/**
 * @brief Add Characteristic with 128-bit UUID, properties, and data
 * @returns attribute value handle
 * @see ATT_PROPERTY_* in ble/att_db.h
 */
uint16_t att_db_util_add_characteristic_uuid128(uint8_t * udid128,
    uint16_t properties, uint8_t * data, uint16_t data_len);

/**
 * @brief Get address of constructed ATT DB
 */
uint8_t * att_db_util_get_address(void);

/**
 * @brief Get size of constructed ATT DB
 */
uint16_t att_db_util_get_size(void);
```

19.3. BLE ATT Server API.

```
/*
 * @brief setup ATT server
 * @param db attribute database created by compile-gatt.ph
 * @param read_callback, see att_db.h, can be NULL
 * @param write_callback, see attl.h, can be NULL
 */
void att_server_init(uint8_t const * db, att_read_callback_t
    read_callback, att_write_callback_t write_callback);

/*
 * @brief register packet handler for ATT server events:
 *        - ATT_EVENT_CAN_SEND_NOW
 *        - ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE
 *        - ATT_EVENT_MTU_EXCHANGE_COMPLETE
 * @param handler
 */
void att_server_register_packet_handler(btstack_packet_handler_t
    handler);

/*
 * @brief tests if a notification or indication can be send right
     now
 * @param con_handle
 * @return 1, if packet can be sent
 */
int  att_server_can_send_packet_now(hci_con_handle_t con_handle);

/**
 * @brief Request emission of ATT_EVENT_CAN_SEND_NOW as soon as
     possible
 * @note ATT_EVENT_CAN_SEND_NOW might be emitted during call to this
     function
 *       so packet handler should be ready to handle it
 * @param con_handle
 */
void att_server_request_can_send_now_event(hci_con_handle_t
    con_handle);

/**
 * @brief Request callback when sending is possible
 * @note callback might happend during call to this function
 * @param callback_registration to point to callback function and
     context information
 * @param con_handle
 */
void att_server_register_can_send_now_callback(
    btstack_context_callback_registration_t * callback_registration,
     hci_con_handle_t con_handle);

/*
 * @brief notify client about attribute value change
 * @param con_handle
 * @param attribute_handle
 * @param value
 * @param value_len
 * @return 0 if ok, error otherwise
 */
int att_server_notify(hci_con_handle_t con_handle, uint16_t
    attribute_handle, uint8_t *value, uint16_t value_len);

/*
 * @brief indicate value change to client. client is supposed to
     reply with an indication_response
 * @param con_handle
 * @param attribute_handle
 * @param value
 * @param value_len
 * @return 0 if ok, error otherwise
 */
int att_server_indicate(hci_con_handle_t con_handle, uint16_t
    attribute_handle, uint8_t *value, uint16_t value_len);
```

19.4. **BLE GATT Client API.**

```
typedef struct {
    uint16_t start_group_handle;
    uint16_t end_group_handle;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} gatt_client_service_t;

typedef struct {
    uint16_t start_handle;
    uint16_t value_handle;
    uint16_t end_handle;
    uint16_t properties;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} gatt_client_characteristic_t;

typedef struct {
    uint16_t handle;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} gatt_client_characteristic_descriptor_t;

/**
 * @brief Set up GATT client.
 */
void gatt_client_init(void);

/**
 * @brief MTU is available after the first query has completed. If
     status is equal to 0, it returns the real value, otherwise the
     default value of 23.
 */
uint8_t gatt_client_get_mtu(hci_con_handle_t con_handle, uint16_t *
   mtu);

/**
 * @brief Returns if the GATT client is ready to receive a query. It
     is used with daemon.
 */
int gatt_client_is_ready(hci_con_handle_t con_handle);

/**
 * @brief Discovers all primary services. For each found service, an
     le_service_event_t with type set to
    GATT_EVENT_SERVICE_QUERY_RESULT will be generated and passed to
     the registered callback. The gatt_complete_event_t, with type
    set to GATT_EVENT_QUERY_COMPLETE, marks the end of discovery.
 */
uint8_t gatt_client_discover_primary_services(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle);

/**
 * @brief Discovers a specific primary service given its UUID. This
    service may exist multiple times. For each found service, an
    le_service_event_t with type set to
    GATT_EVENT_SERVICE_QUERY_RESULT will be generated and passed to
     the registered callback. The gatt_complete_event_t, with type
    set to GATT_EVENT_QUERY_COMPLETE, marks the end of discovery.
 */
uint8_t gatt_client_discover_primary_services_by_uuid16(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t uuid16);
uint8_t gatt_client_discover_primary_services_by_uuid128(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   const uint8_t * uuid);

/**
 * @brief Finds included services within the specified service. For
    each found included service, an le_service_event_t with type
    set to GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT will be
    generated and passed to the registered callback. The
    gatt_complete_event_t with type set to
    GATT_EVENT_QUERY_COMPLETE, marks the end of discovery.
    Information about included service type (primary/secondary) can
     be retrieved either by sending an ATT find information request
     for the returned start group handle (returning the handle and
    the UUID for primary or secondary service) or by comparing the
    service to the list of all primary services.
 */
uint8_t gatt_client_find_included_services_for_service(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_service_t *service);

/**
```

```
 * @brief Discovers all characteristics within the specified service
     . For each found characteristic, an le_characteristics_event_t
     with type set to GATT_EVENT_CHARACTERISTIC_QUERY_RESULT will be
      generated and passed to the registered callback. The
     gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, marks the end of discovery.
 */
uint8_t gatt_client_discover_characteristics_for_service(
    btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    gatt_client_service_t  *service);

/**
 * @brief The following four functions are used to discover all
     characteristics within the specified service or handle range,
     and return those that match the given UUID. For each found
     characteristic, an le_characteristic_event_t with type set to
     GATT_EVENT_CHARACTERISTIC_QUERY_RESULT will be generated and
     passed to the registered callback. The gatt_complete_event_t
     with type set to GATT_EVENT_QUERY_COMPLETE, marks the end of
     discovery.
 */
uint8_t
    gatt_client_discover_characteristics_for_handle_range_by_uuid16(
    btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t start_handle, uint16_t end_handle, uint16_t uuid16);
uint8_t
    gatt_client_discover_characteristics_for_handle_range_by_uuid128
    (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
     uint16_t start_handle, uint16_t end_handle, uint8_t  * uuid);
uint8_t gatt_client_discover_characteristics_for_service_by_uuid16(
    btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    gatt_client_service_t  *service, uint16_t  uuid16);
uint8_t gatt_client_discover_characteristics_for_service_by_uuid128(
    btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    gatt_client_service_t  *service, uint8_t  * uuid128);

/**
 * @brief Discovers attribute handle and UUID of a characteristic
     descriptor within the specified characteristic. For each found
     descriptor, an le_characteristic_descriptor_event_t with type
     set to GATT_EVENT_ALL_CHARACTERISTIC_DESCRIPTORS_QUERY_RESULT
     will be generated and passed to the registered callback. The
     gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, marks the end of discovery.
 */
uint8_t gatt_client_discover_characteristic_descriptors(
    btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    gatt_client_characteristic_t  *characteristic);

/**
```

```
 * @brief Reads the characteristic value using the characteristic's
     value handle. If the characteristic value is found, an
     le_characteristic_value_event_t with type set to
     GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT will be generated
     and passed to the registered callback. The
     gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, marks the end of read.
 */
uint8_t gatt_client_read_value_of_characteristic(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_t  *characteristic);
uint8_t gatt_client_read_value_of_characteristic_using_value_handle(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t characteristic_value_handle);

/**
 * @brief Reads the characteric value of all characteristics with
     the uuid. For each found, an le_characteristic_value_event_t
     with type set to GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT
     will be generated and passed to the registered callback. The
     gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, marks the end of read.
 */
uint8_t gatt_client_read_value_of_characteristics_by_uuid16(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t start_handle, uint16_t end_handle, uint16_t uuid16);
uint8_t gatt_client_read_value_of_characteristics_by_uuid128(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t start_handle, uint16_t end_handle, uint8_t * uuid128);

/**
 * @brief Reads the long characteristic value using the
     characteristic's value handle. The value will be returned in
     several blobs. For each blob, an
     le_characteristic_value_event_t with type set to
     GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT and updated value
     offset will be generated and passed to the registered callback.
      The gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, mark the end of read.
 */
uint8_t gatt_client_read_long_value_of_characteristic(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_t  *characteristic);
uint8_t
   gatt_client_read_long_value_of_characteristic_using_value_handle
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t characteristic_value_handle);
uint8_t
   gatt_client_read_long_value_of_characteristic_using_value_handle_with_offset
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t characteristic_value_handle, uint16_t offset);

/*
 * @brief Read multiple characteristic values
 * @param number handles
 * @param list_of_handles list of handles
 */
uint8_t gatt_client_read_multiple_characteristic_values(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   int num_value_handles, uint16_t * value_handles);

/**
 * @brief Writes the characteristic value using the characteristic's
      value handle without an acknowledgment that the write was
     successfully performed.
 */
uint8_t gatt_client_write_value_of_characteristic_without_response(
   hci_con_handle_t con_handle, uint16_t
   characteristic_value_handle, uint16_t length, uint8_t  * data);

/**
 * @brief Writes the authenticated characteristic value using the
     characteristic's value handle without an acknowledgment that
     the write was successfully performed.
 */
uint8_t gatt_client_signed_write_without_response(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t handle, uint16_t message_len, uint8_t  * message);

/**
 * @brief Writes the characteristic value using the characteristic's
      value handle. The gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE, marks the end of write. The write is
      successfully performed, if the event's status field is set to
     0.
 */
uint8_t gatt_client_write_value_of_characteristic(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t characteristic_value_handle, uint16_t length, uint8_t
   * data);
uint8_t gatt_client_write_long_value_of_characteristic(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t characteristic_value_handle, uint16_t length, uint8_t
   * data);
uint8_t gatt_client_write_long_value_of_characteristic_with_offset(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t characteristic_value_handle, uint16_t offset, uint16_t
   length, uint8_t  * data);

/**
 * @brief Writes of the long characteristic value using the
     characteristic's value handle. It uses server response to
     validate that the write was correctly received. The
     gatt_complete_event_t with type set to
     GATT_EVENT_QUERY_COMPLETE marks the end of write. The write is
     successfully performed, if the event's status field is set to
     0.
 */
```

```
uint8_t gatt_client_reliable_write_long_value_of_characteristic(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   uint16_t characteristic_value_handle, uint16_t length, uint8_t
   * data);

/**
 * @brief Reads the characteristic descriptor using its handle. If
    the characteristic descriptor is found, an
    le_characteristic_descriptor_event_t with type set to
    GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT will be
    generated and passed to the registered callback. The
    gatt_complete_event_t with type set to
    GATT_EVENT_QUERY_COMPLETE, marks the end of read.
 */
uint8_t gatt_client_read_characteristic_descriptor(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_descriptor_t * descriptor);
uint8_t
   gatt_client_read_characteristic_descriptor_using_descriptor_handle
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle);

/**
 * @brief Reads the long characteristic descriptor using its handle.
     It will be returned in several blobs. For each blob, an
    le_characteristic_descriptor_event_t with type set to
    GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT will be
    generated and passed to the registered callback. The
    gatt_complete_event_t with type set to
    GATT_EVENT_QUERY_COMPLETE, marks the end of read.
 */
uint8_t gatt_client_read_long_characteristic_descriptor(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_descriptor_t * descriptor);
uint8_t
   gatt_client_read_long_characteristic_descriptor_using_descriptor_handle
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle);
uint8_t
   gatt_client_read_long_characteristic_descriptor_using_descriptor_handle_with_offs
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle, uint16_t offset);

/**
 * @brief Writes the characteristic descriptor using its handle. The
     gatt_complete_event_t with type set to
    GATT_EVENT_QUERY_COMPLETE, marks the end of write. The write is
     successfully performed, if the event's status field is set to
    0.
 */
uint8_t gatt_client_write_characteristic_descriptor(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_descriptor_t * descriptor, uint16_t
   length, uint8_t * data);
```

```
uint8_t
   gatt_client_write_characteristic_descriptor_using_descriptor_handle
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle, uint16_t length, uint8_t * data);
uint8_t gatt_client_write_long_characteristic_descriptor(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_descriptor_t * descriptor, uint16_t
   length, uint8_t * data);
uint8_t
   gatt_client_write_long_characteristic_descriptor_using_descriptor_handle
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle, uint16_t length, uint8_t * data);
uint8_t
   gatt_client_write_long_characteristic_descriptor_using_descriptor_handle_with_off
   (btstack_packet_handler_t callback, hci_con_handle_t con_handle,
    uint16_t descriptor_handle, uint16_t offset, uint16_t length,
   uint8_t * data);

/**
 * @brief Writes the client characteristic configuration of the
     specified characteristic. It is used to subscribe for
     notifications or indications of the characteristic value. For
     notifications or indications specify:
     GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION resp.
     GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION as
     configuration value.
 */
uint8_t gatt_client_write_client_characteristic_configuration(
   btstack_packet_handler_t callback, hci_con_handle_t con_handle,
   gatt_client_characteristic_t * characteristic, uint16_t
   configuration);

/**
 * @brief Register for notifications and indications of a
     characteristic enabled by
     gatt_client_write_client_characteristic_configuration
 * @param notification struct used to store registration
 * @param packet_handler
 * @param con_handle
 * @param characteristic
 */
void gatt_client_listen_for_characteristic_value_updates(
   gatt_client_notification_t * notification,
   btstack_packet_handler_t packet_handler, hci_con_handle_t
   con_handle, gatt_client_characteristic_t * characteristic);

/**
 * @brief Stop listening to characteristic value updates registered
     with gatt_client_listen_for_characteristic_value_updates
 * @param notification struct used in
     gatt_client_listen_for_characteristic_value_updates
 */
void gatt_client_stop_listening_for_characteristic_value_updates(
   gatt_client_notification_t * notification);
```

```
/**
 * @brief -> gatt complete event
 */
uint8_t gatt_client_prepare_write(btstack_packet_handler_t callback,
    hci_con_handle_t con_handle, uint16_t attribute_handle,
   uint16_t offset, uint16_t length, uint8_t * data);

/**
 * @brief -> gatt complete event
 */
uint8_t gatt_client_execute_write(btstack_packet_handler_t callback,
    hci_con_handle_t con_handle);

/**
 * @brief -> gatt complete event
 */
uint8_t gatt_client_cancel_write(btstack_packet_handler_t callback,
   hci_con_handle_t con_handle);
```

19.5. **BLE Device Database API.**

```
/**
 * @brief init
 */
void le_device_db_init(void);


/**
 * @brief sets local bd addr. allows for db per Bluetooth controller
 * @param bd_addr
 */
void le_device_db_set_local_bd_addr(bd_addr_t bd_addr);

/**
 * @brief add device to db
 * @param addr_type, address of the device
 * @param irk of the device
 * @returns index if successful, -1 otherwise
 */
int le_device_db_add(int addr_type, bd_addr_t addr, sm_key_t irk);

/**
 * @brief get number of devices in db for enumeration
 * @returns number of device in db
 */
int le_device_db_count(void);

/**
 * @brief get device information: addr type and address needed to
    identify device
 * @param index
 * @param addr_type, address of the device as output
 * @param irk of the device
 */
void le_device_db_info(int index, int * addr_type, bd_addr_t addr,
   sm_key_t irk);


/**
 * @brief set remote encryption info
 * @brief index
 * @brief ediv
 * @brief rand
 * @brief ltk
 * @brief key size
 * @brief authenticated
 * @brief authorized
 */
void le_device_db_encryption_set(int index, uint16_t ediv, uint8_t
   rand[8], sm_key_t ltk, int key_size, int authenticated, int
   authorized);

/**
 * @brief get remote encryption info
 * @brief index
 * @brief ediv
 * @brief rand
 * @brief ltk
 * @brief key size
 * @brief authenticated
 * @brief authorized
 */
void le_device_db_encryption_get(int index, uint16_t * ediv, uint8_t
    rand[8], sm_key_t ltk,  int * key_size, int * authenticated,
   int * authorized);

#ifdef ENABLE_LE_SIGNED_WRITE

/**
 * @brief set local signing key for this device
 * @param index
 * @param signing key as input
 */
void le_device_db_local_csrk_set(int index, sm_key_t csrk);

/**
 * @brief get local signing key for this device
 * @param index
 * @param signing key as output
 */
void le_device_db_local_csrk_get(int index, sm_key_t csrk);

/**
 * @brief set remote signing key for this device
 * @param index
 * @param signing key as input
 */
void le_device_db_remote_csrk_set(int index, sm_key_t csrk);

/**
 * @brief get remote signing key for this device
 * @param index
 * @param signing key as output
 */
void le_device_db_remote_csrk_get(int index, sm_key_t csrk);

/**
 * @brief query last used/seen signing counter
 * @param index
 * @returns next expected counter, 0 after devices was added
 */
uint32_t le_device_db_remote_counter_get(int index);

/**
 * @brief update signing counter
 * @param index
 * @param counter to store
 */
void le_device_db_remote_counter_set(int index, uint32_t counter);

/**
 * @brief query last used/seen signing counter
 * @param index
 * @returns next expected counter, 0 after devices was added
 */
uint32_t le_device_db_local_counter_get(int index);

/**
 * @brief update signing counter
 * @param index
 * @param counter to store
 */
void le_device_db_local_counter_set(int index, uint32_t counter);

#endif

/**
 * @brief free device
 * @param index
 */
void le_device_db_remove(int index);

void le_device_db_dump(void);
```

19.6. BLE Security Manager API.

```
/**
 * @brief Initializes the Security Manager, connects to L2CAP
 */
```

```
void sm_init(void);

/**
 * @brief Set secret ER key for key generation as described in Core
     V4.0, Vol 3, Part G, 5.2.2
 * @param er
 */
void sm_set_er(sm_key_t er);

/**
 * @brief Set secret IR key for key generation as described in Core
     V4.0, Vol 3, Part G, 5.2.2
 */
void sm_set_ir(sm_key_t ir);

/**
 *
 * @brief Registers OOB Data Callback. The callback should set the
     oob_data and return 1 if OOB data is availble
 * @param get_oob_data_callback
 */
void sm_register_oob_data_callback( int (*get_oob_data_callback)(
    uint8_t addres_type, bd_addr_t addr, uint8_t * oob_data));

/**
 * @brief Add event packet handler.
 */
void sm_add_event_handler(btstack_packet_callback_registration_t *
    callback_handler);

/**
 * @brief Limit the STK generation methods. Bonding is stopped if
     the resulting one isn't in the list
 * @param OR combination of SM_STK_GENERATION_METHOD_
 */
void sm_set_accepted_stk_generation_methods(uint8_t
    accepted_stk_generation_methods);

/**
 * @brief Set the accepted encryption key size range. Bonding is
     stopped if the result isn't within the range
 * @param min_size (default 7)
 * @param max_size (default 16)
 */
void sm_set_encryption_key_size_range(uint8_t min_size, uint8_t
    max_size);

/**
 * @brief Sets the requested authentication requirements, bonding
     yes/no, MITM yes/no, SC yes/no, keypress yes/no
 * @param OR combination of SM_AUTHREQ_ flags
 */
void sm_set_authentication_requirements(uint8_t auth_req);
```

```
/**
 * @brief Sets the available IO Capabilities
 * @param IO_CAPABILITY_
 */
void sm_set_io_capabilities(io_capability_t io_capability);

/**
 * @brief Let Peripheral request an encrypted connection right after
      connecting
 * @note Not used normally. Bonding is triggered by access to
     protected attributes in ATT Server
 */
void sm_set_request_security(int enable);

/**
 * @brief Trigger Security Request
 * @note Not used normally. Bonding is triggered by access to
     protected attributes in ATT Server
 */
void sm_send_security_request(hci_con_handle_t con_handle);

/**
 * @brief Decline bonding triggered by event before
 * @param con_handle
 */
void sm_bonding_decline(hci_con_handle_t con_handle);

/**
 * @brief Confirm Just Works bonding
 * @param con_handle
 */
void sm_just_works_confirm(hci_con_handle_t con_handle);

/**
 * @brief Confirm value from SM_EVENT_NUMERIC_COMPARISON_REQUEST for
      Numeric Comparison bonding
 * @param con_handle
 */
void sm_numeric_comparison_confirm(hci_con_handle_t con_handle);

/**
 * @brief Reports passkey input by user
 * @param con_handle
 * @param passkey in [0..999999]
 */
void sm_passkey_input(hci_con_handle_t con_handle, uint32_t passkey)
    ;

/**
 * @brief Send keypress notification for keyboard only devices
 * @param con_handle
 * @param action see SM_KEYPRESS_* in bluetooth.h
 */
```

```
void sm_keypress_notification(hci_con_handle_t con_handle, uint8_t
    action);

/**
 *
 * @brief Get encryption key size.
 * @param con_handle
 * @return 0 if not encrypted, 7-16 otherwise
 */
int sm_encryption_key_size(hci_con_handle_t con_handle);

/**
 * @brief Get authentication property.
 * @param con_handle
 * @return 1 if bonded with OOB/Passkey (AND MITM protection)
 */
int sm_authenticated(hci_con_handle_t con_handle);

/**
 * @brief Queries authorization state.
 * @param con_handle
 * @return authorization_state for the current session
 */
authorization_state_t sm_authorization_state(hci_con_handle_t
    con_handle);

/**
 * @brief Used by att_server.c to request user authorization.
 * @param con_handle
 */
void sm_request_pairing(hci_con_handle_t con_handle);

/**
 * @brief Report user authorization decline.
 * @param con_handle
 */
void sm_authorization_decline(hci_con_handle_t con_handle);

/**
 * @brief Report user authorization grant.
 * @param con_handle
 */
void sm_authorization_grant(hci_con_handle_t con_handle);


/**
 * @brief Check if CMAC AES engine is ready
 * @return ready
 */
 int sm_cmac_ready(void);

/*
 * @brief Generic CMAC AES
 * @param key
 * @param message_len
 * @param get_byte_callback
 * @param done_callback
 * @note hash is 16 bytes in big endian
 */
void sm_cmac_general_start(const sm_key_t key, uint16_t message_len,
     uint8_t (*get_byte_callback)(uint16_t offset), void (*
    done_callback)(uint8_t * hash));

/**
 * @brief Support for signed writes, used by att_server.
 * @note Message is in little endian to allows passing in ATT PDU
     without flipping.
 * @note signing data: [opcode, attribute_handle, message,
     sign_counter]
 * @note calculated hash in done_callback is big endian and has 16
     byte.
 * @param key
 * @param opcde
 * @param attribute_handle
 * @param message_len
 * @param message
 * @param sign_counter
 */
void sm_cmac_signed_write_start(const sm_key_t key, uint8_t opcode,
    uint16_t attribute_handle, uint16_t message_len, const uint8_t *
     message, uint32_t sign_counter, void (*done_callback)(uint8_t *
     hash));

/*
 * @brief Match address against bonded devices
 * @return 0 if successfully added to lookup queue
 * @note Triggers SM_IDENTITY_RESOLVING_* events
 */
int sm_address_resolution_lookup(uint8_t addr_type, bd_addr_t addr);

/**
 * @brief Identify device in LE Device DB.
 * @param handle
 * @return index from le_device_db or -1 if not found/identified
 */
int sm_le_device_index(hci_con_handle_t con_handle );

/**
 * @brief Set Elliptic Key Public/Private Keypair
 * @note Using the same key for more than one device is not
     recommended.
 * @param qx 32 bytes
 * @param qy 32 bytes
 * @param d  32 bytes
 */
void sm_use_fixed_ec_keypair(uint8_t * qx, uint8_t * qy, uint8_t * d
    );
```

19.7. BNEP API.

```
/**
 * @brief Set up BNEP.
 */
void bnep_init(void);

/**
 * @brief Check if a data packet can be send out.
 */
int bnep_can_send_packet_now(uint16_t bnep_cid);

/**
 * @brief Request emission of BNEP_CAN_SEND_NOW as soon as possible
 * @note BNEP_CAN_SEND_NOW might be emitted during call to this
     function
 *       so packet handler should be ready to handle it
 * @param bnep_cid
 */
void bnep_request_can_send_now_event(uint16_t bnep_cid);

/**
 * @brief Send a data packet.
 */
int bnep_send(uint16_t bnep_cid, uint8_t *packet, uint16_t len);

/**
 * @brief Set the network protocol filter.
 */
int bnep_set_net_type_filter(uint16_t bnep_cid, bnep_net_filter_t *
    filter, uint16_t len);

/**
 * @brief Set the multicast address filter.
 */
int bnep_set_multicast_filter(uint16_t bnep_cid, bnep_multi_filter_t
     *filter, uint16_t len);

/**
 * @brief Set security level required for incoming connections, need
      to be called before registering services.
 */
void bnep_set_required_security_level(gap_security_level_t
    security_level);

/**
 * @brief Creates BNEP connection (channel) to a given server on a
     remote device with baseband address. A new baseband connection
     will be initiated if necessary.
 */
int bnep_connect(btstack_packet_handler_t packet_handler, bd_addr_t
    addr, uint16_t l2cap_psm, uint16_t uuid_src, uint16_t uuid_dest)
    ;
```

```
/**
 * @brief Disconnects BNEP channel with given identifier.
 */
void bnep_disconnect(bd_addr_t addr);

/**
 * @brief Registers BNEP service, set a maximum frame size and
     assigns a packet handler. On embedded systems, use NULL for
     connection parameter.
 */
uint8_t bnep_register_service(btstack_packet_handler_t
   packet_handler, uint16_t service_uuid, uint16_t max_frame_size);

/**
 * @brief Unregister BNEP service.
 */
void bnep_unregister_service(uint16_t service_uuid);
```

19.8. Link Key DB API.

```
typedef struct {

    // management
    void (*open)(void);
    void (*set_local_bd_addr)(bd_addr_t bd_addr);
    void (*close)(void);

    // link key
    int  (*get_link_key)(bd_addr_t bd_addr, link_key_t link_key,
       link_key_type_t * type);
    void (*put_link_key)(bd_addr_t bd_addr, link_key_t link_key,
       link_key_type_t   type);
    void (*delete_link_key)(bd_addr_t bd_addr);

} btstack_link_key_db_t;
```

19.9. HSP Headset API.

```
/**
 * @brief Set up HSP HS.
 * @param rfcomm_channel_nr
 */
void hsp_hs_init(uint8_t rfcomm_channel_nr);

/**
 * @brief Create HSP Headset (HS) SDP service record.
 * @param service Empty buffer in which a new service record will be
      stored.
 * @param rfcomm_channel_nr
 * @param name
 * @param have_remote_audio_control
 */
```

```
void hsp_hs_create_sdp_record(uint8_t * service, uint32_t
    service_record_handle, int rfcomm_channel_nr, const char * name,
     uint8_t have_remote_audio_control);

/**
 * @brief Register packet handler to receive HSP HS events.
 *
 * The HSP HS event has type HCI_EVENT_HSP_META with following
     subtypes:
 * - HSP_SUBEVENT_RFCOMM_CONNECTION_COMPLETE
 * - HSP_SUBEVENT_RFCOMM_DISCONNECTION_COMPLETE
 * - HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE
 * - HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE
 * - HSP_SUBEVENT_RING
 * - HSP_SUBEVENT_MICROPHONE_GAIN_CHANGED
 * - HSP_SUBEVENT_SPEAKER_GAIN_CHANGED
 * - HSP_SUBEVENT_AG_INDICATION
 *
 * @param callback
 */
void hsp_hs_register_packet_handler(btstack_packet_handler_t
    callback);

/**
 * @brief Connect to HSP Audio Gateway.
 *
 * Perform SDP query for an RFCOMM service on a remote device,
 * and establish an RFCOMM connection if such service is found.
     Reception of the
 * HSP_SUBEVENT_RFCOMM_CONNECTION_COMPLETE with status 0
 * indicates if the connection is successfully established.
 *
 * @param bd_addr
 */
void hsp_hs_connect(bd_addr_t bd_addr);

/**
 * @brief Disconnect from HSP Audio Gateway
 *
 * Releases the RFCOMM channel. Reception of the
 * HSP_SUBEVENT_RFCOMM_DISCONNECTION_COMPLETE with status 0
 * indicates if the connection is successfully released.
 * @param bd_addr
 */
void hsp_hs_disconnect(void);


/**
 * @brief Send button press action. Toggle establish/release of
     audio connection.
 */
void hsp_hs_send_button_press(void);

/**
```

```
 * @brief Triger establishing audio connection.
 *
 * Reception of the HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE with
     status 0
 * indicates if the audio connection is successfully established.
 * @param bd_addr
 */
void hsp_hs_establish_audio_connection(void);

/**
 * @brief Trigger releasing audio connection.
 *
 * Reception of the HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE with
     status 0
 * indicates if the connection is successfully released.
 * @param bd_addr
 */
void hsp_hs_release_audio_connection(void);

/**
 * @brief Set microphone gain.
 *
 * The new gain value will be confirmed by the HSP Audio Gateway.
 * A HSP_SUBEVENT_MICROPHONE_GAIN_CHANGED event will be received.
 * @param gain Valid range: [0,15]
 */
void hsp_hs_set_microphone_gain(uint8_t gain);

/**
 * @brief Set speaker gain.
 *
 * The new gain value will be confirmed by the HSP Audio Gateway.
 * A HSP_SUBEVENT_SPEAKER_GAIN_CHANGED event will be received.
 * @param gain - valid range: [0,15]
 */
void hsp_hs_set_speaker_gain(uint8_t gain);



/**
 * @brief Enable custom indications.
 *
 * Custom indications are disabled by default.
 * When enabled, custom indications are received via the
     HSP_SUBEVENT_AG_INDICATION.
 * @param enable
 */
void hsp_hs_enable_custom_indications(int enable);

/**
 * @brief Send answer to custom indication.
 *
 * On HSP_SUBEVENT_AG_INDICATION, the client needs to respond
 * with this function with the result to the custom indication
 * @param result
 */
int hsp_hs_send_result(const char * result);
```

19.10. HSP Audio Gateway API.

```
/**
 * @brief Set up HSP AG.
 * @param rfcomm_channel_nr
 */
void hsp_ag_init(uint8_t rfcomm_channel_nr);

/**
 * @brief Create HSP Audio Gateway (AG) SDP service record.
 * @param service Empty buffer in which a new service record will be
     stored.
 * @param service_record_handle
 * @param rfcomm_channel_nr
 * @param name
 */
void hsp_ag_create_sdp_record(uint8_t * service, uint32_t
    service_record_handle, int rfcomm_channel_nr, const char * name)
    ;

/**
 * @brief Register packet handler to receive HSP AG events.

 * The HSP AG event has type HCI_EVENT_HSP_META with following
     subtypes:
 * - HSP_SUBEVENT_RFCOMM_CONNECTION_COMPLETE
 * - HSP_SUBEVENT_RFCOMM_DISCONNECTION_COMPLETE
 * - HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE
 * - HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE
 * - HSP_SUBEVENT_MICROPHONE_GAIN_CHANGED
 * - HSP_SUBEVENT_SPEAKER_GAIN_CHANGED
 * - HSP_SUBEVENT_HS_COMMAND
 *
 * @param callback
 */
void hsp_ag_register_packet_handler(btstack_packet_handler_t
    callback);

/**
 * @brief Connect to HSP Headset.
 *
 * Perform SDP query for an RFCOMM service on a remote device,
 * and establish an RFCOMM connection if such service is found.
     Reception of the
 * HSP_SUBEVENT_RFCOMM_CONNECTION_COMPLETE with status 0
 * indicates if the connection is successfully established.
 *
 * @param bd_addr
 */
```

```
void hsp_ag_connect(bd_addr_t bd_addr);

/**
 * @brief Disconnect from HSP Headset
 *
 * Reception of the HSP_SUBEVENT_RFCOMM_DISCONNECTION_COMPLETE with
     status 0
 * indicates if the connection is successfully released.
 * @param bd_addr
 */
void hsp_ag_disconnect(void);


/**
 * @brief Establish audio connection.
 *
 * Reception of the HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE with
     status 0
 * indicates if the audio connection is successfully established.
 * @param bd_addr
 */
void hsp_ag_establish_audio_connection(void);

/**
 * @brief Release audio connection.
 *
 * Reception of the HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE with
     status 0
 * indicates if the connection is successfully released.
 * @param bd_addr
 */
void hsp_ag_release_audio_connection(void);

/**
 * @brief Set microphone gain.
 * @param gain Valid range: [0,15]
 */
void hsp_ag_set_microphone_gain(uint8_t gain);

/**
 * @brief Set speaker gain.
 * @param gain Valid range: [0,15]
 */
void hsp_ag_set_speaker_gain(uint8_t gain);

/**
 * @brief Start ringing because of incoming call.
 */
void hsp_ag_start_ringing(void);

/**
 * @brief Stop ringing (e.g. call was terminated).
 */
void hsp_ag_stop_ringing(void);
```

```
/**
 * @brief Enable custom AT commands.
 *
 * Custom commands are disabled by default.
 * When enabled, custom AT commands are received via the
    HSP_SUBEVENT_HS_COMMAND.
 * @param enable
 */
void hsp_ag_enable_custom_commands(int enable);

/**
 * @brief Send a custom AT command to HSP Headset.
 *
 * On HSP_SUBEVENT_AG_INDICATION, the client needs to respond
 * with this function with the result to the custom command.
 * @param result
 */
int hsp_ag_send_result(char * result);
```

19.11. HFP Hands-Free API.

```
/**
 * @brief Create HFP Hands-Free (HF) SDP service record.
 * @param service
 * @param rfcomm_channel_nr
 * @param name
 * @param suported_features 32-bit bitmap, see HFP_HFSF_* values in
    hfp.h
 * @param wide_band_speech supported
 */
void hfp_hf_create_sdp_record(uint8_t * service, uint32_t
   service_record_handle, int rfcomm_channel_nr, const char * name,
    uint16_t supported_features, int wide_band_speech);

/**
 * @brief Set up HFP Hands-Free (HF) device without additional
    supported features.
 * @param rfcomm_channel_nr
 */
void hfp_hf_init(uint16_t rfcomm_channel_nr);

/**
 * @brief Set codecs.
 * @param codecs_nr
 * @param codecs
 */
void hfp_hf_init_codecs(int codecs_nr, uint8_t * codecs);

/**
 * @brief Set supported features.
 * @param supported_features 32-bit bitmap, see HFP_HFSF_* values in
     hfp.h
 */
void hfp_hf_init_supported_features(uint32_t supported_features);

/**
 * @brief Set HF indicators.
 * @param indicators_nr
 * @param indicators
 */
void hfp_hf_init_hf_indicators(int indicators_nr, uint16_t *
   indicators);


/**
 * @brief Register callback for the HFP Hands-Free (HF) client.
 * @param callback
 */
void hfp_hf_register_packet_handler(btstack_packet_handler_t
   callback);

/**
 * @brief Establish RFCOMM connection with the AG with given
     Bluetooth address,
 * and perform service level connection (SLC) agreement:
 * - exchange supported features
 * - retrieve Audio Gateway (AG) indicators and their status
 * - enable indicator status update in the AG
 * - notify the AG about its own available codecs, if possible
 * - retrieve the AG information describing the call hold and
     multiparty services, if possible
 * - retrieve which HF indicators are enabled on the AG, if possible
 * The status of SLC connection establishment is reported via
 * HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_ESTABLISHED.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_establish_service_level_connection(bd_addr_t bd_addr);

/**
 * @brief Release the RFCOMM channel and the audio connection
     between the HF and the AG.
 * The status of releasing the SLC connection is reported via
 * HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_RELEASED.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_release_service_level_connection(hci_con_handle_t
   acl_handle);

/**
 * @brief Enable status update for all indicators in the AG.
 * The status field of the HFP_SUBEVENT_COMPLETE reports if the
     command was accepted.
 * The status of an AG indicator is reported via
     HFP_SUBEVENT_AG_INDICATOR_STATUS_CHANGED.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_enable_status_update_for_all_ag_indicators(
    hci_con_handle_t acl_handle);

/**
 * @brief Disable status update for all indicators in the AG.
 * The status field of the HFP_SUBEVENT_COMPLETE reports if the
     command was accepted.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_disable_status_update_for_all_ag_indicators(
    hci_con_handle_t acl_handle);

/**
 * @brief Enable or disable status update for the individual
     indicators in the AG using bitmap.
 * The status field of the HFP_SUBEVENT_COMPLETE reports if the
     command was accepted.
 * The status of an AG indicator is reported via
     HFP_SUBEVENT_AG_INDICATOR_STATUS_CHANGED.
 *
 * @param bd_addr Bluetooth address of the AG
 * @param indicators_status_bitmap 32-bit bitmap, 0 - indicator is
     disabled, 1 - indicator is enabled
 */
void hfp_hf_set_status_update_for_individual_ag_indicators(
    hci_con_handle_t acl_handle, uint32_t indicators_status_bitmap);

/**
 * @brief Query the name of the currently selected Network operator
     by AG.
 *
 * The name is restricted to max 16 characters. The result is
     reported via
 * HFP_SUBEVENT_NETWORK_OPERATOR_CHANGED subtype
 * containing network operator mode, format and name.
 * If no operator is selected, format and operator are omitted.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_query_operator_selection(hci_con_handle_t acl_handle);

/**
 * @brief Enable Extended Audio Gateway Error result codes in the AG
     .
 * Whenever there is an error relating to the functionality of the
     AG as a
 * result of AT command, the AG shall send +CME ERROR. This error is
      reported via
 * HFP_SUBEVENT_EXTENDED_AUDIO_GATEWAY_ERROR, see hfp_cme_error_t in
      hfp.h
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_enable_report_extended_audio_gateway_error_result_code(
    hci_con_handle_t acl_handle);

/**
 * @brief Disable Extended Audio Gateway Error result codes in the
     AG.
 *
 * @param bd_addr Bluetooth address of the AG
 */
 void hfp_hf_disable_report_extended_audio_gateway_error_result_code
     (hci_con_handle_t acl_handle);

/**
 * @brief Establish audio connection.
 * The status of audio connection establishment is reported via
 * HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_establish_audio_connection(hci_con_handle_t acl_handle);

/**
 * @brief Release audio connection.
 * The status of releasing of the audio connection is reported via
 * HFP_SUBEVENT_AUDIO_CONNECTION_RELEASED.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_release_audio_connection(hci_con_handle_t acl_handle);

/**
 * @brief Answer incoming call.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_answer_incoming_call(hci_con_handle_t acl_handle);

/**
 * @brief Reject incoming call.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_reject_incoming_call(hci_con_handle_t acl_handle);

/**
 * @brief Release all held calls or sets User Determined User Busy (
     UDUB) for a waiting call.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_user_busy(hci_con_handle_t acl_handle);

/**
 * @brief Release all active calls (if any exist) and accepts the
     other (held or waiting) call.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_end_active_and_accept_other(hci_con_handle_t acl_handle)
    ;

/**
 * @brief Place all active calls (if any exist) on hold and accepts
     the other (held or waiting) call.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_swap_calls(hci_con_handle_t acl_handle);

/**
 * @brief Add a held call to the conversation.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_join_held_call(hci_con_handle_t acl_handle);

/**
 * @brief Connect the two calls and disconnects the subscriber from
     both calls (Explicit Call
Transfer).
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_connect_calls(hci_con_handle_t acl_handle);

/**
 * @brief Terminate an incoming or an outgoing call.
 * HFP_SUBEVENT_CALL_TERMINATED is sent upon call termination.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_terminate_call(hci_con_handle_t acl_handle);

/**
 * @brief Initiate outgoing voice call by providing the destination
     phone number to the AG.
 * @param bd_addr Bluetooth address of the AG
 * @param number
 */
void hfp_hf_dial_number(hci_con_handle_t acl_handle, char * number);

/**
 * @brief Initiate outgoing voice call using the memory dialing
     feature of the AG.
 * @param bd_addr Bluetooth address of the AG
 * @param memory_id
 */
void hfp_hf_dial_memory(hci_con_handle_t acl_handle, int memory_id);

/**
 * @brief Initiate outgoing voice call by recalling the last number
     dialed by the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_redial_last_number(hci_con_handle_t acl_handle);
```

```
/*
 * @brief Enable the   Call Waiting notification  function in the
     AG.
 * The AG shall send the corresponding result code to the HF
     whenever
 * an incoming call is waiting during an ongoing call. In that event
     ,
 * the HFP_SUBEVENT_CALL_WAITING_NOTIFICATION is emitted.
 *
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_activate_call_waiting_notification(hci_con_handle_t
    acl_handle);

/*
 * @brief Disable the   Call Waiting notification  function in
     the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_deactivate_call_waiting_notification(hci_con_handle_t
    acl_handle);

/*
 * @brief Enable the   Calling Line Identification notification
     function in the AG.
 * The AG shall issue the corresponding result code just after every
     RING indication,
 * when the HF is alerted in an incoming call. In that event,
 * the HFP_SUBEVENT_CALLING_LINE_INDETIFICATION_NOTIFICATION is
     emitted.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_activate_calling_line_notification(hci_con_handle_t
    acl_handle);

/*
 * @brief Disable the   Calling Line Identification notification
     function in the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_deactivate_calling_line_notification(hci_con_handle_t
    acl_handle);


/*
 * @brief Activate echo canceling and noise reduction in the AG. By
     default,
 * if the AG supports its own embedded echo canceling and/or noise
     reduction
 * functions, it shall have them activated until this function is
     called.
 * If the AG does not support any echo canceling and noise reduction
     functions,
 * it shall respond with the ERROR indicator (TODO)
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_activate_echo_canceling_and_noise_reduction(
    hci_con_handle_t acl_handle);

/*
 * @brief Deactivate echo canceling and noise reduction in the AG.
 */
void hfp_hf_deactivate_echo_canceling_and_noise_reduction(
    hci_con_handle_t acl_handle);

/*
 * @brief Activate voice recognition function.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_activate_voice_recognition_notification(hci_con_handle_t
     acl_handle);

/*
 * @brief Dectivate voice recognition function.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_deactivate_voice_recognition_notification(
    hci_con_handle_t acl_handle);

/*
 * @brief Set microphone gain.
 * @param bd_addr Bluetooth address of the AG
 * @param gain Valid range: [0,15]
 */
void hfp_hf_set_microphone_gain(hci_con_handle_t acl_handle, int
    gain);

/*
 * @brief Set speaker gain.
 * @param bd_addr Bluetooth address of the AG
 * @param gain Valid range: [0,15]
 */
void hfp_hf_set_speaker_gain(hci_con_handle_t acl_handle, int gain);

/*
 * @brief Instruct the AG to transmit a DTMF code.
 * @param bd_addr Bluetooth address of the AG
 * @param dtmf_code
 */
void hfp_hf_send_dtmf_code(hci_con_handle_t acl_handle, char code);

/*
 * @brief Read numbers from the AG for the purpose of creating
 * a unique voice tag and storing the number and its linked voice
 * tag in the HF s memory.
 * The number is reported via HFP_SUBEVENT_NUMBER_FOR_VOICE_TAG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_request_phone_number_for_voice_tag(hci_con_handle_t
    acl_handle);

/*
 * @brief Query the list of current calls in AG.
 * The result is received via HFP_SUBEVENT_ENHANCED_CALL_STATUS.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_query_current_call_status(hci_con_handle_t acl_handle);

/*
 * @brief Release a call with index in the AG.
 * @param bd_addr Bluetooth address of the AG
 * @param index
 */
void hfp_hf_release_call_with_index(hci_con_handle_t acl_handle, int
     index);

/*
 * @brief Place all parties of a multiparty call on hold with the
 * exception of the specified call.
 * @param bd_addr Bluetooth address of the AG
 * @param index
 */
void hfp_hf_private_consultation_with_call(hci_con_handle_t
    acl_handle, int index);

/*
 * @brief Query the status of the  Response and Hold  state of
     the AG.
 * The result is reported via HFP_SUBEVENT_RESPONSE_AND_HOLD_STATUS.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_rrh_query_status(hci_con_handle_t acl_handle);

/*
 * @brief Put an incoming call on hold in the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_rrh_hold_call(hci_con_handle_t acl_handle);

/*
 * @brief Accept held incoming call in the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_rrh_accept_held_call(hci_con_handle_t acl_handle);

/*
 * @brief Reject held incoming call in the AG.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_rrh_reject_held_call(hci_con_handle_t acl_handle);
```

```
/*
 * @brief Query the AG subscriber number.
 * The result is reported via
     HFP_SUBEVENT_SUBSCRIBER_NUMBER_INFORMATION.
 * @param bd_addr Bluetooth address of the AG
 */
void hfp_hf_query_subscriber_number(hci_con_handle_t acl_handle);

/*
 * @brief Set HF indicator.
 * @param bd_addr Bluetooth address of the AG
 * @param assigned_number
 * @param value
 */
void hfp_hf_set_hf_indicator(hci_con_handle_t acl_handle, int
    assigned_number, int value);
```

19.12. **HFP Audio Gateway API.**

```
typedef struct {
    uint8_t type;
    const char * number;
} hfp_phone_number_t;

/**
 * @brief Create HFP Audio Gateway (AG) SDP service record.
 * @param service
 * @param rfcomm_channel_nr
 * @param name
 * @param ability_to_reject_call
 * @param suported_features 32-bit bitmap, see HFP_AGSF_* values in
     hfp.h
 * @param wide_band_speech supported
 */
void hfp_ag_create_sdp_record(uint8_t * service, uint32_t
    service_record_handle, int rfcomm_channel_nr, const char * name,
     uint8_t ability_to_reject_call, uint16_t supported_features,
    int wide_band_speech);

/**
 * @brief Set up HFP Audio Gateway (AG) device without additional
     supported features.
 * @param rfcomm_channel_nr
 */
void hfp_ag_init(uint16_t rfcomm_channel_nr);

/**
 * @brief Set codecs.
 * @param codecs_nr
 * @param codecs
 */
void hfp_ag_init_codecs(int codecs_nr, uint8_t * codecs);
```

```
/**
 * @brief Set supported features.
 * @param supported_features 32-bit bitmap, see HFP_AGSF_* values in
     hfp.h
 */
void hfp_ag_init_supported_features(uint32_t supported_features);

/**
 * @brief Set AG indicators.
 * @param indicators_nr
 * @param indicators
 */
void hfp_ag_init_ag_indicators(int ag_indicators_nr,
   hfp_ag_indicator_t * ag_indicators);

/**
 * @brief Set HF indicators.
 * @param indicators_nr
 * @param indicators
 */
void hfp_ag_init_hf_indicators(int hf_indicators_nr,
   hfp_generic_status_indicator_t * hf_indicators);

/**
 * @brief Set Call Hold services.
 * @param indicators_nr
 * @param indicators
 */
void hfp_ag_init_call_hold_services(int call_hold_services_nr, const
    char * call_hold_services[]);


/**
 * @brief Register callback for the HFP Audio Gateway (AG) client.
 * @param callback
 */
void hfp_ag_register_packet_handler(btstack_packet_handler_t
   callback);

/**
 * @brief Enable in-band ring tone.
 * @param use_in_band_ring_tone
 */
void hfp_ag_set_use_in_band_ring_tone(int use_in_band_ring_tone);


// actions used by local device / user

/**
 * @brief Establish RFCOMM connection, and perform service level
    connection agreement:
 * - exchange of supported features
 * - report Audio Gateway (AG) indicators and their status
 * - enable indicator status update in the AG
 * - accept the information about available codecs in the Hands-Free
      (HF), if sent
 * - report own information describing the call hold and multiparty
     services, if possible
 * - report which HF indicators are enabled on the AG, if possible
 * The status of SLC connection establishment is reported via
 * HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_ESTABLISHED.
 *
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_establish_service_level_connection(bd_addr_t bd_addr);

/**
 * @brief Release the RFCOMM channel and the audio connection
     between the HF and the AG.
 * If the audio connection exists, it will be released.
 * The status of releasing the SLC connection is reported via
 * HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_RELEASED.
 *
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_release_service_level_connection(hci_con_handle_t
    acl_handle);

/**
 * @brief Establish audio connection.
 * The status of Audio connection establishment is reported via is
     reported via
 * HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE.
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_establish_audio_connection(hci_con_handle_t acl_handle);

/**
 * @brief Release audio connection.
 * The status of releasing the Audio connection is reported via is
     reported via
 * HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE.
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_release_audio_connection(hci_con_handle_t acl_handle);

/**
 * @brief Put the current call on hold, if it exists, and accept
     incoming call.
 */
void hfp_ag_answer_incoming_call(void);

/**
 * @brief Join held call with active call.
 */
void hfp_ag_join_held_call(void);

/**
```

```
 * @brief Reject incoming call, if exists, or terminate active call.
 */
void hfp_ag_terminate_call(void);

/*
 * @brief Put incoming call on hold.
 */
void hfp_ag_hold_incoming_call(void);

/*
 * @brief Accept the held incoming call.
 */
void hfp_ag_accept_held_incoming_call(void);

/*
 * @brief Reject the held incoming call.
 */
void hfp_ag_reject_held_incoming_call(void);

/*
 * @brief Set microphone gain.
 * @param bd_addr Bluetooth address of the HF
 * @param gain Valid range: [0,15]
 */
void hfp_ag_set_microphone_gain(hci_con_handle_t acl_handle, int
    gain);

/*
 * @brief Set speaker gain.
 * @param bd_addr Bluetooth address of the HF
 * @param gain Valid range: [0,15]
 */
void hfp_ag_set_speaker_gain(hci_con_handle_t acl_handle, int gain);

/*
 * @brief Set battery level.
 * @param level Valid range: [0,5]
 */
void hfp_ag_set_battery_level(int level);

/*
 * @brief Clear last dialed number.
 */
void hfp_ag_clear_last_dialed_number(void);

/*
 * @brief Notify the HF that an incoming call is waiting
 * during an ongoing call. The notification will be sent only if the
      HF has
 * has previously enabled the "Call Waiting notification" in the AG.
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_notify_incoming_call_waiting(hci_con_handle_t acl_handle
    );
```

```
// Voice Recognition

/*
 * @brief Activate voice recognition.
 * @param bd_addr Bluetooth address of the HF
 * @param activate
 */
void hfp_ag_activate_voice_recognition(hci_con_handle_t acl_handle,
    int activate);

/*
 * @brief Send a phone number back to the HF.
 * @param bd_addr Bluetooth address of the HF
 * @param phone_number
 */
void hfp_ag_send_phone_number_for_voice_tag(hci_con_handle_t
    acl_handle, const char * phone_number);

/*
 * @brief Reject sending a phone number to the HF.
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_reject_phone_number_for_voice_tag(hci_con_handle_t
    acl_handle);

/**
 * @brief Store phone number with initiated call.
 * @param type
 * @param number
 */
void hfp_ag_set_clip(uint8_t type, const char * number);


// Cellular Actions

/**
 * @brief Pass the accept incoming call event to the AG.
 */
void hfp_ag_incoming_call(void);

/**
 * @brief Pass the reject outgoing call event to the AG.
 */
void hfp_ag_outgoing_call_rejected(void);

/**
 * @brief Pass the accept outgoing call event to the AG.
 */
void hfp_ag_outgoing_call_accepted(void);

/**
 * @brief Pass the outgoing call ringing event to the AG.
 */
```

```
void hfp_ag_outgoing_call_ringing(void);

/**
 * @brief Pass the outgoing call established event to the AG.
 */
void hfp_ag_outgoing_call_established(void);

/**
 * @brief Pass the call droped event to the AG.
 */
void hfp_ag_call_dropped(void);

/*
 * @brief Set network registration status.
 * @param status 0 - not registered, 1 - registered
 */
void hfp_ag_set_registration_status(int status);

/*
 * @brief Set network signal strength.
 * @param strength [0-5]
 */
void hfp_ag_set_signal_strength(int strength);

/*
 * @brief Set roaming status.
 * @param status 0 - no roaming, 1 - roaming active
 */
void hfp_ag_set_roaming_status(int status);

/*
 * @brief Set subcriber number information, e.g. the phone number
 * @param numbers
 * @param numbers_count
 */
void hfp_ag_set_subcriber_number_information(hfp_phone_number_t *
   numbers, int numbers_count);

/*
 * @brief Called by cellular unit after a DTMF code was transmitted,
     so that the next one can be emitted.
 * @param bd_addr Bluetooth address of the HF
 */
void hfp_ag_send_dtmf_code_done(hci_con_handle_t acl_handle);

/**
 * @brief Report Extended Audio Gateway Error result codes in the AG
    .
 * Whenever there is an error relating to the functionality of the
    AG as a
 * result of AT command, the AG shall send +CME ERROR:
 * - +CME ERROR: 0  - AG failure
 * - +CME ERROR: 1  - no connection to phone
 * - +CME ERROR: 3  - operation not allowed
 * - +CME ERROR: 4  - operation not supported
 * - +CME ERROR: 5  - PH-SIM PIN required
 * - +CME ERROR: 10 - SIM not inserted
 * - +CME ERROR: 11 - SIM PIN required
 * - +CME ERROR: 12 - SIM PUK required
 * - +CME ERROR: 13 - SIM failure
 * - +CME ERROR: 14 - SIM busy
 * - +CME ERROR: 16 - incorrect password
 * - +CME ERROR: 17 - SIM PIN2 required
 * - +CME ERROR: 18 - SIM PUK2 required
 * - +CME ERROR: 20 - memory full
 * - +CME ERROR: 21 - invalid index
 * - +CME ERROR: 23 - memory failure
 * - +CME ERROR: 24 - text string too long
 * - +CME ERROR: 25 - invalid characters in text string
 * - +CME ERROR: 26 - dial string too long
 * - +CME ERROR: 27 - invalid characters in dial string
 * - +CME ERROR: 30 - no network service
 * - +CME ERROR: 31 - network Timeout.
 * - +CME ERROR: 32 - network not allowed     Emergency calls only
 *
 * @param bd_addr Bluetooth address of the HF
 * @param error
 */
void hfp_ag_report_extended_audio_gateway_error_result_code(
    hci_con_handle_t acl_handle, hfp_cme_error_t error);
```

19.13. **PAN API.**

```
/**
 * @brief Creates SDP record for PANU BNEP service in provided empty
     buffer.
 * @note Make sure the buffer is big enough.
 *
 * @param service is an empty buffer to store service record
 * @param service_record_handle for new service
 * @param network_packet_types array of types terminated by a 0x0000
     entry
 * @param name if NULL, the default service name will be assigned
 * @param description if NULL, the default service description will
    be assigned
 * @param security_desc
 */
void pan_create_panu_sdp_record(uint8_t *service, uint32_t
    service_record_handle, uint16_t * network_packet_types, const
    char *name,
     const char *description, security_description_t security_desc);

/**
 * @brief Creates SDP record for GN BNEP service in provided empty
    buffer.
 * @note Make sure the buffer is big enough.
 *
 * @param service is an empty buffer to store service record
 * @param service_record_handle for new service
 * @param network_packet_types array of types terminated by a 0x0000
     entry
 * @param name if NULL, the default service name will be assigned
 * @param description if NULL, the default service description will
    be assigned
 * @param security_desc
 * @param IPv4Subnet is optional subnet definition, e.g.
    "10.0.0.0/8"
 * @param IPv6Subnet is optional subnet definition given in the
    standard IETF format with the absolute attribute IDs
 */
void pan_create_gn_sdp_service(uint8_t *service, uint32_t
    service_record_handle, uint16_t * network_packet_types, const
    char *name,
     const char *description, security_description_t security_desc,
        const char *IPv4Subnet,
     const char *IPv6Subnet);

/**
 * @brief Creates SDP record for NAP BNEP service in provided empty
    buffer.
 * @note Make sure the buffer is big enough.
 *
 * @param service is an empty buffer to store service record
 * @param service_record_handle for new service
 * @param name if NULL, the default service name will be assigned
 * @param network_packet_types array of types terminated by a 0x0000
     entry
 * @param description if NULL, the default service description will
    be assigned
 * @param security_desc
 * @param net_access_type type of available network access
 * @param max_net_access_rate based on net_access_type measured in
    byte/s
 * @param IPv4Subnet is optional subnet definition, e.g.
    "10.0.0.0/8"
 * @param IPv6Subnet is optional subnet definition given in the
    standard IETF format with the absolute attribute IDs
 */
void pan_create_nap_sdp_record(uint8_t *service, uint32_t
    service_record_handle, uint16_t * network_packet_types, const
    char *name,
     const char *description, security_description_t security_desc,
        net_access_type_t net_access_type,
     uint32_t max_net_access_rate, const char *IPv4Subnet, const char
         *IPv6Subnet);
```

19.14. RFCOMM API.

```
/**
 * @brief Set up RFCOMM.
 */
void rfcomm_init(void);

/**
 * @brief Set security level required for incoming connections, need
      to be called before registering services.
 */
void rfcomm_set_required_security_level(gap_security_level_t
    security_level);

/*
 * @brief Create RFCOMM connection to a given server channel on a
     remote deivce.
 * This channel will automatically provide enough credits to the
     remote side.
 * @param addr
 * @param server_channel
 * @param out_cid
 * @result status
 */
uint8_t rfcomm_create_channel(btstack_packet_handler_t
    packet_handler, bd_addr_t addr, uint8_t server_channel, uint16_t
     * out_cid);

/*
 * @brief Create RFCOMM connection to a given server channel on a
     remote deivce.
 * This channel will use explicit credit management. During channel
     establishment, an initial  amount of credits is provided.
 * @param addr
 * @param server_channel
 * @param initial_credits
 * @param out_cid
 * @result status
 */
uint8_t rfcomm_create_channel_with_initial_credits(
    btstack_packet_handler_t packet_handler, bd_addr_t addr, uint8_t
     server_channel, uint8_t initial_credits, uint16_t * out_cid);

/**
 * @brief Disconnects RFCOMM channel with given identifier.
 */
void rfcomm_disconnect(uint16_t rfcomm_cid);

/**
 * @brief Registers RFCOMM service for a server channel and a
     maximum frame size, and assigns a packet handler.
 * This channel provides credits automatically to the remote side ->
      no flow control
 * @param packet handler for all channels of this service
 * @param channel
 * @param max_frame_size
 */
```

```
uint8_t rfcomm_register_service(btstack_packet_handler_t
    packet_handler, uint8_t channel, uint16_t max_frame_size);

/**
 * @brief Registers RFCOMM service for a server channel and a
     maximum frame size, and assigns a packet handler.
 * This channel will use explicit credit management. During channel
     establishment, an initial amount of credits is provided.
 * @param packet handler for all channels of this service
 * @param channel
 * @param max_frame_size
 * @param initial_credits
 */
uint8_t rfcomm_register_service_with_initial_credits(
    btstack_packet_handler_t packet_handler, uint8_t channel,
    uint16_t max_frame_size, uint8_t initial_credits);

/**
 * @brief Unregister RFCOMM service.
 */
void rfcomm_unregister_service(uint8_t service_channel);

/**
 * @brief Accepts incoming RFCOMM connection.
 */
void rfcomm_accept_connection(uint16_t rfcomm_cid);

/**
 * @brief Deny incoming RFCOMM connection.
 */
void rfcomm_decline_connection(uint16_t rfcomm_cid);

/**
 * @brief Grant more incoming credits to the remote side for the
     given RFCOMM channel identifier.
 */
void rfcomm_grant_credits(uint16_t rfcomm_cid, uint8_t credits);

/**
 * @brief Checks if RFCOMM can send packet.
 * @param rfcomm_cid
 * @result != 0 if can send now
 */
int rfcomm_can_send_packet_now(uint16_t rfcomm_cid);

/**
 * @brief Request emission of RFCOMM_EVENT_CAN_SEND_NOW as soon as
     possible
 * @note RFCOMM_EVENT_CAN_SEND_NOW might be emitted during call to
     this function
 *       so packet handler should be ready to handle it
 * @param rfcomm_cid
 */
void rfcomm_request_can_send_now_event(uint16_t rfcomm_cid);
```

```
/**
 * @brief Sends RFCOMM data packet to the RFCOMM channel with given
    identifier.
 * @param rfcomm_cid
 */
int  rfcomm_send(uint16_t rfcomm_cid, uint8_t *data, uint16_t len);

/**
 * @brief Sends Local Line Status, see LINE_STATUS_..
 * @param rfcomm_cid
 * @param line_status
 */
int rfcomm_send_local_line_status(uint16_t rfcomm_cid, uint8_t
   line_status);

/**
 * @brief Send local modem status. see MODEM_STAUS_..
 * @param rfcomm_cid
 * @param modem_status
 */
int rfcomm_send_modem_status(uint16_t rfcomm_cid, uint8_t
   modem_status);

/**
 * @brief Configure remote port
 * @param rfcomm_cid
 * @param baud_rate
 * @param data_bits
 * @param stop_bits
 * @param parity
 * @param flow_control
 */
int rfcomm_send_port_configuration(uint16_t rfcomm_cid, rpn_baud_t
   baud_rate, rpn_data_bits_t data_bits, rpn_stop_bits_t stop_bits,
    rpn_parity_t parity, rpn_flow_control_t flow_control);

/**
 * @brief Query remote port
 * @param rfcomm_cid
 */
int rfcomm_query_port_configuration(uint16_t rfcomm_cid);

/**
 * @brief Query max frame size
 * @param rfcomm_cid
 */
uint16_t  rfcomm_get_max_frame_size(uint16_t rfcomm_cid);

/**
 * @brief Allow to create RFCOMM packet in outgoing buffer.
 * if (rfcomm_can_send_packet_now(cid)){
 *     rfcomm_reserve_packet_buffer();
 *     uint8_t * buffer = rfcomm_get_outgoing_buffer();
 *      uint16_t buffer_size = rfcomm_get_max_frame_size(cid);
 *      // .. setup data in buffer with len
 *      rfcomm_send_prepared(cid, len)
 * }
 */
int        rfcomm_reserve_packet_buffer(void);
uint8_t *  rfcomm_get_outgoing_buffer(void);
int        rfcomm_send_prepared(uint16_t rfcomm_cid, uint16_t len);
void       rfcomm_release_packet_buffer(void);

/**
 * CRC8 functions using ETSI TS 101 369 V6.3.0.
 * Only used by RFCOMM
 */
uint8_t crc8_check(uint8_t *data, uint16_t len, uint8_t check_sum);
uint8_t crc8_calc(uint8_t *data, uint16_t len);
```

19.15. SDP Client API.

```
typedef struct de_state {
    uint8_t  in_state_GET_DE_HEADER_LENGTH ;
    uint32_t addon_header_bytes;
    uint32_t de_size;
    uint32_t de_offset;
} de_state_t;

void de_state_init(de_state_t * state);
int  de_state_size(uint8_t eventByte, de_state_t *de_state);

/**
 * @brief Checks if the SDP Client is ready
 * @return 1 when no query is active
 */
int sdp_client_ready(void);

/**
 * @brief Queries the SDP service of the remote device given a
     service search pattern and a list of attribute IDs.
 * The remote data is handled by the SDP parser. The SDP parser
     delivers attribute values and done event via the callback.
 * @param callback for attributes values and done event
 * @param remote address
 * @param des_service_search_pattern
 * @param des_attribute_id_list
 */
uint8_t sdp_client_query(btstack_packet_handler_t callback,
   bd_addr_t remote, const uint8_t * des_service_search_pattern,
   const uint8_t * des_attribute_id_list);

/*
 * @brief Searches SDP records on a remote device for all services
     with a given UUID.
 * @note calls sdp_client_query with service search pattern based on
      uuid16
 */
uint8_t sdp_client_query_uuid16(btstack_packet_handler_t callback,
   bd_addr_t remote, uint16_t uuid16);

/*
 * @brief Searches SDP records on a remote device for all services
     with a given UUID.
 * @note calls sdp_client_query with service search pattern based on
      uuid128
 */
uint8_t sdp_client_query_uuid128(btstack_packet_handler_t callback,
   bd_addr_t remote, const uint8_t* uuid128);


/**
 * @brief Retrieves all attribute IDs of a SDP record specified by
     its service record handle and a list of attribute IDs.
 * The remote data is handled by the SDP parser. The SDP parser
     delivers attribute values and done event via the callback.
 * @note only provided if ENABLE_SDP_EXTRA_QUERIES is defined
 * @param callback for attributes values and done event
 * @param remote address
 * @param search_service_record_handle
 * @param des_attributeIDList
 */
uint8_t sdp_client_service_attribute_search(btstack_packet_handler_t
     callback, bd_addr_t remote, uint32_t
   search_service_record_handle, const uint8_t *
   des_attributeIDList);

/**
 * @brief Query the list of SDP records that match a given service
     search pattern.
 * The remote data is handled by the SDP parser. The SDP parser
     delivers attribute values and done event via the callback.
 * @note only provided if ENABLE_SDP_EXTRA_QUERIES is defined
 * @param callback for attributes values and done event
 * @param remote address
 * @param des_service_search_pattern
 */
uint8_t sdp_client_service_search(btstack_packet_handler_t callback,
     bd_addr_t remote, const uint8_t * des_service_search_pattern);

#ifdef ENABLE_SDP_EXTRA_QUERIES
void sdp_client_parse_service_record_handle_list(uint8_t* packet,
   uint16_t total_count, uint16_t current_count);
#endif
```

19.16. SDP RFCOMM Query API.

```
/**
```

```
 * @brief Searches SDP records on a remote device for RFCOMM
     services with a given 16-bit UUID.
 * @note calls sdp_service_search_pattern_for_uuid16 that uses
     global buffer
 */
uint8_t sdp_client_query_rfcomm_channel_and_name_for_uuid(
    btstack_packet_handler_t callback, bd_addr_t remote, uint16_t
    uuid);

/**
 * @brief Searches SDP records on a remote device for RFCOMM
     services with a given 128-bit UUID.
 * @note calls sdp_service_search_pattern_for_uuid128 that uses
     global buffer
 */
uint8_t sdp_client_query_rfcomm_channel_and_name_for_uuid128(
    btstack_packet_handler_t callback, bd_addr_t remote, const
    uint8_t * uuid128);

/**
 * @brief Searches SDP records on a remote device for RFCOMM
     services with a given service search pattern.
 */
uint8_t sdp_client_query_rfcomm_channel_and_name_for_search_pattern(
    btstack_packet_handler_t callback, bd_addr_t remote, const
    uint8_t * des_serviceSearchPattern);
```

19.17. SDP Server API.

```
/**
 * @brief Set up SDP.
 */
void sdp_init(void);

/**
 * @brief Register Service Record with database using
     ServiceRecordHandle stored in record
 * @pre AttributeIDs are in ascending order
 * @pre ServiceRecordHandle is first attribute and valid
 * @param record is not copied!
 * @result status
 */
uint8_t sdp_register_service(const uint8_t * record);

/**
 * @brief Unregister service record internally.
 */
void sdp_unregister_service(uint32_t service_record_handle);
```

19.18. SDP Utils API.

```
typedef enum {
    DE_NIL = 0,
    DE_UINT,
    DE_INT,
    DE_UUID,
    DE_STRING,
    DE_BOOL,
    DE_DES,
    DE_DEA,
    DE_URL
} de_type_t;

typedef enum {
    DE_SIZE_8 = 0,
    DE_SIZE_16,
    DE_SIZE_32,
    DE_SIZE_64,
    DE_SIZE_128,
    DE_SIZE_VAR_8,
    DE_SIZE_VAR_16,
    DE_SIZE_VAR_32
} de_size_t;

// MARK: DateElement
void       de_dump_data_element(const uint8_t * record);
int        de_get_len(const uint8_t * header);
de_size_t  de_get_size_type(const uint8_t * header);
de_type_t  de_get_element_type(const uint8_t * header);
int        de_get_header_size(const uint8_t * header);
int        de_element_get_uint16(const uint8_t * element, uint16_t *
    value);
int        de_get_data_size(const uint8_t * header);
uint32_t   de_get_uuid32(const uint8_t * element);
int        de_get_normalized_uuid(uint8_t *uuid128, const uint8_t *
    element);

void       de_create_sequence(uint8_t * header);
void       de_store_descriptor_with_len(uint8_t * header, de_type_t
    type, de_size_t size, uint32_t len);
uint8_t * de_push_sequence(uint8_t *header);
void       de_pop_sequence(uint8_t * parent, uint8_t * child);
void       de_add_number(uint8_t *seq, de_type_t type, de_size_t size
    , uint32_t value);
void       de_add_data( uint8_t *seq, de_type_t type, uint16_t size,
    uint8_t *data);

void       de_add_uuid128(uint8_t * seq, uint8_t * uuid);

// MARK: DES iterator
typedef struct {
    uint8_t * element;
    uint16_t pos;
    uint16_t length;
} des_iterator_t;
```

```
int des_iterator_init(des_iterator_t * it, uint8_t * element);
int  des_iterator_has_more(des_iterator_t * it);
de_type_t des_iterator_get_type (des_iterator_t * it);
uint16_t des_iterator_get_size (des_iterator_t * it);
uint8_t * des_iterator_get_element(des_iterator_t * it);
void des_iterator_next(des_iterator_t * it);

// MARK: SDP
uint16_t  sdp_append_attributes_in_attributeIDList(uint8_t *record,
    uint8_t *attributeIDList, uint16_t startOffset, uint16_t
    maxBytes, uint8_t *buffer);
uint8_t * sdp_get_attribute_value_for_attribute_id(uint8_t * record,
     uint16_t attributeID);
uint8_t   sdp_set_attribute_value_for_attribute_id(uint8_t * record,
     uint16_t attributeID, uint32_t value);
int       sdp_record_matches_service_search_pattern(uint8_t *record,
     uint8_t *serviceSearchPattern);
int       spd_get_filtered_size(uint8_t *record, uint8_t *
    attributeIDList);
int       sdp_filter_attributes_in_attributeIDList(uint8_t *record,
    uint8_t *attributeIDList, uint16_t startOffset, uint16_t
    maxBytes, uint16_t *usedBytes, uint8_t *buffer);
int       sdp_attribute_list_constains_id(uint8_t *attributeIDList,
    uint16_t attributeID);
int       sdp_traversal_match_pattern(uint8_t * element, de_type_t
    attributeType, de_size_t size, void *my_context);

/*
 * @brief Returns service search pattern for given UUID-16
 * @note Uses fixed buffer
 */
uint8_t* sdp_service_search_pattern_for_uuid16(uint16_t uuid16);

/*
 * @brief Returns service search pattern for given UUID-128
 * @note Uses fixed buffer
 */
uint8_t* sdp_service_search_pattern_for_uuid128(const uint8_t *
    uuid128);
```

19.19. BLE Advertisements Parser API.

```
typedef struct ad_context {
     const uint8_t * data;
     uint8_t    offset;
     uint8_t    length;
} ad_context_t;

// Advertising or Scan Response data iterator
void ad_iterator_init(ad_context_t *context, uint8_t ad_len, const
    uint8_t * ad_data);
int  ad_iterator_has_more(const ad_context_t * context);
void ad_iterator_next(ad_context_t * context);
```

```
// Access functions
uint8_t         ad_iterator_get_data_type(const ad_context_t *
   context);
uint8_t         ad_iterator_get_data_len(const ad_context_t *
   context);
const uint8_t * ad_iterator_get_data(const ad_context_t * context);

// convenience function on complete advertisements
int ad_data_contains_uuid16(uint8_t ad_len, const uint8_t * ad_data,
    uint16_t uuid);
int ad_data_contains_uuid128(uint8_t ad_len, const uint8_t * ad_data
   , const uint8_t * uuid128);
```

19.20. BTstack Chipset API.

19.21. BTstack Hardware Control API.

19.22. HCI Event Getter API.

```
/**
 * @brief Get event type
 * @param event
 * @return type of event
 */
static inline uint8_t hci_event_packet_get_type(const uint8_t *
   event){
    return event[0];
}

/***
 * @brief Get subevent code for ancs event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_ancs_meta_get_subevent_code(const
   uint8_t * event){
    return event[2];
}
/***
 * @brief Get subevent code for avdtp event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_avdtp_meta_get_subevent_code(const
   uint8_t * event){
    return event[2];
}
/***
 * @brief Get subevent code for a2dp event
 * @param event packet
 * @return subevent_code
 */
```

```
static inline uint8_t hci_event_a2dp_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for avrcp event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_avrcp_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for goep event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_goep_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for hfp event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_hfp_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for hsp event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_hsp_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for pbap event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_pbap_meta_get_subevent_code(const
    uint8_t * event){
     return event[2];
}
/***
 * @brief Get subevent code for le event
 * @param event packet
 * @return subevent_code
 */
```

```
static inline uint8_t hci_event_le_meta_get_subevent_code(const
    uint8_t * event){
    return event[2];
}
/***
 * @brief Get subevent code for hid event
 * @param event packet
 * @return subevent_code
 */
static inline uint8_t hci_event_hid_meta_get_subevent_code(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field status from event HCI_EVENT_INQUIRY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_inquiry_complete_get_status(const
    uint8_t * event){
    return event[2];
}

/**
 * @brief Get field num_responses from event
     HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return num_responses
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_inquiry_result_get_num_responses(
    const uint8_t * event){
    return event[2];
}
/**
 * @brief Get field bd_addr from event HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_inquiry_result_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field page_scan_repetition_mode from event
     HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return page_scan_repetition_mode
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    hci_event_inquiry_result_get_page_scan_repetition_mode(const
    uint8_t * event){
     return event[9];
}
/**
 * @brief Get field reserved1 from event HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return reserved1
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_inquiry_result_get_reserved1(const
    uint8_t * event){
     return event[10];
}
/**
 * @brief Get field reserved2 from event HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return reserved2
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_inquiry_result_get_reserved2(const
    uint8_t * event){
     return event[11];
}
/**
 * @brief Get field class_of_device from event
     HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return class_of_device
 * @note: btstack_type 3
 */
static inline uint32_t hci_event_inquiry_result_get_class_of_device(
    const uint8_t * event){
     return little_endian_read_24(event, 12);
}
/**
 * @brief Get field clock_offset from event HCI_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return clock_offset
 * @note: btstack_type 2
 */
static inline uint16_t hci_event_inquiry_result_get_clock_offset(
    const uint8_t * event){
     return little_endian_read_16(event, 15);
}

/**
 * @brief Get field status from event HCI_EVENT_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
```

```
static inline uint8_t hci_event_connection_complete_get_status(const
     uint8_t * event){
    return event[2];
}
/**
 * @brief Get field connection_handle from event
     HCI_EVENT_CONNECTION_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_connection_complete_get_connection_handle(const
    uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event HCI_EVENT_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_connection_complete_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[5], bd_addr);
}
/**
 * @brief Get field link_type from event
     HCI_EVENT_CONNECTION_COMPLETE
 * @param event packet
 * @return link_type
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_connection_complete_get_link_type(
    const uint8_t * event){
    return event[11];
}
/**
 * @brief Get field encryption_enabled from event
     HCI_EVENT_CONNECTION_COMPLETE
 * @param event packet
 * @return encryption_enabled
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_connection_complete_get_encryption_enabled(const
    uint8_t * event){
    return event[12];
}

/**
 * @brief Get field bd_addr from event HCI_EVENT_CONNECTION_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_connection_request_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[2], bd_addr);
}
/**
 * @brief Get field class_of_device from event
     HCI_EVENT_CONNECTION_REQUEST
 * @param event packet
 * @return class_of_device
 * @note: btstack_type 3
 */
static inline uint32_t
    hci_event_connection_request_get_class_of_device(const uint8_t *
     event){
    return little_endian_read_24(event, 8);
}
/**
 * @brief Get field link_type from event
     HCI_EVENT_CONNECTION_REQUEST
 * @param event packet
 * @return link_type
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_connection_request_get_link_type(
    const uint8_t * event){
    return event[11];
}

/**
 * @brief Get field status from event
     HCI_EVENT_DISCONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_disconnection_complete_get_status(
    const uint8_t * event){
    return event[2];
}
/**
 * @brief Get field connection_handle from event
     HCI_EVENT_DISCONNECTION_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_disconnection_complete_get_connection_handle(const
    uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
```

```
 * @brief Get field reason from event
     HCI_EVENT_DISCONNECTION_COMPLETE
 * @param event packet
 * @return reason
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_disconnection_complete_get_reason(
    const uint8_t * event){
     return event[5];
}

/**
 * @brief Get field status from event
     HCI_EVENT_AUTHENTICATION_COMPLETE_EVENT
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_authentication_complete_event_get_status(const uint8_t
     * event){
     return event[2];
}
/**
 * @brief Get field connection_handle from event
     HCI_EVENT_AUTHENTICATION_COMPLETE_EVENT
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_authentication_complete_event_get_connection_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}

/**
 * @brief Get field status from event
     HCI_EVENT_REMOTE_NAME_REQUEST_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_remote_name_request_complete_get_status(const uint8_t
    * event){
     return event[2];
}
/**
 * @brief Get field bd_addr from event
     HCI_EVENT_REMOTE_NAME_REQUEST_COMPLETE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
   hci_event_remote_name_request_complete_get_bd_addr(const uint8_t
    * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field remote_name from event
    HCI_EVENT_REMOTE_NAME_REQUEST_COMPLETE
 * @param event packet
 * @return remote_name
 * @note: btstack_type N
 */
static inline const char *
   hci_event_remote_name_request_complete_get_remote_name(const
   uint8_t * event){
    return (const char *) &event[9];
}

/**
 * @brief Get field status from event HCI_EVENT_ENCRYPTION_CHANGE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_encryption_change_get_status(const
   uint8_t * event){
    return event[2];
}
/**
 * @brief Get field connection_handle from event
    HCI_EVENT_ENCRYPTION_CHANGE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_event_encryption_change_get_connection_handle(const uint8_t
   * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field encryption_enabled from event
    HCI_EVENT_ENCRYPTION_CHANGE
 * @param event packet
 * @return encryption_enabled
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_event_encryption_change_get_encryption_enabled(const uint8_t
    * event){
    return event[5];
}
```

```
/**
 * @brief Get field status from event
     HCI_EVENT_CHANGE_CONNECTION_LINK_KEY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_change_connection_link_key_complete_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field connection_handle from event
     HCI_EVENT_CHANGE_CONNECTION_LINK_KEY_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_change_connection_link_key_complete_get_connection_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}

/**
 * @brief Get field status from event
     HCI_EVENT_MASTER_LINK_KEY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_master_link_key_complete_get_status(
    const uint8_t * event){
     return event[2];
}
/**
 * @brief Get field connection_handle from event
     HCI_EVENT_MASTER_LINK_KEY_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_master_link_key_complete_get_connection_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field key_flag from event
     HCI_EVENT_MASTER_LINK_KEY_COMPLETE
 * @param event packet
 * @return key_flag
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_master_link_key_complete_get_key_flag(const uint8_t *
    event){
    return event[5];
}

/**
 * @brief Get field num_hci_command_packets from event
     HCI_EVENT_COMMAND_COMPLETE
 * @param event packet
 * @return num_hci_command_packets
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_command_complete_get_num_hci_command_packets(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field command_opcode from event
     HCI_EVENT_COMMAND_COMPLETE
 * @param event packet
 * @return command_opcode
 * @note: btstack_type 2
 */
static inline uint16_t hci_event_command_complete_get_command_opcode
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field return_parameters from event
     HCI_EVENT_COMMAND_COMPLETE
 * @param event packet
 * @return return_parameters
 * @note: btstack_type R
 */
static inline const uint8_t *
    hci_event_command_complete_get_return_parameters(const uint8_t *
     event){
    return &event[5];
}

/**
 * @brief Get field status from event HCI_EVENT_COMMAND_STATUS
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_command_status_get_status(const
    uint8_t * event){
    return event[2];
}
/**
```

```
 * @brief Get field num_hci_command_packets from event
     HCI_EVENT_COMMAND_STATUS
 * @param event packet
 * @return num_hci_command_packets
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_command_status_get_num_hci_command_packets(const
    uint8_t * event){
    return event[3];
}
/**
 * @brief Get field command_opcode from event
     HCI_EVENT_COMMAND_STATUS
 * @param event packet
 * @return command_opcode
 * @note: btstack_type 2
 */
static inline uint16_t hci_event_command_status_get_command_opcode(
    const uint8_t * event){
    return little_endian_read_16(event, 4);
}

/**
 * @brief Get field hardware_code from event
     HCI_EVENT_HARDWARE_ERROR
 * @param event packet
 * @return hardware_code
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_hardware_error_get_hardware_code(
    const uint8_t * event){
    return event[2];
}

/**
 * @brief Get field status from event HCI_EVENT_ROLE_CHANGE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_role_change_get_status(const uint8_t
     * event){
    return event[2];
}
/**
 * @brief Get field bd_addr from event HCI_EVENT_ROLE_CHANGE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_role_change_get_bd_addr(const uint8_t *
     event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field role from event HCI_EVENT_ROLE_CHANGE
 * @param event packet
 * @return role
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_role_change_get_role(const uint8_t *
     event){
    return event[9];
}

/**
 * @brief Get field status from event HCI_EVENT_MODE_CHANGE_EVENT
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_mode_change_event_get_status(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field handle from event HCI_EVENT_MODE_CHANGE_EVENT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_event_mode_change_event_get_handle(const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field mode from event HCI_EVENT_MODE_CHANGE_EVENT
 * @param event packet
 * @return mode
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_mode_change_event_get_mode(const
    uint8_t * event){
    return event[5];
}
/**
 * @brief Get field interval from event HCI_EVENT_MODE_CHANGE_EVENT
 * @param event packet
 * @return interval
 * @note: btstack_type 2
 */
static inline uint16_t hci_event_mode_change_event_get_interval(
    const uint8_t * event){
    return little_endian_read_16(event, 6);
}

/**
```

```
 * @brief Get field bd_addr from event HCI_EVENT_PIN_CODE_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_pin_code_request_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}

/**
 * @brief Get field bd_addr from event HCI_EVENT_LINK_KEY_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_link_key_request_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}

/**
 * @brief Get field link_type from event
     HCI_EVENT_DATA_BUFFER_OVERFLOW
 * @param event packet
 * @return link_type
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_data_buffer_overflow_get_link_type(
    const uint8_t * event){
     return event[2];
}

/**
 * @brief Get field handle from event HCI_EVENT_MAX_SLOTS_CHANGED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_event_max_slots_changed_get_handle(const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field lmp_max_slots from event
     HCI_EVENT_MAX_SLOTS_CHANGED
 * @param event packet
 * @return lmp_max_slots
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_max_slots_changed_get_lmp_max_slots(
    const uint8_t * event){
     return event[4];
}
```

```
/**
 * @brief Get field status from event
     HCI_EVENT_READ_CLOCK_OFFSET_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_read_clock_offset_complete_get_status(const uint8_t *
    event){
     return event[2];
}
/**
 * @brief Get field handle from event
     HCI_EVENT_READ_CLOCK_OFFSET_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_event_read_clock_offset_complete_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field clock_offset from event
     HCI_EVENT_READ_CLOCK_OFFSET_COMPLETE
 * @param event packet
 * @return clock_offset
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_read_clock_offset_complete_get_clock_offset(const
    uint8_t * event){
     return little_endian_read_16(event, 5);
}

/**
 * @brief Get field status from event
     HCI_EVENT_CONNECTION_PACKET_TYPE_CHANGED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_connection_packet_type_changed_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field handle from event
     HCI_EVENT_CONNECTION_PACKET_TYPE_CHANGED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_event_connection_packet_type_changed_get_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field packet_types from event
     HCI_EVENT_CONNECTION_PACKET_TYPE_CHANGED
 * @param event packet
 * @return packet_types
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_event_connection_packet_type_changed_get_packet_types(const
    uint8_t * event){
     return little_endian_read_16(event, 5);
}

/**
 * @brief Get field num_responses from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return num_responses
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_inquiry_result_with_rssi_get_num_responses(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field bd_addr from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_inquiry_result_with_rssi_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field page_scan_repetition_mode from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return page_scan_repetition_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_inquiry_result_with_rssi_get_page_scan_repetition_mode
    (const uint8_t * event){
```

```
    return event[9];
}
/**
 * @brief Get field reserved from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return reserved
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_event_inquiry_result_with_rssi_get_reserved(const uint8_t *
   event){
    return event[10];
}
/**
 * @brief Get field class_of_device from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return class_of_device
 * @note: btstack_type 3
 */
static inline uint32_t
   hci_event_inquiry_result_with_rssi_get_class_of_device(const
   uint8_t * event){
    return little_endian_read_24(event, 11);
}
/**
 * @brief Get field clock_offset from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return clock_offset
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_event_inquiry_result_with_rssi_get_clock_offset(const
   uint8_t * event){
    return little_endian_read_16(event, 14);
}
/**
 * @brief Get field rssi from event
     HCI_EVENT_INQUIRY_RESULT_WITH_RSSI
 * @param event packet
 * @return rssi
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_inquiry_result_with_rssi_get_rssi(
   const uint8_t * event){
    return event[16];
}

/**
 * @brief Get field status from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_synchronous_connection_complete_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field handle from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_event_synchronous_connection_complete_get_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
    hci_event_synchronous_connection_complete_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[5], bd_addr);
}
/**
 * @brief Get field link_type from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return link_type
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_synchronous_connection_complete_get_link_type(const
    uint8_t * event){
     return event[11];
}
/**
 * @brief Get field transmission_interval from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return transmission_interval
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_synchronous_connection_complete_get_transmission_interval
    (const uint8_t * event){
```

```
    return event[12];
}
/**
 * @brief Get field retransmission_interval from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return retransmission_interval
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_event_synchronous_connection_complete_get_retransmission_interval
   (const uint8_t * event){
    return event[13];
}
/**
 * @brief Get field rx_packet_length from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return rx_packet_length
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_event_synchronous_connection_complete_get_rx_packet_length(
   const uint8_t * event){
    return little_endian_read_16(event, 14);
}
/**
 * @brief Get field tx_packet_length from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return tx_packet_length
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_event_synchronous_connection_complete_get_tx_packet_length(
   const uint8_t * event){
    return little_endian_read_16(event, 16);
}
/**
 * @brief Get field air_mode from event
     HCI_EVENT_SYNCHRONOUS_CONNECTION_COMPLETE
 * @param event packet
 * @return air_mode
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_event_synchronous_connection_complete_get_air_mode(const
   uint8_t * event){
    return event[18];
}

/**
 * @brief Get field num_responses from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return num_responses
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_extended_inquiry_response_get_num_responses(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field bd_addr from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_extended_inquiry_response_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field page_scan_repetition_mode from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return page_scan_repetition_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_extended_inquiry_response_get_page_scan_repetition_mode
    (const uint8_t * event){
    return event[9];
}
/**
 * @brief Get field reserved from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return reserved
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_event_extended_inquiry_response_get_reserved(const uint8_t *
     event){
    return event[10];
}
/**
 * @brief Get field class_of_device from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return class_of_device
 * @note: btstack_type 3
 */
static inline uint32_t
    hci_event_extended_inquiry_response_get_class_of_device(const
    uint8_t * event){
```

```
    return little_endian_read_24(event, 11);
}
/**
 * @brief Get field clock_offset from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return clock_offset
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_event_extended_inquiry_response_get_clock_offset(const
   uint8_t * event){
    return little_endian_read_16(event, 14);
}
/**
 * @brief Get field rssi from event
     HCI_EVENT_EXTENDED_INQUIRY_RESPONSE
 * @param event packet
 * @return rssi
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_extended_inquiry_response_get_rssi(
   const uint8_t * event){
    return event[16];
}

/**
 * @brief Get field status from event
     HCI_EVENT_ENCRYPTION_KEY_REFRESH_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_event_encryption_key_refresh_complete_get_status(const
   uint8_t * event){
    return event[2];
}
/**
 * @brief Get field handle from event
     HCI_EVENT_ENCRYPTION_KEY_REFRESH_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   hci_event_encryption_key_refresh_complete_get_handle(const
   uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field bd_addr from event
     HCI_EVENT_USER_CONFIRMATION_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_user_confirmation_request_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}
/**
 * @brief Get field numeric_value from event
     HCI_EVENT_USER_CONFIRMATION_REQUEST
 * @param event packet
 * @return numeric_value
 * @note: btstack_type 4
 */
static inline uint32_t
    hci_event_user_confirmation_request_get_numeric_value(const
    uint8_t * event){
     return little_endian_read_32(event, 8);
}

/**
 * @brief Get field bd_addr from event
     HCI_EVENT_USER_PASSKEY_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_user_passkey_request_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}

/**
 * @brief Get field bd_addr from event
     HCI_EVENT_REMOTE_OOB_DATA_REQUEST
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_remote_oob_data_request_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}

/**
 * @brief Get field status from event
     HCI_EVENT_SIMPLE_PAIRING_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_simple_pairing_complete_get_status(
    const uint8_t * event){
```

```
    return event[2];
}
/**
 * @brief Get field bd_addr from event
     HCI_EVENT_SIMPLE_PAIRING_COMPLETE
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void hci_event_simple_pairing_complete_get_bd_addr(
   const uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[3], bd_addr);
}

/**
 * @brief Get field state from event BTSTACK_EVENT_STATE
 * @param event packet
 * @return state
 * @note: btstack_type 1
 */
static inline uint8_t btstack_event_state_get_state(const uint8_t *
   event){
    return event[2];
}

/**
 * @brief Get field number_connections from event
     BTSTACK_EVENT_NR_CONNECTIONS_CHANGED
 * @param event packet
 * @return number_connections
 * @note: btstack_type 1
 */
static inline uint8_t
   btstack_event_nr_connections_changed_get_number_connections(
   const uint8_t * event){
    return event[2];
}



/**
 * @brief Get field discoverable from event
     BTSTACK_EVENT_DISCOVERABLE_ENABLED
 * @param event packet
 * @return discoverable
 * @note: btstack_type 1
 */
static inline uint8_t
   btstack_event_discoverable_enabled_get_discoverable(const
   uint8_t * event){
    return event[2];
}

/**
 * @brief Get field active from event HCI_EVENT_TRANSPORT_SLEEP_MODE
 * @param event packet
 * @return active
 * @note: btstack_type 1
 */
static inline uint8_t hci_event_transport_sleep_mode_get_active(
    const uint8_t * event){
     return event[2];
}

/**
 * @brief Get field handle from event HCI_EVENT_SCO_CAN_SEND_NOW
 * @param event packet
 * @param Pointer to storage for handle
 * @note: btstack_type B
 */
static inline void hci_event_sco_can_send_now_get_handle(const
    uint8_t * event, bd_addr_t handle){
     reverse_bd_addr(&event[2], handle);
}

/**
 * @brief Get field status from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t l2cap_event_channel_opened_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field address from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void l2cap_event_channel_opened_get_address(const
    uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[3], address);
}
/**
 * @brief Get field handle from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t l2cap_event_channel_opened_get_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 9);
}
/**
 * @brief Get field psm from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return psm
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_opened_get_psm(const
   uint8_t * event){
    return little_endian_read_16(event, 11);
}
/**
 * @brief Get field local_cid from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_opened_get_local_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 13);
}
/**
 * @brief Get field remote_cid from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return remote_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_opened_get_remote_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 15);
}
/**
 * @brief Get field local_mtu from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return local_mtu
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_opened_get_local_mtu(
   const uint8_t * event){
    return little_endian_read_16(event, 17);
}
/**
 * @brief Get field remote_mtu from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return remote_mtu
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_opened_get_remote_mtu(
   const uint8_t * event){
    return little_endian_read_16(event, 19);
}
/**
 * @brief Get field flush_timeout from event
     L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return flush_timeout
 * @note: btstack_type 2
 */
```

```
static inline uint16_t l2cap_event_channel_opened_get_flush_timeout(
    const uint8_t * event){
    return little_endian_read_16(event, 21);
}
/**
 * @brief Get field incoming from event L2CAP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t l2cap_event_channel_opened_get_incoming(const
    uint8_t * event){
    return event[23];
}

/**
 * @brief Get field local_cid from event L2CAP_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_channel_closed_get_local_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 2);
}

/**
 * @brief Get field address from event
     L2CAP_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void l2cap_event_incoming_connection_get_address(const
     uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[2], address);
}
/**
 * @brief Get field handle from event
     L2CAP_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    l2cap_event_incoming_connection_get_handle(const uint8_t * event
    ){
    return little_endian_read_16(event, 8);
}
/**
 * @brief Get field psm from event L2CAP_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return psm
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_incoming_connection_get_psm(const
     uint8_t * event){
    return little_endian_read_16(event, 10);
}
/**
 * @brief Get field local_cid from event
     L2CAP_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_incoming_connection_get_local_cid
   (const uint8_t * event){
    return little_endian_read_16(event, 12);
}
/**
 * @brief Get field remote_cid from event
     L2CAP_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return remote_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   l2cap_event_incoming_connection_get_remote_cid(const uint8_t *
   event){
    return little_endian_read_16(event, 14);
}

/**
 * @brief Get field handle from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_REQUEST
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   l2cap_event_connection_parameter_update_request_get_handle(const
    uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field interval_min from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_REQUEST
 * @param event packet
 * @return interval_min
 * @note: btstack_type 2
 */
static inline uint16_t
   l2cap_event_connection_parameter_update_request_get_interval_min
   (const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
```

```
 * @brief Get field interval_max from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_REQUEST
 * @param event packet
 * @return interval_max
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_connection_parameter_update_request_get_interval_max
    (const uint8_t * event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field latencey from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_REQUEST
 * @param event packet
 * @return latencey
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_connection_parameter_update_request_get_latencey(
    const uint8_t * event){
     return little_endian_read_16(event, 8);
}
/**
 * @brief Get field timeout_multiplier from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_REQUEST
 * @param event packet
 * @return timeout_multiplier
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_connection_parameter_update_request_get_timeout_multiplier
    (const uint8_t * event){
     return little_endian_read_16(event, 10);
}

/**
 * @brief Get field handle from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_RESPONSE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    l2cap_event_connection_parameter_update_response_get_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field result from event
     L2CAP_EVENT_CONNECTION_PARAMETER_UPDATE_RESPONSE
 * @param event packet
 * @return result
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_connection_parameter_update_response_get_result(
    const uint8_t * event){
     return little_endian_read_16(event, 4);
}

/**
 * @brief Get field local_cid from event L2CAP_EVENT_CAN_SEND_NOW
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_can_send_now_get_local_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 2);
}

/**
 * @brief Get field address_type from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return address_type
 * @note: btstack_type 1
 */
static inline uint8_t
    l2cap_event_le_incoming_connection_get_address_type(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field address from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void l2cap_event_le_incoming_connection_get_address(
    const uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[3], address);
}
/**
 * @brief Get field handle from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    l2cap_event_le_incoming_connection_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 9);
}
/**
```

```
 * @brief Get field psm from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return psm
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_incoming_connection_get_psm(
    const uint8_t * event){
    return little_endian_read_16(event, 11);
}
/**
 * @brief Get field local_cid from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_le_incoming_connection_get_local_cid(const uint8_t *
     event){
    return little_endian_read_16(event, 13);
}
/**
 * @brief Get field remote_cid from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return remote_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_le_incoming_connection_get_remote_cid(const uint8_t
    * event){
    return little_endian_read_16(event, 15);
}
/**
 * @brief Get field remote_mtu from event
     L2CAP_EVENT_LE_INCOMING_CONNECTION
 * @param event packet
 * @return remote_mtu
 * @note: btstack_type 2
 */
static inline uint16_t
    l2cap_event_le_incoming_connection_get_remote_mtu(const uint8_t
    * event){
    return little_endian_read_16(event, 17);
}

/**
 * @brief Get field status from event L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
```

```
static inline uint8_t l2cap_event_le_channel_opened_get_status(const
     uint8_t * event){
    return event[2];
}
/**
 * @brief Get field address_type from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return address_type
 * @note: btstack_type 1
 */
static inline uint8_t l2cap_event_le_channel_opened_get_address_type
   (const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field address from event L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void l2cap_event_le_channel_opened_get_address(const
   uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[4], address);
}
/**
 * @brief Get field handle from event L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   l2cap_event_le_channel_opened_get_handle(const uint8_t * event){
    return little_endian_read_16(event, 10);
}
/**
 * @brief Get field incoming from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t l2cap_event_le_channel_opened_get_incoming(
   const uint8_t * event){
    return event[12];
}
/**
 * @brief Get field psm from event L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return psm
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_channel_opened_get_psm(const
   uint8_t * event){
    return little_endian_read_16(event, 13);
}
/**
 * @brief Get field local_cid from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_channel_opened_get_local_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 15);
}
/**
 * @brief Get field remote_cid from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return remote_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_channel_opened_get_remote_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 17);
}
/**
 * @brief Get field local_mtu from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return local_mtu
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_channel_opened_get_local_mtu(
   const uint8_t * event){
    return little_endian_read_16(event, 19);
}
/**
 * @brief Get field remote_mtu from event
     L2CAP_EVENT_LE_CHANNEL_OPENED
 * @param event packet
 * @return remote_mtu
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_channel_opened_get_remote_mtu(
   const uint8_t * event){
    return little_endian_read_16(event, 21);
}

/**
 * @brief Get field local_cid from event
     L2CAP_EVENT_LE_CHANNEL_CLOSED
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
```

```
static inline uint16_t l2cap_event_le_channel_closed_get_local_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}

/**
 * @brief Get field local_cid from event L2CAP_EVENT_LE_CAN_SEND_NOW
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_can_send_now_get_local_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}

/**
 * @brief Get field local_cid from event L2CAP_EVENT_LE_PACKET_SENT
 * @param event packet
 * @return local_cid
 * @note: btstack_type 2
 */
static inline uint16_t l2cap_event_le_packet_sent_get_local_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}

/**
 * @brief Get field status from event RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t rfcomm_event_channel_opened_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field bd_addr from event RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void rfcomm_event_channel_opened_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[3], bd_addr);
}
/**
 * @brief Get field con_handle from event
     RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return con_handle
 * @note: btstack_type 2
 */
```

```
static inline uint16_t rfcomm_event_channel_opened_get_con_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 9);
}
/**
 * @brief Get field server_channel from event
      RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return server_channel
 * @note: btstack_type 1
 */
static inline uint8_t rfcomm_event_channel_opened_get_server_channel
    (const uint8_t * event){
     return event[11];
}
/**
 * @brief Get field rfcomm_cid from event
      RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t rfcomm_event_channel_opened_get_rfcomm_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 12);
}
/**
 * @brief Get field max_frame_size from event
      RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return max_frame_size
 * @note: btstack_type 2
 */
static inline uint16_t
    rfcomm_event_channel_opened_get_max_frame_size(const uint8_t *
    event){
     return little_endian_read_16(event, 14);
}
/**
 * @brief Get field incoming from event RFCOMM_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t rfcomm_event_channel_opened_get_incoming(const
     uint8_t * event){
     return event[16];
}

/**
 * @brief Get field rfcomm_cid from event
      RFCOMM_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t rfcomm_event_channel_closed_get_rfcomm_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 2);
}

/**
 * @brief Get field bd_addr from event
     RFCOMM_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void rfcomm_event_incoming_connection_get_bd_addr(
   const uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[2], bd_addr);
}
/**
 * @brief Get field server_channel from event
     RFCOMM_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return server_channel
 * @note: btstack_type 1
 */
static inline uint8_t
   rfcomm_event_incoming_connection_get_server_channel(const
   uint8_t * event){
    return event[8];
}
/**
 * @brief Get field rfcomm_cid from event
     RFCOMM_EVENT_INCOMING_CONNECTION
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   rfcomm_event_incoming_connection_get_rfcomm_cid(const uint8_t *
   event){
    return little_endian_read_16(event, 9);
}

/**
 * @brief Get field rfcomm_cid from event
     RFCOMM_EVENT_REMOTE_LINE_STATUS
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   rfcomm_event_remote_line_status_get_rfcomm_cid(const uint8_t *
   event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field line_status from event
     RFCOMM_EVENT_REMOTE_LINE_STATUS
 * @param event packet
 * @return line_status
 * @note: btstack_type 1
 */
static inline uint8_t
   rfcomm_event_remote_line_status_get_line_status(const uint8_t *
   event){
    return event[4];
}

/**
 * @brief Get field rfcomm_cid from event
     RFCOMM_EVENT_REMOTE_MODEM_STATUS
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   rfcomm_event_remote_modem_status_get_rfcomm_cid(const uint8_t *
   event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field modem_status from event
     RFCOMM_EVENT_REMOTE_MODEM_STATUS
 * @param event packet
 * @return modem_status
 * @note: btstack_type 1
 */
static inline uint8_t
   rfcomm_event_remote_modem_status_get_modem_status(const uint8_t
   * event){
    return event[4];
}

/**
 * @brief Get field rfcomm_cid from event RFCOMM_EVENT_CAN_SEND_NOW
 * @param event packet
 * @return rfcomm_cid
 * @note: btstack_type 2
 */
static inline uint16_t rfcomm_event_can_send_now_get_rfcomm_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 2);
}

/**
 * @brief Get field status from event SDP_EVENT_QUERY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t sdp_event_query_complete_get_status(const
   uint8_t * event){
    return event[2];
}

/**
 * @brief Get field rfcomm_channel from event
     SDP_EVENT_QUERY_RFCOMM_SERVICE
 * @param event packet
 * @return rfcomm_channel
 * @note: btstack_type 1
 */
static inline uint8_t
   sdp_event_query_rfcomm_service_get_rfcomm_channel(const uint8_t
   * event){
    return event[2];
}
/**
 * @brief Get field name from event SDP_EVENT_QUERY_RFCOMM_SERVICE
 * @param event packet
 * @return name
 * @note: btstack_type T
 */
static inline const char * sdp_event_query_rfcomm_service_get_name(
   const uint8_t * event){
    return (const char *) &event[3];
}

/**
 * @brief Get field record_id from event
     SDP_EVENT_QUERY_ATTRIBUTE_BYTE
 * @param event packet
 * @return record_id
 * @note: btstack_type 2
 */
static inline uint16_t sdp_event_query_attribute_byte_get_record_id(
   const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field attribute_id from event
     SDP_EVENT_QUERY_ATTRIBUTE_BYTE
 * @param event packet
 * @return attribute_id
 * @note: btstack_type 2
 */
static inline uint16_t
   sdp_event_query_attribute_byte_get_attribute_id(const uint8_t *
   event){
    return little_endian_read_16(event, 4);
}
/**
```

```
 * @brief Get field attribute_length from event
     SDP_EVENT_QUERY_ATTRIBUTE_BYTE
 * @param event packet
 * @return attribute_length
 * @note: btstack_type 2
 */
static inline uint16_t
    sdp_event_query_attribute_byte_get_attribute_length(const
    uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field data_offset from event
     SDP_EVENT_QUERY_ATTRIBUTE_BYTE
 * @param event packet
 * @return data_offset
 * @note: btstack_type 2
 */
static inline uint16_t
    sdp_event_query_attribute_byte_get_data_offset(const uint8_t *
    event){
    return little_endian_read_16(event, 8);
}
/**
 * @brief Get field data from event SDP_EVENT_QUERY_ATTRIBUTE_BYTE
 * @param event packet
 * @return data
 * @note: btstack_type 1
 */
static inline uint8_t sdp_event_query_attribute_byte_get_data(const
    uint8_t * event){
    return event[10];
}

/**
 * @brief Get field record_id from event
     SDP_EVENT_QUERY_ATTRIBUTE_VALUE
 * @param event packet
 * @return record_id
 * @note: btstack_type 2
 */
static inline uint16_t sdp_event_query_attribute_value_get_record_id
    (const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field attribute_id from event
     SDP_EVENT_QUERY_ATTRIBUTE_VALUE
 * @param event packet
 * @return attribute_id
 * @note: btstack_type 2
 */
```

```
static inline uint16_t
    sdp_event_query_attribute_value_get_attribute_id(const uint8_t *
     event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field attribute_length from event
     SDP_EVENT_QUERY_ATTRIBUTE_VALUE
 * @param event packet
 * @return attribute_length
 * @note: btstack_type L
 */
static inline int
    sdp_event_query_attribute_value_get_attribute_length(const
    uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field attribute_value from event
     SDP_EVENT_QUERY_ATTRIBUTE_VALUE
 * @param event packet
 * @return attribute_value
 * @note: btstack_type V
 */
static inline const uint8_t *
    sdp_event_query_attribute_value_get_attribute_value(const
    uint8_t * event){
    return &event[8];
}

/**
 * @brief Get field total_count from event
     SDP_EVENT_QUERY_SERVICE_RECORD_HANDLE
 * @param event packet
 * @return total_count
 * @note: btstack_type 2
 */
static inline uint16_t
    sdp_event_query_service_record_handle_get_total_count(const
    uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field record_index from event
     SDP_EVENT_QUERY_SERVICE_RECORD_HANDLE
 * @param event packet
 * @return record_index
 * @note: btstack_type 2
 */
static inline uint16_t
    sdp_event_query_service_record_handle_get_record_index(const
    uint8_t * event){
    return little_endian_read_16(event, 4);
}
```

```
/**
 * @brief Get field record_handle from event
     SDP_EVENT_QUERY_SERVICE_RECORD_HANDLE
 * @param event packet
 * @return record_handle
 * @note: btstack_type 4
 */
static inline uint32_t
    sdp_event_query_service_record_handle_get_record_handle(const
    uint8_t * event){
    return little_endian_read_32(event, 6);
}

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event GATT_EVENT_QUERY_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t gatt_event_query_complete_get_handle(
    const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field status from event GATT_EVENT_QUERY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t gatt_event_query_complete_get_status(const
    uint8_t * event){
    return event[4];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     GATT_EVENT_SERVICE_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_service_query_result_get_handle(const uint8_t * event
    ){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field service from event
     GATT_EVENT_SERVICE_QUERY_RESULT
 * @param event packet
 * @param Pointer to storage for service
 * @note: btstack_type X
 */
static inline void gatt_event_service_query_result_get_service(const
    uint8_t * event, gatt_client_service_t * service){
    gatt_client_deserialize_service(event, 4, service);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     GATT_EVENT_CHARACTERISTIC_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_characteristic_query_result_get_handle(const uint8_t
    * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field characteristic from event
     GATT_EVENT_CHARACTERISTIC_QUERY_RESULT
 * @param event packet
 * @param Pointer to storage for characteristic
 * @note: btstack_type Y
 */
static inline void
    gatt_event_characteristic_query_result_get_characteristic(const
    uint8_t * event, gatt_client_characteristic_t * characteristic){
    gatt_client_deserialize_characteristic(event, 4, characteristic)
        ;
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_included_service_query_result_get_handle(const
    uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field include_handle from event
     GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT
 * @param event packet
 * @return include_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    gatt_event_included_service_query_result_get_include_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field service from event
      GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT
 * @param event packet
 * @param Pointer to storage for service
 * @note: btstack_type X
 */
static inline void
    gatt_event_included_service_query_result_get_service(const
    uint8_t * event, gatt_client_service_t * service){
     gatt_client_deserialize_service(event, 6, service);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
      GATT_EVENT_ALL_CHARACTERISTIC_DESCRIPTORS_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_all_characteristic_descriptors_query_result_get_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field characteristic_descriptor from event
      GATT_EVENT_ALL_CHARACTERISTIC_DESCRIPTORS_QUERY_RESULT
 * @param event packet
 * @param Pointer to storage for characteristic_descriptor
 * @note: btstack_type Z
 */
static inline void
    gatt_event_all_characteristic_descriptors_query_result_get_characteristic_descrip
    (const uint8_t * event, gatt_client_characteristic_descriptor_t
    * characteristic_descriptor){
     gatt_client_deserialize_characteristic_descriptor(event, 4,
         characteristic_descriptor);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
      GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_characteristic_value_query_result_get_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field value_handle from event
     GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    gatt_event_characteristic_value_query_result_get_value_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field value_length from event
     GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value_length
 * @note: btstack_type L
 */
static inline int
    gatt_event_characteristic_value_query_result_get_value_length(
    const uint8_t * event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field value from event
     GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t *
    gatt_event_characteristic_value_query_result_get_value(const
    uint8_t * event){
     return &event[8];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_long_characteristic_value_query_result_get_handle(
    const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field value_handle from event
     GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    gatt_event_long_characteristic_value_query_result_get_value_handle
    (const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field value_offset from event
     GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value_offset
 * @note: btstack_type 2
 */
static inline uint16_t
    gatt_event_long_characteristic_value_query_result_get_value_offset
    (const uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field value_length from event
     GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value_length
 * @note: btstack_type L
 */
static inline int
    gatt_event_long_characteristic_value_query_result_get_value_length
    (const uint8_t * event){
    return little_endian_read_16(event, 8);
}
/**
 * @brief Get field value from event
     GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t *
    gatt_event_long_characteristic_value_query_result_get_value(
    const uint8_t * event){
    return &event[10];
}
```

```
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event GATT_EVENT_NOTIFICATION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t gatt_event_notification_get_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field value_handle from event GATT_EVENT_NOTIFICATION
 * @param event packet
 * @return value_handle
 * @note: btstack_type 2
 */
static inline uint16_t gatt_event_notification_get_value_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field value_length from event GATT_EVENT_NOTIFICATION
 * @param event packet
 * @return value_length
 * @note: btstack_type L
 */
static inline int gatt_event_notification_get_value_length(const
    uint8_t * event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field value from event GATT_EVENT_NOTIFICATION
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t * gatt_event_notification_get_value(
    const uint8_t * event){
     return &event[8];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event GATT_EVENT_INDICATION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t gatt_event_indication_get_handle(
    const uint8_t * event){
```

```
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field value_handle from event GATT_EVENT_INDICATION
 * @param event packet
 * @return value_handle
 * @note: btstack_type 2
 */
static inline uint16_t gatt_event_indication_get_value_handle(const
    uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field value_length from event GATT_EVENT_INDICATION
 * @param event packet
 * @return value_length
 * @note: btstack_type L
 */
static inline int gatt_event_indication_get_value_length(const
    uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field value from event GATT_EVENT_INDICATION
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t * gatt_event_indication_get_value(const
    uint8_t * event){
    return &event[8];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_characteristic_descriptor_query_result_get_handle(
    const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field descriptor_handle from event
     GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor_handle
 * @note: btstack_type 2
 */
```

```
static inline uint16_t
    gatt_event_characteristic_descriptor_query_result_get_descriptor_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field descriptor_length from event
      GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor_length
 * @note: btstack_type L
 */
static inline int
    gatt_event_characteristic_descriptor_query_result_get_descriptor_length
    (const uint8_t * event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field descriptor from event
      GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor
 * @note: btstack_type V
 */
static inline const uint8_t *
    gatt_event_characteristic_descriptor_query_result_get_descriptor
    (const uint8_t * event){
     return &event[8];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
      GATT_EVENT_LONG_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    gatt_event_long_characteristic_descriptor_query_result_get_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field descriptor_offset from event
      GATT_EVENT_LONG_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor_offset
 * @note: btstack_type 2
 */
static inline uint16_t
    gatt_event_long_characteristic_descriptor_query_result_get_descriptor_offset
    (const uint8_t * event){
```

```
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field descriptor_length from event
     GATT_EVENT_LONG_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor_length
 * @note: btstack_type L
 */
static inline int
   gatt_event_long_characteristic_descriptor_query_result_get_descriptor_length
   (const uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field descriptor from event
     GATT_EVENT_LONG_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT
 * @param event packet
 * @return descriptor
 * @note: btstack_type V
 */
static inline const uint8_t *
   gatt_event_long_characteristic_descriptor_query_result_get_descriptor
   (const uint8_t * event){
    return &event[8];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event GATT_EVENT_MTU
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t gatt_event_mtu_get_handle(const
   uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field MTU from event GATT_EVENT_MTU
 * @param event packet
 * @return MTU
 * @note: btstack_type 2
 */
static inline uint16_t gatt_event_mtu_get_MTU(const uint8_t * event)
   {
    return little_endian_read_16(event, 4);
}
#endif

/**
 * @brief Get field handle from event
     ATT_EVENT_MTU_EXCHANGE_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    att_event_mtu_exchange_complete_get_handle(const uint8_t * event
    ){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field MTU from event ATT_EVENT_MTU_EXCHANGE_COMPLETE
 * @param event packet
 * @return MTU
 * @note: btstack_type 2
 */
static inline uint16_t att_event_mtu_exchange_complete_get_MTU(const
     uint8_t * event){
     return little_endian_read_16(event, 4);
}

/**
 * @brief Get field status from event
     ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    att_event_handle_value_indication_complete_get_status(const
    uint8_t * event){
     return event[2];
}
/**
 * @brief Get field conn_handle from event
     ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE
 * @param event packet
 * @return conn_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    att_event_handle_value_indication_complete_get_conn_handle(const
     uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field attribute_handle from event
     ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE
 * @param event packet
 * @return attribute_handle
 * @note: btstack_type 2
 */
static inline uint16_t
    att_event_handle_value_indication_complete_get_attribute_handle(
    const uint8_t * event){
```

```
    return little_endian_read_16(event, 5);
}


/**
 * @brief Get field status from event BNEP_EVENT_SERVICE_REGISTERED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t bnep_event_service_registered_get_status(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field service_uuid from event
    BNEP_EVENT_SERVICE_REGISTERED
 * @param event packet
 * @return service_uuid
 * @note: btstack_type 2
 */
static inline uint16_t
   bnep_event_service_registered_get_service_uuid(const uint8_t *
   event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field status from event BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t bnep_event_channel_opened_get_status(const
   uint8_t * event){
    return event[2];
}
/**
 * @brief Get field bnep_cid from event BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return bnep_cid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_channel_opened_get_bnep_cid(const
   uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field source_uuid from event BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return source_uuid
 * @note: btstack_type 2
 */
```

```
static inline uint16_t bnep_event_channel_opened_get_source_uuid(
    const uint8_t * event){
     return little_endian_read_16(event, 5);
}
/**
 * @brief Get field destination_uuid from event
     BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return destination_uuid
 * @note: btstack_type 2
 */
static inline uint16_t
    bnep_event_channel_opened_get_destination_uuid(const uint8_t *
    event){
     return little_endian_read_16(event, 7);
}
/**
 * @brief Get field mtu from event BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @return mtu
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_channel_opened_get_mtu(const
    uint8_t * event){
     return little_endian_read_16(event, 9);
}
/**
 * @brief Get field remote_address from event
     BNEP_EVENT_CHANNEL_OPENED
 * @param event packet
 * @param Pointer to storage for remote_address
 * @note: btstack_type B
 */
static inline void bnep_event_channel_opened_get_remote_address(
    const uint8_t * event, bd_addr_t remote_address){
     reverse_bd_addr(&event[11], remote_address);
}

/**
 * @brief Get field bnep_cid from event BNEP_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @return bnep_cid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_channel_closed_get_bnep_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field source_uuid from event BNEP_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @return source_uuid
 * @note: btstack_type 2
 */
```

```
static inline uint16_t bnep_event_channel_closed_get_source_uuid(
    const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field destination_uuid from event
     BNEP_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @return destination_uuid
 * @note: btstack_type 2
 */
static inline uint16_t
    bnep_event_channel_closed_get_destination_uuid(const uint8_t *
    event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field remote_address from event
     BNEP_EVENT_CHANNEL_CLOSED
 * @param event packet
 * @param Pointer to storage for remote_address
 * @note: btstack_type B
 */
static inline void bnep_event_channel_closed_get_remote_address(
    const uint8_t * event, bd_addr_t remote_address){
    reverse_bd_addr(&event[8], remote_address);
}

/**
 * @brief Get field bnep_cid from event BNEP_EVENT_CHANNEL_TIMEOUT
 * @param event packet
 * @return bnep_cid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_channel_timeout_get_bnep_cid(const
     uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field source_uuid from event
     BNEP_EVENT_CHANNEL_TIMEOUT
 * @param event packet
 * @return source_uuid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_channel_timeout_get_source_uuid(
    const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field destination_uuid from event
     BNEP_EVENT_CHANNEL_TIMEOUT
 * @param event packet
 * @return destination_uuid
 * @note: btstack_type 2
 */
static inline uint16_t
    bnep_event_channel_timeout_get_destination_uuid(const uint8_t *
    event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field remote_address from event
     BNEP_EVENT_CHANNEL_TIMEOUT
 * @param event packet
 * @param Pointer to storage for remote_address
 * @note: btstack_type B
 */
static inline void bnep_event_channel_timeout_get_remote_address(
    const uint8_t * event, bd_addr_t remote_address){
     reverse_bd_addr(&event[8], remote_address);
}
/**
 * @brief Get field channel_state from event
     BNEP_EVENT_CHANNEL_TIMEOUT
 * @param event packet
 * @return channel_state
 * @note: btstack_type 1
 */
static inline uint8_t bnep_event_channel_timeout_get_channel_state(
    const uint8_t * event){
     return event[14];
}

/**
 * @brief Get field bnep_cid from event BNEP_EVENT_CAN_SEND_NOW
 * @param event packet
 * @return bnep_cid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_can_send_now_get_bnep_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field source_uuid from event BNEP_EVENT_CAN_SEND_NOW
 * @param event packet
 * @return source_uuid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_can_send_now_get_source_uuid(const
     uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field destination_uuid from event
     BNEP_EVENT_CAN_SEND_NOW
 * @param event packet
 * @return destination_uuid
 * @note: btstack_type 2
 */
static inline uint16_t bnep_event_can_send_now_get_destination_uuid(
    const uint8_t * event){
     return little_endian_read_16(event, 6);
}
/**
 * @brief Get field remote_address from event
      BNEP_EVENT_CAN_SEND_NOW
 * @param event packet
 * @param Pointer to storage for remote_address
 * @note: btstack_type B
 */
static inline void bnep_event_can_send_now_get_remote_address(const
    uint8_t * event, bd_addr_t remote_address){
     reverse_bd_addr(&event[8], remote_address);
}

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_JUST_WORKS_REQUEST
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_just_works_request_get_handle(const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event SM_EVENT_JUST_WORKS_REQUEST
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_just_works_request_get_addr_type(
    const uint8_t * event){
     return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_JUST_WORKS_REQUEST
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_just_works_request_get_address(const
    uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
```

```
 * @brief Get field handle from event SM_EVENT_JUST_WORKS_CANCEL
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t sm_event_just_works_cancel_get_handle
   (const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event SM_EVENT_JUST_WORKS_CANCEL
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_just_works_cancel_get_addr_type(const
    uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_JUST_WORKS_CANCEL
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_just_works_cancel_get_address(const
   uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_PASSKEY_DISPLAY_NUMBER
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   sm_event_passkey_display_number_get_handle(const uint8_t * event
   ){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_PASSKEY_DISPLAY_NUMBER
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_passkey_display_number_get_addr_type(
   const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_PASSKEY_DISPLAY_NUMBER
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_passkey_display_number_get_address(const
     uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
/**
 * @brief Get field passkey from event
     SM_EVENT_PASSKEY_DISPLAY_NUMBER
 * @param event packet
 * @return passkey
 * @note: btstack_type 4
 */
static inline uint32_t sm_event_passkey_display_number_get_passkey(
    const uint8_t * event){
    return little_endian_read_32(event, 11);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_PASSKEY_DISPLAY_CANCEL
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_passkey_display_cancel_get_handle(const uint8_t * event
    ){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_PASSKEY_DISPLAY_CANCEL
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_passkey_display_cancel_get_addr_type(
    const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_PASSKEY_DISPLAY_CANCEL
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_passkey_display_cancel_get_address(const
     uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_PASSKEY_INPUT_NUMBER
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_passkey_input_number_get_handle(const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_PASSKEY_INPUT_NUMBER
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_passkey_input_number_get_addr_type(
    const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_PASSKEY_INPUT_NUMBER
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_passkey_input_number_get_address(const
    uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_PASSKEY_INPUT_CANCEL
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_passkey_input_cancel_get_handle(const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
```

```
 * @brief Get field addr_type from event
     SM_EVENT_PASSKEY_INPUT_CANCEL
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_passkey_input_cancel_get_addr_type(
    const uint8_t * event){
     return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_PASSKEY_INPUT_CANCEL
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_passkey_input_cancel_get_address(const
    uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_NUMERIC_COMPARISON_REQUEST
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_numeric_comparison_request_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_NUMERIC_COMPARISON_REQUEST
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_numeric_comparison_request_get_addr_type(const uint8_t
    * event){
     return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_NUMERIC_COMPARISON_REQUEST
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
```

```
static inline void sm_event_numeric_comparison_request_get_address(
    const uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
/**
 * @brief Get field passkey from event
     SM_EVENT_NUMERIC_COMPARISON_REQUEST
 * @param event packet
 * @return passkey
 * @note: btstack_type 4
 */
static inline uint32_t
    sm_event_numeric_comparison_request_get_passkey(const uint8_t *
    event){
     return little_endian_read_32(event, 11);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_NUMERIC_COMPARISON_CANCEL
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_numeric_comparison_cancel_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_NUMERIC_COMPARISON_CANCEL
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_numeric_comparison_cancel_get_addr_type(const uint8_t *
     event){
     return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_NUMERIC_COMPARISON_CANCEL
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_numeric_comparison_cancel_get_address(
    const uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
```

```
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_IDENTITY_RESOLVING_STARTED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_identity_resolving_started_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_IDENTITY_RESOLVING_STARTED
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_identity_resolving_started_get_addr_type(const uint8_t
    * event){
     return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_IDENTITY_RESOLVING_STARTED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_identity_resolving_started_get_address(
    const uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_IDENTITY_RESOLVING_FAILED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_identity_resolving_failed_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 2);
}
/**
```

```
 * @brief Get field addr_type from event
     SM_EVENT_IDENTITY_RESOLVING_FAILED
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_identity_resolving_failed_get_addr_type(const uint8_t *
     event){
    return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_IDENTITY_RESOLVING_FAILED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_identity_resolving_failed_get_address(
    const uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_identity_resolving_succeeded_get_handle(const uint8_t *
     event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_identity_resolving_succeeded_get_addr_type(const
    uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_identity_resolving_succeeded_get_address
    (const uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
/**
 * @brief Get field identity_addr_type from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @return identity_addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_identity_resolving_succeeded_get_identity_addr_type(
    const uint8_t * event){
     return event[11];
}
/**
 * @brief Get field identity_address from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @param Pointer to storage for identity_address
 * @note: btstack_type B
 */
static inline void
    sm_event_identity_resolving_succeeded_get_identity_address(const
     uint8_t * event, bd_addr_t identity_address){
     reverse_bd_addr(&event[12], identity_address);
}
/**
 * @brief Get field index_internal from event
     SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED
 * @param event packet
 * @return index_internal
 * @note: btstack_type 2
 */
static inline uint16_t
    sm_event_identity_resolving_succeeded_get_index_internal(const
    uint8_t * event){
     return little_endian_read_16(event, 18);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_AUTHORIZATION_REQUEST
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_authorization_request_get_handle(const uint8_t * event)
    {
```

```
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_AUTHORIZATION_REQUEST
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_authorization_request_get_addr_type(
   const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event
     SM_EVENT_AUTHORIZATION_REQUEST
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_authorization_request_get_address(const
   uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_AUTHORIZATION_RESULT
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   sm_event_authorization_result_get_handle(const uint8_t * event){
    return little_endian_read_16(event, 2);
}
/**
 * @brief Get field addr_type from event
     SM_EVENT_AUTHORIZATION_RESULT
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_authorization_result_get_addr_type(
   const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_AUTHORIZATION_RESULT
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
```

```
static inline void sm_event_authorization_result_get_address(const
    uint8_t * event, bd_addr_t address){
     reverse_bd_addr(&event[5], address);
}
/**
 * @brief Get field authorization_result from event
     SM_EVENT_AUTHORIZATION_RESULT
 * @param event packet
 * @return authorization_result
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_authorization_result_get_authorization_result(const
    uint8_t * event){
     return event[11];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_KEYPRESS_NOTIFICATION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    sm_event_keypress_notification_get_handle(const uint8_t * event)
    {
     return little_endian_read_16(event, 2);
}
/**
 * @brief Get field action from event SM_EVENT_KEYPRESS_NOTIFICATION
 * @param event packet
 * @return action
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_keypress_notification_get_action(
    const uint8_t * event){
     return event[4];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event SM_EVENT_IDENTITY_CREATED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t sm_event_identity_created_get_handle(
    const uint8_t * event){
     return little_endian_read_16(event, 2);
}
/**
```

```
 * @brief Get field addr_type from event SM_EVENT_IDENTITY_CREATED
 * @param event packet
 * @return addr_type
 * @note: btstack_type 1
 */
static inline uint8_t sm_event_identity_created_get_addr_type(const
    uint8_t * event){
    return event[4];
}
/**
 * @brief Get field address from event SM_EVENT_IDENTITY_CREATED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void sm_event_identity_created_get_address(const
    uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[5], address);
}
/**
 * @brief Get field identity_addr_type from event
     SM_EVENT_IDENTITY_CREATED
 * @param event packet
 * @return identity_addr_type
 * @note: btstack_type 1
 */
static inline uint8_t
    sm_event_identity_created_get_identity_addr_type(const uint8_t *
     event){
    return event[11];
}
/**
 * @brief Get field identity_address from event
     SM_EVENT_IDENTITY_CREATED
 * @param event packet
 * @param Pointer to storage for identity_address
 * @note: btstack_type B
 */
static inline void sm_event_identity_created_get_identity_address(
    const uint8_t * event, bd_addr_t identity_address){
    reverse_bd_addr(&event[12], identity_address);
}
#endif

/**
 * @brief Get field handle from event GAP_EVENT_SECURITY_LEVEL
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t gap_event_security_level_get_handle(
    const uint8_t * event){
    return little_endian_read_16(event, 2);
}
```

```
/**
 * @brief Get field security_level from event
     GAP_EVENT_SECURITY_LEVEL
 * @param event packet
 * @return security_level
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_security_level_get_security_level(
    const uint8_t * event){
    return event[4];
}

/**
 * @brief Get field status from event
     GAP_EVENT_DEDICATED_BONDING_COMPLETED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    gap_event_dedicated_bonding_completed_get_status(const uint8_t *
     event){
    return event[2];
}
/**
 * @brief Get field address from event
     GAP_EVENT_DEDICATED_BONDING_COMPLETED
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void gap_event_dedicated_bonding_completed_get_address
    (const uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[3], address);
}

/**
 * @brief Get field advertising_event_type from event
     GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @return advertising_event_type
 * @note: btstack_type 1
 */
static inline uint8_t
    gap_event_advertising_report_get_advertising_event_type(const
    uint8_t * event){
    return event[2];
}
/**
 * @brief Get field address_type from event
     GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @return address_type
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_advertising_report_get_address_type(
    const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field address from event GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @param Pointer to storage for address
 * @note: btstack_type B
 */
static inline void gap_event_advertising_report_get_address(const
    uint8_t * event, bd_addr_t address){
    reverse_bd_addr(&event[4], address);
}
/**
 * @brief Get field rssi from event GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @return rssi
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_advertising_report_get_rssi(const
    uint8_t * event){
    return event[10];
}
/**
 * @brief Get field data_length from event
     GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @return data_length
 * @note: btstack_type J
 */
static inline int gap_event_advertising_report_get_data_length(const
     uint8_t * event){
    return event[11];
}
/**
 * @brief Get field data from event GAP_EVENT_ADVERTISING_REPORT
 * @param event packet
 * @return data
 * @note: btstack_type V
 */
static inline const uint8_t * gap_event_advertising_report_get_data(
    const uint8_t * event){
    return &event[12];
}

/**
 * @brief Get field bd_addr from event GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
```

```
static inline void gap_event_inquiry_result_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[2], bd_addr);
}
/**
 * @brief Get field page_scan_repetition_mode from event
     GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return page_scan_repetition_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    gap_event_inquiry_result_get_page_scan_repetition_mode(const
    uint8_t * event){
     return event[8];
}
/**
 * @brief Get field class_of_device from event
     GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return class_of_device
 * @note: btstack_type 3
 */
static inline uint32_t gap_event_inquiry_result_get_class_of_device(
    const uint8_t * event){
     return little_endian_read_24(event, 9);
}
/**
 * @brief Get field clock_offset from event GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return clock_offset
 * @note: btstack_type 2
 */
static inline uint16_t gap_event_inquiry_result_get_clock_offset(
    const uint8_t * event){
     return little_endian_read_16(event, 12);
}
/**
 * @brief Get field rssi_available from event
     GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return rssi_available
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_inquiry_result_get_rssi_available(
    const uint8_t * event){
     return event[14];
}
/**
 * @brief Get field rssi from event GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return rssi
 * @note: btstack_type 1
 */
```

```
static inline uint8_t gap_event_inquiry_result_get_rssi(const
    uint8_t * event){
    return event[15];
}
/**
 * @brief Get field name_available from event
     GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return name_available
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_inquiry_result_get_name_available(
    const uint8_t * event){
    return event[16];
}
/**
 * @brief Get field name_len from event GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return name_len
 * @note: btstack_type J
 */
static inline int gap_event_inquiry_result_get_name_len(const
    uint8_t * event){
    return event[17];
}
/**
 * @brief Get field name from event GAP_EVENT_INQUIRY_RESULT
 * @param event packet
 * @return name
 * @note: btstack_type V
 */
static inline const uint8_t * gap_event_inquiry_result_get_name(
    const uint8_t * event){
    return &event[18];
}

/**
 * @brief Get field status from event GAP_EVENT_INQUIRY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t gap_event_inquiry_complete_get_status(const
    uint8_t * event){
    return event[2];
}

/**
 * @brief Get field status from event
     HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
```

```
static inline uint8_t hci_subevent_le_connection_complete_get_status
    (const uint8_t * event){
     return event[3];
}
/**
 * @brief Get field connection_handle from event
      HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_subevent_le_connection_complete_get_connection_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field role from event
      HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return role
 * @note: btstack_type 1
 */
static inline uint8_t hci_subevent_le_connection_complete_get_role(
    const uint8_t * event){
     return event[6];
}
/**
 * @brief Get field peer_address_type from event
      HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return peer_address_type
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_subevent_le_connection_complete_get_peer_address_type(const
    uint8_t * event){
     return event[7];
}
/**
 * @brief Get field peer_address from event
      HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for peer_address
 * @note: btstack_type B
 */
static inline void
    hci_subevent_le_connection_complete_get_peer_address(const
    uint8_t * event, bd_addr_t peer_address){
     reverse_bd_addr(&event[8], peer_address);
}
/**
 * @brief Get field conn_interval from event
      HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return conn_interval
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_complete_get_conn_interval(const
    uint8_t * event){
     return little_endian_read_16(event, 14);
}
/**
 * @brief Get field conn_latency from event
     HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return conn_latency
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_complete_get_conn_latency(const
    uint8_t * event){
     return little_endian_read_16(event, 16);
}
/**
 * @brief Get field supervision_timeout from event
     HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return supervision_timeout
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_complete_get_supervision_timeout(
    const uint8_t * event){
     return little_endian_read_16(event, 18);
}
/**
 * @brief Get field master_clock_accuracy from event
     HCI_SUBEVENT_LE_CONNECTION_COMPLETE
 * @param event packet
 * @return master_clock_accuracy
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_subevent_le_connection_complete_get_master_clock_accuracy(
    const uint8_t * event){
     return event[20];
}

/**
 * @brief Get field status from event
     HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    hci_subevent_le_connection_update_complete_get_status(const
    uint8_t * event){
    return event[3];
}
/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_subevent_le_connection_update_complete_get_connection_handle
    (const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field conn_interval from event
     HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE
 * @param event packet
 * @return conn_interval
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_update_complete_get_conn_interval(
    const uint8_t * event){
    return little_endian_read_16(event, 6);
}
/**
 * @brief Get field conn_latency from event
     HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE
 * @param event packet
 * @return conn_latency
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_update_complete_get_conn_latency(
    const uint8_t * event){
    return little_endian_read_16(event, 8);
}
/**
 * @brief Get field supervision_timeout from event
     HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE
 * @param event packet
 * @return supervision_timeout
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_connection_update_complete_get_supervision_timeout
    (const uint8_t * event){
    return little_endian_read_16(event, 10);
}
```

```
/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_READ_REMOTE_USED_FEATURES_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
//  static inline hci_con_handle_t
    hci_subevent_le_read_remote_used_features_complete_get_connection_handle
    (const uint8_t * event){
//      not implemented yet
//  }
/**
 * @brief Get field random_number from event
     HCI_SUBEVENT_LE_READ_REMOTE_USED_FEATURES_COMPLETE
 * @param event packet
 * @return random_number
 * @note: btstack_type D
 */
//  static inline const uint8_t *
    hci_subevent_le_read_remote_used_features_complete_get_random_number
    (const uint8_t * event){
//      not implemented yet
//  }
/**
 * @brief Get field encryption_diversifier from event
     HCI_SUBEVENT_LE_READ_REMOTE_USED_FEATURES_COMPLETE
 * @param event packet
 * @return encryption_diversifier
 * @note: btstack_type 2
 */
//  static inline uint16_t
    hci_subevent_le_read_remote_used_features_complete_get_encryption_diversifier
    (const uint8_t * event){
//      not implemented yet
//  }

/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_LONG_TERM_KEY_REQUEST
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
//  static inline hci_con_handle_t
    hci_subevent_le_long_term_key_request_get_connection_handle(
    const uint8_t * event){
//      not implemented yet
//  }
/**
 * @brief Get field random_number from event
     HCI_SUBEVENT_LE_LONG_TERM_KEY_REQUEST
 * @param event packet
 * @return random_number
 * @note: btstack_type D
 */
//  static inline const uint8_t *
    hci_subevent_le_long_term_key_request_get_random_number(const
    uint8_t * event){
//       not implemented yet
//  }
/**
 * @brief Get field encryption_diversifier from event
     HCI_SUBEVENT_LE_LONG_TERM_KEY_REQUEST
 * @param event packet
 * @return encryption_diversifier
 * @note: btstack_type 2
 */
//  static inline uint16_t
    hci_subevent_le_long_term_key_request_get_encryption_diversifier
    (const uint8_t * event){
//       not implemented yet
//  }

/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hci_subevent_le_remote_connection_parameter_request_get_connection_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field interval_min from event
     HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST
 * @param event packet
 * @return interval_min
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_remote_connection_parameter_request_get_interval_min
    (const uint8_t * event){
     return little_endian_read_16(event, 5);
}
/**
 * @brief Get field interval_max from event
     HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST
 * @param event packet
 * @return interval_max
 * @note: btstack_type 2
 */
static inline uint16_t
    hci_subevent_le_remote_connection_parameter_request_get_interval_max
    (const uint8_t * event){
```

```
    return little_endian_read_16(event, 7);
}
/**
 * @brief Get field latency from event
     HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST
 * @param event packet
 * @return latency
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_remote_connection_parameter_request_get_latency(
   const uint8_t * event){
    return little_endian_read_16(event, 9);
}
/**
 * @brief Get field timeout from event
     HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST
 * @param event packet
 * @return timeout
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_remote_connection_parameter_request_get_timeout(
   const uint8_t * event){
    return little_endian_read_16(event, 11);
}

/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   hci_subevent_le_data_length_change_get_connection_handle(const
   uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field max_tx_octets from event
     HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE
 * @param event packet
 * @return max_tx_octets
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_data_length_change_get_max_tx_octets(const
   uint8_t * event){
    return little_endian_read_16(event, 5);
}
/**
 * @brief Get field max_tx_time from event
     HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE
 * @param event packet
 * @return max_tx_time
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_data_length_change_get_max_tx_time(const uint8_t
    * event){
    return little_endian_read_16(event, 7);
}
/**
 * @brief Get field max_rx_octets from event
    HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE
 * @param event packet
 * @return max_rx_octets
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_data_length_change_get_max_rx_octets(const
   uint8_t * event){
    return little_endian_read_16(event, 9);
}
/**
 * @brief Get field max_rx_time from event
    HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE
 * @param event packet
 * @return max_rx_time
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_data_length_change_get_max_rx_time(const uint8_t
    * event){
    return little_endian_read_16(event, 11);
}

/**
 * @brief Get field status from event
    HCI_SUBEVENT_LE_READ_LOCAL_P256_PUBLIC_KEY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_subevent_le_read_local_p256_public_key_complete_get_status(
   const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field dhkey_x from event
    HCI_SUBEVENT_LE_READ_LOCAL_P256_PUBLIC_KEY_COMPLETE
 * @param event packet
 * @param Pointer to storage for dhkey_x
 * @note: btstack_type Q
 */
```

```
static inline void
    hci_subevent_le_read_local_p256_public_key_complete_get_dhkey_x(
    const uint8_t * event, uint8_t * dhkey_x){
     reverse_bytes(&event[4], dhkey_x, 32);
}
/**
 * @brief Get field dhkey_y from event
     HCI_SUBEVENT_LE_READ_LOCAL_P256_PUBLIC_KEY_COMPLETE
 * @param event packet
 * @param Pointer to storage for dhkey_y
 * @note: btstack_type Q
 */
static inline void
    hci_subevent_le_read_local_p256_public_key_complete_get_dhkey_y(
    const uint8_t * event, uint8_t * dhkey_y){
     reverse_bytes(&event[36], dhkey_y, 32);
}

/**
 * @brief Get field status from event
     HCI_SUBEVENT_LE_GENERATE_DHKEY_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_subevent_le_generate_dhkey_complete_get_status(const uint8_t
     * event){
     return event[3];
}
/**
 * @brief Get field dhkey from event
     HCI_SUBEVENT_LE_GENERATE_DHKEY_COMPLETE
 * @param event packet
 * @param Pointer to storage for dhkey
 * @note: btstack_type Q
 */
static inline void hci_subevent_le_generate_dhkey_complete_get_dhkey
    (const uint8_t * event, uint8_t * dhkey){
     reverse_bytes(&event[4], dhkey, 32);
}

/**
 * @brief Get field status from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hci_subevent_le_enhanced_connection_complete_get_status(const
    uint8_t * event){
     return event[3];
}
```

```
/**
 * @brief Get field connection_handle from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return connection_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
   hci_subevent_le_enhanced_connection_complete_get_connection_handle
   (const uint8_t * event){
    return little_endian_read_16(event, 4);
}
/**
 * @brief Get field role from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return role
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_subevent_le_enhanced_connection_complete_get_role(const
   uint8_t * event){
    return event[6];
}
/**
 * @brief Get field peer_address_type from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return peer_address_type
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_subevent_le_enhanced_connection_complete_get_peer_address_type
   (const uint8_t * event){
    return event[7];
}
/**
 * @brief Get field perr_addresss from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for perr_addresss
 * @note: btstack_type B
 */
static inline void
   hci_subevent_le_enhanced_connection_complete_get_perr_addresss(
   const uint8_t * event, bd_addr_t perr_addresss){
    reverse_bd_addr(&event[8], perr_addresss);
}
/**
 * @brief Get field local_resolvable_private_addres from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for local_resolvable_private_addres
 * @note: btstack_type B
 */
static inline void
   hci_subevent_le_enhanced_connection_complete_get_local_resolvable_private_addres
   (const uint8_t * event, bd_addr_t
   local_resolvable_private_addres){
    reverse_bd_addr(&event[14], local_resolvable_private_addres);
}
/**
 * @brief Get field peer_resolvable_private_addres from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @param Pointer to storage for peer_resolvable_private_addres
 * @note: btstack_type B
 */
static inline void
   hci_subevent_le_enhanced_connection_complete_get_peer_resolvable_private_addres
   (const uint8_t * event, bd_addr_t peer_resolvable_private_addres
   ){
    reverse_bd_addr(&event[20], peer_resolvable_private_addres);
}
/**
 * @brief Get field conn_interval from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return conn_interval
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_enhanced_connection_complete_get_conn_interval(
   const uint8_t * event){
    return little_endian_read_16(event, 26);
}
/**
 * @brief Get field conn_latency from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return conn_latency
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_enhanced_connection_complete_get_conn_latency(
   const uint8_t * event){
    return little_endian_read_16(event, 28);
}
/**
 * @brief Get field supervision_timeout from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return supervision_timeout
 * @note: btstack_type 2
 */
static inline uint16_t
   hci_subevent_le_enhanced_connection_complete_get_supervision_timeout
   (const uint8_t * event){
    return little_endian_read_16(event, 30);
}
/**
 * @brief Get field master_clock_accuracy from event
     HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE
 * @param event packet
 * @return master_clock_accuracy
 * @note: btstack_type 1
 */
static inline uint8_t
   hci_subevent_le_enhanced_connection_complete_get_master_clock_accuracy
   (const uint8_t * event){
    return event[32];
}

/**
 * @brief Get field status from event
     HSP_SUBEVENT_RFCOMM_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
   hsp_subevent_rfcomm_connection_complete_get_status(const uint8_t
    * event){
    return event[3];
}

/**
 * @brief Get field status from event
     HSP_SUBEVENT_RFCOMM_DISCONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
   hsp_subevent_rfcomm_disconnection_complete_get_status(const
   uint8_t * event){
    return event[3];
}

/**
 * @brief Get field status from event
     HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
   hsp_subevent_audio_connection_complete_get_status(const uint8_t
   * event){
    return event[3];
}
/**
```

```
 * @brief Get field handle from event
     HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hsp_subevent_audio_connection_complete_get_handle(const uint8_t
    * event){
    return little_endian_read_16(event, 4);
}

/**
 * @brief Get field status from event
     HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hsp_subevent_audio_disconnection_complete_get_status(const
    uint8_t * event){
    return event[3];
}


/**
 * @brief Get field gain from event
     HSP_SUBEVENT_MICROPHONE_GAIN_CHANGED
 * @param event packet
 * @return gain
 * @note: btstack_type 1
 */
static inline uint8_t hsp_subevent_microphone_gain_changed_get_gain(
    const uint8_t * event){
    return event[3];
}

/**
 * @brief Get field gain from event
     HSP_SUBEVENT_SPEAKER_GAIN_CHANGED
 * @param event packet
 * @return gain
 * @note: btstack_type 1
 */
static inline uint8_t hsp_subevent_speaker_gain_changed_get_gain(
    const uint8_t * event){
    return event[3];
}

/**
 * @brief Get field value_length from event HSP_SUBEVENT_HS_COMMAND
 * @param event packet
 * @return value_length
 * @note: btstack_type J
 */
static inline int hsp_subevent_hs_command_get_value_length(const
    uint8_t * event){
     return event[3];
}
/**
 * @brief Get field value from event HSP_SUBEVENT_HS_COMMAND
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t * hsp_subevent_hs_command_get_value(
    const uint8_t * event){
     return &event[4];
}

/**
 * @brief Get field value_length from event
     HSP_SUBEVENT_AG_INDICATION
 * @param event packet
 * @return value_length
 * @note: btstack_type J
 */
static inline int hsp_subevent_ag_indication_get_value_length(const
    uint8_t * event){
     return event[3];
}
/**
 * @brief Get field value from event HSP_SUBEVENT_AG_INDICATION
 * @param event packet
 * @return value
 * @note: btstack_type V
 */
static inline const uint8_t * hsp_subevent_ag_indication_get_value(
    const uint8_t * event){
     return &event[4];
}

/**
 * @brief Get field status from event
     HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_service_level_connection_established_get_status(
    const uint8_t * event){
     return event[3];
}
/**
 * @brief Get field con_handle from event
     HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_ESTABLISHED
 * @param event packet
 * @return con_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hfp_subevent_service_level_connection_established_get_con_handle
    (const uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field bd_addr from event
     HFP_SUBEVENT_SERVICE_LEVEL_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
    hfp_subevent_service_level_connection_established_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[6], bd_addr);
}


/**
 * @brief Get field status from event
     HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_audio_connection_established_get_status(const
    uint8_t * event){
     return event[3];
}
/**
 * @brief Get field handle from event
     HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hfp_subevent_audio_connection_established_get_handle(const
    uint8_t * event){
     return little_endian_read_16(event, 4);
}
/**
 * @brief Get field bd_addr from event
     HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
```

```
static inline void
    hfp_subevent_audio_connection_established_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[6], bd_addr);
}
/**
 * @brief Get field negotiated_codec from event
     HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED
 * @param event packet
 * @return negotiated_codec
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_audio_connection_established_get_negotiated_codec(
    const uint8_t * event){
    return event[12];
}


/**
 * @brief Get field status from event HFP_SUBEVENT_COMPLETE
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_complete_get_status(const uint8_t
     * event){
    return event[3];
}

/**
 * @brief Get field indicator_index from event
     HFP_SUBEVENT_AG_INDICATOR_STATUS_CHANGED
 * @param event packet
 * @return indicator_index
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_ag_indicator_status_changed_get_indicator_index(
    const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field indicator_status from event
     HFP_SUBEVENT_AG_INDICATOR_STATUS_CHANGED
 * @param event packet
 * @return indicator_status
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_ag_indicator_status_changed_get_indicator_status(
    const uint8_t * event){
    return event[4];
}
```

```
/**
 * @brief Get field indicator_name from event
     HFP_SUBEVENT_AG_INDICATOR_STATUS_CHANGED
 * @param event packet
 * @return indicator_name
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_ag_indicator_status_changed_get_indicator_name(
    const uint8_t * event){
     return (const char *) &event[5];
}

/**
 * @brief Get field network_operator_mode from event
     HFP_SUBEVENT_NETWORK_OPERATOR_CHANGED
 * @param event packet
 * @return network_operator_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_network_operator_changed_get_network_operator_mode(
    const uint8_t * event){
     return event[3];
}
/**
 * @brief Get field network_operator_format from event
     HFP_SUBEVENT_NETWORK_OPERATOR_CHANGED
 * @param event packet
 * @return network_operator_format
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_network_operator_changed_get_network_operator_format
    (const uint8_t * event){
     return event[4];
}
/**
 * @brief Get field network_operator_name from event
     HFP_SUBEVENT_NETWORK_OPERATOR_CHANGED
 * @param event packet
 * @return network_operator_name
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_network_operator_changed_get_network_operator_name(
    const uint8_t * event){
     return (const char *) &event[5];
}

/**
 * @brief Get field error from event
     HFP_SUBEVENT_EXTENDED_AUDIO_GATEWAY_ERROR
 * @param event packet
 * @return error
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_extended_audio_gateway_error_get_error(const
    uint8_t * event){
     return event[3];
}




/**
 * @brief Get field number from event
     HFP_SUBEVENT_PLACE_CALL_WITH_NUMBER
 * @param event packet
 * @return number
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_place_call_with_number_get_number(const uint8_t *
    event){
     return (const char *) &event[3];
}


/**
 * @brief Get field number from event
     HFP_SUBEVENT_NUMBER_FOR_VOICE_TAG
 * @param event packet
 * @return number
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_number_for_voice_tag_get_number(const uint8_t *
    event){
     return (const char *) &event[3];
}

/**
 * @brief Get field dtmf from event HFP_SUBEVENT_TRANSMIT_DTMF_CODES
 * @param event packet
 * @return dtmf
 * @note: btstack_type T
 */
static inline const char * hfp_subevent_transmit_dtmf_codes_get_dtmf
    (const uint8_t * event){
     return (const char *) &event[3];
}




/**
```

```
 * @brief Get field status from event HFP_SUBEVENT_SPEAKER_VOLUME
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_speaker_volume_get_status(const
    uint8_t * event){
     return event[3];
}
/**
 * @brief Get field gain from event HFP_SUBEVENT_SPEAKER_VOLUME
 * @param event packet
 * @return gain
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_speaker_volume_get_gain(const
    uint8_t * event){
     return event[4];
}

/**
 * @brief Get field status from event HFP_SUBEVENT_MICROPHONE_VOLUME
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_microphone_volume_get_status(
    const uint8_t * event){
     return event[3];
}
/**
 * @brief Get field gain from event HFP_SUBEVENT_MICROPHONE_VOLUME
 * @param event packet
 * @return gain
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_microphone_volume_get_gain(const
    uint8_t * event){
     return event[4];
}

/**
 * @brief Get field type from event
     HFP_SUBEVENT_CALL_WAITING_NOTIFICATION
 * @param event packet
 * @return type
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_call_waiting_notification_get_type(const uint8_t *
    event){
     return event[3];
}
/**
```

```
 * @brief Get field number from event
     HFP_SUBEVENT_CALL_WAITING_NOTIFICATION
 * @param event packet
 * @return number
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_call_waiting_notification_get_number(const uint8_t
    * event){
    return (const char *) &event[4];
}

/**
 * @brief Get field type from event
     HFP_SUBEVENT_CALLING_LINE_IDENTIFICATION_NOTIFICATION
 * @param event packet
 * @return type
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_calling_line_identification_notification_get_type(
    const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field number from event
     HFP_SUBEVENT_CALLING_LINE_IDENTIFICATION_NOTIFICATION
 * @param event packet
 * @return number
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_calling_line_identification_notification_get_number
    (const uint8_t * event){
    return (const char *) &event[4];
}

/**
 * @brief Get field clcc_idx from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return clcc_idx
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_enhanced_call_status_get_clcc_idx
    (const uint8_t * event){
    return event[3];
}
/**
 * @brief Get field clcc_dir from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return clcc_dir
 * @note: btstack_type 1
 */
static inline uint8_t hfp_subevent_enhanced_call_status_get_clcc_dir
   (const uint8_t * event){
    return event[4];
}
/**
 * @brief Get field clcc_status from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return clcc_status
 * @note: btstack_type 1
 */
static inline uint8_t
   hfp_subevent_enhanced_call_status_get_clcc_status(const uint8_t
   * event){
    return event[5];
}
/**
 * @brief Get field clcc_mpty from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return clcc_mpty
 * @note: btstack_type 1
 */
static inline uint8_t
   hfp_subevent_enhanced_call_status_get_clcc_mpty(const uint8_t *
   event){
    return event[6];
}
/**
 * @brief Get field bnip_type from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return bnip_type
 * @note: btstack_type 1
 */
static inline uint8_t
   hfp_subevent_enhanced_call_status_get_bnip_type(const uint8_t *
   event){
    return event[7];
}
/**
 * @brief Get field bnip_number from event
     HFP_SUBEVENT_ENHANCED_CALL_STATUS
 * @param event packet
 * @return bnip_number
 * @note: btstack_type T
 */
static inline const char *
   hfp_subevent_enhanced_call_status_get_bnip_number(const uint8_t
   * event){
    return (const char *) &event[8];
}
```

```
/**
 * @brief Get field status from event
     HFP_SUBEVENT_SUBSCRIBER_NUMBER_INFORMATION
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_subscriber_number_information_get_status(const
    uint8_t * event){
     return event[3];
}
/**
 * @brief Get field bnip_type from event
     HFP_SUBEVENT_SUBSCRIBER_NUMBER_INFORMATION
 * @param event packet
 * @return bnip_type
 * @note: btstack_type 1
 */
static inline uint8_t
    hfp_subevent_subscriber_number_information_get_bnip_type(const
    uint8_t * event){
     return event[4];
}
/**
 * @brief Get field bnip_number from event
     HFP_SUBEVENT_SUBSCRIBER_NUMBER_INFORMATION
 * @param event packet
 * @return bnip_number
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_subscriber_number_information_get_bnip_number(const
     uint8_t * event){
     return (const char *) &event[5];
}

/**
 * @brief Get field value from event
     HFP_SUBEVENT_RESPONSE_AND_HOLD_STATUS
 * @param event packet
 * @return value
 * @note: btstack_type T
 */
static inline const char *
    hfp_subevent_response_and_hold_status_get_value(const uint8_t *
    event){
     return (const char *) &event[3];
}

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event ANCS_SUBEVENT_CLIENT_CONNECTED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    ancs_subevent_client_connected_get_handle(const uint8_t * event)
    {
     return little_endian_read_16(event, 3);
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     ANCS_SUBEVENT_CLIENT_NOTIFICATION
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    ancs_subevent_client_notification_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field attribute_id from event
     ANCS_SUBEVENT_CLIENT_NOTIFICATION
 * @param event packet
 * @return attribute_id
 * @note: btstack_type 2
 */
static inline uint16_t
    ancs_subevent_client_notification_get_attribute_id(const uint8_t
     * event){
     return little_endian_read_16(event, 5);
}
/**
 * @brief Get field text from event
     ANCS_SUBEVENT_CLIENT_NOTIFICATION
 * @param event packet
 * @return text
 * @note: btstack_type T
 */
static inline const char *
    ancs_subevent_client_notification_get_text(const uint8_t * event
    ){
     return (const char *) &event[7];
}
#endif

#ifdef ENABLE_BLE
/**
 * @brief Get field handle from event
     ANCS_SUBEVENT_CLIENT_DISCONNECTED
 * @param event packet
 * @return handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    ancs_subevent_client_disconnected_get_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 3);
}
#endif

/**
 * @brief Get field avdtp_cid from event
      AVDTP_SUBEVENT_SIGNALING_ACCEPT
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avdtp_subevent_signaling_accept_get_avdtp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
      AVDTP_SUBEVENT_SIGNALING_ACCEPT
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t avdtp_subevent_signaling_accept_get_local_seid
    (const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field signal_identifier from event
      AVDTP_SUBEVENT_SIGNALING_ACCEPT
 * @param event packet
 * @return signal_identifier
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_accept_get_signal_identifier(const
    uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avdtp_cid from event
      AVDTP_SUBEVENT_SIGNALING_REJECT
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avdtp_subevent_signaling_reject_get_avdtp_cid
    (const uint8_t * event){
```

```
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_REJECT
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t avdtp_subevent_signaling_reject_get_local_seid
   (const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field signal_identifier from event
     AVDTP_SUBEVENT_SIGNALING_REJECT
 * @param event packet
 * @return signal_identifier
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_reject_get_signal_identifier(const
   uint8_t * event){
    return event[6];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_GENERAL_REJECT
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   avdtp_subevent_signaling_general_reject_get_avdtp_cid(const
   uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_GENERAL_REJECT
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_general_reject_get_local_seid(const
   uint8_t * event){
    return event[5];
}
/**
 * @brief Get field signal_identifier from event
     AVDTP_SUBEVENT_SIGNALING_GENERAL_REJECT
 * @param event packet
 * @return signal_identifier
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_general_reject_get_signal_identifier(
    const uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_connection_established_get_avdtp_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event
     AVDTP_SUBEVENT_SIGNALING_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
    avdtp_subevent_signaling_connection_established_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[5], bd_addr);
}
/**
 * @brief Get field status from event
     AVDTP_SUBEVENT_SIGNALING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_connection_established_get_status(const
     uint8_t * event){
     return event[11];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_CONNECTION_RELEASED
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
```

```
static inline uint16_t
    avdtp_subevent_signaling_connection_released_get_avdtp_cid(const
     uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_SEP_FOUND
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_sep_found_get_avdtp_cid(const uint8_t *
     event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_SIGNALING_SEP_FOUND
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_sep_found_get_remote_seid(const uint8_t
     * event){
    return event[5];
}
/**
 * @brief Get field in_use from event
     AVDTP_SUBEVENT_SIGNALING_SEP_FOUND
 * @param event packet
 * @return in_use
 * @note: btstack_type 1
 */
static inline uint8_t avdtp_subevent_signaling_sep_found_get_in_use(
    const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field media_type from event
     AVDTP_SUBEVENT_SIGNALING_SEP_FOUND
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_sep_found_get_media_type(const uint8_t
    * event){
    return event[7];
}
/**
```

```
 * @brief Get field sep_type from event
     AVDTP_SUBEVENT_SIGNALING_SEP_FOUND
 * @param event packet
 * @return sep_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_sep_found_get_sep_type(const uint8_t *
    event){
    return event[8];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_avdtp_cid
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_local_seid
    (const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_remote_seid
    (const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field media_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_capability_get_media_type
   (const uint8_t * event){
    return event[7];
}
/**
 * @brief Get field sampling_frequency_bitmap from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return sampling_frequency_bitmap
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_capability_get_sampling_frequency_bitmap
   (const uint8_t * event){
    return event[8];
}
/**
 * @brief Get field channel_mode_bitmap from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return channel_mode_bitmap
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_capability_get_channel_mode_bitmap
   (const uint8_t * event){
    return event[9];
}
/**
 * @brief Get field block_length_bitmap from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return block_length_bitmap
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_capability_get_block_length_bitmap
   (const uint8_t * event){
    return event[10];
}
/**
 * @brief Get field subbands_bitmap from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return subbands_bitmap
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_capability_get_subbands_bitmap
   (const uint8_t * event){
    return event[11];
}
```

```
/**
 * @brief Get field allocation_method_bitmap from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return allocation_method_bitmap
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_allocation_method_bitmap
    (const uint8_t * event){
     return event[12];
}
/**
 * @brief Get field min_bitpool_value from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return min_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_min_bitpool_value
    (const uint8_t * event){
     return event[13];
}
/**
 * @brief Get field max_bitpool_value from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CAPABILITY
 * @param event packet
 * @return max_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_capability_get_max_bitpool_value
    (const uint8_t * event){
     return event[14];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_media_codec_other_capability_get_avdtp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_other_capability_get_local_seid
   (const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_other_capability_get_remote_seid
   (const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field media_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_other_capability_get_media_type
   (const uint8_t * event){
    return event[7];
}
/**
 * @brief Get field media_codec_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return media_codec_type
 * @note: btstack_type 2
 */
static inline uint16_t
   avdtp_subevent_signaling_media_codec_other_capability_get_media_codec_type
   (const uint8_t * event){
    return little_endian_read_16(event, 8);
}
/**
 * @brief Get field media_codec_information_len from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return media_codec_information_len
 * @note: btstack_type L
 */
static inline int
   avdtp_subevent_signaling_media_codec_other_capability_get_media_codec_informatio
   (const uint8_t * event){
    return little_endian_read_16(event, 10);
}
/**
 * @brief Get field media_codec_information from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CAPABILITY
 * @param event packet
 * @return media_codec_information
 * @note: btstack_type V
 */
static inline const uint8_t *
    avdtp_subevent_signaling_media_codec_other_capability_get_media_codec_information
    (const uint8_t * event){
     return &event[12];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_media_codec_sbc_configuration_get_avdtp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_configuration_get_local_seid
    (const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_sbc_configuration_get_remote_seid
    (const uint8_t * event){
     return event[6];
}
/**
 * @brief Get field reconfigure from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return reconfigure
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_reconfigure
   (const uint8_t * event){
    return event[7];
}
/**
 * @brief Get field media_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_media_type
   (const uint8_t * event){
    return event[8];
}
/**
 * @brief Get field sampling_frequency from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return sampling_frequency
 * @note: btstack_type 2
 */
static inline uint16_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_sampling_frequency
   (const uint8_t * event){
    return little_endian_read_16(event, 9);
}
/**
 * @brief Get field channel_mode from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return channel_mode
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_channel_mode
   (const uint8_t * event){
    return event[11];
}
/**
 * @brief Get field num_channels from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return num_channels
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_num_channels
   (const uint8_t * event){
    return event[12];
}
/**
 * @brief Get field block_length from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return block_length
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_block_length
   (const uint8_t * event){
    return event[13];
}
/**
 * @brief Get field subbands from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return subbands
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_subbands
   (const uint8_t * event){
    return event[14];
}
/**
 * @brief Get field allocation_method from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return allocation_method
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_allocation_method
   (const uint8_t * event){
    return event[15];
}
/**
 * @brief Get field min_bitpool_value from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return min_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_min_bitpool_value
   (const uint8_t * event){
    return event[16];
}
/**
 * @brief Get field max_bitpool_value from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return max_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_sbc_configuration_get_max_bitpool_value
   (const uint8_t * event){
    return event[17];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   avdtp_subevent_signaling_media_codec_other_configuration_get_avdtp_cid
   (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_other_configuration_get_local_seid
   (const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   avdtp_subevent_signaling_media_codec_other_configuration_get_remote_seid
   (const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field reconfigure from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return reconfigure
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    avdtp_subevent_signaling_media_codec_other_configuration_get_reconfigure
    (const uint8_t * event){
     return event[7];
}
/**
 * @brief Get field media_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_signaling_media_codec_other_configuration_get_media_type
    (const uint8_t * event){
     return event[8];
}
/**
 * @brief Get field media_codec_type from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_type
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_signaling_media_codec_other_configuration_get_media_codec_type
    (const uint8_t * event){
     return little_endian_read_16(event, 9);
}
/**
 * @brief Get field media_codec_information_len from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_information_len
 * @note: btstack_type L
 */
static inline int
    avdtp_subevent_signaling_media_codec_other_configuration_get_media_codec_informa
    (const uint8_t * event){
     return little_endian_read_16(event, 11);
}
/**
 * @brief Get field media_codec_information from event
     AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_information
 * @note: btstack_type V
 */
static inline const uint8_t *
    avdtp_subevent_signaling_media_codec_other_configuration_get_media_codec_informat
    (const uint8_t * event){
     return &event[13];
}
```

```
/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_streaming_connection_established_get_avdtp_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
    avdtp_subevent_streaming_connection_established_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[5], bd_addr);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_streaming_connection_established_get_local_seid(
    const uint8_t * event){
     return event[11];
}
/**
 * @brief Get field remote_seid from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_streaming_connection_established_get_remote_seid(
    const uint8_t * event){
     return event[12];
}
/**
 * @brief Get field status from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_streaming_connection_established_get_status(const
     uint8_t * event){
    return event[13];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_RELEASED
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_streaming_connection_released_get_avdtp_cid(const
     uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_STREAMING_CONNECTION_RELEASED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_streaming_connection_released_get_local_seid(
    const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field avdtp_cid from event
     AVDTP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
 * @param event packet
 * @return avdtp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avdtp_subevent_streaming_can_send_media_packet_now_get_avdtp_cid
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     AVDTP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    avdtp_subevent_streaming_can_send_media_packet_now_get_local_seid
    (const uint8_t * event){
```

```
    return event[5];
}
/**
 * @brief Get field sequence_number from event
     AVDTP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
 * @param event packet
 * @return sequence_number
 * @note: btstack_type 2
 */
static inline uint16_t
   avdtp_subevent_streaming_can_send_media_packet_now_get_sequence_number
   (const uint8_t * event){
    return little_endian_read_16(event, 6);
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   a2dp_subevent_streaming_can_send_media_packet_now_get_a2dp_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
   a2dp_subevent_streaming_can_send_media_packet_now_get_local_seid
   (const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   a2dp_subevent_signaling_media_codec_sbc_configuration_get_a2dp_cid
   (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
```

```
 * @brief Get field int_seid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return int_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_int_seid
    (const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field acp_seid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return acp_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_acp_seid
    (const uint8_t * event){
     return event[6];
}
/**
 * @brief Get field reconfigure from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return reconfigure
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_reconfigure
    (const uint8_t * event){
     return event[7];
}
/**
 * @brief Get field media_type from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_media_type
    (const uint8_t * event){
     return event[8];
}
/**
 * @brief Get field sampling_frequency from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return sampling_frequency
 * @note: btstack_type 2
 */
```

```
static inline uint16_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_sampling_frequency
    (const uint8_t * event){
     return little_endian_read_16(event, 9);
}
/**
 * @brief Get field channel_mode from event
      A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return channel_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_channel_mode
    (const uint8_t * event){
     return event[11];
}
/**
 * @brief Get field num_channels from event
      A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return num_channels
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_num_channels
    (const uint8_t * event){
     return event[12];
}
/**
 * @brief Get field block_length from event
      A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return block_length
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_block_length
    (const uint8_t * event){
     return event[13];
}
/**
 * @brief Get field subbands from event
      A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return subbands
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_subbands
    (const uint8_t * event){
     return event[14];
}
/**
```

```
 * @brief Get field allocation_method from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return allocation_method
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_allocation_method
    (const uint8_t * event){
    return event[15];
}
/**
 * @brief Get field min_bitpool_value from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return min_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_min_bitpool_value
    (const uint8_t * event){
    return event[16];
}
/**
 * @brief Get field max_bitpool_value from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 * @param event packet
 * @return max_bitpool_value
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_sbc_configuration_get_max_bitpool_value
    (const uint8_t * event){
    return event[17];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_a2dp_cid
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field int_seid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return int_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_int_seid
    (const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field acp_seid from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return acp_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_acp_seid
    (const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field reconfigure from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return reconfigure
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_reconfigure
    (const uint8_t * event){
    return event[7];
}
/**
 * @brief Get field media_type from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_type
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_media_type
    (const uint8_t * event){
    return event[8];
}
/**
 * @brief Get field media_codec_type from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_type
 * @note: btstack_type 2
 */
static inline uint16_t
    a2dp_subevent_signaling_media_codec_other_configuration_get_media_codec_type
    (const uint8_t * event){
    return little_endian_read_16(event, 9);
}
```

```
/**
 * @brief Get field media_codec_information_len from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_information_len
 * @note: btstack_type L
 */
static inline int
    a2dp_subevent_signaling_media_codec_other_configuration_get_media_codec_informati
    (const uint8_t * event){
     return little_endian_read_16(event, 11);
}
/**
 * @brief Get field media_codec_information from event
     A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION
 * @param event packet
 * @return media_codec_information
 * @note: btstack_type V
 */
static inline const uint8_t *
    a2dp_subevent_signaling_media_codec_other_configuration_get_media_codec_informati
    (const uint8_t * event){
     return &event[13];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_STREAM_ESTABLISHED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_stream_established_get_a2dp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event
     A2DP_SUBEVENT_STREAM_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void a2dp_subevent_stream_established_get_bd_addr(
    const uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[5], bd_addr);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_STREAM_ESTABLISHED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    a2dp_subevent_stream_established_get_local_seid(const uint8_t *
    event){
     return event[11];
}
/**
 * @brief Get field remote_seid from event
     A2DP_SUBEVENT_STREAM_ESTABLISHED
 * @param event packet
 * @return remote_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_stream_established_get_remote_seid(const uint8_t *
     event){
     return event[12];
}
/**
 * @brief Get field status from event
     A2DP_SUBEVENT_STREAM_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_stream_established_get_status(
    const uint8_t * event){
     return event[13];
}

/**
 * @brief Get field a2dp_cid from event A2DP_SUBEVENT_STREAM_STARTED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_stream_started_get_a2dp_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_STREAM_STARTED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_stream_started_get_local_seid(
    const uint8_t * event){
     return event[5];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_STREAM_SUSPENDED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_stream_suspended_get_a2dp_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_STREAM_SUSPENDED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_stream_suspended_get_local_seid(
   const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_STREAM_RELEASED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_stream_released_get_a2dp_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_STREAM_RELEASED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_stream_released_get_local_seid(
   const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_COMMAND_ACCEPTED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_command_accepted_get_a2dp_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}
```

```
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_COMMAND_ACCEPTED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_command_accepted_get_local_seid(
    const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field signal_identifier from event
     A2DP_SUBEVENT_COMMAND_ACCEPTED
 * @param event packet
 * @return signal_identifier
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_command_accepted_get_signal_identifier(const
    uint8_t * event){
    return event[6];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_COMMAND_REJECTED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t a2dp_subevent_command_rejected_get_a2dp_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_COMMAND_REJECTED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t a2dp_subevent_command_rejected_get_local_seid(
    const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field signal_identifier from event
     A2DP_SUBEVENT_COMMAND_REJECTED
 * @param event packet
 * @return signal_identifier
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    a2dp_subevent_command_rejected_get_signal_identifier(const
    uint8_t * event){
    return event[6];
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_INCOMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    a2dp_subevent_incoming_connection_established_get_a2dp_cid(const
     uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field bd_addr from event
     A2DP_SUBEVENT_INCOMING_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void
    a2dp_subevent_incoming_connection_established_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[5], bd_addr);
}

/**
 * @brief Get field a2dp_cid from event
     A2DP_SUBEVENT_SIGNALING_CONNECTION_RELEASED
 * @param event packet
 * @return a2dp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    a2dp_subevent_signaling_connection_released_get_a2dp_cid(const
    uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field local_seid from event
     A2DP_SUBEVENT_SIGNALING_CONNECTION_RELEASED
 * @param event packet
 * @return local_seid
 * @note: btstack_type 1
 */
static inline uint8_t
    a2dp_subevent_signaling_connection_released_get_local_seid(const
     uint8_t * event){
    return event[5];
}

/**
 * @brief Get field status from event
     AVRCP_SUBEVENT_CONNECTION_ESTABLISHED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_connection_established_get_status(const uint8_t *
     event){
    return event[3];
}
/**
 * @brief Get field bd_addr from event
     AVRCP_SUBEVENT_CONNECTION_ESTABLISHED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void avrcp_subevent_connection_established_get_bd_addr
    (const uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[4], bd_addr);
}
/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_CONNECTION_ESTABLISHED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_connection_established_get_avrcp_cid(const
    uint8_t * event){
    return little_endian_read_16(event, 10);
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_CONNECTION_RELEASED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_connection_released_get_avrcp_cid(const uint8_t *
     event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avrcp_subevent_now_playing_info_get_avrcp_cid
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_now_playing_info_get_command_type(const uint8_t *
     event){
    return event[5];
}
/**
 * @brief Get field track from event AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return track
 * @note: btstack_type 1
 */
static inline uint8_t avrcp_subevent_now_playing_info_get_track(
    const uint8_t * event){
    return event[6];
}
/**
 * @brief Get field total_tracks from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return total_tracks
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_now_playing_info_get_total_tracks(const uint8_t *
     event){
    return event[7];
}
/**
 * @brief Get field song_length from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return song_length
 * @note: btstack_type 4
 */
static inline uint32_t
    avrcp_subevent_now_playing_info_get_song_length(const uint8_t *
    event){
    return little_endian_read_32(event, 8);
}
```

```
/**
 * @brief Get field title_len from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return title_len
 * @note: btstack_type J
 */
static inline int avrcp_subevent_now_playing_info_get_title_len(
    const uint8_t * event){
     return event[12];
}
/**
 * @brief Get field title from event AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return title
 * @note: btstack_type V
 */
static inline const uint8_t *
    avrcp_subevent_now_playing_info_get_title(const uint8_t * event)
    {
     return &event[13];
}
/**
 * @brief Get field artist_len from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return artist_len
 * @note: btstack_type J
 */
static inline int avrcp_subevent_now_playing_info_get_artist_len(
    const uint8_t * event){
     return event[13 + event[12]];
}
/**
 * @brief Get field artist from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return artist
 * @note: btstack_type V
 */
static inline const uint8_t *
    avrcp_subevent_now_playing_info_get_artist(const uint8_t * event
    ){
     return &event[13 + event[12] + 1];
}
/**
 * @brief Get field album_len from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return album_len
 * @note: btstack_type J
 */
static inline int avrcp_subevent_now_playing_info_get_album_len(
    const uint8_t * event){
```

```
    return event[13 + event[12] + 1 + event[13 + event[12]]];
}
/**
 * @brief Get field album from event AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return album
 * @note: btstack_type V
 */
static inline const uint8_t *
   avrcp_subevent_now_playing_info_get_album(const uint8_t * event)
   {
    return &event[13 + event[12] + 1 + event[13 + event[12]] + 1];
}
/**
 * @brief Get field genre_len from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return genre_len
 * @note: btstack_type J
 */
static inline int avrcp_subevent_now_playing_info_get_genre_len(
   const uint8_t * event){
    return event[13 + event[12] + 1 + event[13 + event[12]] + 1 +
        event[13 + event[12] + 1 + event[13 + event[12]]]];
}
/**
 * @brief Get field genre from event AVRCP_SUBEVENT_NOW_PLAYING_INFO
 * @param event packet
 * @return genre
 * @note: btstack_type V
 */
static inline const uint8_t *
   avrcp_subevent_now_playing_info_get_genre(const uint8_t * event)
   {
    return &event[13 + event[12] + 1 + event[13 + event[12]] + 1 +
        event[13 + event[12] + 1 + event[13 + event[12]]] + 1];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_SHUFFLE_AND_REPEAT_MODE
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
   avrcp_subevent_shuffle_and_repeat_mode_get_avrcp_cid(const
   uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_SHUFFLE_AND_REPEAT_MODE
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_shuffle_and_repeat_mode_get_command_type(const
    uint8_t * event){
     return event[5];
}
/**
 * @brief Get field repeat_mode from event
     AVRCP_SUBEVENT_SHUFFLE_AND_REPEAT_MODE
 * @param event packet
 * @return repeat_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_shuffle_and_repeat_mode_get_repeat_mode(const
    uint8_t * event){
     return event[6];
}
/**
 * @brief Get field shuffle_mode from event
     AVRCP_SUBEVENT_SHUFFLE_AND_REPEAT_MODE
 * @param event packet
 * @return shuffle_mode
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_shuffle_and_repeat_mode_get_shuffle_mode(const
    uint8_t * event){
     return event[7];
}

/**
 * @brief Get field avrcp_cid from event AVRCP_SUBEVENT_PLAY_STATUS
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avrcp_subevent_play_status_get_avrcp_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_PLAY_STATUS
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t avrcp_subevent_play_status_get_command_type(
    const uint8_t * event){
     return event[5];
}
```

```
/**
 * @brief Get field song_length from event
     AVRCP_SUBEVENT_PLAY_STATUS
 * @param event packet
 * @return song_length
 * @note: btstack_type 4
 */
static inline uint32_t avrcp_subevent_play_status_get_song_length(
    const uint8_t * event){
    return little_endian_read_32(event, 6);
}
/**
 * @brief Get field song_position from event
     AVRCP_SUBEVENT_PLAY_STATUS
 * @param event packet
 * @return song_position
 * @note: btstack_type 4
 */
static inline uint32_t avrcp_subevent_play_status_get_song_position(
    const uint8_t * event){
    return little_endian_read_32(event, 10);
}
/**
 * @brief Get field play_status from event
     AVRCP_SUBEVENT_PLAY_STATUS
 * @param event packet
 * @return play_status
 * @note: btstack_type 1
 */
static inline uint8_t avrcp_subevent_play_status_get_play_status(
    const uint8_t * event){
    return event[14];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOTIFICATION_PLAYBACK_STATUS_CHANGED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_notification_playback_status_changed_get_avrcp_cid
    (const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOTIFICATION_PLAYBACK_STATUS_CHANGED
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    avrcp_subevent_notification_playback_status_changed_get_command_type
    (const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field play_status from event
     AVRCP_SUBEVENT_NOTIFICATION_PLAYBACK_STATUS_CHANGED
 * @param event packet
 * @return play_status
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_playback_status_changed_get_play_status
    (const uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOTIFICATION_TRACK_CHANGED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_notification_track_changed_get_avrcp_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOTIFICATION_TRACK_CHANGED
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_track_changed_get_command_type(const
     uint8_t * event){
     return event[5];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOTIFICATION_NOW_PLAYING_CONTENT_CHANGED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_notification_now_playing_content_changed_get_avrcp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOTIFICATION_NOW_PLAYING_CONTENT_CHANGED
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_now_playing_content_changed_get_command_type
    (const uint8_t * event){
     return event[5];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOTIFICATION_AVAILABLE_PLAYERS_CHANGED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_notification_available_players_changed_get_avrcp_cid
    (const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOTIFICATION_AVAILABLE_PLAYERS_CHANGED
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_available_players_changed_get_command_type
    (const uint8_t * event){
     return event[5];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOTIFICATION_VOLUME_CHANGED
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_notification_volume_changed_get_avrcp_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_NOTIFICATION_VOLUME_CHANGED
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_volume_changed_get_command_type(
    const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field absolute_volume from event
      AVRCP_SUBEVENT_NOTIFICATION_VOLUME_CHANGED
 * @param event packet
 * @return absolute_volume
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_notification_volume_changed_get_absolute_volume(
    const uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avrcp_cid from event
      AVRCP_SUBEVENT_SET_ABSOLUTE_VOLUME_RESPONSE
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_set_absolute_volume_response_get_avrcp_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
      AVRCP_SUBEVENT_SET_ABSOLUTE_VOLUME_RESPONSE
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_set_absolute_volume_response_get_command_type(
    const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field absolute_volume from event
      AVRCP_SUBEVENT_SET_ABSOLUTE_VOLUME_RESPONSE
 * @param event packet
 * @return absolute_volume
 * @note: btstack_type 1
 */
```

```
static inline uint8_t
    avrcp_subevent_set_absolute_volume_response_get_absolute_volume(
    const uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avrcp_cid from event
      AVRCP_SUBEVENT_ENABLE_NOTIFICATION_COMPLETE
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_enable_notification_complete_get_avrcp_cid(const
    uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
      AVRCP_SUBEVENT_ENABLE_NOTIFICATION_COMPLETE
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_enable_notification_complete_get_command_type(
    const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field notification_id from event
      AVRCP_SUBEVENT_ENABLE_NOTIFICATION_COMPLETE
 * @param event packet
 * @return notification_id
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_enable_notification_complete_get_notification_id(
    const uint8_t * event){
     return event[6];
}

/**
 * @brief Get field avrcp_cid from event
      AVRCP_SUBEVENT_OPERATION_START
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avrcp_subevent_operation_start_get_avrcp_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
```

```
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_OPERATION_START
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_operation_start_get_command_type(const uint8_t *
    event){
     return event[5];
}
/**
 * @brief Get field operation_id from event
     AVRCP_SUBEVENT_OPERATION_START
 * @param event packet
 * @return operation_id
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_operation_start_get_operation_id(const uint8_t *
    event){
     return event[6];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_OPERATION_COMPLETE
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_operation_complete_get_avrcp_cid(const uint8_t *
    event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_OPERATION_COMPLETE
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_operation_complete_get_command_type(const uint8_t
     * event){
     return event[5];
}
/**
 * @brief Get field operation_id from event
     AVRCP_SUBEVENT_OPERATION_COMPLETE
 * @param event packet
 * @return operation_id
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_operation_complete_get_operation_id(const uint8_t
     * event){
    return event[6];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_PLAYER_APPLICATION_VALUE_RESPONSE
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_player_application_value_response_get_avrcp_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field command_type from event
     AVRCP_SUBEVENT_PLAYER_APPLICATION_VALUE_RESPONSE
 * @param event packet
 * @return command_type
 * @note: btstack_type 1
 */
static inline uint8_t
    avrcp_subevent_player_application_value_response_get_command_type
    (const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_UNIT_INFO_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avrcp_subevent_unit_info_query_get_avrcp_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_SUBUNIT_INFO_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
```

```
static inline uint16_t
    avrcp_subevent_subunit_info_query_get_avrcp_cid(const uint8_t *
    event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field offset from event
     AVRCP_SUBEVENT_SUBUNIT_INFO_QUERY
 * @param event packet
 * @return offset
 * @note: btstack_type 1
 */
static inline uint8_t avrcp_subevent_subunit_info_query_get_offset(
    const uint8_t * event){
    return event[5];
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_COMPANY_IDS_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_company_ids_query_get_avrcp_cid(const uint8_t *
    event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_EVENT_IDS_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t avrcp_subevent_event_ids_query_get_avrcp_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_PLAY_STATUS_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_play_status_query_get_avrcp_cid(const uint8_t *
    event){
    return little_endian_read_16(event, 3);
}
```

```
/**
 * @brief Get field avrcp_cid from event
     AVRCP_SUBEVENT_NOW_PLAYING_INFO_QUERY
 * @param event packet
 * @return avrcp_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    avrcp_subevent_now_playing_info_query_get_avrcp_cid(const
    uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field goep_cid from event
     GOEP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return goep_cid
 * @note: btstack_type 2
 */
static inline uint16_t goep_subevent_connection_opened_get_goep_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field status from event
     GOEP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t goep_subevent_connection_opened_get_status(
    const uint8_t * event){
    return event[5];
}
/**
 * @brief Get field bd_addr from event
     GOEP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void goep_subevent_connection_opened_get_bd_addr(const
     uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[6], bd_addr);
}
/**
 * @brief Get field con_handle from event
     GOEP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return con_handle
 * @note: btstack_type H
 */
```

```
static inline hci_con_handle_t
   goep_subevent_connection_opened_get_con_handle(const uint8_t *
   event){
    return little_endian_read_16(event, 12);
}
/**
 * @brief Get field incoming from event
     GOEP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t goep_subevent_connection_opened_get_incoming(
   const uint8_t * event){
    return event[14];
}

/**
 * @brief Get field goep_cid from event
     GOEP_SUBEVENT_CONNECTION_CLOSED
 * @param event packet
 * @return goep_cid
 * @note: btstack_type 2
 */
static inline uint16_t goep_subevent_connection_closed_get_goep_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field goep_cid from event GOEP_SUBEVENT_CAN_SEND_NOW
 * @param event packet
 * @return goep_cid
 * @note: btstack_type 2
 */
static inline uint16_t goep_subevent_can_send_now_get_goep_cid(const
     uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field pbap_cid from event
     PBAP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return pbap_cid
 * @note: btstack_type 2
 */
static inline uint16_t pbap_subevent_connection_opened_get_pbap_cid(
   const uint8_t * event){
    return little_endian_read_16(event, 3);
}
/**
 * @brief Get field status from event
     PBAP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t pbap_subevent_connection_opened_get_status(
    const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field bd_addr from event
      PBAP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
static inline void pbap_subevent_connection_opened_get_bd_addr(const
     uint8_t * event, bd_addr_t bd_addr){
     reverse_bd_addr(&event[6], bd_addr);
}
/**
 * @brief Get field con_handle from event
      PBAP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return con_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    pbap_subevent_connection_opened_get_con_handle(const uint8_t *
    event){
     return little_endian_read_16(event, 12);
}
/**
 * @brief Get field incoming from event
      PBAP_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t pbap_subevent_connection_opened_get_incoming(
    const uint8_t * event){
     return event[14];
}

/**
 * @brief Get field goep_cid from event
      PBAP_SUBEVENT_CONNECTION_CLOSED
 * @param event packet
 * @return goep_cid
 * @note: btstack_type 2
 */
static inline uint16_t pbap_subevent_connection_closed_get_goep_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
```

```
/**
 * @brief Get field goep_cid from event
     PBAP_SUBEVENT_OPERATION_COMPLETED
 * @param event packet
 * @return goep_cid
 * @note: btstack_type 2
 */
static inline uint16_t
    pbap_subevent_operation_completed_get_goep_cid(const uint8_t *
    event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field status from event
     PBAP_SUBEVENT_OPERATION_COMPLETED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t pbap_subevent_operation_completed_get_status(
    const uint8_t * event){
     return event[5];
}

/**
 * @brief Get field hid_cid from event
     HID_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return hid_cid
 * @note: btstack_type 2
 */
static inline uint16_t hid_subevent_connection_opened_get_hid_cid(
    const uint8_t * event){
     return little_endian_read_16(event, 3);
}
/**
 * @brief Get field status from event HID_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return status
 * @note: btstack_type 1
 */
static inline uint8_t hid_subevent_connection_opened_get_status(
    const uint8_t * event){
     return event[5];
}
/**
 * @brief Get field bd_addr from event
     HID_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @param Pointer to storage for bd_addr
 * @note: btstack_type B
 */
```

```
static inline void hid_subevent_connection_opened_get_bd_addr(const
    uint8_t * event, bd_addr_t bd_addr){
    reverse_bd_addr(&event[6], bd_addr);
}
/**
 * @brief Get field con_handle from event
     HID_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return con_handle
 * @note: btstack_type H
 */
static inline hci_con_handle_t
    hid_subevent_connection_opened_get_con_handle(const uint8_t *
    event){
    return little_endian_read_16(event, 12);
}
/**
 * @brief Get field incoming from event
     HID_SUBEVENT_CONNECTION_OPENED
 * @param event packet
 * @return incoming
 * @note: btstack_type 1
 */
static inline uint8_t hid_subevent_connection_opened_get_incoming(
    const uint8_t * event){
    return event[14];
}

/**
 * @brief Get field hid_cid from event
     HID_SUBEVENT_CONNECTION_CLOSED
 * @param event packet
 * @return hid_cid
 * @note: btstack_type 2
 */
static inline uint16_t hid_subevent_connection_closed_get_hid_cid(
    const uint8_t * event){
    return little_endian_read_16(event, 3);
}

/**
 * @brief Get field hid_cid from event HID_SUBEVENT_CAN_SEND_NOW
 * @param event packet
 * @return hid_cid
 * @note: btstack_type 2
 */
static inline uint16_t hid_subevent_can_send_now_get_hid_cid(const
    uint8_t * event){
    return little_endian_read_16(event, 3);
}
```

19.23. BTstack Memory Management API.

```
/**
 * @brief Initializes BTstack memory pools.
 */
void btstack_memory_init(void);
```

19.24. **BTstack Linked List API.**

```
typedef struct btstack_linked_item {
    struct btstack_linked_item *next; // <-- next element in list,
        or NULL
} btstack_linked_item_t;

typedef btstack_linked_item_t * btstack_linked_list_t;

typedef struct {
    int advance_on_next;
    btstack_linked_item_t * prev;   // points to the item before the
         current one
    btstack_linked_item_t * curr;   // points to the current item (
        to detect item removal)
} btstack_linked_list_iterator_t;


// test if list is empty
int                       btstack_linked_list_empty(
   btstack_linked_list_t * list);
// add item to list as first element
void                      btstack_linked_list_add(
   btstack_linked_list_t * list, btstack_linked_item_t *item);
// add item to list as last element
void                      btstack_linked_list_add_tail(
   btstack_linked_list_t * list, btstack_linked_item_t *item);
// pop (get + remove) first element
btstack_linked_item_t * btstack_linked_list_pop(
   btstack_linked_list_t * list);
// remove item from list
int                       btstack_linked_list_remove(
   btstack_linked_list_t * list, btstack_linked_item_t *item);
// get first element
btstack_linked_item_t * btstack_linked_list_get_first_item(
   btstack_linked_list_t * list);
// find the last item in the list
btstack_linked_item_t * btstack_linked_list_get_last_item(
   btstack_linked_list_t * list);

/**
 * @brief Counts number of items in list
 * @returns number of items in list
 */
int btstack_linked_list_count(btstack_linked_list_t * list);

//
// iterator for linked lists. allows to remove current element.
// robust against removal of current element by
    btstack_linked_list_remove.
//
void            btstack_linked_list_iterator_init(
    btstack_linked_list_iterator_t * it, btstack_linked_list_t *
    list);
int             btstack_linked_list_iterator_has_next(
    btstack_linked_list_iterator_t * it);
btstack_linked_item_t * btstack_linked_list_iterator_next(
    btstack_linked_list_iterator_t * it);
void            btstack_linked_list_iterator_remove(
    btstack_linked_list_iterator_t * it);
```

19.25. Run Loop API.

```
/**
 * @brief Init main run loop. Must be called before any other run
     loop call.
 *
 * Use btstack_run_loop_$(btstack_run_loop_TYPE)_get_instance() from
      btstack_run_loop_$(btstack_run_loop_TYPE).h to get instance
 */
void btstack_run_loop_init(const btstack_run_loop_t * run_loop);

/**
 * @brief Set timer based on current time in milliseconds.
 */
void btstack_run_loop_set_timer(btstack_timer_source_t * ts,
    uint32_t timeout_in_ms);

/**
 * @brief Set callback that will be executed when timer expires.
 */
void btstack_run_loop_set_timer_handler(btstack_timer_source_t * ts,
     void (*process)(btstack_timer_source_t *_ts));

/**
 * @brief Set context for this timer
 */
void btstack_run_loop_set_timer_context(btstack_timer_source_t * ts,
     void * context);

/**
 * @brief Get context for this timer
 */
void * btstack_run_loop_get_timer_context(btstack_timer_source_t *
    ts);

/**
 * @brief Add timer source.
 */
void btstack_run_loop_add_timer(btstack_timer_source_t * timer);
```

```
/**
 * @brief Remove timer source.
 */
int  btstack_run_loop_remove_timer(btstack_timer_source_t * timer);

/**
 * @brief Get current time in ms
 * @note 32-bit ms counter will overflow after approx. 52 days
 */
uint32_t btstack_run_loop_get_time_ms(void);

/**
 * @brief Set data source callback.
 */
void btstack_run_loop_set_data_source_handler(btstack_data_source_t
    * data_source, void (*process)(btstack_data_source_t *_ds,
    btstack_data_source_callback_type_t callback_type));

/**
 * @brief Set data source file descriptor.
 * @param data_source
 * @param fd file descriptor
 * @note No effect if port doensn't have file descriptors
 */
void btstack_run_loop_set_data_source_fd(btstack_data_source_t *
    data_source, int fd);

/**
 * @brief Get data source file descriptor.
 * @param data_source
 */
int btstack_run_loop_get_data_source_fd(btstack_data_source_t *
    data_source);

/**
 * @brief Enable callbacks for a data source
 * @param data_source to remove
 * @param callback types to enable
 */
void btstack_run_loop_enable_data_source_callbacks(
    btstack_data_source_t * data_source, uint16_t callbacks);

/**
 * @brief Enable callbacks for a data source
 * @param data_source to remove
 * @param callback types to disable
 */
void btstack_run_loop_disable_data_source_callbacks(
    btstack_data_source_t * data_source, uint16_t callbacks);

/**
 * @brief Add data source to run loop
 * @param data_source to add
 */
```

```
void btstack_run_loop_add_data_source(btstack_data_source_t *
    data_source);

/**
 * @brief Remove data source from run loop
 * @param data_source to remove
 */
int btstack_run_loop_remove_data_source(btstack_data_source_t *
    data_source);

/**
 * @brief Execute configured run loop. This function does not return
     .
 */
void btstack_run_loop_execute(void);
```

19.26. Common Utils API.

```
/**
 * @brief Minimum function for uint32_t
 * @param a
 * @param b
 * @return value
 */
uint32_t btstack_min(uint32_t a, uint32_t b);

/**
 * @brief Maximum function for uint32_t
 * @param a
 * @param b
 * @return value
 */
uint32_t btstack_max(uint32_t a, uint32_t b);


/**
 * @brief Read 16/24/32 bit little endian value from buffer
 * @param buffer
 * @param position in buffer
 * @return value
 */
uint16_t little_endian_read_16(const uint8_t * buffer, int position)
    ;
uint32_t little_endian_read_24(const uint8_t * buffer, int position)
    ;
uint32_t little_endian_read_32(const uint8_t * buffer, int position)
    ;

/**
 * @brief Write 16/32 bit little endian value into buffer
 * @param buffer
 * @param position in buffer
 * @param value
 */
void little_endian_store_16(uint8_t *buffer, uint16_t position,
    uint16_t value);
void little_endian_store_32(uint8_t *buffer, uint16_t position,
    uint32_t value);

/**
 * @brief Read 16/24/32 bit big endian value from buffer
 * @param buffer
 * @param position in buffer
 * @return value
 */
uint32_t big_endian_read_16( const uint8_t * buffer, int pos);
uint32_t big_endian_read_24( const uint8_t * buffer, int pos);
uint32_t big_endian_read_32( const uint8_t * buffer, int pos);

/**
 * @brief Write 16/32 bit big endian value into buffer
 * @param buffer
 * @param position in buffer
 * @param value
 */
void big_endian_store_16(uint8_t *buffer, uint16_t pos, uint16_t
    value);
void big_endian_store_24(uint8_t *buffer, uint16_t pos, uint32_t
    value);
void big_endian_store_32(uint8_t *buffer, uint16_t pos, uint32_t
    value);


/**
 * @brief Swap bytes in 16 bit integer
 */
static inline uint16_t btstack_flip_16(uint16_t value){
    return (uint16_t)((value & 0xff) << 8) | (value >> 8);
}

/**
 * @brief Check for big endian system
 * @returns 1 if on big endian
 */
static inline int btstack_is_big_endian(void){
    uint16_t sample = 0x0100;
    return *(uint8_t*) &sample;
}

/**
 * @brief Check for little endian system
 * @returns 1 if on little endian
 */
static inline int btstack_is_little_endian(void){
    uint16_t sample = 0x0001;
    return *(uint8_t*) &sample;
}
```

```
/**
 * @brief Copy from source to destination and reverse byte order
 * @param src
 * @param dest
 * @param len
 */
void reverse_bytes  (const uint8_t *src, uint8_t * dest, int len);

/**
 * @brief Wrapper around reverse_bytes for common buffer sizes
 * @param src
 * @param dest
 */
void reverse_24 (const uint8_t *src, uint8_t * dest);
void reverse_48 (const uint8_t *src, uint8_t * dest);
void reverse_56 (const uint8_t *src, uint8_t * dest);
void reverse_64 (const uint8_t *src, uint8_t * dest);
void reverse_128(const uint8_t *src, uint8_t * dest);
void reverse_256(const uint8_t *src, uint8_t * dest);

void reverse_bd_addr(const bd_addr_t src, bd_addr_t dest);

/**
 * @brief ASCII character for 4-bit nibble
 * @return character
 */
char char_for_nibble(int nibble);

/**
 * @brif 4-bit nibble from ASCII character
 * @return value
 */
int nibble_for_char(char c);

/**
 * @brief Compare two Bluetooth addresses
 * @param a
 * @param b
 * @return true if equal
 */
int bd_addr_cmp(const bd_addr_t a, const bd_addr_t b);

/**
 * @brief Copy Bluetooth address
 * @param dest
 * @param src
 */
void bd_addr_copy(bd_addr_t dest, const bd_addr_t src);

/**
 * @brief Use printf to write hexdump as single line of data
 */
void printf_hexdump(const void *data, int size);
```

```
/**
 * @brief Create human readable representation for UUID128
 * @note uses fixed global buffer
 * @return pointer to UUID128 string
 */
char * uuid128_to_str(const uint8_t * uuid);

/**
 * @brief Create human readable represenationt of Bluetooth address
 * @note uses fixed global buffer
 * @return pointer to Bluetooth address string
 */
char * bd_addr_to_str(const bd_addr_t addr);

/**
 * @brief Parse Bluetooth address
 * @param address_string
 * @param buffer for parsed address
 * @return 1 if string was parsed successfully
 */
int sscanf_bd_addr(const char * addr_string, bd_addr_t addr);

/**
 * @brief Constructs UUID128 from 16 or 32 bit UUID using Bluetooth
     base UUID
 * @param uuid128 output buffer
 * @param short_uuid
 */
void uuid_add_bluetooth_prefix(uint8_t * uuid128, uint32_t
    short_uuid);

/**
 * @brief Checks if UUID128 has Bluetooth base UUID prefix
 * @param uui128 to test
 * @return 1 if it can be expressed as UUID32
 */
int  uuid_has_bluetooth_prefix(const uint8_t * uuid128);

/**
 * @brief Parse unsigned number
 * @param str to parse
 * @return value
 */
uint32_t btstack_atoi(const char *str);
```

19.27. GAP API.

```
// Classic + LE

/**
 * @brief Disconnect connection with handle
 * @param handle
 */
uint8_t gap_disconnect(hci_con_handle_t handle);

/**
 * @brief Get connection type
 * @param con_handle
 * @result connection_type
 */
gap_connection_type_t gap_get_connection_type(hci_con_handle_t
    connection_handle);

// Classic

/**
 * @brief Sets local name.
 * @note has to be done before stack starts up
 * @note Default name is 'BTstack 00:00:00:00:00:00'
 * @note '00:00:00:00:00:00' in local_name will be replaced with
     actual bd addr
 * @param name is not copied, make sure memory is accessible during
     stack startup
 */
void gap_set_local_name(const char * local_name);

/**
 * @brief Set Extended Inquiry Response data
 * @note has to be done before stack starts up
 * @note If not set, local name will be used for EIR data (see
     gap_set_local_name)
 * @note '00:00:00:00:00:00' in local_name will be replaced with
     actual bd addr
 * @param eir_data size 240 bytes, is not copied make sure memory is
      accessible during stack startup
 */
void gap_set_extended_inquiry_response(const uint8_t * data);

/**
 * @brief Set class of device that will be set during Bluetooth init
     .
 * @note has to be done before stack starts up
 */
void gap_set_class_of_device(uint32_t class_of_device);

/**
 * @brief Enable/disable bonding. Default is enabled.
 * @param enabled
 */
void gap_set_bondable_mode(int enabled);

/**
 * @brief Get bondable mode.
 * @return 1 if bondable
 */
int gap_get_bondable_mode(void);
```

```
/* Configure Secure Simple Pairing */

/**
 * @brief Enable will enable SSP during init.
 */
void gap_ssp_set_enable(int enable);

/**
 * @brief Set IO Capability. BTstack will return capability to SSP
     requests
 */
void gap_ssp_set_io_capability(int ssp_io_capability);

/**
 * @brief Set Authentication Requirements using during SSP
 */
void gap_ssp_set_authentication_requirement(int
    authentication_requirement);

/**
 * @brief If set, BTstack will confirm a numeric comparison and
     enter '000000' if requested.
 */
void gap_ssp_set_auto_accept(int auto_accept);

/**
 * @brief Start dedicated bonding with device. Disconnect after
     bonding.
 * @param device
 * @param request MITM protection
 * @return error, if max num acl connections active
 * @result GAP_DEDICATED_BONDING_COMPLETE
 */
int gap_dedicated_bonding(bd_addr_t device, int
    mitm_protection_required);

gap_security_level_t gap_security_level_for_link_key_type(
    link_key_type_t link_key_type);
gap_security_level_t gap_security_level(hci_con_handle_t con_handle)
    ;

void gap_request_security_level(hci_con_handle_t con_handle,
    gap_security_level_t level);

int  gap_mitm_protection_required_for_security_level(
    gap_security_level_t level);

// LE

/**
 * @brief Set parameters for LE Scan
 */
```

```
void gap_set_scan_parameters(uint8_t scan_type, uint16_t
    scan_interval, uint16_t scan_window);

/**
 * @brief Start LE Scan
 */
void gap_start_scan(void);

/**
 * @brief Stop LE Scan
 */
void gap_stop_scan(void);

/**
 * @brief Enable privacy by using random addresses
 * @param random_address_type to use (incl. OFF)
 */
void gap_random_address_set_mode(gap_random_address_type_t
    random_address_type);

/**
 * @brief Get privacy mode
 */
gap_random_address_type_t gap_random_address_get_mode(void);

/**
 * @brief Sets update period for random address
 * @param period_ms in ms
 */
 void gap_random_address_set_update_period(int period_ms);

/**
 * @brief Sets a fixed random address for advertising
 * @param addr
 * @note Sets random address mode to type off
 */
void gap_random_address_set(bd_addr_t addr);

/**
 * @brief Set Advertisement Data
 * @param advertising_data_length
 * @param advertising_data (max 31 octets)
 * @note data is not copied, pointer has to stay valid
 * @note '00:00:00:00:00:00' in advertising_data will be replaced
     with actual bd addr
 */
void gap_advertisements_set_data(uint8_t advertising_data_length,
    uint8_t * advertising_data);

/**
 * @brief Set Advertisement Paramters
 * @param adv_int_min
 * @param adv_int_max
 * @param adv_type
 * @param direct_address_type
 * @param direct_address
 * @param channel_map
 * @param filter_policy
 * @note own_address_type is used from gap_random_address_set_mode
 */
void gap_advertisements_set_params(uint16_t adv_int_min, uint16_t
    adv_int_max, uint8_t adv_type,
     uint8_t direct_address_typ, bd_addr_t direct_address, uint8_t
        channel_map, uint8_t filter_policy);

/**
 * @brief Enable/Disable Advertisements. OFF by default.
 * @param enabled
 */
void gap_advertisements_enable(int enabled);

/**
 * @brief Set Scan Response Data
 *
 * @note For scan response data, scannable undirected advertising (
    ADV_SCAN_IND) need to be used
 *
 * @param advertising_data_length
 * @param advertising_data (max 31 octets)
 * @note data is not copied, pointer has to stay valid
 * @note '00:00:00:00:00:00' in scan_response_data will be replaced
    with actual bd addr
 */
void gap_scan_response_set_data(uint8_t scan_response_data_length,
    uint8_t * scan_response_data);

/**
 * @brief Set connection parameters for outgoing connections
 * @param conn_interval_min (unit: 1.25ms), default: 10 ms
 * @param conn_interval_max (unit: 1.25ms), default: 30 ms
 * @param conn_latency, default: 4
 * @param supervision_timeout (unit: 10ms), default: 720 ms
 * @param min_ce_length (unit: 0.625ms), default: 10 ms
 * @param max_ce_length (unit: 0.625ms), default: 30 ms
 */

void gap_set_connection_parameters(uint16_t conn_interval_min,
    uint16_t conn_interval_max,
     uint16_t conn_latency, uint16_t supervision_timeout, uint16_t
        min_ce_length, uint16_t max_ce_length);

/**
 * @brief Request an update of the connection parameter for a given
    LE connection
 * @param handle
 * @param conn_interval_min (unit: 1.25ms)
 * @param conn_interval_max (unit: 1.25ms)
 * @param conn_latency
 * @param supervision_timeout (unit: 10ms)
 * @returns 0 if ok
 */
int gap_request_connection_parameter_update(hci_con_handle_t
    con_handle, uint16_t conn_interval_min,
     uint16_t conn_interval_max, uint16_t conn_latency, uint16_t
        supervision_timeout);

/**
 * @brief Updates the connection parameters for a given LE
     connection
 * @param handle
 * @param conn_interval_min (unit: 1.25ms)
 * @param conn_interval_max (unit: 1.25ms)
 * @param conn_latency
 * @param supervision_timeout (unit: 10ms)
 * @returns 0 if ok
 */
int gap_update_connection_parameters(hci_con_handle_t con_handle,
    uint16_t conn_interval_min,
     uint16_t conn_interval_max, uint16_t conn_latency, uint16_t
        supervision_timeout);

/**
 * @brief Set accepted connection parameter range
 * @param range
 */
void gap_get_connection_parameter_range(
    le_connection_parameter_range_t * range);

/**
 * @brief Get accepted connection parameter range
 * @param range
 */
void gap_set_connection_parameter_range(
    le_connection_parameter_range_t * range);

/**
 * @brief Connect to remote LE device
 */
uint8_t gap_connect(bd_addr_t addr, bd_addr_type_t addr_type);

/**
 * @brief Cancel connection process initiated by gap_connect
 */
uint8_t gap_connect_cancel(void);

/**
 * @brief Auto Connection Establishment - Start Connecting to device
 * @param address_typ
 * @param address
 * @returns 0 if ok
 */
```

```
int gap_auto_connection_start(bd_addr_type_t address_typ, bd_addr_t
    address);

/**
 * @brief Auto Connection Establishment - Stop Connecting to device
 * @param address_typ
 * @param address
 * @returns 0 if ok
 */
int gap_auto_connection_stop(bd_addr_type_t address_typ, bd_addr_t
    address);

/**
 * @brief Auto Connection Establishment - Stop everything
 * @note  Convenience function to stop all active auto connection
      attempts
 */
void gap_auto_connection_stop_all(void);

// Classic

/**
 * @brief Override page scan mode. Page scan mode enabled by l2cap
      when services are registered
 * @note Might be used to reduce power consumption while Bluetooth
      module stays powered but no (new)
 *        connections are expected
 */
void gap_connectable_control(uint8_t enable);

/**
 * @brief Allows to control if device is discoverable. OFF by
      default.
 */
void gap_discoverable_control(uint8_t enable);

/**
 * @brief Gets local address.
 */
void gap_local_bd_addr(bd_addr_t address_buffer);

/**
 * @brief Deletes link key for remote device with baseband address.
 * @param addr
 */
void gap_drop_link_key_for_bd_addr(bd_addr_t addr);

/**
 * @brief Store link key for remote device with baseband address
 * @param addr
 * @param link_key
 * @param link_key_type
 */
```

```
void gap_store_link_key_for_bd_addr(bd_addr_t addr, link_key_t
    link_key, link_key_type_t type);

/**
 * @brief Start GAP Classic Inquiry
 * @param duration in 1.28s units
 * @return 0 if ok
 * @events: GAP_EVENT_INQUIRY_RESULT, GAP_EVENT_INQUIRY_COMPLETE
 */
int gap_inquiry_start(uint8_t duration_in_1280ms_units);

/**
 * @brief Stop GAP Classic Inquiry
 * @brief Stop GAP Classic Inquiry
 * @returns 0 if ok
 * @events: GAP_EVENT_INQUIRY_COMPLETE
 */
int gap_inquiry_stop(void);

/**
 * @brief Remote Name Request
 * @param addr
 * @param page_scan_repetition_mode
 * @param clock_offset only used when bit 15 is set - pass 0 if not
     known
 * @events: HCI_EVENT_REMOTE_NAME_REQUEST_COMPLETE
 */
int gap_remote_name_request(bd_addr_t addr, uint8_t
    page_scan_repetition_mode, uint16_t clock_offset);

/**
 * @brief Legacy Pairing Pin Code Response
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_pin_code_response(bd_addr_t addr, const char * pin);

/**
 * @brief Abort Legacy Pairing
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_pin_code_negative(bd_addr_t addr);

/**
 * @brief SSP Passkey Response
 * @param addr
 * @param passkey [0..999999]
 * @return 0 if ok
 */
int gap_ssp_passkey_response(bd_addr_t addr, uint32_t passkey);
```

```
/**
 * @brief Abort SSP Passkey Entry/Pairing
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_ssp_passkey_negative(bd_addr_t addr);

/**
 * @brief Accept SSP Numeric Comparison
 * @param addr
 * @param passkey
 * @return 0 if ok
 */
int gap_ssp_confirmation_response(bd_addr_t addr);

/**
 * @brief Abort SSP Numeric Comparison/Pairing
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_ssp_confirmation_negative(bd_addr_t addr);



// LE

/**
 * @brief Get own addr type and address used for LE
 */
void gap_le_get_own_address(uint8_t * addr_type, bd_addr_t addr);
```

19.28. HCI API.

```
// HCI init and configuration



/**
 * @brief Set up HCI. Needs to be called before any other function.
 */
void hci_init(const hci_transport_t *transport, const void *config);

/**
 * @brief Configure Bluetooth chipset driver. Has to be called
     before power on, or right after receiving the local version
     information.
 */
void hci_set_chipset(const btstack_chipset_t *chipset_driver);

/**
 * @brief Configure Bluetooth hardware control. Has to be called
     before power on.
 */
void hci_set_control(const btstack_control_t *hardware_control);

/**
 * @brief Configure Bluetooth hardware control. Has to be called
     before power on.
 */
void hci_set_link_key_db(btstack_link_key_db_t const * link_key_db);

/**
 * @brief Set callback for Bluetooth Hardware Error
 */
void hci_set_hardware_error_callback(void (*fn)(uint8_t error));

/**
 * @brief Set Public BD ADDR - passed on to Bluetooth chipset during
      init if supported in bt_control_h
 */
void hci_set_bd_addr(bd_addr_t addr);

/**
 * @brief Configure Voice Setting for use with SCO data in HSP/HFP
 */
void hci_set_sco_voice_setting(uint16_t voice_setting);

/**
 * @brief Get SCO Voice Setting
 * @return current voice setting
 */
uint16_t hci_get_sco_voice_setting(void);

/**
 * @brief Set inquiry mode: standard, with RSSI, with RSSI +
     Extended Inquiry Results. Has to be called before power on.
 * @param inquriy_mode see bluetooth_defines.h
 */
void hci_set_inquiry_mode(inquiry_mode_t mode);

/**
 * @brief Requests the change of BTstack power mode.
 */
int  hci_power_control(HCI_POWER_MODE mode);

/**
 * @brief Shutdown HCI
 */
void hci_close(void);


// Callback registration


/**
 * @brief Add event packet handler.
 */
void hci_add_event_handler(btstack_packet_callback_registration_t *
    callback_handler);

/**
 * @brief Registers a packet handler for ACL data. Used by L2CAP
 */
void hci_register_acl_packet_handler(btstack_packet_handler_t
    handler);

/**
 * @brief Registers a packet handler for SCO data. Used for HSP and
     HFP profiles.
 */
void hci_register_sco_packet_handler(btstack_packet_handler_t
    handler);


// Sending HCI Commands

/**
 * @brief Check if CMD packet can be sent to controller
 */
int hci_can_send_command_packet_now(void);

/**
 * @brief Creates and sends HCI command packets based on a template
     and a list of parameters. Will return error if outgoing data
     buffer is occupied.
 */
int hci_send_cmd(const hci_cmd_t *cmd, ...);


// Sending SCO Packets

/** @brief Get SCO packet length for current SCO Voice setting
 *  @note  Using SCO packets of the exact length is required for USB
      transfer
 *  @return Length of SCO packets in bytes (not audio frames) incl.
     3 byte header
 */
int hci_get_sco_packet_length(void);

/**
 * @brief Request emission of HCI_EVENT_SCO_CAN_SEND_NOW as soon as
     possible
 * @note HCI_EVENT_SCO_CAN_SEND_NOW might be emitted during call to
     this function
 *        so packet handler should be ready to handle it
 */
void hci_request_sco_can_send_now_event(void);

/**
```

```
 * @brief Check HCI packet buffer and if SCO packet can be sent to
     controller
 */
int hci_can_send_sco_packet_now(void);

/**
 * @brief Check if SCO packet can be sent to controller
 */
int hci_can_send_prepared_sco_packet_now(void);

/**
 * @brief Send SCO packet prepared in HCI packet buffer
 */
int hci_send_sco_packet_buffer(int size);


// Outgoing packet buffer, also used for SCO packets
// see hci_can_send_prepared_sco_packet_now amn
    hci_send_sco_packet_buffer

/**
 * Reserves outgoing packet buffer.
 * @return 1 on success
 */
int hci_reserve_packet_buffer(void);

/**
 * Get pointer for outgoing packet buffer
 */
uint8_t* hci_get_outgoing_packet_buffer(void);

/**
 * Release outgoing packet buffer\
 * @note only called instead of hci_send_preparared
 */
void hci_release_packet_buffer(void);
```

19.29. HCI Logging API.

```
typedef enum {
    HCI_DUMP_BLUEZ = 0,
    HCI_DUMP_PACKETLOGGER,
    HCI_DUMP_STDOUT
} hci_dump_format_t;

/*
 * @brief
 */
void hci_dump_open(const char *filename, hci_dump_format_t format);

/*
 * @brief
 */
```

```
void hci_dump_set_max_packets(int packets); // -1 for unlimited

/*
 * @brief
 */
void hci_dump_packet(uint8_t packet_type, uint8_t in, uint8_t *
    packet, uint16_t len);

/*
 * @brief
 */
void hci_dump_log(int log_level, const char * format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 2, 3)));
#endif
;

/*
 * @brief
 */
void hci_dump_enable_log_level(int log_level, int enable);

/*
 * @brief
 */
void hci_dump_close(void);
```

19.30. **HCI Transport API.**

```
/* HCI packet types */
typedef struct {
    /**
     * transport name
     */
    const char * name;

    /**
     * init transport
     * @param transport_config
     */
    void   (*init) (const void *transport_config);

    /**
     * open transport connection
     */
    int    (*open)(void);

    /**
     * close transport connection
     */
    int    (*close)(void);

    /**
```

```
     * register packet handler for HCI packets: ACL, SCO, and Events
     */
    void    (*register_packet_handler)(void (*handler)(uint8_t
       packet_type, uint8_t *packet, uint16_t size));

    /**
     * support async transport layers, e.g. IRQ driven without
        buffers
     */
    int     (*can_send_packet_now)(uint8_t packet_type);

    /**
     * send packet
     */
    int     (*send_packet)(uint8_t packet_type, uint8_t *packet, int
       size);

    /**
     * extension for UART transport implementations
     */
    int     (*set_baudrate)(uint32_t baudrate);

    /**
     * extension for UART H5 on CSR: reset BCSP/H5 Link
     */
    void    (*reset_link)(void);

    /**
     * extension for USB transport implementations: config SCO
        connections
     */
    void    (*set_sco_config)(uint16_t voice_setting, int
       num_connections);

} hci_transport_t;

typedef enum {
    HCI_TRANSPORT_CONFIG_UART,
    HCI_TRANSPORT_CONFIG_USB
} hci_transport_config_type_t;

typedef struct {
    hci_transport_config_type_t type;
} hci_transport_config_t;

typedef struct {
    hci_transport_config_type_t type; // ==
       HCI_TRANSPORT_CONFIG_UART
    uint32_t   baudrate_init; // initial baud rate
    uint32_t   baudrate_main; // = 0: same as initial baudrate
    int        flowcontrol;   //
    const char *device_name;
} hci_transport_config_uart_t;
```

```
// inline various hci_transport_X.h files

/*
 * @brief Setup H4 instance with uart_driver
 * @param uart_driver to use
 */
const hci_transport_t * hci_transport_h4_instance(const
    btstack_uart_block_t * uart_driver);

/*
 * @brief Setup H5 instance with uart_driver
 * @param uart_driver to use
 */
const hci_transport_t * hci_transport_h5_instance(const
    btstack_uart_block_t * uart_driver);

/*
 * @brief Enable H5 Low Power Mode: enter sleep mode after x ms of
     inactivity
 * @param inactivity_timeout_ms or 0 for off
 */
void hci_transport_h5_set_auto_sleep(uint16_t inactivity_timeout_ms)
    ;

/*
 * @brief Enable BSCP mode H5, by enabling event parity
 */
void hci_transport_h5_enable_bcsp_mode(void);

/*
 * @brief
 */
const hci_transport_t * hci_transport_usb_instance(void);

/**
 * @brief Specify USB Bluetooth device via port numbers from root to
      device
 */
void hci_transport_usb_set_path(int len, uint8_t * port_numbers);
```

19.31. **L2CAP API.**

```
/**
 * @brief Set up L2CAP and register L2CAP with HCI layer.
 */
void l2cap_init(void);

/**
 * @brief Registers packet handler for LE Connection Parameter
     Update events
 */
```

```
void l2cap_register_packet_handler(void (*handler)(uint8_t
    packet_type, uint16_t channel, uint8_t *packet, uint16_t size));

/**
 * @brief Get max MTU for Classic connections based on btstack
      configuration
 */
uint16_t l2cap_max_mtu(void);

/**
 * @brief Get max MTU for LE connections based on btstack
      configuration
 */
uint16_t l2cap_max_le_mtu(void);

/**
 * @brief Creates L2CAP channel to the PSM of a remote device with
      baseband address. A new baseband connection will be initiated
      if necessary.
 * @param packet_handler
 * @param address
 * @param psm
 * @param mtu
 * @param local_cid
 * @return status
 */
uint8_t l2cap_create_channel(btstack_packet_handler_t packet_handler
    , bd_addr_t address, uint16_t psm, uint16_t mtu, uint16_t *
    out_local_cid);

/**
 * @brief Creates L2CAP channel to the PSM of a remote device with
      baseband address using Enhanced Retransmission Mode.
 *        A new baseband connection will be initiated if necessary.
 * @param packet_handler
 * @param address
 * @param psm
 * @param ertm_config
 * @param buffer to store reassembled rx packet, out-of-order
      packets and unacknowledged outgoing packets with their
      tretransmission timers
 * @param size of buffer
 * @param local_cid
 * @return status
 */
uint8_t l2cap_create_ertm_channel(btstack_packet_handler_t
    packet_handler, bd_addr_t address, uint16_t psm,
     l2cap_ertm_config_t * ertm_contig, uint8_t * buffer, uint32_t
        size, uint16_t * out_local_cid);

/**
 * @brief Disconnects L2CAP channel with given identifier.
 */
void l2cap_disconnect(uint16_t local_cid, uint8_t reason);
```

```
/**
 * @brief Queries the maximal transfer unit (MTU) for L2CAP channel
     with given identifier.
 */
uint16_t l2cap_get_remote_mtu_for_local_cid(uint16_t local_cid);

/**
 * @brief Sends L2CAP data packet to the channel with given
     identifier.
 */
int l2cap_send(uint16_t local_cid, uint8_t *data, uint16_t len);

/**
 * @brief Registers L2CAP service with given PSM and MTU, and
     assigns a packet handler.
 */
uint8_t l2cap_register_service(btstack_packet_handler_t
    packet_handler, uint16_t psm, uint16_t mtu, gap_security_level_t
     security_level);

/**
 * @brief Unregisters L2CAP service with given PSM.
 */
uint8_t l2cap_unregister_service(uint16_t psm);

/**
 * @brief Accepts incoming L2CAP connection.
 */
void l2cap_accept_connection(uint16_t local_cid);

/**
 * @brief Accepts incoming L2CAP connection for Enhanced
     Retransmission Mode
 * @param local_cid
 * @param ertm_config
 * @param buffer to store reassembled rx packet, out-of-order
     packets and unacknowledged outgoing packets with their
     tretransmission timers
 * @param size of buffer
 * @return status
 */
uint8_t l2cap_accept_ertm_connection(uint16_t local_cid,
    l2cap_ertm_config_t * ertm_contig, uint8_t * buffer, uint32_t
    size);

/**
 * @brief Deny incoming L2CAP connection.
 */
void l2cap_decline_connection(uint16_t local_cid);

/**
 * @brief Check if outgoing buffer is available and that there's
     space on the Bluetooth module
 */
int  l2cap_can_send_packet_now(uint16_t local_cid);

/**
 * @brief Request emission of L2CAP_EVENT_CAN_SEND_NOW as soon as
     possible
 * @note L2CAP_EVENT_CAN_SEND_NOW might be emitted during call to
     this function
 *       so packet handler should be ready to handle it
 * @param local_cid
 */
void l2cap_request_can_send_now_event(uint16_t local_cid);

/**
 * @brief Reserve outgoing buffer
 */
int  l2cap_reserve_packet_buffer(void);

/**
 * @brief Get outgoing buffer and prepare data.
 */
uint8_t *l2cap_get_outgoing_buffer(void);

/**
 * @brief Send L2CAP packet prepared in outgoing buffer to channel
 */
int l2cap_send_prepared(uint16_t local_cid, uint16_t len);

/**
 * @brief Release outgoing buffer (only needed if
     l2cap_send_prepared is not called)
 */
void l2cap_release_packet_buffer(void);


//
// LE Connection Oriented Channels feature with the LE Credit Based
    Flow Control Mode == LE Data Channel
//


/**
 * @brief Register L2CAP LE Data Channel service
 * @note MTU and initial credits are specified in
     l2cap_le_accept_connection(..) call
 * @param packet_handler
 * @param psm
 * @param security_level
 */
uint8_t l2cap_le_register_service(btstack_packet_handler_t
    packet_handler, uint16_t psm, gap_security_level_t
    security_level);

/**
```

```
 * @brief Unregister L2CAP LE Data Channel service
 * @param psm
 */

uint8_t l2cap_le_unregister_service(uint16_t psm);

/*
 * @brief Accept incoming LE Data Channel connection
 * @param local_cid              L2CAP LE Data Channel Identifier
 * @param receive_buffer         buffer used for reassembly of L2CAP
    LE Information Frames into service data unit (SDU) with given
    MTU
 * @param receive_buffer_size    buffer size equals MTU
 * @param initial_credits        Number of initial credits provided
    to peer or L2CAP_LE_AUTOMATIC_CREDITS to enable automatic
    credits
 */

uint8_t l2cap_le_accept_connection(uint16_t local_cid, uint8_t *
   receive_sdu_buffer, uint16_t mtu, uint16_t initial_credits);

/**
 * @brief Deny incoming LE Data Channel connection due to resource
    constraints
 * @param local_cid              L2CAP LE Data Channel Identifier
 */

uint8_t l2cap_le_decline_connection(uint16_t local_cid);

/**
 * @brief Create LE Data Channel
 * @param packet_handler         Packet handler for this connection
 * @param con_handle             ACL-LE HCI Connction Handle
 * @param psm                    Service PSM to connect to
 * @param receive_buffer         buffer used for reassembly of L2CAP
    LE Information Frames into service data unit (SDU) with given
    MTU
 * @param receive_buffer_size    buffer size equals MTU
 * @param initial_credits        Number of initial credits provided
    to peer or L2CAP_LE_AUTOMATIC_CREDITS to enable automatic
    credits
 * @param security_level         Minimum required security level
 * @param out_local_cid          L2CAP LE Channel Identifier is
    stored here
 */
uint8_t l2cap_le_create_channel(btstack_packet_handler_t
   packet_handler, hci_con_handle_t con_handle,
    uint16_t psm, uint8_t * receive_sdu_buffer, uint16_t mtu,
        uint16_t initial_credits, gap_security_level_t
        security_level,
    uint16_t * out_local_cid);

/**
 * @brief Provide credtis for LE Data Channel
 * @param local_cid              L2CAP LE Data Channel Identifier
 * @param credits                Number additional credits for peer
 */
uint8_t l2cap_le_provide_credits(uint16_t cid, uint16_t credits);

/**
 * @brief Check if packet can be scheduled for transmission
 * @param local_cid              L2CAP LE Data Channel Identifier
 */
int l2cap_le_can_send_now(uint16_t cid);

/**
 * @brief Request emission of L2CAP_EVENT_LE_CAN_SEND_NOW as soon as
     possible
 * @note L2CAP_EVENT_CAN_SEND_NOW might be emitted during call to
    this function
 *        so packet handler should be ready to handle it
 * @param local_cid              L2CAP LE Data Channel Identifier
 */
uint8_t l2cap_le_request_can_send_now_event(uint16_t cid);

/**
 * @brief Send data via LE Data Channel
 * @note Since data larger then the maximum PDU needs to be
    segmented into multiple PDUs, data needs to stay valid until
    ... event
 * @param local_cid              L2CAP LE Data Channel Identifier
 * @param data                   data to send
 * @param size                   data size
 */
uint8_t l2cap_le_send_data(uint16_t cid, uint8_t * data, uint16_t
   size);

/**
 * @brief Disconnect from LE Data Channel
 * @param local_cid              L2CAP LE Data Channel Identifier
 */
uint8_t l2cap_le_disconnect(uint16_t cid);
```

20. Events and Errors

20.1. **L2CAP Events.** L2CAP events and data packets are delivered to the packet handler specified by *l2cap_register_service* resp. *l2cap_create_channel*. Data packets have the L2CAP_DATA_PACKET packet type. L2CAP provides the following events:

- L2CAP_EVENT_CHANNEL_OPENED - sent if channel establishment is done. Status not equal zero indicates an error. Possible errors: out of memory; connection terminated by local host, when the connection to remote device fails.
- L2CAP_EVENT_CHANNEL_CLOSED - emitted when channel is closed. No status information is provided.

- L2CAP_EVENT_INCOMING_CONNECTION - received when the connection is requested by remote. Connection accept and decline are performed with *l2cap_accept_connection* and *l2cap_decline_connecti-on* respectively.
- L2CAP_EVENT_CAN_SEND_NOW - Indicates that an L2CAP data packet could be sent on the reported l2cap_cid. It is emitted after a call to *l2cap_request_can_send_now*. See Sending L2CAP Data

Event	Event Code
L2CAP_EVENT_CHANNEL_OPENED	0x70
L2CAP_EVENT_CHANNEL_CLOSED	0x71
L2CAP_EVENT_INCOMING_CONNECTION	0x72
L2CAP_EVENT_CAN_SEND_NOW	0x78

Table 12: L2CAP Events.

L2CAP event paramaters, with size in bits:

- L2CAP_EVENT_CHANNEL_OPENED:
  - *event(8), len(8), status(8), address(48), handle(16), psm(16), local_cid(16), remote_cid(16), local_mtu(16), remote_mtu(16)*

- L2CAP_EVENT_CHANNEL_CLOSED:
  - *event (8), len(8), channel(16)*
- L2CAP_EVENT_INCOMING_CONNECTION:
  - *event(8), len(8), address(48), handle(16), psm (16), local_cid(16), remote_cid (16)*
- L2CAP_EVENT_CAN_SEND_NOW:
  - *event(8), len(8), local_cid(16)

20.2. **RFCOMM Events.** All RFCOMM events and data packets are currently delivered to the packet handler specified by *rfcomm_register_packet_handler*. Data packets have the *DATA*PACKET packet type. Here is the list of events provided by RFCOMM:

- RFCOMM_EVENT_INCOMING_CONNECTION - received when the connection is requested by remote. Connection accept and decline are performed with *rfcomm_accept_connection* and *rfcomm_decline_connection* respectively.
- RFCOMM_EVENT_CHANNEL_CLOSED - emitted when channel is closed. No status information is provided.
- RFCOMM_EVENT_CHANNEL_OPENED - sent if channel establishment is done. Status not equal zero indicates an error. Possible errors: an L2CAP error, out of memory.

- RFCOMM_EVENT_CAN_SEND_NOW - Indicates that an RFCOMM data packet could be sent on the reported rfcomm_cid. It is emitted after a call to *rfcomm_request_can_send_now*. See Sending RFCOMM Data

Event	Event Code
RFCOMM_EVENT_CHANNEL_OPENED	0x80
RFCOMM_EVENT_CHANNEL_CLOSED	0x81
RFCOMM_EVENT_INCOMING_CONNECTION	0x82
RFCOMM_EVENT_CAN_SEND_NOW	0x89

Table 13: RFCOMM Events.

RFCOMM event paramaters, with size in bits:

- RFCOMM_EVENT_CHANNEL_OPENED:
  - *event(8), len(8), status(8), address(48), handle(16), server_channel(8), rfcomm_cid(16), max_frame_size(16)*
- RFCOMM_EVENT_CHANNEL_CLOSED:
  - *event(8), len(8), rfcomm_cid(16)*
- RFCOMM_EVENT_INCOMING_CONNECTION:
  - *event(8), len(8), address(48), channel (8), rfcomm_cid(16)*
- RFCOMM_EVENT_CAN_SEND_NOW:
  - *event(8), len(8), rfcomm_cid(16)

Error	Error Code
BTSTACK_MEMORY_ALLOC_FAILED	0x56
BTSTACK_ACL_BUFFERS_FULL	0x57
L2CAP_COMMAND_REJECT_REASON_COMMAND_NOT_UNDERSTOOD	0x60
L2CAP_COMMAND_REJECT_REASON_SIGNALING_MTU_EXCEEDED	0x61
L2CAP_COMMAND_REJECT_REASON_INVALID_CID_IN_REQUEST	0x62
L2CAP_CONNECTION_RESPONSE_RESULT_SUCCESSFUL	0x63
L2CAP_CONNECTION_RESPONSE_RESULT_PENDING	0x64
L2CAP_CONNECTION_RESPONSE_RESULT_REFUSED_PSM	0x65
L2CAP_CONNECTION_RESPONSE_RESULT_REFUSED_SECURITY	0x66
L2CAP_CONNECTION_RESPONSE_RESULT_REFUSED_RESOURCES	0x65
L2CAP_CONFIG_RESPONSE_RESULT_SUCCESSFUL	0x66
L2CAP_CONFIG_RESPONSE_RESULT_UNACCEPTABLE_PARAMS	0x67
L2CAP_CONFIG_RESPONSE_RESULT_REJECTED	0x68

Error	Error Code
L2CAP_CONFIG_RESPONSE_RESULT_UNKNOWN_OPTIONS	0x69
L2CAP_SERVICE_ALREADY_REGISTERED	0x6a
RFCOMM_MULTIPLEXER_STOPPED	0x70
RFCOMM_NO_OUTGOING_CREDITS	0x72
SDP_HANDLE_ALREADY_REGISTERED	0x80

Table 14: Errors.

20.3. Errors.

21. Migration to v1.0

We already had two listings with the Bluetooth SIG, but no official BTstack v1.0 release. After the second listing, we decided that it's time for a major overhaul of the API - making it easier for new users.

In the following, we provide an overview of the changes and guidelines for updating an existing code base. At the end, we present a command line tool and as an alternative a Web service, that can simplify the migration to the new v1.0 API.

22. Changes

22.1. Repository structure.

22.1.1. include/btstack *folder.* The header files had been split between *src/* and *include/btstack/*. Now, the *include/* folder is gone and the headers are provided in *src/*, *src/classic/*, *src/ble/* or by the *platform/* folders. There's a new *src/btstack.h* header that includes all BTstack headers.

22.1.2. *Plural folder names.* Folder with plural names have been renamed to the singular form, examples: *doc*, *chipset*, *platform*, *tool*.

22.1.3. *ble and src folders.* The *ble* folder has been renamed into *src/ble*. Files that are relevant for using BTstack in Classic mode, have been moved to *src/-classic*. Files in *src* that are specific to individual platforms have been moved to *platform* or *port*.

22.1.4. *platform and port folders.* The *platforms* folder has been split into *platform* and "port" folders. The *port* folder contains a complete BTstack port of for a specific setup consisting of an MCU and a Bluetooth chipset. Code that didn't belong to the BTstack core in *src* have been moved to *platform* or *port* as well, e.g. *hci_transport_h4_dma.c*.

22.1.5. *Common file names.* Types with generic names like *linked_list* have been prefixed with btstack_ (e.g. btstack_linked_list.h) to avoid problems when integrating with other environments like vendor SDKs.

22.2. **Defines and event names.** All defines that originate from the Bluetooth Specification are now in *src/bluetooth.h.* Addition defines by BTstack are collected in *src/btstack_defines.h.* All events names have the form MODULE_EVENT_NAME now.

22.3. **Function names.**

- The *internal suffix has been an artifact from the iOS port. All public functions with the* internal suffix have been stripped of this suffix.
- Types with generic names like linked_list have been prefixed with btstack_ (e.g. btstack_linked_list) to avoid problems when integrating with other environments like vendor SDKs.

22.4. **Packet Handlers.** We streamlined the use of packet handlers and their types. Most callbacks are now of type *btstack_packet_handler_t* and receive a pointer to an HCI Event packet. Often a void * connection was the first argument - this has been removed.

To facilitate working with HCI Events and get rid of manually calculating offsets into packets, we're providing auto-generated getters for all fields of all events in *src/hci_event.h.* All functions are defined as static inline, so they are not wasting any program memory if not used. If used, the memory footprint should be identical to accessing the field directly via offsets into the packet. Feel free to start using these getters as needed.

22.5. **Event Forwarding.** In the past, events have been forwarded up the stack from layer to layer, with the undesired effect that an app that used both *att_server* and *security_manager* would get each HCI event twice. To fix this and other potential issues, this has been cleaned up by providing a way to register multiple listeners for HCI events. If you need to receive HCI or GAP events, you can now directly register your callback with *hci_add_event_handler.*

22.6. **util folder.** The *utils* folder has been renamed into *btstack_util* to avoid compile issues with other frameworks.

- The functions to read/store values in little/bit endian have been renamed into big/little_endian_read/write_16/24/32.
- The functions to reverse bytes swap16/24/32/48/64/128/X habe been renamed to reverse_16/24/32/48/64/128/X.

22.7. **btstack_config.h.**

- *btstack-config.h* is now *btstack_config.h*
- Defines have been sorted: HAVE_ specify features that are particular to your port. ENABLE_ features can be added as needed.
- *NO* has been replaced with *NR* for the BTstack static memory allocation, e.g., *MAX_NO_HCI_CONNECTIONS8 -> MAX_NR_HCI_CONNECTIONS*
- The #define EMBEDDED is dropped, i.e. the distinction if the API is for embedded or not has been removed.

22.8. **Linked List.**

- The file has been renamed to btstack_linked_list.
- All functions are prefixed with btstack_.
- The user data field has been removed as well.

22.9. **Run Loop.**

- The files have been renamed to *btstack_run_loop_*
- To allow for simpler integration of custom run loop implementations, *run_loop_init(...)* is now called with a pointer to a *run_loop_t* function table instead of using an enum that needs to be defined inside the BTstack sources.
- Timers now have a new context field that can be used by the user.

22.10. **HCI Setup.** In the past, hci_init(...) was called with an hci_transport_t, the transport configuration, remote_device_t and a bt_control_t objects. Besides cleaning up the types (see remote_device_db and bt_control below), hci_init only requires the hci_transport_t and it's configuration.

The used *btstack_chipset_t*, *bt_control_t*, or *link_key_db_t* can now be set with *hci_set_chipset*, *hci_set_control*, and *hci_set_link_key_db* respectively.

22.10.1. *remote_device_db.* Has been split into *src/classic/btstack_link_key_db*, *platform/daemon/btstack_device_name_db*, and *platform/daemon/rfcomm_service_db.*

22.10.2. *bt_control.* Has been split into *src/btstack_chipset.h* and *src/bstack_control.h*

22.11. **HCI / GAP.** HCI functions that are commonly placed in GAP have been moved from *src/hci.h* to *src/gap.h*

22.12. **RFCOMM.** In contrast to L2CAP, RFCOMM did not allow to register individual packet handlers for each service and outgoing connection. This has been fixed and the general *rfcomm_register_packet_handler(...)* has been removed.

22.13. **SPP Server.** The function to create an SPP SDP record has been moved into *spp_server.h*

22.14. **SDP Client.**

- SDP Query structs have been replaced by HCI events. You can use btstack_event.h to access the fields.

22.15. **SDP Server.**

- Has been renamed to *src/classic/sdp_server.h.*
- The distinction if the API is for embedded or not has been removed.

22.16. **Security Manager.**

- In all Security Manager calls that refer to an active connection, pass in the current handle instead of addr + addr type.
- All Security Manager events are now regular HCI Events instead of sm_* structs
- Multiple packet handler can be registered with *sm_add_event_handler(. . .)* similar to HCI.

22.17. **GATT Client.**

- All GATT Client events are now regular HCI Events instead of gatt_* structs.
- The subclient_id has been replaced by a complete handler for GATT Client requests

22.18. **ANCS Client.** Renamed to *src/ble/ancs_client*

22.19. **Flow control / DAEMON_EVENT_HCI_PACKET_SENT.** In BTstack, you can only send a packet with most protocols (L2CAP, RFCOMM, ATT) if the outgoing buffer is free and also per-protocol constraints are satisfied, e.g., there are outgoing credits for RFCOMM.

Before v1.0, we suggested to check with *l2cap_can_send_packet_now(..)* or *rfcomm_can_send_packet(..)* whenever an HCI event was received. This has been cleaned up and streamlined in v1.0.

Now, when there is a need to send a packet, you can call *rcomm_request_can_send_now(..)* / *l2cap_request_can_send_now(..)* to let BTstack know that you want to send a packet. As soon as it becomes possible to send a RFCOMM_EVENT_CAN_SEND_NOW/L2CAP_EV will be emitted and you can directly react on that and send a packet. After that, you can request another "can send event" if needed.

22.20. **Daemon.**

- Not tested since API migration!
- Moved into *platform/daemon/*
- Header for clients: *platform/daemon/btstack_client.h*
- Java bindings are now at *platform/daemon/bindings*

22.21. **Migration to v1.0 with a script.** To simplify the migration to the new v1.0 API, we've provided a tool in *tool/migration_to_v1.0* that fixes include path changes, handles all function renames and also updates the packet handlers to the btstack_packet_handler_t type. It also has a few rules that try to rename file names in Makefile as good as possible.

It's possible to migrate most of the provided embedded examples to v1.0. The one change that it cannot handle is the switch from structs to HCI Events for the SDP, GATT, and ANCS clients. The migration tool updates the packet handler signature to btstack_packet_handler_t, but it does not convert the field accesses to sue the appropriate getters in *btstack_event.h*. This has to be done manually, but it is straight forward. E.g., to read the field *status* of the GATT_EVENT_QUERY_COMPLETE, you call call gatt_event_query_complete_get_status(packet).

22.21.1. *Requirements.*

- bash
- sed
- Coccinelle. On Debian-based distributions, it's available via apt. On OS X, it can be installed via Homebrew.

22.21.2. *Usage.*

```
tool/migration_to_v1.0/migration.sh PATH_TO_ORIGINAL_PROJECT
    PATH_TO_MIGRATED_PROJECT
```

The tool first creates a copy of the original project and then uses sed and coccinelle to update the source files.

22.22. **Migration to v1.0 with a Web Service.** BlueKitchen GmbH is providing a web service to help migrate your sources if you don't want or cannot install Coccinelle yourself.