master-af0dac01d
December 27, 2025

BTstack Manual

Including Quickstart Guide

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

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.

#Quick Start

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

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.

Let's Go

The easiest way to try BTstack is on a regular desktop setup like macOS, Linux or Windows together with a standard USB Bluetooth Controller. Running BTstack on desktop speeds up the development cycle a lot and also provides direct access to full packet log files in cases something doesn't work as expected. The same code can then later be run unmodified on an embedded target.

For macOS and Linux, please see libusb port. For Windows, please see windows-winusb port.

Or checkout the list of existing ports

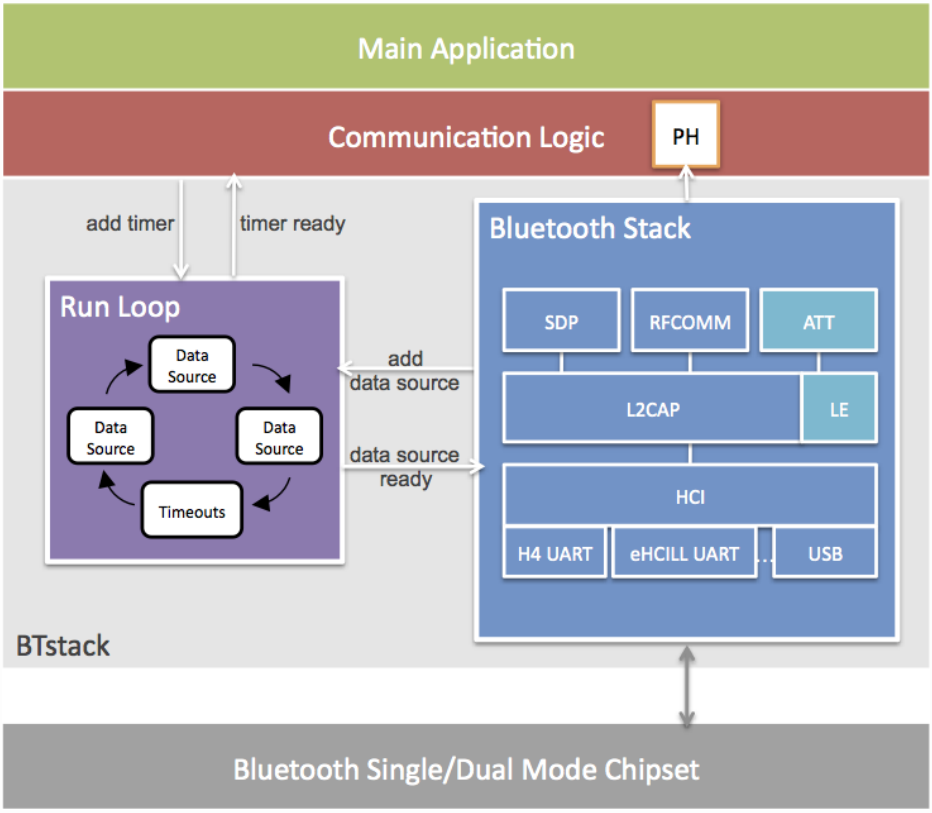
#BTstack Architecture

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 below 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.



Architecture of a BTstack-based application.

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 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.

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.

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.

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.

#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.

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.

HAVE_* directives

System properties:

#define	Description
HAVE_AES128	Use platform AES128 engine
HAVE_BTSTACK_STDIN	STDIN is available for CLI interface
HAVE_LWIP	lwIP is available
HAVE_MALLOC	Use dynamic memory
HAVE_MBEDTLS_ECC_P256	mbedTLS provides NIST P-256 operations e.g. for LE Secure Connections

Embedded platform properties:

#define	Description
HAVE_EMBEDDED_TIME_MS	System provides time in milliseconds
HAVE_EMBEDDED_TICK	System provides tick interrupt
HAVE_HAL_AUDIO	Audio HAL is available
HAVE_HAL_AUDIO_SINK_ STEREO_ONLY	Duplicate samples for mono playback

FreeRTOS platform properties:

#define	Description
HAVE_FREERTOS_ INCLUDE_PREFIX	FreeRTOS headers are in ‘freertos’ folder (e.g. ESP32’s esp-idf)

POSIX platform properties:

#define	Description
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

Chipset properties:

#define	Description
HAVE_BCM_PCM2	PCM2 is used and requires additional configuration
HAVE_BCM_PCM_NBS_16KHZ	NBS is up/downsampled, use 16 kHz sample rate for NBS

ENABLE_* directives

BTstack properties:

#define	Description
ENABLE_A2DP_EXPLICIT_CONFIG	Let application configure strea endpoint (skip auto-config of S endpoint)
ENABLE_AIROC_DOWNLOAD_MODE	Enable AIROC (newer Infineon Controller PatchRAM downloa
ENABLE_ATT_DELAYED_RESPONSE	Enable support for delayed AT operations, see GATT Server
ENABLE_AVDTP_ACCEPTOR_ EXPLICIT_START_STREAM_ CONFIRMATION	Allow accept or reject of strea on A2DP_SUBEVENT_ START_STREAM_REQUESTED
ENABLE_BCM_PCM_WBS	Enable support for Wide-Band codec in BCM controller, requir ENABLE_SCO_OVER_PCM
ENABLE_BLE	Enable BLE related code in HC L2CAP
ENABLE_CC256X_ASSISTED_HFP	

#define	Description
	Enable support for Assisted HF mode in CC256x Controller, req ENABLE_SCO_OVER_PCM
ENABLE_CC256X_BAUDRATE_ CHANGE_FLOWCONTROL_ BUG_WORKAROUND	Enable workaround for bug in CC256x Flow Control during ba rate change, see chipset docs.
ENABLE_CLASSIC	Enable Classic related code in H and L2CAP
ENABLE_CLASSIC_OOB_PAIRING	Enable support for classic Out- Band (OOB) pairing
ENABLE_CONTROLLER_ DUMP_PACKETS	Dump number of packets in Controller per type for debuggi
ENABLE_CONTROLLER_ WARM_BOOT	Enable stack startup without p cycle (if supported/possible)
ENABLE_CROSS_TRANSPORT_ KEY_DERIVATION	Enable Cross-Transport Key Derivation (CTKD) for Secure Connections
ENABLE_CYPRESS_BAUDRATE_ CHANGE_FLOWCONTROL_ BUG_WORKAROUND	Enable workaround for bug in CYW2070x Flow Control during rate change, similar to CC256x.
ENABLE_EHCILL	Enable eHCILL low power mod CC256x/WL18xx chipsets
ENABLE_EXPLICIT_BR_EDR_ SECURITY_MANAGER	Report BR/EDR Security Manag support in L2CAP Information Response
ENABLE_EXPLICIT_ CONNECTABLE_MODE_CONTROL	Disable calls to control Connec Mode by L2CAP
ENABLE_EXPLICIT_DEDICATED_ BONDING_DISCONNECT	Keep connection after dedicate bonding is complete
ENABLE_EXPLICIT_IO_ CAPABILITIES_REPLY	Let application trigger sending Capabilities (Negative) Reply
ENABLE_EXPLICIT_LINK_ KEY_REPLY	Let application trigger sending Key (Negative) Response, allow asynchronous link key lookup
ENABLE_EXPLICIT_PAIRING_ON_SECURITY_REQUEST	

#define	Description
	Let application trigger LE Pairi upon SM_EVENT_SECURITY_RE
ENABLE_GATT_CLIENT_ PAIRING	Enable GATT Client to start pai and retry operation on security
ENABLE_GATT_CLIENT_ CACHING	Enable GATT Service Client to c Characteristics in TLV
ENABLE_H5	Enable support for SLIP mode i `btstack_uart.h` drivers for HCI (‘Three-Wire Mode’)
ENABLE_HCI_COMMAND_STATUS_ DISCARDED_FOR_FAILED_ CONNECTIONS WORKAROUND	Track connection handle for HC Commands and assume comma has failed if disonnect event fo connection is received
ENABLE_HCI_CONTROLLER_ TO_HOST_FLOW_CONTROL	Enable HCI Controller to Host F Control, see below
ENABLE_HCI_SERIALIZED_ CONTROLLER_OPERATIONS	Serialize Inquiry, Remote Name Request, and Create Connection operations
ENABLE_HFP_AT_MESSAGES	Enable `HFP_SUBEVENT_AT_MESSAG` and `HFP_SUBEVENT_AT_MESSAGE_RECEI` events
ENABLE_HFP_WIDE_BAND_ PEECH	Enable support for mSBC codec in HFP profile for Wide-Band S
ENABLE_L2CAP_ENHANCED_ CREDIT_BASED_FLOW_ CONTROL_MODE	Enable Enhanced credit-based control mode for L2CAP Chann
ENABLE_L2CAP_ENHANCED_ RETRANSMISSION_MODE	Enable Enhanced Retransmissi Mode for L2CAP Channels. Mandatory for AVRCP Browsin
ENABLE_L2CAP_LE_ CREDIT_BASED_FLOW_ CONTROL_MODE	Enable LE credit-based flow-co mode for L2CAP channels
ENABLE_LE_CENTRAL	Enable support for LE Central I HCI and Security Manager

#define	Description
ENABLE_LE_DATA_ LENGTH_EXTENSION	Enable LE Data Length Extensi support
ENABLE_LE_ENHANCED_ CONNECTION_COMPLETE_EVENT	Enable LE Enhanced Connectio Complete Event v1 & v2
ENABLE_LE_EXTENDED_ ADVERTISING	Enable extended advertising a scanning
ENABLE_LE_LIMIT_ACL_ FRAGMENT_BY_MAX_OCTETS	Force HCI to fragment ACL-LE to fit into over-the-air packet
ENABLE_LE_PERIODIC_ ADVERTISING	Enable periodic advertising an scanning
ENABLE_LE_PERIPHERAL	Enable support for LE Peripher in HCI and Security Manager
ENABLE_LE_PRIVACY_ ADDRESS_RESOLUTION	Enable address resolution for resolvable private addresses in Controller
ENABLE_LE_PROACTIVE_ AUTHENTICATION	Enable automatic encryption fo bonded devices on re-connect
ENABLE_LE_SECURE_ CONNECTIONS	Enable LE Secure Connections
ENABLE_LE_SECURE_ CONNECTIONS_DEBUG_KEY	Enable support for LE Secure Connection debug keys for test
ENABLE_LE_SET_ ADV_PARAMS_ON_RANDOM_ ADDRESS_CHANGE	Send HCI LE Set Advertising Pa after HCI LE Set Random Addre workaround for Controller Bug
ENABLE_LE_SIGNED_WRITE	Enable LE Signed Writes in ATT
ENABLE_LE_WHITELIST *TOUCH_AFTER* RESOLVING_LIST_UPDATE	Enable Workaround for Contro bug
ENABLE_LOG_BTSTACK_ EVENTS	Log internal/custom BTstack ev
ENABLE_LOG_DEBUG	Enable log_debug messages
ENABLE_LOG_ERROR	Enable log_error messages
ENABLE_LOG_INFO	Enable log_info messages

#define	Description
ENABLE_MICRO_ECC_FOR_ LE_SECURE_CONNECTIONS	Use micro-ecc library for ECC operations
ENABLE_MODPLAYER	Enable HXCMOD player in btstack_audio_generator and examples
ENABLE_MUTUAL_ AUTHENTICATION_FOR_ LEGACY_SECURE_CONNECTIONS	Re-authentication after connec was encrypted to avoid BIAS A Not needed for min encryption size of 16
ENABLE_PRINTF_TO_LOG	Log printf into packet log
ENABLE_RTK_PCM_WBS	Enable support for Wide-Band codec in Realtek controller, req ENABLE_SCO_OVER_PCM
ENABLE_SCO_OVER_HCI	Enable SCO over HCI for chipse supported)
ENABLE_SCO_OVER_PCM	Enable SCO ofer PCM/I2S for ch (if supported)
ENABLE_SEGGER_RTT	Use SEGGER RTT for console ou and packet log, see additional c
ENABLE_TLV_FLASH_ EXPLICIT_DELETE_FIELD	Enable use of explicit delete fie TLV Flash implementation - rec when flash value cannot be overwritten with zero
ENABLE_TLV_FLASH_ WRITE_ONCE	Enable storing of emtpy tag ins overwriting existing tag - requi when flash value cannot be overwritten at all
ENABLE_VORBIS	Enable OGG-Vorbis support in btstack_audio_genrator and ex

Notes:

- ENABLE_MICRO_ECC_FOR_LE_SECURE_CONNECTIONS: Only some Bluetooth 4.2+ controllers (e.g., EM9304, ESP32) support the necessary HCI commands for ECC. Other reason 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, or if the ECC HCI Commands are unreliable.

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_FLOW_CONTROL. 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:

#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

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 recommended. The large ACL buffer is required for 3-DH5 packets to be used.

#define	Description
HCI_ACL_PAYLOAD_SIZE	Max size of HCI ACL payloads
HCI_ACL_CHUNK_SIZE_ALIGNMENT	Alignment of ACL chunk size, can be used to align HCI transport writes
HCI_INCOMING_PRE_BUFFER_SIZE	Number of bytes reserved before actual data for incoming HCI packets
MAX_NR_BNEP_CHANNELS	Max number of BNEP channels
MAX_NR_BNEP_SERVICES	Max number of BNEP services
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 connections
MAX_NR_L2CAP_SERVICES	Max number of L2CAP services
MAX_NR_RFCOMM_CHANNELS	Max number of RFCOMM connections
MAX_NR_RFCOMM_MULTIPLEXERS	Max number of RFCOMM multiplexers, with one multiplexer per HCI connection
MAX_NR_RFCOMM_SERVICES	Max number of RFCOMM services
MAX_NR_SERVICE_RECORD_ITEMS	Max number of SDP service records
MAX_NR_SM_LOOKUP_ENTRIES	

#define	Description
	Max number of items in Security Manager lookup queue
MAX_NR_WHITELIST_ENTRIES	Max number of items in GAP LE Whitelist to connect to

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 NVM_NUM_LINK_KEYS  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 stored in persistent memory.

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 is implemented using Flash memory. Then, the number of stored entries are limited by:

#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
NVN_NUM_GATT_SERVER_CCC	Max number of 'Client Characteristic Configuration' values that can be stored by GATT Server

HCI Dump Stdout directives

Allow to truncate HCI ACL and SCO packets to reduce console output for debugging audio applications.

#define	Description
HCI_DUMP_STDOUT_MAX_SIZE_ACL	Max size of ACL packets to log via stdout
HCI_DUMP_STDOUT_MAX_SIZE_SCO	Max size of SCO packets to log via stdout
HCI_DUMP_STDOUT_MAX_SIZE_ISO	Max size of ISO packets to log via stdout

SEGGER Real Time Transfer (RTT) directives

SEGGER RTT improves on the use of an UART for debugging with higher throughput and less overhead. In addition, it allows for direct logging in PacketLogger/BlueZ format via the provided JLinkRTTLogger tool.

When enabled with `ENABLE_SEGGER_RTT` and `hci_dump_init()` can be called with an `hci_dunp_segger_stdout_get_instance()` for textual output and `hci_dump_segger_binary_get_instance()` for binary output. With the latter, you can select `HCI_DUMP_BLUEZ` or `HCI_DUMP_PACKETLOGGER`, format. For RTT, the following directives are used to configure the up channel:

#define	Default	Description
SEGGER_RTT_PACKETLOG_MODE	SEGGER_RTT_MODE_NO_BLOCK_SKIP	SEGGER_RTT_MODE_NO_BLOCK_SKIP to skip messages if buffer is full, or, SEGGER_RTT_MODE_BLOCK_IF_FIFO_FULL to block
SEGGER_RTT_PACKETLOG_CHANNEL	1	Channel to use for packet log. Channel 0 is used for terminal
SEGGER_RTT_PACKETLOG_BUFFER_SIZE	1024	Size of outgoing ring buffer. Increase if you cannot block but get ‘message skipped’ warnings.

LE Audio Configuration

The following list of directives are all set with default values. Modify them if you need support for different number of: - BASS subgroups, - ASCS Sink and Source endpoints, - PACS Client Sink and Source Endpoint records, - number of services for VOCS, or - if you, in rare case, need to extend the buffers.

#define	Default	Description
MAX_SIZE_AICS_AUDIO_INPUT_DESCRIPTION	30	AICS: Size of audio input description.
MAX_NR_ASCS_SERVER_SINK_ENDPOINTS	3	ASCS Server: Number of Audio Sink endpoints.
MAX_NR_ASCS_SERVER_SOURCE_ENDPOINTS	3	ASCS Server: Number of Audio Source endpoints.
MAX_NR_ASCS_CLIENT_SINK_ENDPOINTS	3	

#define	Default	Description
		ASCS Client: Number of Audio Sink endpoints.
MAX_NR_ASCS_CLIENT_SOURCE_ENDPOINTS	3	ASCS Client: Number of Audio Source endpoints.
MAX_SIZE_ASCS_SERVER_LONG_WRITE_BUFFER	100	ASCS Server: Size of long write buffer.
MAX_NR_BASS_SUBGROUPS	10	BASS: Number of subgroups.
MAX_SIZE_BASS_SERVER_LONG_WRITE_BUFFER	512	BASS Server: Size of long write buffer.
MAX_SIZE_BASS_SERVER_NOTIFICATION_BUFFER	200	BASS Server: Size of notification buffer.
MAX_SIZE_BASS_CLIENT_LONG_WRITE_BUFFER	512	BASS Client: Size of long write buffer.
MAX_SIZE_HAS_PRESET_RECORD_NAME	40	HAS: Size of preset record name.
MAX_SIZE_MCS_SERVER_SEARCH_CONTROL_POINT_COMMAND	100	MCS Server: Size of search command buffer.
MAX_SIZE_MCS_SERVER_ICON_URL	50	MCS Server: Size of icon URL.
MAX_SIZE_MCS_SERVER_TRACK_TITLE	50	MCS Server: Size of track title.
MAX_SIZE_MCS_CLIENT_SEARCH_CONTROL_POINT_COMMAND	100	MCS Client: Size of search command buffer.
	3	

#define	Default	Description
MAX_NR_OTS_SERVER_SERVER_FILTERS		OTS Server: Number of filters.
MAX_SIZE_OTS_STRING	60	OTS Server: Size of string.
MAX_NR_PACS_CLIENT_SINK_ENDPOINT_RECORDS	3	PACS Client: Number of Audio Sink endpoint records.
MAX_NR_PACS_CLIENT_SOURCE_ENDPOINT_RECORDS	3	PACS Client: Number of Audio Source endpoint records.
MAX_SIZE_TBS_CLIENT_SERIALISATION_BUFFER	255	TBS Client: Size of serialization buffer.
MAX_NR_VOCS_SERVER_SERVICES	5	VOCS Server: Number of services.
MAX_SIZE_VOCS_AUDIO_OUTPUT_DESCRIPTION	30	VOCS: Size of audio output description.
MAX_NR_TBS_SERVER_URI	252	TBS Server: Size of URI.
MAX_NR_TBS_SERVER_PROVIDER_NAME	30	TBS Server: Size of provider name.
MAX_NR_TBS_SERVER_UNIFORM_CALLER_IDENTIFIER	10	TBS Server: Size of uniform caller identifier.
MAX_NR_TBS_SERVER_URI_SCHEMES_LIST	30	TBS Server: Size of URI schemes list.
MAX_NR_TBS_SERVER_CALL_CONTROL_POINT_NOTIFICATION	3	TBS Server: Size of call control notification.

#define	Default	Description
MAX_SIZE_TBS_CLIENT_SERIALISATION_BUFFER	255	TBS Client: Size of serialisation buffer.

_

Run-time configuration

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 below shows a minimal platform setup for 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());

  // enable logging
  hci_dump_init(hci_dump_embedded_stdout_get_instance());



  // 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();
}
```

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

The call to *hci_dump_init* configures BTstack to output all Bluetooth packets and its own debug and error message using printf with BTstack's millisecond timestamps.s as tim. 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.

Source tree structure

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

Path	Description
chipset	Support for individual Bluetooth Controller 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/*.

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. In addition, most implementations also allow to trigger a poll of the data sources from interrupt context, or, execute a function from a different thread.

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. Data sources need to be configured upon what event they are called back. They can be configured to be polled (*DATA_SOURCE_CALLBACK_POLL*), on read ready (*DATA_SOURCE_CALLBACK_READ*), or on write ready (*DATA_SOURCE_CALLBACK_WRITE*).

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:

- CoreFoundation: implementation for iOS and OS X applications
- Embedded: the main implementation for embedded systems, especially without an RTOS.
- FreeRTOS: implementation to run BTstack on a dedicated FreeRTOS thread
- POSIX: implementation for POSIX systems based on the select() call.
- Qt: implementation for the Qt 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, FreeRTOS), or the platform provides a way to wait for a data source to become ready for read or write (CoreFoundation, POSIX, Qt, 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 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());
```

If the run loop allows to trigger polling of data sources from interrupt context, *btstack_run_loop_poll_data_sources_from_irq*.

On multi-threaded environments, e.g., FreeRTOS, POSIX, WINDOWS, *btstack_run_loop_execute_code_on_main_thread* can be used to schedule a callback on the main loop.

The complete Run loop API is provided here.

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_poll_data_sources_from_irq* function. The call to *btstack_run_loop_poll_data_sources_from_irq* 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.

While there is no threading, *btstack_run_loop_poll_data_sources_from_irq* allows to reduce stack size by scheduling a continuation.

Run Loop FreeRTOS

The FreeRTOS run loop is used on a dedicated FreeRTOS thread and it uses a FreeRTOS queue to schedule callbacks on the run loop. In each iteration:

- all data sources are polled
- all scheduled callbacks are executed
- all expired timers are called
- finally, it gets the next timeout. It then waits for a 'trigger' or the next timeout, if set.

It supports both *btstack_run_loop_poll_data_sources_from_irq* as well as *btstack_run_loop_execute_code_on_main_thread*.

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.

It supports both *btstack_run_loop_poll_data_sources_from_irq* as well as *btstack_run_loop_execute_code_on_main_thread*.

Run loop CoreFoundation (OS X/iOS)

This run loop directly maps BTstack'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.

It currently only supports *btstack_run_loop_execute_code_on_main_thread*.

Run Lop Qt

This run loop directly maps BTstack's data source and timer source with Qt Core objects. It supports ready to read and write similar to the POSIX implementation.

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

It supports both *btstack_run_loop_poll_data_sources_from_irq* as well as *btstack_run_loop_execute_code_on_main_thread*.

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.

It supports both *btstack_run_loop_poll_data_sources_from_irq* as well as *btstack_run_loop_execute_code_on_main_thread*.

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.

It currently only supports *btstack_run_loop_execute_code_on_main_thread*.

HCI Transport configuration

The HCI initialization has to adapt BTstack 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.

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.

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 below.

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_initial_credits
RFCOMM channel packet handler	rfcomm_create_channel and rfcomm_create_channel_with_initial_credits

Table: 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_report*. 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);
```

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 and the following implementations:

```
void hci_dump_init(const hci_dump_t * hci_dump_implementation);
```

Platform	File	Description
POSIX	`hci_dump_posix_fs.c`	HCI log file for Apple PacketLogger and Wireshark
POSIX	`hci_dump_posix_stdout.c`	Console output via printf
Embedded	`hci_dump_embedded_stdout.c`	Console output via printf
Embedded	`hci_dump_segger_stdout.c`	Console output via SEGGER RTT
Embedded	`hci_dump_segger_binary.c`	

Platform	File	Description
		HCI log file for Apple PacketLogger via SEGGER RTT

On POSIX systems, you can call *hci_dump_init* with a *hci_dump_posix_fs_get_instance()* and configure the path and output format with *hci_dump_posix_fs_open(const char* path, hci_dump_format_t format) *where format can be* HCI_DUMP_BLUEZ* or *HCI_DUMP_PACKETLOGGER*. The resulting file can be analyzed with Wireshark or the Apple's PacketLogger tool.

On embedded systems without a file system, you either log to an UART console via printf or use SEGGER RTT. For printf output you pass *hci_dump_embedded_stdout_get_instance()* to *hci_dump_init()*. With RTT, you can choose between textual output similar to printf, and binary output. For textual output, you can provide the *hci_dump_segger_stdout_get_instance()*.

It will log all HCI packets to the UART console via printf or RTT Terminal. 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.

For less overhead and higher logging speed, you can directly log in binary format by passing *hci_dump_segger_rtt_binary_get_instance()* and selecting the output format by calling *hci_dump_segger_rtt_binary_open(hci_dump_format_t format)* with the same format as above.

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.

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.

#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

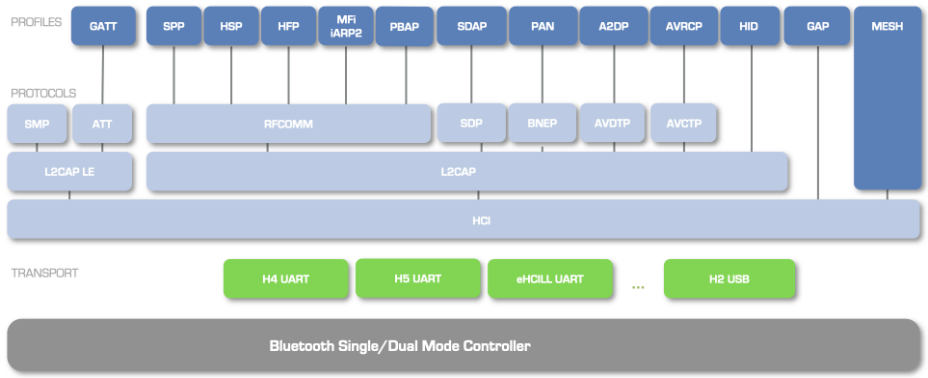
connection. This profile mainly make use of the Host Controller Interface (HCI) protocol, the lowest protocol in the stack hierarchy which implements a command interface to the Bluetooth chipset.

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 below 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.



Architecture of a BTstack-based application.

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.

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 below shows the OGFs provided by BTstack in 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
```

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 below, and src/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 below 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)
};
```

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 below 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: Supported Format Specifiers of HCI Command Parameter.

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 below illustrates how to manually set the device name with the HCI Write Local Name command.

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

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.

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.

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 below.

```
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();
}
```

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 below provides L2CAP service example 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);
}
```

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(cahnnel_id)* 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_event* function returns. The L2CAP_EVENT_CAN_SEND_NOW indicates a channel ID on which sending is possible.

Please note that the guarantee that a packet can be sent is only valid when the event is received. After returning from the packet handler, BTstack might need to send itself.

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_cbm_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_cbm_provide_credits*. If *L2CAP_LE_AUTOMATIC_CREDITS* 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_cbm_register_service* and *l2cap_cbm_unregister_service* are used to manage local services.
- *l2cap_cbm_accept_connection* and *l2cap_cbm_decline_connection* are used to accept or deny an incoming connection request.
- *l2cap_cbm_create_channel* creates an outgoing connections.
- *l2cap_cbm_can_send_now* checks if a packet can be scheduled for transmission now.
- *l2cap_cbm_request_can_send_now_event* requests an *L2CAP_EVENT_LE_CAN_SEND_NOW* event as soon as possible.
- *l2cap_cbm_disconnect* closes the connection.

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.

No RFCOMM packet boundaries

As RFCOMM emulates a serial port, it does not preserve packet boundaries.

On most operating systems, RFCOMM/SPP will be modeled as a pipe that allows to write a block of bytes. The OS and the Bluetooth Stack are free to buffer and chunk this data in any way it seems fit. In your BTstack application, you will therefore receive this data in the same order, but there are no guarantees as how it might be fragmented into multiple chunks.

If you need to preserve the concept of sending a packet with a specific size over RFCOMM, the simplest way is to prefix the data with a 2 or 4 byte length field and then reconstruct the packet on the receiving side.

Please note, that due to BTstack's 'no buffers' policy, BTstack will send outgoing RFCOMM data immediately and implicitly preserve the packet boundaries, i.e., it will send the data as a single RFCOMM packet in a single L2CAP packet, which will arrive in one piece. While this will hold between two BTstack instances, it's not a good idea to rely on implementation details and rather prefix the data as described.

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.

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_CLOSED events, and RFCOMM data packets, as shown in Listing below.

```
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();
}
```

The RFCOMM channel will stay open until either side closes it with rfcomm_disconnect. The RFCOMM multiplexer will be closed by the peer that closes the last RFCOMM channel.

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 below provides the RFCOMM service example 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);
}
```

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 below.

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

If the management of credits is manual, credits are provided by the application such that it can manage its receive buffers explicitly, see Listing below.

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 below.

```
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);
}


void processing(){
    // process incoming data packet
```

```
    ...
    // provide new credit
    rfcomm_grant_credits(rfcomm_channel_id, 1);
}
```

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.

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 below.

Please note that the guarantee that a packet can be sent is only valid when the event is received. After returning from the packet handler, BTstack might need to send itself.

```
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;
                ...
            }
            ...
        }
    }
}
```

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*.

SDP - Service Discovery Protocol

The SDP protocol allows to announce services and discover services provided by a remote Bluetooth device.

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.

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 below.

```
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;
}
```

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.

Receive BNEP events

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

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*.

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.

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.

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.

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.

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.

To only allow LE Secure Connections, you can call *sm_set_secure_connections_only_mode(true)*.

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.

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 require 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*

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.

User interaction during Pairing

BTstack will inform the app about pairing via theses events:

- *SM_EVENT_PAIRING_STARTED*: inform user that pairing has started
- *SM_EVENT_PAIRING_COMPLETE*: inform user that pairing is complete, see status

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_PASSKEY_DISPLAY_CANCEL*: cancel show passkey to user
- *SM_EVENT_NUMERIC_COMPARISON_REQUEST*: show a passkey to the user and request confirmation

To accept Just Works/Numeric Comparison, or provide a Passkey, *sm_just_works_confirm* or *sm_passkey_input* can be called respectively. Othwerise, *sm_bonding_decline* aborts the pairing.

After the bonding process, *SM_EVENT_PAIRING_COMPLETE*, is emitted. Any active dialog can be closed on this.

Connection with Bonded Devices

During pairing, two devices exchange bonding information, notably a Long-Term Key (LTK) and their respective Identity Resolving Key (IRK). On a subsequent connection, the Securit Manager will use this information to establish an encrypted connection.

To inform about this, the following events are emitted:

- *SM_EVENT_REENCRYPTION_STARTED*: we have stored bonding information and either trigger encryption (as Central), or, sent a security request (as Peripheral).
- *SM_EVENT_REENCRYPTION_COMPLETE*: re-encryption is complete. If the remote device does not have a stored LTK, the status code will be *ERROR_CODE_PIN_OR_KEY_MISSING*.

The *SM_EVENT_REENCRYPTION_COMPLETE* with *ERROR_CODE_PIN_OR_KEY_MISSING* can be caused:

- if the remote device was reset or the bonding was removed, or,
- we’re connected to an attacker that uses the Bluetooth address of a bonded device.

In Peripheral role, pairing will start even in case of an re-encryption error. It might be helpful to inform the user about the lost bonding or reject it right away due to security considerations.

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.

Cross-transport Key Derivation (CTKD) for LE Secure Connections

In a dual-mode configuration, BTstack generates an BR/EDR Link Key from the LE LTK via the Link Key Conversion functions *h6* , (or *h7* if supported) when *ENABLE_CROSS_TRANSPORT_KEY_DERIVATION* is defined. The derived key then stored in local LE Device DB.

The main use case for this is connections with smartphones. E.g. iOS provides APIs for LE scanning and connection, but none for BR/EDR. This allows an application to connect and pair with a device and also later setup a BR/EDR connection without the need for the smartphone user to use the system Settings menu.

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. BTstack does not support LE Secure Connections via LE Transport currently.

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*.

#Profiles

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

A2DP - Advanced Audio Distribution

The A2DP profile defines how to stream audio over a Bluetooth connection from one device, such as a mobile phone, to another device such as a headset. A device that acts as source of audio stream implements the A2DP Source role. Similarly, a device that receives an audio stream implements the A2DP Sink role. As such, the A2DP service allows uni-directional transfer of an audio stream, from single channel mono, up to two channel stereo. Our implementation includes mandatory support for the low-complexity SBC codec. Signaling for optional codes (FDK AAC, LDAC, APTX) is supported as well, by you need to provide your own codec library.

AVRCP - Audio/Video Remote Control Profile

The AVRCP profile defines how audio playback on a remote device (e.g. a music app on a smartphone) can be controlled as well as how to state changes such as volume, information on currently played media, battery, etc. can be received from a remote device (e.g. a speaker). Usually, each device implements two roles: - The Controller role allows to query information on currently played media, such are title, artist and album, as well as to control the playback, i.e. to play, stop, repeat, etc. - The Target role responds to commands, e.g. playback control, and queries, e.g. playback status, media info, from the Controller currently played media.

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 `hci.h` and `gap.h`. Please check both.

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 below 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);
          ...
     }
 }
```

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 below.

```
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;
       ...
    }
}
```

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.

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 below.

```
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;
       ...
    }
}
```

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.

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.

SPP - Serial Port Profile

The SPP profile defines how to set up virtual serial ports and connect two Bluetooth enabled devices. Please keep in mind that a serial port does not preserve packet boundaries if you try to send data as packets and read about RFCOMM packet boundaries.

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.

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.

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.

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.

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.

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

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.

Supported Features

The supported features define the HFP capabilities of the device. The enumeration unfortunately differs between HF and AG sides.

The AG supported features are set by combining the flags that start with HFP_AGSF_xx and calling hfp_ag_init_supported_features, followed by creating SDP record for the service using the same feature set.

Similarly, the HF supported features are a combination of HFP_HFSF_xx flags and are configured by calling hfp_hf_init_supported_features, as well as creating an SDP record.

Define for AG Supported Feature	Description
HFP_AGSF_ THREE_WAY_CALLING	Three-way calling
HFP_AGSF_ EC_NR_FUNCTION	Echo Canceling and/or Noise Reduction function
HFP_AGSF_ VOICE_RECOGNITION_FUNCTION	Voice recognition function

Define for AG Supported Feature	Description
HFP_AGSF_ IN_BAND_RING_TONE	In-band ring tone capability
HFP_AGSF_ ATTACH_A_NUMBER_TO_A_VOICE_TAG	Attach a number to a voice tag
HFP_AGSF_ ABILITY_TO_REJECT_A_CALL	Ability to reject a call
HFP_AGSF_ ENHANCED_CALL_STATUS	Enhanced call status
HFP_AGSF_ ENHANCED_CALL_CONTROL	Enhanced call control
HFP_AGSF_ EXTENDED_ERROR_RESULT_CODES	Extended Error Result Codes
HFP_AGSF_ CODEC_NEGOTIATION	Codec negotiation
HFP_AGSF_ HF_INDICATORS	HF Indicators
HFP_AGSF_ ESCO_S4	eSCO S4 (and T2) Settings Supported
HFP_AGSF_ ENHANCED_VOICE_RECOGNITION_STATUS	Enhanced voice recognition status
HFP_AGSF_ VOICE_RECOGNITION_TEXT	Voice recognition text

Define for HF Supported Feature	Description
HFP_HFSF_ THREE_WAY_CALLING	Three-way calling
HFP_HFSF_ EC_NR_FUNCTION	Echo Canceling and/or Noise Reduction function
HFP_HFSF_ CLI_PRESENTATION_CAPABILITY	CLI presentation capability
HFP_HFSF_ VOICE_RECOGNITION_FUNCTION	Voice recognition function
HFP_HFSF_ REMOTE_VOLUME_CONTROL	Remote volume control

Define for HF Supported Feature	Description
HFP_HFSF_ ATTACH_A_NUMBER_TO_A_VOICE_TAG	Attach a number to a voice tag
HFP_HFSF_ ENHANCED_CALL_STATUS	Enhanced call status
HFP_HFSF_ ENHANCED_CALL_CONTROL	Enhanced call control
HFP_HFSF_ CODEC_NEGOTIATION	Codec negotiation
HFP_HFSF_ HF_INDICATORS	HF Indicators
HFP_HFSF_ ESCO_S4	eSCO S4 (and T2) Settings Supported
HFP_HFSF_ ENHANCED_VOICE_RECOGNITION_STATUS	Enhanced voice recognition status
HFP_HFSF_ VOICE_RECOGNITION_TEXT	Voice recognition text

Audio Voice Recognition Activation

Audio voice recognition (AVR) requires that HF and AG have the following features enabled:

- HF: HFP_HFSF_VOICE_RECOGNITION_FUNCTION and
- AG: HFP_AGSF_VOICE_RECOGNITION_FUNCTION.

It can be activated or deactivated on both sides by calling:

```
// AG
uint8_t hfp_ag_activate_voice_recognition(hci_con_handle_t
acl_handle);
uint8_t hfp_ag_deactivate_voice_recognition(hci_con_handle_t
acl_handle);

// HF
uint8_t hfp_hf_activate_voice_recognition(hci_con_handle_t
acl_handle);
```

```
uint8_t hfp_hf_deactivate_voice_recognition(hci_con_handle_t
acl_handle);
```

On activation change, the HFP_SUBEVENT_VOICE_RECOGNITION_(DE)ACTIVATED event will be emitted with status field set to ERROR_CODE_SUCCESS on success.

Voice recognition will stay active until either the deactivation command is called, or until the current Service Level Connection between the AG and the HF is dropped for any reason.

Use cases	Expected behavior
No previous audio connection, AVR activated then deactivated	Audio connection will be opened by AG upon AVR activation, and upon AVR deactivation closed
AVR activated and deactivated during existing audio connection	Audio remains active upon AVR deactivation
Call to close audio connection during active AVR session	The audio connection shut down will be refused
AVR activated, but audio connection failed to be established	AVR will stay activated

Beyond the audio routing and voice recognition activation capabilities, the rest of the voice recognition functionality is implementation dependent - the stack only provides the signaling for this.

Enhanced Audio Voice Recognition

Similarly to AVR, Enhanced Audio voice recognition (eAVR) requires that HF and AG have the following features enabled:

- HF: HFP_HFSF_ENHANCED_VOICE_RECOGNITION_STATUS and
- AG: HFP_AGSF_ENHANCED_VOICE_RECOGNITION_STATUS.

In addition, to allow textual representation of audio that is parsed by eAVR (note that parsing is not part of Bluetooth specification), both devices must enable:

- HF: HFP_HFSF_VOICE_RECOGNITION_TEXT and

• AG: HFP_AGSF_VOICE_RECOGNITION_TEXT.

eAVR implements the same use cases as AVR (see previous section) and it can be activated or deactivated using the same API as for AVR, see above.

When eAVR and audio channel are established there are several additional commands that can be sent:

HFP Role	eVRA API	Description
HF	hfp_hf_enhanced_voice_recognition_report_ready_for_audio	Ready to accept audio input.
AG	hfp_ag_enhanced_voice_recognition_report_ready_for_audio	Voice recognition engine is ready to accept audio input.
AG	hfp_ag_enhanced_voice_recognition_report_sending_audio	The voice recognition engine will play a sound, e.g. starting sound.
AG	hfp_ag_enhanced_voice_recognition_report_processing_input	Voice recognition engine is processing the audio input.
AG	hfp_ag_enhanced_voice_recognition_send_message	Report textual representation from the voice recognition engine.

HID - Human-Interface Device Profile

The HID profile allows an HID Host to connect to one or more HID Devices and communicate with them. Examples of Bluetooth HID devices are keyboards, mice, joysticks, gamepads, remote controls, and

also voltmeters and temperature sensors. Typical HID hosts would be a personal computer, tablets, gaming console, industrial machine, or data-recording device.

Please refer to:

- HID Host API and hid_host_demo for the HID Host role
- HID Device API, hid_keyboard_demo and hid_mouse_demo for the HID Device role.

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.

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.

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 .

GATT - Generic Attribute Profile

The GATT profile uses ATT Attributes to represent a hierarchical structure of GATT Services and GATT Characteristics. A GATT Service is a standardized entity which is performing a specific function on a device. For example, the main purpose of the Battery Service is to enable connected devices to report battery level information. GATT

Services that define the main functionality of a device are known as Primary Services. Services can also include other Services, known as Included (Secondary) Services.

Each Service has one or more Characteristics. Each Characteristic has meta data attached like its type or its properties that declare the set of allowed operations and their access permissions. This hierarchy of GATT Characteristics and Services are queried and modified via ATT operations.

Officially adopted GATT Services and Characteristics are identified by unique 16-bit UUIDs, whereas a custom made Services use 128-bit UUIDs. A complete list of officially adopted GATT Services is available on the Bluetooth Developer Portal.

As an example, the Scan Parameters Service (UUID: 0x1813) is used to optimize the way a Central device (like a smartphone or tablet) scans for advertising packets from a Peripheral device (like a BLE sensor, wearable, or HID device). It allows the Peripheral device to suggest preferred scanning parameters by means of two Characteristics:

- Scan Interval Window (0x2A31) – mandatory Characteristic comprising two 16-bit values: LE_Scan_Interval and LE_Scan_Window. It stores the Client's scan parameters and is only writable by the Client.
- Scan Refresh (0x2A31) – optional Characteristic used to notify the Client that the Server requires an update of the Scan Interval Window value.

Two connected Bluetooth LE devices exchange data by taking on two GATT roles: one acts as the GATT Server, and the other as the GATT Client.

GATT Client

The GATT Client's role is to discover available Services and Characteristics on the GATT Server. Depending on the Service configuration, it can read or write values of a Characteristic, or subscribe to receive updates when those values change.

If you’re looking to implement a standard GATT (Generic Attribute Profile) Service Client, your starting point will be the corresponding Bluetooth GATT Service Profile specification provided by the Bluetooth SIG. This document outlines how a GATT Client interacts with a GATT Server, but only those steps after the service has been discovered, characteristics have been queried, and all allowed notifications and indications have been subscribed to. While there is no mystery to this setup process — it follows a well-defined sequence of queries and responses — its implementation is repetitive and time-consuming.

To streamline this effort, we introduced a GATT Service Client component that encapsulates the entire discover-and-subscribe workflow. The component removes the overhead of boilerplate setup code, so that developers can now dive straight into implementing the core logic defined by the GATT Service Profile specification.

GATT Client Discovery Workflow

Before a GATT Client can perform operations on Characteristics or process responses from the GATT Server, it must first complete a sequence of GATT queries, typically including:

- Discover services
- Query characteristics and their subscription properties
- Subscribe to notifications or indications, if allowed

GATT queries in most Bluetooth stacks, including BTstack, are asynchronous. Therefore, a state machine is required to manage the sequence of operations. One part of this state machine is responsible for issuing GATT queries - one at a time. The other part handles the GATT Server responses. For example, a “discover all primary services with UUID X” query may result in multiple GATT events, one for each matching service, followed by a final “service query complete” event indicating that all results have been received. These two parts are illustrated below in pseudocode.

Pseudocode: Send GATT Query

```
switch (STATE):
    case W2_QUERY_SERVICES:
        - Set state to W4_SERVICES_RESULT and wait for
```

```
SERVICE_QUERY_RESULT GATT event
        - Execute query to search for all services with the given
UUID

    case W2_QUERY_CHARACTERISTICS:
        - Set state to W4_QUERY_CHARACTERISTICS_RESULT
        - Execute query to get all characteristics of the current
service

    case W2_SUBSCRIBE:
        - Set state to W4_SUBSCRIPTION
        - Subscribe to notifications or indications
```

Pseudocode: Handle GATT Response

```
switch (STATE):
    case W4_SERVICES_RESULT:
        switch (GATT_EVENT):
            case SERVICE_QUERY_RESULT:
               * A matching service was found
               * Store the service in the client’s service list
            case SERVICE_QUERY_COMPLETE:
               * Set state to W2_QUERY_CHARACTERISTICS

    case W4_QUERY_CHARACTERISTICS_RESULT:
        switch (GATT_EVENT):
            case CHARACTERISTIC_QUERY_RESULT:
               * A characteristic for a given service was found
               * Store the characteristic and its properties in
the service’s characteristic list
            case CHARACTERISTIC_QUERY_COMPLETE:
               * All characteristics for the current service have
been reported
               * Choose the first characteristic that supports
notification or indication
               * If such characteristic exists set state to
W2_SUBSCRIBE, otherwise to CONNECTED

    case W4_SUBSCRIPTION:
        switch (GATT_EVENT):
            case CHARACTERISTIC_QUERY_COMPLETE:
```

```
                * Characteristic was subscribed
                * Choose the next characteristic that supports
notification or indication
                * If such characteristic exists set state to
W2_SUBSCRIBE, otherwise to CONNECTED
```

While not conceptually complex, implementing the described discovery workflow can be tedious and time-consuming, diverting attention from the core development of the GATT Client. To simplify this process, we introduced the GATT Service Client component (btstack/src/ble/gatt_service_client.h) that encapsulates all discovery steps - from querying services to subscribing to all allowed Characteristics. Let's see how it it is used...

GATT Service Client Component Setup

To setup a GATT Service Client component for a custom service, we need to initialize both the GATT Client as well as the GATT Service Client and then register an instance of the new GATT Service Client. This usually happens before the Bluetooth stack is started:

- gatt_client_init function to initialize the GATT Client,
- gatt_service_client_init function to initialize the GATT Service Client component, and
- gatt_service_client_register_client_with_uuid(16|128)s function to register the new GATT Service Client instance with the list of Characteristic UUIDs (16-bit or 128-bit respectively). The connection and notification callback is used to receive GATT Service Client connected and disconnected events, as well as notifications and identifications.

The following code snippet shows the setup code of the GATT Streamer Client (btstack/examples/le_streamer_client.c), which we rewrote as our first experiment. The GATT Streamer provides a custom GATT Service created to test the maximal GATT throughput - its core functionality is to stream data over GATT as fast as possible using write without response operations and notifications on the RX Characteristic.

```
# define LE_STREAMER_SERVICE_NUM_CHARACTERISTICS      2
// LE Streamer GATT Service Client instance
```

```
static gatt_service_client_t
le_streamer_client;

// 128-bit (custom) LE Streamer Service UUID
static uuid128_t LE_STREAMER_SERVICE_UUID = {
    0x00, 0x00, 0xFF, 0x10, 0x00, 0x00, 0x10, 0x00, 0x80, 0x00,
0x00, 0x80, 0x5F, 0x9B, 0x34, 0xFB
};

// 128-bit UUIDs if the LE Streamer Service Characteristics
static enum {
    LE_STREAMER_SERVICE_CHARACTERISTIC_INDEX_RX = 0,
    LE_STREAMER_SERVICE_CHARACTERISTIC_INDEX_TX
} le_streamer_service_characteristic_index_t;

static const uuid128_t le_streamer_uuid128s[] = {
    { 0x00, 0x00, 0xFF, 0x11, 0x00, 0x00, 0x10, 0x00, 0x80, 0x00,
0x00, 0x80, 0x5F, 0x9B, 0x34, 0xFB}, // RX
    { 0x00, 0x00, 0xFF, 0x12, 0x00, 0x00, 0x10, 0x00, 0x80, 0x00,
0x00, 0x80, 0x5F, 0x9B, 0x34, 0xFB}  // TX
};

/*
 *  Callback to receive GATT Service Client events: connected,
disconnected, notifications and identifications
 */
static void gatt_service_client_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size);

static void setup(void){
    ...
    gatt_client_init();
    gatt_service_client_init();

gatt_service_client_register_client_uuid128(&le_streamer_client,
&gatt_service_client_handler,

le_streamer_uuid128s, LE_STREAMER_SERVICE_NUM_CHARACTERISTICS);
    …
}
```

GATT Service Client Component Connect

After the setup is performed and a Bluetooth HCI connection between two devices has been established, we can trigger the GATT Service discovery workflow by calling a single function gatt_service_client_connect_primary_service_with_uuid(16|128). This function requires the connection handle, the GATT Service UUID as well as storage for the information on the characteristics that will be discovered during the discover-and-subscribe phase. The following code snippet shows how to place a call to this function directly after the HCI connection has been established. The value ERROR_CODE_SUCCESS of the returned status indicates that the service discovery workflow has started. The end of the discovery is marked by the GATTSERVICE_SUBEVENT_CLIENT_CONNECTED event in the gatt_service_client_handler.

```
static hci_con_handle_t                     connection_handle;
static gatt_service_client_connection_t
le_streamer_connection;
static gatt_service_client_characteristic_t
le_streamer_connection_characteristics_storage[NUM_CHARACTERISTICS]

/*
 - Callback to receive GATT Service Client events: connected,
disconnected, notifications and identifications
 */
static void gatt_service_client_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size){
    if (packet_type != HCI_EVENT_PACKET) return;
    switch(hci_event_packet_get_type(packet){
        case HCI_EVENT_GATTSERVICE_META:
            switch
(hci_event_gattservice_meta_get_subevent_code(packet)) {
                case GATTSERVICE_SUBEVENT_CLIENT_CONNECTED:
                    …
                    break;
                case GATTSERVICE_SUBEVENT_CLIENT_DISCONNECTED:
                    …
                    break;
                default:
```

```
                    break;
            }
            break;
        case GATT_EVENT_NOTIFICATION:
            …
            break;
        case GATT_EVENT_INDICATION:
            …
            break;
        default:
            break;
    }
}

static void hci_event_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size){
    if (hci_event_packet_get_type(packet) == HCI_EVENT_META_GAP) {
        if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE){

            uint8_t status =
gatt_service_client_connect_primary_service_with_uuid128(
                                        connection_handle,
                                        &le_streamer_client,
                                        &le_streamer_connection,
                                        &LE_STREAMER_SERVICE_UUID,

le_streamer_connection_characteristics_storage,

LE_STREAMER_SERVICE_NUM_CHARACTERISTICS);
            …
        }
    }
    …
}

static void setup(void){
     ...
    hci_event_callback_registration.callback = &hci_event_handler;
    hci_add_event_handler(&hci_event_callback_registration);
```

```
    …
}
```

GATT Service Client Queries

If the status of the received GATTSERVICE_SUBEVENT_CLIENT_CONNECTED event is ERROR_CODE_SUCCESS, we can start scheduling service specific GATT Queries if needed. The complete list of supported GATT queries is defined in (btstack/src/ble/gatt_client.h). From this point on a query will most likely be either read, write or write without response on a specific characteristic of the GATT Service. These queries use a characteristic's attribute value handle to identify the characteristic. The attribute value handle can be accessed using the gatt_service_client_characteristic_value_handle_for_index function. The index parameter for this function matches the order of the UUIDs stored in the UUID array.

The following code snippet shows how to perform write without response.

```
static uint8_t  data[200];
static uint16_t data_len;

static void
le_streamer_write_without_response(le_streamer_client_connection_t
* connection){
    uint8_t status =
gatt_client_write_value_of_characteristic_without_response(
                            connection_handle,

gatt_service_client_characteristic_value_handle_for_index(connectic
CHARACTERISTIC_INDEX_RX),
                            data_len, data);
    …
}
```

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

GATT Client Caching

When accessing a remote GATT Server, one needs to send multiple GATT queries to find all Services and Characteristics before being able to access them. If you only access a single service or two, this is fast enough and usually not noticeable. On the other end, a device implementing LE Audio Unicast Gateway needs to query and connect to multiple services, e.g. CSIS, PACS, ASCS, MCS, VCS. Depending on the used connection parameters and the responsiveness of the remote device, this can take quite long. We measured up to a whole minute connecting to an LE Audio Headset.

To reduce this time, GATT Client Caching can be used. The general idea is that the remote GATT database usually does not change. Instead of repeating the GATT queries on every connect, it's much faster to store the results locally. However, we need to make sure that we avoid the case that the GATT database acutally changes and we use incorrect ATT Attribute Handles. For this, devices can provide the GATT Database Hash Characteristic since Bluetooth Core Spec v5.1. The Database Hash is calculated over the structure, not the values, of the complete GATT Database. If the Database Hash matches the one from the last connect, we can be sure that the ATT Attribute Handles haven't changed and all services are still there.

In BTstack, you can enable tracking of this GATT Database Hash by enabling `ENABLE_GATT_CLIENT_CACHING` in btstack_config.h. The GATT Client will automatically cache and track changes to the Database Hash and also register for Service Changed notifications. In both cases, a `GATTSERVICE_SUBEVENT_GATT_SERVICE_CHANGED` or a `GATTSERVICE_SUBEVENT_GATT_DATABASE_HASH` will be emitted.

In addition, the GATT Service Client will automatically make use of these mechanisms to cache information about the Characteristics required by it's clients without changes to your existing code.

For this caching to work, a few conditions need to be true:

- `ENABLE_GATT_CLIENT_CACHING` is defined.
- Your application supports bonding.

- Your application instructs the GATT Client to use an encrypted connection by calling `gatt_client_set_required_security_level` with a level > 0.

- Your applications usually connects to remote GATT Services via the GATT Service Client in the same order. The GATT Service Client will work correctly even if the order changes, but it will need to repeat the query to refresh the cached data.

Please note: if the remote database changes while a GATT Service Client is connected to it, it will emit a `GATTSERVICE_SUBEVENT_CLIENT_DISCONNECTED` event with status set to `ERROR_CODE_UNSPECIFIED_ERROR` and consider the connection as closed to avoid sending or receiving incorrect data. If required, the application can then restart their logic and request the GATT Service Client to reconnect to the remote service again.

In addition to the GATT Service Client requests that are cached, the tracking of the remote Database Hash allows your application to cache data on a higher level if data is unlikely to change. For example, an LE Audio Unicast Gateway needs to use the Coordinated Set Identification Service (CSIS) to figure out if the connected device is part of a set of devices. It also needs to use the Published Audio Capabilities Service (PACS) to learn what audio codecs are supported in detail. In both cases, the application can cache the values assuming that they don't change during the lifetime of a remote device.

To make use of this, your application can register itself with the GATT Client via `gatt_client_add_service_changed_handler(..)` to get `GATTSERVICE_SUBEVENT_GATT_DATABASE_HASH` events which contains both the Database Hash as well as a local 16 bit version. In addition, you can use the LE Device Index you can get from the Security Manage with `sm_le_device_index(..)` to construct a unique tag for use with the BTstack's TLV. To invalidate the cached data, you need to respond to `GAP_SUBEVENT_BONDING_DELETED` event received by registering for HCI events via `hci_add_event_handler(..)`.

GATT Server

The GATT Server's role is to store Characteristics and make them available through Services. The GATT Server can notify the client when the value of a Characteristic changes and it responds to the Client's read and write requests.

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 - provided by `att_db.h` and `att_server.h` - 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 below.

```
// 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}
...
```

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

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

Properties can be a list of properties combined using '|'

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 additional Client Configuration Characteristic.

Property	Description
READ	Characteristic can be read
WRITE	Characteristic can be written using Write Request
WRITE_WITHOUT_RESPONSE	Characteristic can be written using Write Command
NOTIFY	Characteristic allows notifications by server
INDICATE	Characteristic allows indication by server
DYNAMIC	Read or writes to Characteristic are handled by application

To require encryption or authentication before a Characteristic can be accessed, you can add one or more of the following properties:

Property	Description
AUTHENTICATION_REQUIRED	Read and Write operations require Authentication
READ_ENCRYPTED	Read operations require Encryption
READ_AUTHENTICATED	Read operations require Authentication
WRITE_ENCRYPTED	Write operations require Encryption
WRITE_AUTHENTICATED	Write operations require Authentication
ENCRYPTION_KEY_SIZE_X	Require encryption size >= X, with W in [7..16]

For example, Volume State Characteristic (Voice Control Service) requires:

- Mandatory Properties: Read, Notify
- Security Permissions: Encryption Required

In addition, its read is handled by application. We can model this Characteristic as follows:

```
CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_VOLUME_STATE, DYNAMIC
| READ | NOTIFY | ENCRYPTION_KEY_SIZE_16
```

To use already implemented GATT Services, you can import it using the *#import <service_name.gatt>* 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_to_send_notification* to request a callback, when yuo can send the Notification.

Deferred Handling of ATT Read / Write Requests

If your application cannot handle an ATT Read Request in the *att_read_callback* in some situations, you can enable support for this by adding ENABLE_ATT_DELAYED_RESPONSE to *btstack_config.h*. Now, you can store the requested attribute handle and return *ATT_READ_RESPONSE_PENDING* instead of the length of the provided data when you don't have the data ready. For ATT operations that read more than one attribute, your *att_read_callback* might get called multiple times as well. To let you know that all necessary attribute handles have been 'requested' by the *att_server*, you'll get a final *att_read_callback* with the attribute handle of *ATT_READ_RESPONSE_PENDING*. When you've got the data for all requested attributes ready, you can call *att_server_response_ready*, which will trigger processing of the current request. Please keep in

mind that there is only one active ATT operation and that it has a 30 second timeout after which the ATT server is considered defunct by the GATT Client.

Similarly, you can return ATT_ERROR_WRITE_RESPONSE_PENDING in the *att_write_callback*. The ATT Server will not respond to the ATT Client in this case and wait for your code to call *att_server_response_ready*, which then triggers the *att_write_callback* again.

Please have a look at the ATT Delayed Response example.

Implementing Standard GATT Services

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

1. Get the Service specification from Bluetooth SIG
2. Find the Service Characteristics table and their properties
3. Create .gatt file from Service Characteristics table
4. Implement Service server, e.g., battery_service_server.c

Step 1:

All GATT Service specifications can be found here.

Step 2:

The Service Characteristics table is usually in chapter “Service Characteristics”.

Let’s have a look at an actual example, the Battery Service Specification v1.0. In it, we find this:

Characteristic	Ref	Mandatory/Optional
Battery Level	3.1	M

So, the Battery Service has a single mandatory Characteristic “Battery Level”.

Properties	Mandatory/Optional/Excluded
Broadcast	x

Properties	Mandatory/Optional/Excluded
Read	M
Write without Response	x
Write	x
Notify	O
Indicate	x
Signed Write	x
Reliable Write	x
Writable Auxiliaries	x

The Battery Level Characteristic must supports Read and optionally allows for Notifications.

Step 3:

Following the Battery Service v1.0 example, let's create `battery_service.gatt`.

BTstack has a list of most GATT Service and Characteristics UUIDs in `src/bluetooth_gatt.h`, which can be used in .gatt files.

Missing UUIDs can be found in Bluetooth SIG Bitbucket repo: - Service UUIDs - Characteristic UUIDs

First we add the Primary Service definition:

```
// Battery Service v1.0
PRIMARY_SERVICE, ORG_BLUETOOTH_SERVICE_BATTERY_SERVICE
```

Next, we add all Characteristics and map their properties into the format of the .gatt file.

In this example, the Battery Level is dynamic and supports Read and Notification.

```
CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_BATTERY_LEVEL,
DYNAMIC | READ | NOTIFY,
```

Feel free to take a look at already implemented GATT Service .gatt files in `src/ble/gatt-service/`.

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 necessary 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_service_handler()*.

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_with_uuid16*.

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*.

GATT Database Hash

When a GATT Client connects to a GATT Server, it cannot know if the GATT Database has changed and has to discover the provided GATT Services and Characteristics after each connect.

To speed this up, the Bluetooth specification defines a GATT Service Changed Characteristic, with the idea that a GATT Server would notify a bonded GATT Client if its database changed. However, this is quite fragile and it is not clear how it can be implemented in a robust way.

The Bluetooth Core Spec 5.1 introduced the GATT Database Hash Characteristic, which allows for a simple robust mechanism to cache a remote GATT Database. The GATT Database Hash is a 16-byte value that is calculated over the list of Services and Characteristics. If there is any change to the database, the hash will change as well.

To support this on the GATT Server, you only need to add a GATT Service with the GATT Database Characteristic to your .gatt file. The hash value is then calculated by the GATT compiler.

```
PRIMARY_SERVICE, GATT_SERVICE
CHARACTERISTIC, GATT_DATABASE_HASH, READ,
```

Note: make sure to install the PyCryptodome python package as the hash is calculated using AES-CMAC, e.g. with:

```
pip install pycryptodomex
```

GATT Authentication

By default, the GATT Server is responsible for security and the GATT Client does not enforce any kind of authentication. If the GATT Client accesses Characteristic that require encrytion or authentication, the remote GATT Server will return an error, which is returned in the *att status* of the *GATT_EVENT_QUERY_COMPLETE*.

You can define *ENABLE_GATT_CLIENT_PAIRING* to instruct the GATT Client to trigger pairing in this case and to repeat the request.

This model allows for an attacker to spoof another device, but don't require authentication for the Characteristics. As a first improvement, you can define *ENABLE_LE_PROACTIVE_AUTHENTICATION* in *btstack_config.h*. When defined, the GATT Client will request the Security Manager to re-encrypt the connection if there is stored bonding information available. If this fails, the *GATT_EVENT_QUERY_COMPLETE* will return with the att status *ATT_ERROR_BONDING_INFORMATION_MISSING*.

With *ENABLE_LE_PROACTIVE_AUTHENTICATION* defined and in Central role, you need to delete the local bonding information if the remote lost its bonding information, e.g. because of a device reset. See *example/sm_pairing_central.c*.

Even with the Proactive Authentication, your device may still connect to an attacker that provides the same advertising data as your actual device. If the device that you want to connect requires pairing, you can instruct the GATT Client to automatically request an encrypted connection before sending any GATT Client request by calling *gatt_client_set_required_security_level()*. If the device provides sufficient IO capabilities, a MITM attack can then be prevented. We call this 'Mandatory Authentication'.

The following diagrams provide a detailed overview about the GATT Client security mechanisms in different configurations:

- Reactive Authentication as Central
- Reactive Authentication as Peripheral
- Proactive Authentication as Central
- Proactive Authentication as Peripheral
- Mandatory Authentication as Central
- Mandatory Authentication as Peripheral

#Implemented GATT Services

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

To use an already implemented GATT Service Server, you only have to add it to your application's GATT file with: - `#import <service_name.gatt>` 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.

Battery Service Server

The Battery Service allows to query your device's battery level in a standardized way.

To use with your application, add `#import <battery_service.gatt>` to your .gatt file. 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.

See Battery Service Server API.

Battery Service V1 Server

The Battery Service allows to query your device's battery level in a standardized way.

To use with your application, add `#import <battery_service.gatt>` to your .gatt file. 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.

See Battery Service V1 Server API.

Bond Management Service Server

The Bond Management service server defines how a peer Bluetooth device can manage the storage of bond information, especially the deletion of it, on the Bluetooth device supporting this service.

To use with your application, add `#import <bond_management_service.gatt>` to your .gatt file. After adding it to your .gatt file, you call *bond_management_service_server_init(supported_features)*. The input parameter "supported_features" is a bitmap, which defines how the bond information will be deleted, see BMF_DELETE_* flags below.

See Bond Management Service Server API.

Cycling Power Service Server

The Cycling Power Service allows to query device's power- and force-related data and optionally speed- and cadence-related data for use in sports and fitness applications.

To use with your application, add `#import <cycling_power_service.gatt>` to your .gatt file.

See Cycling Power Service Server API.

Cycling Speed and Cadence Service Server

The Cycling Speed and Cadence Service allows to query device's speed- and cadence-related data for use in sports and fitness applications.

To use with your application, add `#import <cycling_speed_and_cadence_service.gatt>` to your .gatt file.

See Cycling Speed and Cadence Service Server API.

Device Information Service Server

Th eDevice Information Service allows to query manufacturer and/or vendor information about a device.

To use with your application, add `#import <device_information_service.gatt>` to your .gatt file.

Note: instead of calling all setters, you can create a local copy of the .gatt file and remove all Characteristics that are not relevant for your application and define all fixed values in the .gatt file.

See Device Information Service Server API.

Heart Rate Service Server

The heart rate service server provides heart rate measurements via notifications.

Each notification reports the heart rate measurement in beats per minute, and if enabled, the total energy expended in kilo Joules, as well as RR-intervals in 1/1024 seconds resolution.

The Energy Expended field represents the accumulated energy expended in kilo Joules since the last time it was reset. If the maximum value of 65535 kilo Joules is reached, it will remain at this value, until a reset command from the client is received.

The RR-Interval represents the time between two consecutive R waves in an Electrocardiogram (ECG) waveform. If needed, the RR-Intervals are sent in multiple notifications.

To use with your application, add `#import <heart_rate_service.gatt>` to your .gatt file. After adding it to your .gatt file, you call *heart_rate_server_init(body_sensor_location, energy_expended_supported)* with the intended sensor location, and a flag indicating if energy expanded is supported.

If heart rate measurement changes, you can call *heart_rate_service_server_update_heart_rate_values(heart_rate_bpm, service_sensor_contact_status, rr_interval_count, rr_intervals)*. This function will trigger sending Notifications if the client enables them.

If energy expanded is supported, you can call *heart_rate_service_add_energy_expended(energy_expended_kJ)* with the newly expanded energy. The accumulated energy expended value will be emitted with the next heart rate measurement.

See Heart Rate Service Server API.

HIDS Device

Implementation of the GATT HIDS Device To use with your application, add ‘#import <hids.gatt>’ to your .gatt file

See HIDS Device API.

Immediate Alert Service Server

To use with your application, add `#import <immediate_alert_service.gatt>` to your .gatt file.

See Immediate Alert Service Server API.

Link Loss Service Server

See Link Loss Service Server API.

Nordic SPP Service Server

The Nordic SPP Service is implementation of the Nordic SPP-like profile.

To use with your application, add `#import <nordic_spp_service.gatt>` to your .gatt file and call all functions below. All strings and blobs need to stay valid after calling the functions.

See Nordic SPP Service Server API.

Scan Parameters Service Server

The Scan Parameters Service enables a remote GATT Client to store the LE scan parameters it is using locally. These parameters can be utilized by the application to optimize power consumption and/or reconnection latency.

To use with your application, add `#import <scan_parameters_service.gatt>` to your .gatt file and call all functions below. All strings and blobs need to stay valid after calling the functions.

See Scan Parameters Service Server API.

Tx Power Service Server

The TX Power service exposes a device's current transmit power level when in a connection. There shall only be one instance of the Tx Power service on a device.

To use with your application, add `#import <tx_power_service.gatt>` to your .gatt file. After adding it to your .gatt file, you call *tx_power_service_server_init(value)* with the device's current transmit power level value.

If the power level value changes, you can call *tx_power_service_server_set_level(tx_power_level_dBm)*. The service does not support sending Notifications.

See Tx Power Service Server API.

u-blox SPP Service Server

The u-blox SPP Service is implementation of the u-Blox SPP-like profile.

To use with your application, add `#import <ublox_spp_service.gatt>` to your .gatt file and call all functions below. All strings and blobs need to stay valid after calling the functions.

See u-blox SPP Service Server API.

#Implemented GATT Clients

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

ANCS Client

The ANCS Client implements Notification Consumer (NC) of the Apple Notification Center Service (ANCS).

See ANCS Client API.

Battery Service Client

The Battery Service Client connects to the Battery Services of a remote device and queries its battery level values. Level updates are either received via notifications (if supported by the remote Battery Service), or by manual polling.

See Battery Service Client API.

Device Information Service Client

The Device Information Service Client retrieves the following information from a remote device: - manufacturer name- model number - serial number - hardware revision- firmware revision- software revision- system ID - IEEE regulatory certification- PNP ID - UDI for medical devices

See Device Information Service Client API.

Immediate Alert Service Client

The Immediate Alert Service Client

See Immediate Alert Service Client API.

Link Loss Service Client

The Link Loss Service Client

See Link Loss Service Client API.

Scan Parameters Service Client

The Scan Parameters Service Client allows to store its LE scan parameters on a remote device such that the remote device can utilize this information to optimize power consumption and/or reconnection latency.

See Scan Parameters Service Client API.

Tx Power Service Client

The Immediate Alert Service Client

See Tx Power Service Client API.

#Examples

In this section, we will describe a number of examples from the *example* folder. Here is a list of existing examples:

- Hello World example:
  - led_counter: Hello World - Blinking an LED without Bluetooth.
- GAP examples:
  - gap_inquiry: GAP Classic Inquiry.
  - gap_link_keys: GAP Link Key Management (Classic).
- Low Energy examples:
  - gap_le_advertisements: GAP LE Advertisements Scanner.
  - gatt_browser: GATT Client - Discover Primary Services.
  - gatt_counter: GATT Server - Heartbeat Counter over GATT.
  - gatt_streamer_server: Performance - Stream Data over GATT (Server).
  - gatt_battery_query: GATT Battery Service Client.

    - gatt_device_information_query: GATT Device Information Service Client.
    - gatt_heart_rate_client: GATT Heart Rate Sensor Client .
    - nordic_spp_le_counter: LE Nordic SPP-like Heartbeat Server .
    - nordic_spp_le_streamer: LE Nordic SPP-like Streamer Server .
    - ublox_spp_le_counter: LE u-blox SPP-like Heartbeat Server.
    - sm_pairing_central: LE Central - Test Pairing Methods.
    - sm_pairing_peripheral: LE Peripheral - Test Pairing Methods.
    - le_credit_based_flow_control_mode_client: LE Credit-Based Flow-Control Mode Client - Send Data over L2CAP.
    - le_credit_based_flow_control_mode_server: LE Credit-Based Flow-Control Mode Server - Receive data over L2CAP.
    - att_delayed_response: LE Peripheral - Delayed Response.
    - ancs_client_demo: LE ANCS Client - Apple Notification Service.
    - le_mitm: LE Man-in-the-Middle Tool.
- Performance examples:
    - le_streamer_client: Performance - Stream Data over GATT (Client).
    - gatt_streamer_server: Performance - Stream Data over GATT (Server).
    - le_credit_based_flow_control_mode_client: LE Credit-Based Flow-Control Mode Client - Send Data over L2CAP.
    - le_credit_based_flow_control_mode_server: LE Credit-Based Flow-Control Mode Server - Receive data over L2CAP.
    - spp_streamer_client: Performance - Stream Data over SPP (Client).
    - spp_streamer: Performance - Stream Data over SPP (Server).

- Audio examples:
  - a2dp_sink_demo: A2DP Sink - Receive Audio Stream and Control Playback.
  - a2dp_source_demo: A2DP Source - Stream Audio and Control Volume.
  - avrcp_browsing_client: AVRCP Browsing - Browse Media Players and Media Information.
  - hfp_ag_demo: HFP AG - Audio Gateway.
  - hfp_hf_demo: HFP HF - Hands-Free.
  - hsp_ag_demo: HSP AG - Audio Gateway.
  - hsp_hs_demo: HSP HS - Headset.
  - sine_player: Audio Driver - Play Sine .
  - mod_player: Audio Driver - Play 80’s MOD Song.
  - audio_duplex: Audio Driver - Forward Audio from Source to Sink.
- SPP Server examples:
  - spp_counter: SPP Server - Heartbeat Counter over RFCOMM.
  - spp_flowcontrol: SPP Server - RFCOMM Flow Control.
- Networking examples:
  - pan_lwip_http_server: PAN - lwIP HTTP and DHCP Server .
  - panu_demo: BNEP/PANU (Linux only).
- HID examples:
  - hid_keyboard_demo: HID Keyboard Classic.
  - hid_mouse_demo: HID Mouse Classic.
  - hid_host_demo: HID Host Classic.
  - hog_keyboard_demo: HID Keyboard LE.

    - hog_mouse_demo: HID Mouse LE.
    - hog_boot_host_demo: HID Boot Host LE.

- Dual Mode examples:
    - spp_and_gatt_counter: Dual Mode - SPP and LE Counter.
    - gatt_streamer_server: Performance - Stream Data over GATT (Server).

- SDP Queries examples:
    - sdp_general_query: SDP Client - Query Remote SDP Records.
    - sdp_rfcomm_query: SDP Client - Query RFCOMM SDP record.
    - sdp_bnep_query: SDP Client - Query BNEP SDP record.

- Phone Book Access example:
    - pbap_client_demo: PBAP Client - Get Contacts from Phonebook Server.

- Testing example:
    - dut_mode_classic: Testing - Enable Device Under Test (DUT) Mode for Classic.

Hello World - Blinking an LED without Bluetooth

Source Code: led_counter.c

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.

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 here 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
  counter++;

  // print and log
  log_info("BTstack Counter %u",   counter);
  printf("BTstack counter %04u\n\r", counter);

  // toggle LED
  hal_led_toggle();

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

Main Application Setup

Listing here shows main application 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;
}
```

GAP Classic Inquiry

Source Code: gap_inquiry.c

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.

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.

Main Application Setup

Listing here shows main application 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;

  // enabled EIR
  hci_set_inquiry_mode(INQUIRY_MODE_RSSI_AND_EIR);

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

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

GAP Link Key Management (Classic)

Source Code: gap_link_keys.c

Shows how to iterate over the Classic Link Keys stored in NVS Link Keys are per device-device bonding. If the Bluetooth Controller can be swapped, e.g. on desktop systems, a Link Key DB for each Controller is needed. We need to wait until the Bluetooth Stack has started up and selected the correct Link Key DB based on the Controller's BD_ADDR.

GAP Link Key Logic

List stored link keys

Bluetooth Logic

Wait for Bluetooth startup before listing the stored link keys

Main Application Setup

Listing here shows main application 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);

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

GAP LE Advertisements Scanner

Source Code: gap_le_advertisements.c

This example shows how to scan and parse advertisements.

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 here.

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

```
static void gap_le_advertisements_setup(void){
  // Active scanning, 100% (scan interval = scan window)
  gap_set_scan_parameters(1,48,48);
  gap_start_scan();

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

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 const 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 const 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"
};
```

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("%04X ", 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:
        for (i=0; i<size;i+=2){
          printf("%02X ", data[i]);
        }
        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:
        for (i=0; i<size;i+=2){
          printf("%02X ", data[i]);
        }
        break;
      case BLUETOOTH_DATA_TYPE_MANUFACTURER_SPECIFIC_DATA: //
Manufacturer Specific Data
      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");
}
```

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 here.

```
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;

  bd_addr_t address;
  uint8_t address_type;
  uint8_t event_type;
  int8_t rssi;
  uint8_t length;
  const uint8_t * data;

  switch (hci_event_packet_get_type(packet)) {
    case GAP_EVENT_ADVERTISING_REPORT:
      gap_event_advertising_report_get_address(packet, address);
      event_type =
gap_event_advertising_report_get_advertising_event_type(packet);
      address_type =
gap_event_advertising_report_get_address_type(packet);
      rssi = gap_event_advertising_report_get_rssi(packet);
```

```
      length =
gap_event_advertising_report_get_data_length(packet);
      data = gap_event_advertising_report_get_data(packet);
      printf("Advertisement (legacy) 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;
#ifdef ENABLE_LE_EXTENDED_ADVERTISING
    case GAP_EVENT_EXTENDED_ADVERTISING_REPORT:
      gap_event_extended_advertising_report_get_address(packet,
address);
      event_type =
gap_event_extended_advertising_report_get_advertising_event_type(pa
      address_type =
gap_event_extended_advertising_report_get_address_type(packet);
      rssi =
gap_event_extended_advertising_report_get_rssi(packet);
      length =
gap_event_extended_advertising_report_get_data_length(packet);
      data =
gap_event_extended_advertising_report_get_data(packet);
      printf("Advertisement (extended) 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;
#endif
    default:
      break;
  }
}
```

GATT Client - Discover Primary Services

Source Code: gatt_browser.c

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.

GATT client setup

In the setup phase, a GATT client must register the HCI and GATT client packet handlers, as shown in Listing here. Additionally, the security 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){

  // 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);
```

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

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 here.

```
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_META_GAP:
      // wait for connection complete
      if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE) break;
      printf("\nGATT browser - CONNECTED\n");
      connection_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);
      // query primary services

gatt_client_discover_primary_services(handle_gatt_client_event,
connection_handle);
      break;
    case HCI_EVENT_DISCONNECTION_COMPLETE:
      printf("\nGATT browser - DISCONNECTED\n");
      break;
    default:
      break;
  }
}
```

GATT Client event handler

Query results and further queries are handled by the GATT client packet handler, as shown in Listing here. Here, upon receiving 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 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:
      // 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
connection_handle, &service);
        break;
      }
      service_index = 0;
      break;
    default:
      break;
  }
}
```

GATT Server - Heartbeat Counter over GATT

Source Code: gatt_counter.c

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.

Main Application Setup

Listing here shows main application 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;

#ifdef ENABLE_GATT_OVER_CLASSIC
static uint8_t gatt_service_buffer[70];
#endif

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);

// Flags general discoverable, BR/EDR supported (== not supported
flag not set) when ENABLE_GATT_OVER_CLASSIC is enabled
```

```
#ifdef ENABLE_GATT_OVER_CLASSIC
#define APP_AD_FLAGS 0x02
#else
#define APP_AD_FLAGS 0x06
#endif

const uint8_t adv_data[] = {
  // Flags general discoverable
  0x02, BLUETOOTH_DATA_TYPE_FLAGS, APP_AD_FLAGS,
  // 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){

  l2cap_init();

  // setup SM: Display only
  sm_init();

#ifdef ENABLE_GATT_OVER_CLASSIC
  // init SDP, create record for GATT and register with SDP
  sdp_init();
  memset(gatt_service_buffer, 0, sizeof(gatt_service_buffer));
  gatt_create_sdp_record(gatt_service_buffer,
sdp_create_service_record_handle(),
ATT_SERVICE_GATT_SERVICE_START_HANDLE,
ATT_SERVICE_GATT_SERVICE_END_HANDLE);
  btstack_assert(de_get_len( gatt_service_buffer) <=
sizeof(gatt_service_buffer));
  sdp_register_service(gatt_service_buffer);

  // configure Classic GAP
```

```
  gap_set_local_name("GATT Counter BR/EDR 00:00:00:00:00:00");
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_discoverable_control(1);
#endif

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

  // 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);

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

  // register for ATT event
  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);

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

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 here.

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

static void beat(void){
  counter++;
  counter_string_len =
btstack_snprintf_assert_complete(counter_string,
sizeof(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);
  }

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

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);
```

```
  if (packet_type != HCI_EVENT_PACKET) return;

  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_HA
(uint8_t*) counter_string, counter_string_len);
      break;
    default:
      break;
  }
}
```

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 here.

```
// 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_HA
{
    return att_read_callback_handle_blob((const uint8_t
*)counter_string, counter_string_len, offset, buffer,
```

```
buffer_size);
  }
  return 0;
}
```

ATT Write

The only valid ATT writes in this example are to the Client Characteristic Configuration, which configures notification and indication and to the the Characteristic Value. 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. If the ATT handle matches the characteristic value handle, we print the write as hexdump See Listing here.

```
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){
  switch (att_handle){
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_C
      le_notification_enabled = little_endian_read_16(buffer, 0)
== GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION;
      con_handle = connection_handle;
      break;
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
      printf("Write: transaction mode %u, offset %u, data (%u
bytes): ", transaction_mode, offset, buffer_size);
      printf_hexdump(buffer, buffer_size);
      break;
    default:
      break;
  }
  return 0;
}
```

Performance - Stream Data over GATT (Server)

Source Code: gatt_streamer_server.c

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.

  In theory, we should also update the connection parameters, but we already get a connection interval of 30 ms and there's no public way to use a shorter interval with iOS (if we're not implementing an HID device).

  Note: To start the streaming, run the example. On remote device use some GATT Explorer, e.g. LightBlue, BLExplr to enable notifications.

Main Application Setup

Listing here shows main application 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){

  l2cap_init();

  // setup SM: Display only
  sm_init();

#ifdef ENABLE_GATT_OVER_CLASSIC
  // init SDP, create record for GATT and register with SDP
  sdp_init();
  memset(gatt_service_buffer, 0, sizeof(gatt_service_buffer));
  gatt_create_sdp_record(gatt_service_buffer, 0x10001,
```

```
ATT_SERVICE_GATT_SERVICE_START_HANDLE,
ATT_SERVICE_GATT_SERVICE_END_HANDLE);
  sdp_register_service(gatt_service_buffer);
  printf("SDP service record size: %u\n",
de_get_len(gatt_service_buffer));

  // configure Classic GAP
  gap_set_local_name("GATT Streamer BR/EDR 00:00:00:00:00:00");
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_discoverable_control(1);
#endif

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

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

  // register for ATT events
  att_server_register_packet_handler(att_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();
}
```

Track throughput

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

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

static void test_track_evaluate(le_streamer_connection_t *
context) {
  // 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_sent = context->test_data_sent * 1000 /
time_passed;
  int bytes_per_second_received = context->test_data_received *
1000 / time_passed;
  int bytes_per_second_total = bytes_per_second_sent +
bytes_per_second_received;
  printf("%c: Sent %u.%03u kB/s, Received %u.%03u kB/s => Total
%u.%03u kB/s\n", context->name,
    bytes_per_second_sent   / 1000, bytes_per_second_sent   %
1000,
    bytes_per_second_received / 1000, bytes_per_second_received %
1000,
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);
  // store into test data buffer for use with GATT Browser
  btstack_snprintf_best_effort(context->test_data, sizeof(context-
>test_data),
    "Notification Throughput %u.%03u kB/s",
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);
```

```
  // restart
  test_reset(context);
}

static void test_track_data(le_streamer_connection_t * context,
uint32_t bytes_sent, uint32_t bytes_received){
  context->test_data_sent   += bytes_sent;
  context->test_data_received += bytes_received;
  test_track_evaluate(context);
}
```

HCI Packet Handler

The packet handler is used track incoming connections and to stop notifications on disconnect It is also a good place to request the connection parameter update as indicated in the commented code block.

```
static void hci_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;

  uint16_t conn_interval;
  hci_con_handle_t con_handle;
  static const char * const phy_names[] = {
    "Reserved", "1 M", "2 M", "Codec"
  };

  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("To start the streaming, please run the
le_streamer_client example on other device, or use some GATT
Explorer, e.g. LightBlue, BLExplr.\n");
```

```
      }
      break;
    case HCI_EVENT_DISCONNECTION_COMPLETE:
      con_handle =
hci_event_disconnection_complete_get_connection_handle(packet);
      printf("- LE Connection 0x%04x: disconnect, reason %02x\n",
con_handle,
hci_event_disconnection_complete_get_reason(packet));
      break;
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          conn_interval =
gap_subevent_le_connection_complete_get_conn_interval(packet);
          printf("- LE Connection 0x%04x: connected - connection
interval %u.%02u ms, latency %u\n", con_handle, conn_interval *
125 / 100,
            25 * (conn_interval & 3),
gap_subevent_le_connection_complete_get_conn_latency(packet));

          // request min con interval 15 ms for iOS 11+
          printf("- LE Connection 0x%04x: request 15 ms connection
interval\n", con_handle);
          gap_request_connection_parameter_update(con_handle, 12,
12, 4, 0x0048);
          break;
        default:
          break;
      }
      break;
    case HCI_EVENT_LE_META:
      switch (hci_event_le_meta_get_subevent_code(packet)) {
        case HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
```

```
hci_subevent_le_connection_update_complete_get_connection_handle(pa
          conn_interval =
hci_subevent_le_connection_update_complete_get_conn_interval(packet
          printf("- LE Connection 0x%04x: connection update -
connection interval %u.%02u ms, latency %u\n", con_handle,
conn_interval * 125 / 100,
            25 * (conn_interval & 3),
hci_subevent_le_connection_update_complete_get_conn_latency(packet)
          break;
        case HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE:
          con_handle =
hci_subevent_le_data_length_change_get_connection_handle(packet);
          printf("- LE Connection 0x%04x: data length change - max
%u bytes per packet\n", con_handle,

hci_subevent_le_data_length_change_get_max_tx_octets(packet));
          break;
        case HCI_SUBEVENT_LE_PHY_UPDATE_COMPLETE:
          con_handle =
hci_subevent_le_phy_update_complete_get_connection_handle(packet);
          printf("- LE Connection 0x%04x: PHY update - using LE %s
PHY now\n", con_handle,

phy_names[hci_subevent_le_phy_update_complete_get_tx_phy(packet)]);
          break;
        default:
          break;
      }
      break;

    default:
      break;
  }
}
```

ATT Packet Handler

The packet handler is used to track the ATT MTU Exchange and trigger ATT send

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

  int mtu;
  le_streamer_connection_t * context;
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (hci_event_packet_get_type(packet)) {
        case ATT_EVENT_CONNECTED:
          // setup new
          context =
connection_for_conn_handle(HCI_CON_HANDLE_INVALID);
          if (!context) break;
          context->connection_handle =
att_event_connected_get_handle(packet);
          context->test_data_len =
btstack_min(att_server_get_mtu(context->connection_handle) - 3,
sizeof(context->test_data));
          // store into test data buffer for use with GATT Browser
          btstack_snprintf_best_effort(context->test_data,
context  ->test_data_len, "GATT Streamer starting up...");
          printf("%c: ATT connected, handle 0x%04x, test data len
%u\n", context->name, context->connection_handle, context-
>test_data_len);
          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_hanc
          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;
          case ATT_EVENT_DISCONNECTED:
            context =
connection_for_conn_handle(att_event_disconnected_get_handle(packet
            if (!context) break;
            // free connection
            printf("%c: ATT disconnected, handle 0x%04x\n", context-
>name, context->connection_handle);
            context->le_notification_enabled = 0;
            context->connection_handle = HCI_CON_HANDLE_INVALID;
            break;
          default:
            break;
        }
        break;
      default:
        break;
    }
}
```

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];

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

  // track
  test_track_data(context, context->test_data_len, 0);

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

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

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 here.

```
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);

  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_C
      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){
        switch (att_handle){
          case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_C
            context->value_handle =
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
            break;
          default:
            break;
        }
        att_server_request_can_send_now_event(context-
>connection_handle);
      }
      test_reset(context);
      break;
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
      test_track_data(context, 0, buffer_size);
      break;
    default:
      printf("Write to 0x%04x, len %u\n", att_handle,
buffer_size);
      break;
  }
  return 0;
}
```

GATT Battery Service Client

Source Code: gatt_battery_query.c

This example demonstrates how to use the GATT Battery Service client to receive battery level information. The client supports querying of multiple battery services instances of on the remote device. The example scans for remote devices and connects to the first found device and starts the battery service client.

Main Application Setup

The Listing here shows how to setup Battery Service client. Besides calling init() method for each service, you'll also need to register HCI packet handler to handle advertisements, as well as connect and disconect events.

Handling of GATT Battery Service events will be later delegated to a sepparate packet handler, i.e. gatt_client_event_handler.

@note There are two additional files associated with this client to allow a remote device to query out GATT database:

- gatt_battary_query.gatt - contains the declaration of the provided GATT Services and Characteristics.
- gatt_battary_query.h - contains the binary representation of gatt_battary_query.gatt.

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

static void battery_service_client_setup(void){
  // Init L2CAP
  l2cap_init();

  // Setup ATT server - only needed if LE Peripheral does ATT
queries on its own, e.g. Android phones
  att_server_init(profile_data, NULL, NULL);
```

```
  // GATT Client setup
  gatt_client_init();
  // Device Information Service Client setup
  battery_service_client_init();

  sm_init();
  sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);

  hci_event_callback_registration.callback = &hci_event_handler;
  hci_add_event_handler(&hci_event_callback_registration);
}


static void hci_event_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size){
...

  if (packet_type != HCI_EVENT_PACKET){
    return;
  }

  switch (hci_event_packet_get_type(packet)) {
...
    case HCI_EVENT_META_GAP:
      // Wait for connection complete
      if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE) break;

...
      // Get connection handle from event
      connection_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);

      // Connect to remote Battery Service.
      // On succesful connection, the client tries to register for
notifications. If notifications
      // are not supported by remote Battery Service, the client
will automatically poll the battery level - here every 2 seconds.
      // If poll_interval_ms is 0, polling is disabled, and only
notifications will be received (for manual polling,
      // see battery_service_client.h).
```

```
      // All GATT Battery Service events are handled by the
gatt_client_event_handler.
      (void) battery_service_client_connect(connection_handle,
gatt_client_event_handler, 2000, &battery_service_cid);

      app_state = APP_STATE_CONNECTED;
      printf("Battery service connected.\n");
      break;

    case HCI_EVENT_DISCONNECTION_COMPLETE:
      connection_handle = HCI_CON_HANDLE_INVALID;
      // Disconnect battery service
      battery_service_client_disconnect(battery_service_cid);

...
      printf("Disconnected %s\n", bd_addr_to_str(report.address));
      printf("Restart scan.\n");
      app_state = APP_STATE_W4_SCAN_RESULT;
      gap_start_scan();
      break;
    default:
      break;
  }
}

// The gatt_client_event_handler receives following events from
remote device:
//  - GATTSERVICE_SUBEVENT_BATTERY_SERVICE_CONNECTED
//  - GATTSERVICE_SUBEVENT_BATTERY_SERVICE_LEVEL
//
static void gatt_client_event_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size){
...
  uint8_t status;
  uint8_t att_status;

  if (hci_event_packet_get_type(packet) !=
HCI_EVENT_GATTSERVICE_META){
    return;
  }
```

```
  switch (hci_event_gattservice_meta_get_subevent_code(packet)){
    case GATTSERVICE_SUBEVENT_BATTERY_SERVICE_CONNECTED:
      status =
gattservice_subevent_battery_service_connected_get_status(packet);
      switch (status){
        case ERROR_CODE_SUCCESS:
          printf("Battery service client connected, found %d
services, poll bitmap 0x%02x\n",

gattservice_subevent_battery_service_connected_get_num_instances(pa

gattservice_subevent_battery_service_connected_get_poll_bitmap(pack

battery_service_client_read_battery_level(battery_service_cid, 0);
          break;
        default:
          printf("Battery service client connection failed, status
0x%02x.\n", status);
          add_to_blacklist(report.address);
          gap_disconnect(connection_handle);
          break;
      }
      break;

    case GATTSERVICE_SUBEVENT_BATTERY_SERVICE_LEVEL:
      att_status =
gattservice_subevent_battery_service_level_get_att_status(packet);
      if (att_status != ATT_ERROR_SUCCESS){
        printf("Battery level read failed, ATT Error 0x%02x\n",
att_status);
      } else {
        printf("Service index: %d, Battery level: %d\n",

gattservice_subevent_battery_service_level_get_service_index(packet

gattservice_subevent_battery_service_level_get_level(packet));

      }
      break;
```

```
    default:
      break;
  }
}
```

GATT Device Information Service Client

Source Code: gatt_device_information_query.c

This example demonstrates how to use the GATT Device Information Service client to receive device information such as various IDs and revisions. The example scans for remote devices and connects to the first found device. If the remote device provides a Device Information Service, the information is collected and printed in console output, otherwise, the device will be blacklisted and the scan restarted.

Main Application Setup

The Listing here shows how to setup Device Information Service client. Besides calling init() method for each service, you'll also need to register HCI packet handler to handle advertisements, as well as connect and disconect events.

Handling of GATT Device Information Service events will be later delegated to a sepparate packet handler, i.e. gatt_client_event_handler.

@note There are two additional files associated with this client to allow a remote device to query out GATT database:

- gatt_device_information_query.gatt - contains the declaration of the provided GATT Services and Characteristics.
- gatt_device_information_query.h - contains the binary representation of gatt_device_information_query.gatt.

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

```
static void device_information_service_client_setup(void){
  // Init L2CAP
  l2cap_init();

  // Setup ATT server - only needed if LE Peripheral does ATT
queries on its own, e.g. Android phones
  att_server_init(profile_data, NULL, NULL);

  // GATT Client setup
  gatt_client_init();
  // Device Information Service Client setup
  device_information_service_client_init();

  sm_init();
  sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);

  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);
}


static void hci_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size){
...
    case HCI_EVENT_META_GAP:
      // wait for connection complete
      if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE) break;

...
      // get connection handle from event
      connection_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);

      // Connect to remote Device Information Service. The client
will query the remote service and emit events,
      // that will be passed on to gatt_client_event_handler.
      status =
device_information_service_client_query(connection_handle,
gatt_client_event_handler);
      btstack_assert(status == ERROR_CODE_SUCCESS);
```

```
      printf("Device Information connected.\n");

      app_state = APP_STATE_CONNECTED;
      break;
...
  uint8_t att_status = 0;

  if (hci_event_packet_get_type(packet) !=
HCI_EVENT_GATTSERVICE_META){
    return;
  }

  switch (hci_event_gattservice_meta_get_subevent_code(packet)){
    case
GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_MANUFACTURER_NAME:
      att_status =
gattservice_subevent_device_information_manufacturer_name_get_att_s
      if (att_status != ATT_ERROR_SUCCESS){
        printf("Manufacturer Name read failed, ATT Error
0x%02x\n", att_status);
      } else {
        printf("Manufacturer Name: %s\n",
gattservice_subevent_device_information_manufacturer_name_get_value
      }
      break;

    // ...
...
    case GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_DONE:
      att_status =
gattservice_subevent_device_information_serial_number_get_att_statu
      switch (att_status){
        case ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE:
          printf("Device Information service client not found.
\n");
          add_to_blacklist(report.address);
          gap_disconnect(connection_handle);
          break;
        case ATT_ERROR_SUCCESS:
```

```
            printf("Query done\n");
            break;
          default:
            printf("Query failed, ATT Error 0x%02x\n", att_status);
            break;

      }
      if (att_status != ATT_ERROR_SUCCESS){
        if (att_status ==
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE)
        printf("Query failed, ATT Error 0x%02x\n", att_status);
      } else {
        printf("Query done\n");
      }
      break;
    default:
      break;
  }
}
```

GATT Heart Rate Sensor Client

Source Code: gatt_heart_rate_client.c

Connects for Heart Rate Sensor and reports measurements.

LE Nordic SPP-like Heartbeat Server

Source Code: nordic_ssp_le_counter.c

Main Application Setup

Listing here shows main application code. It initializes L2CAP, the Security Manager and configures the ATT Server with the pre-compiled ATT Database generated from $nordic_{s}sp_{l}e_{c}ounter.gatt$. 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 btstack_timer_source_t heartbeat;
static hci_con_handle_t con_handle = HCI_CON_HANDLE_INVALID;
static btstack_context_callback_registration_t send_request;

const uint8_t adv_data[] = {
  // Flags general discoverable, BR/EDR not supported
  2, BLUETOOTH_DATA_TYPE_FLAGS, 0x06,
  // Name
  8, BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME, 'n', 'R', 'F',' ',
'S', 'P', 'P',
  // UUID ...
  17,
BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_128_BIT_SERVICE_CLASS_UUIDS,
0x9e, 0xca, 0xdc, 0x24, 0xe, 0xe5, 0xa9, 0xe0, 0x93, 0xf3, 0xa3,
0xb5, 0x1, 0x0, 0x40, 0x6e,
};
const uint8_t adv_data_len = sizeof(adv_data);
```

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 here.

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

static void beat(void){
  counter++;
  counter_string_len =
btstack_snprintf_assert_complete(counter_string,
sizeof(counter_string), "BTstack counter %03u", counter);
}
```

```
static void nordic_can_send(void * context){
  UNUSED(context);
  printf("SEND: %s\n", counter_string);
  nordic_spp_service_server_send(con_handle, (uint8_t*)
counter_string, counter_string_len);
}

static void heartbeat_handler(struct btstack_timer_source *ts){
  if (con_handle != HCI_CON_HANDLE_INVALID) {
    beat();
    send_request.callback = &nordic_can_send;
    nordic_spp_service_server_request_can_send_now(&send_request,
con_handle);
  }
  btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(ts);
}
```

LE Nordic SPP-like Streamer Server

Source Code: nordic_spp_le_streamer.c

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.

  In theory, we should also update the connection parameters, but we already get a connection interval of 30 ms and there's no public way to use a shorter interval with iOS (if we're not implementing an HID device).

  Note: To start the streaming, run the example. On remote device use some GATT Explorer, e.g. LightBlue, BLExplr to enable notifications.

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(nordic_spp_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(nordic_spp_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: %"PRIu32" 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;
}
```

HCI Packet Handler

The packet handler prints the welcome message and requests a connection paramter update for LE Connections

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

```
  UNUSED(size);

  uint16_t conn_interval;
  hci_con_handle_t con_handle;

  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("To start the streaming, please run nRF Toolbox ->
UART to connect.\n");
      }
      break;
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          conn_interval =
gap_subevent_le_connection_complete_get_conn_interval(packet);
          printf("LE Connection - Connection Interval: %u.%02u
ms\n", conn_interval * 125 / 100, 25 * (conn_interval & 3));
          printf("LE Connection - Connection Latency: %u\n",
gap_subevent_le_connection_complete_get_conn_latency(packet));

          // request min con interval 15 ms for iOS 11+
          printf("LE Connection - Request 15 ms connection
interval\n");
          gap_request_connection_parameter_update(con_handle, 12,
12, 4, 0x0048);
          break;
        default:
          break;
      }
      break;
```

```
    case HCI_EVENT_LE_META:
      switch (hci_event_le_meta_get_subevent_code(packet)) {
        case HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
hci_subevent_le_connection_update_complete_get_connection_handle(pa
          conn_interval =
hci_subevent_le_connection_update_complete_get_conn_interval(packet
          printf("LE Connection - Connection Param update -
connection interval %u.%02u ms, latency %u\n", conn_interval *
125 / 100,
            25 * (conn_interval & 3),
hci_subevent_le_connection_update_complete_get_conn_latency(packet)
          break;
        default:
          break;
      }
      break;
    default:
      break;
  }
}
```

ATT Packet Handler

The packet handler is used to setup and tear down the spp-over-gatt connection and its MTU

```
static void att_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;

  int mtu;
  nordic_spp_le_streamer_connection_t * context;
```

```
  switch (hci_event_packet_get_type(packet)) {
    case ATT_EVENT_CONNECTED:
      // setup new
      context =
connection_for_conn_handle(HCI_CON_HANDLE_INVALID);
      if (!context) break;
      context->counter = 'A';
      context->test_data_len = ATT_DEFAULT_MTU - 4;   // -1 for
nordic 0x01 packet type
      context->connection_handle =
att_event_connected_get_handle(packet);
      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_hanc
      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;

    default:
      break;
  }
}
```

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 nordic_can_send(void * some_context){
  UNUSED(some_context);

  // find next active streaming connection
  int old_connection_index = connection_index;
  while (1){
```

```
    // active found?
    if
((nordic_spp_le_streamer_connections[connection_index].connection_h
= HCI_CON_HANDLE_INVALID) &&

(nordic_spp_le_streamer_connections[connection_index].le_notificati
break;

    // check next
    next_connection_index();

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

  nordic_spp_le_streamer_connection_t * context =
&nordic_spp_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
  nordic_spp_service_server_send(context->connection_handle,
(uint8_t*) context->test_data, context->test_data_len);

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

  // request next send event
  nordic_spp_service_server_request_can_send_now(&context-
>send_request, context->connection_handle);

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

LE u-blox SPP-like Heartbeat Server

Source Code: ublox_spp_le_counter.c

Main Application Setup

Listing here shows main application code. It initializes L2CAP, the Security Manager and configures the ATT Server with the pre-compiled ATT Database generated from *ublox_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 btstack_timer_source_t heartbeat;
static hci_con_handle_t con_handle = HCI_CON_HANDLE_INVALID;
static btstack_context_callback_registration_t send_request;
static btstack_packet_callback_registration_t
hci_event_callback_registration;

const uint8_t adv_data[] = {
  // Flags general discoverable, BR/EDR not supported
  2, BLUETOOTH_DATA_TYPE_FLAGS, 0x06,
  // Name
  5, BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME, '6', '-','5', '6',
  // UUID ...
  17,
BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_128_BIT_SERVICE_CLASS_UUIDS,
0x1, 0xd7, 0xe9, 0x1, 0x4f, 0xf3, 0x44, 0xe7, 0x83, 0x8f, 0xe2,
0x26, 0xb9, 0xe1, 0x56, 0x24,
};

const uint8_t adv_data_len = sizeof(adv_data);
```

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 here.

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

static void beat(void){
  counter++;
  counter_string_len =
btstack_snprintf_assert_complete(counter_string,
sizeof(counter_string), "BTstack counter %03u", counter);
}

static void ublox_can_send(void * context){
  UNUSED(context);
  beat();
  printf("SEND: %s\n", counter_string);
  ublox_spp_service_server_send(con_handle, (uint8_t*)
counter_string, counter_string_len);
}

static void heartbeat_handler(struct btstack_timer_source *ts){
  if (con_handle != HCI_CON_HANDLE_INVALID) {
    send_request.callback = &ublox_can_send;
    ublox_spp_service_server_request_can_send_now(&send_request,
con_handle);
  }
  btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  btstack_run_loop_add_timer(ts);
}
```

Packet Handler

The packet handler is used to:

- stop the counter after a disconnect

```
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:
          con_handle = HCI_CON_HANDLE_INVALID;
          break;
        default:
          break;
      }
      break;
    default:
      break;
  }
}
```

LE Central - Test Pairing Methods

Source Code: sm_pairing_central.c

Depending on the Authentication requiremens and IO Capabilities, the pairing process uses different short and long term key generation method. This example helps explore the different options incl. LE Secure Connections. It scans for advertisements and connects to the first device that lists a random service.

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 here.

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

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

static void sm_pairing_central_setup(void) {
  l2cap_init();

  // setup SM: Display only
  sm_init();

  // setup ATT server
  att_server_init(profile_data, NULL, NULL);

  // setup GATT Client
  gatt_client_init();

  // register handler
  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  sm_event_callback_registration.callback = &sm_packet_handler;
  sm_add_event_handler(&sm_event_callback_registration);


  // Configuration

  // Enable mandatory authentication for GATT Client
  // - if un-encrypted connections are not supported, e.g. when
connecting to own device, this enforces authentication
  // gatt_client_set_required_security_level(LEVEL_2);

  /**
   * Choose ONE of the following configurations
   * Bonding is disabled to allow for repeated testing. It can be
enabled by or'ing
   * SM_AUTHREQ_BONDING to the authentication requirements like
this:
   * sm_set_authentication_requirements( X | SM_AUTHREQ_BONDING)

  // LE Legacy Pairing, Just Works
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_YES_NO);
```

```
  // sm_set_authentication_requirements(0);

  // LE Legacy Pairing, Passkey entry initiator enter, responder
(us) displays
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_MITM_PROTECTION);
  // sm_use_fixed_passkey_in_display_role(FIXED_PASSKEY);

#ifdef ENABLE_LE_SECURE_CONNECTIONS

  // enable LE Secure Connections Only mode - disables Legacy
pairing
  // sm_set_secure_connections_only_mode(true);

  // LE Secure Connections, Just Works
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_YES_NO);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION);

  // LE Secure Connections, Numeric Comparison
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_YES_NO);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);

  // LE Secure Pairing, Passkey entry initiator (us) enters,
responder displays
  // sm_set_io_capabilities(IO_CAPABILITY_KEYBOARD_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);
  // sm_use_fixed_passkey_in_display_role(FIXED_PASSKEY);

  // LE Secure Pairing, Passkey entry initiator (us) displays,
responder enters
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);
```

```
#endif
}
```

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 here.

HCI packet handler

```
static void hci_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;
  hci_con_handle_t con_handle;
  uint8_t status;

  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(1,0x0030, 0x0030);
        gap_start_scan();
      }
      break;
    case GAP_EVENT_ADVERTISING_REPORT:{
      bd_addr_t address;
      gap_event_advertising_report_get_address(packet, address);
      uint8_t address_type =
gap_event_advertising_report_get_address_type(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: addr-type %u, addr %s,
data[%u] ",
      //   address_type, bd_addr_to_str(address), length);
      // printf_hexdump(data, length);
      if (!ad_data_contains_uuid16(length, (uint8_t *) data,
REMOTE_SERVICE)) break;
      printf("Found remote with UUID %04x, connecting...\n",
REMOTE_SERVICE);
      gap_stop_scan();
      gap_connect(address,address_type);
      break;
    }
    case HCI_EVENT_META_GAP:
      // wait for connection complete
      if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE) break;
      con_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);
      printf("Connection complete\n");

      // for testing, choose one of the following actions

      // manually start pairing
      sm_request_pairing(con_handle);

      // gatt client request to authenticated characteristic in
sm_pairing_peripheral (short cut, uses hard-coded value handle)
      //
gatt_client_read_value_of_characteristic_using_value_handle(&hci_pa
con_handle, 0x0009);

      // general gatt client request to trigger mandatory
authentication
      //
gatt_client_discover_primary_services(&hci_packet_handler,
con_handle);
      break;
    case GATT_EVENT_QUERY_COMPLETE:
      status = gatt_event_query_complete_get_att_status(packet);
```

```
      switch (status){
        case ATT_ERROR_INSUFFICIENT_ENCRYPTION:
          printf("GATT Query result: Insufficient Encryption\n");
          break;
        case ATT_ERROR_INSUFFICIENT_AUTHENTICATION:
          printf("GATT Query result: Insufficient
Authentication\n");
          break;
        case ATT_ERROR_BONDING_INFORMATION_MISSING:
          printf("GATT Query result: Bonding Information
Missing\n");
          break;
        case ATT_ERROR_SUCCESS:
          printf("GATT Query result: OK\n");
          break;
        default:
          printf("GATT Query result: 0x%02x\n",
gatt_event_query_complete_get_att_status(packet));
          break;
      }
      break;
    default:
      break;
  }
}

 *


static void sm_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;

  bd_addr_t addr;
  bd_addr_type_t addr_type;

  switch (hci_event_packet_get_type(packet)) {
```

```
    case SM_EVENT_JUST_WORKS_REQUEST:
      printf("Just works requested\n");

sm_just_works_confirm(sm_event_just_works_request_get_handle(packet
      break;
    case SM_EVENT_NUMERIC_COMPARISON_REQUEST:
      printf("Confirming numeric comparison: %"PRIu32"\n",
sm_event_numeric_comparison_request_get_passkey(packet));

sm_numeric_comparison_confirm(sm_event_passkey_display_number_get_h
      break;
    case SM_EVENT_PASSKEY_DISPLAY_NUMBER:
      printf("Display Passkey: %"PRIu32"\n",
sm_event_passkey_display_number_get_passkey(packet));
      break;
    case SM_EVENT_PASSKEY_INPUT_NUMBER:
      printf("Passkey Input requested\n");
      printf("Sending fixed passkey %"PRIu32"\n", (uint32_t)
FIXED_PASSKEY);

sm_passkey_input(sm_event_passkey_input_number_get_handle(packet),
FIXED_PASSKEY);
      break;
    case SM_EVENT_PAIRING_STARTED:
      printf("Pairing started\n");
      break;
    case SM_EVENT_PAIRING_COMPLETE:
      switch (sm_event_pairing_complete_get_status(packet)){
        case ERROR_CODE_SUCCESS:
          printf("Pairing complete, success. CTKD active %u\n",
sm_event_pairing_complete_get_ctkd_active(packet));
          break;
        case ERROR_CODE_CONNECTION_TIMEOUT:
          printf("Pairing failed, timeout\n");
          break;
        case ERROR_CODE_REMOTE_USER_TERMINATED_CONNECTION:
          printf("Pairing failed, disconnected\n");
          break;
        case ERROR_CODE_AUTHENTICATION_FAILURE:
          printf("Pairing failed, authentication failure with
```

```
reason = %u\n", sm_event_pairing_complete_get_reason(packet));
          break;
        default:
          break;
      }
      break;
    case SM_EVENT_REENCRYPTION_STARTED:
      sm_event_reencryption_complete_get_address(packet, addr);
      printf("Bonding information exists for addr type %u,
identity addr %s -> start re-encryption\n",
           sm_event_reencryption_started_get_addr_type(packet),
bd_addr_to_str(addr));
      break;
    case SM_EVENT_REENCRYPTION_COMPLETE:
      switch (sm_event_reencryption_complete_get_status(packet)){
        case ERROR_CODE_SUCCESS:
          printf("Re-encryption complete, success\n");
          break;
        case ERROR_CODE_CONNECTION_TIMEOUT:
          printf("Re-encryption failed, timeout\n");
          break;
        case ERROR_CODE_REMOTE_USER_TERMINATED_CONNECTION:
          printf("Re-encryption failed, disconnected\n");
          break;
        case ERROR_CODE_PIN_OR_KEY_MISSING:
          printf("Re-encryption failed, bonding information
missing\n\n");
          printf("Assuming remote lost bonding information\n");
          printf("Deleting local bonding information and start new
pairing...\n");
          sm_event_reencryption_complete_get_address(packet,
addr);
          addr_type =
sm_event_reencryption_started_get_addr_type(packet);
          gap_delete_bonding(addr_type, addr);

sm_request_pairing(sm_event_reencryption_complete_get_handle(packet
          break;
        default:
          break;
```

```
      }
      break;
    default:
      break;
  }
}
```

The SM packet handler receives Security Manager Events required for pairing. It also receives events generated during Identity Resolving see Listing here.

LE Peripheral - Test Pairing Methods

Source Code: sm_pairing_peripheral.c

Depending on the Authentication requiremens and IO Capabilities, the pairing process uses different short and long term key generation method. This example helps explore the different options incl. LE Secure Connections.

Main Application Setup

Listing here shows main application code. It initializes L2CAP, the Security Manager and configures the ATT Server with the pre-compiled ATT Database generated from $sm_pairing_peripheral.gatt$. Finally, it configures the advertisements and boots the Bluetooth stack. In this example, the Advertisement contains the Flags attribute, the device name, and a 16-bit (test) service 0x1111 The flag 0x06 indicates: LE General Discoverable Mode and BR/EDR not supported. Various examples for IO Capabilites and Authentication Requirements are given below.

```
static btstack_packet_callback_registration_t
sm_event_callback_registration;
static btstack_packet_callback_registration_t
hci_event_callback_registration;

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

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

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, 'S', 'M', ' ',
'P', 'a', 'i', 'r', 'i', 'n', 'g',
  // Incomplete List of 16-bit Service Class UUIDs -- 1111 - only
valid for testing!
  0x03,
BLUETOOTH_DATA_TYPE_INCOMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS,
0x11, 0x11,
};
const uint8_t adv_data_len = sizeof(adv_data);

static void sm_peripheral_setup(void){

  l2cap_init();

  // setup SM: Display only
  sm_init();

  // setup ATT server
  att_server_init(profile_data, NULL, NULL);

  // setup GATT Client
  gatt_client_init();

  // 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);
```

```
  // register handler
  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  sm_event_callback_registration.callback = &sm_packet_handler;
  sm_add_event_handler(&sm_event_callback_registration);

  // Configuration

  // Enable mandatory authentication for GATT Client
  // - if un-encrypted connections are not supported, e.g. when
connecting to own device, this enforces authentication
  // gatt_client_set_required_security_level(LEVEL_2);

  /**
   * Choose ONE of the following configurations
   * Bonding is disabled to allow for repeated testing. It can be
enabled by or'ing
   * SM_AUTHREQ_BONDING to the authentication requirements like
this:
   * sm_set_authentication_requirements( X | SM_AUTHREQ_BONDING)

  // LE Legacy Pairing, Just Works
  // sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
  // sm_set_authentication_requirements(0);

  // LE Legacy Pairing, Passkey entry initiator enter, responder
(us) displays
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_MITM_PROTECTION);
  // sm_use_fixed_passkey_in_display_role(123456);

#ifdef ENABLE_LE_SECURE_CONNECTIONS

  // enable LE Secure Connections Only mode - disables Legacy
pairing
  // sm_set_secure_connections_only_mode(true);
```

```
  // LE Secure Connections, Just Works
  // sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION);

  // LE Secure Connections, Numeric Comparison
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_YES_NO);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);

  // LE Secure Pairing, Passkey entry initiator enter, responder
(us) displays
  // sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);
  // sm_use_fixed_passkey_in_display_role(123456);

  // LE Secure Pairing, Passkey entry initiator displays,
responder (us) enter
  // sm_set_io_capabilities(IO_CAPABILITY_KEYBOARD_ONLY);
  //
sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION|
SM_AUTHREQ_MITM_PROTECTION);
#endif
}
```

Security Manager Packet Handler

The packet handler is used to handle Security Manager events

HCI Packet Handler

```
static void sm_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;

  hci_con_handle_t con_handle;
  bd_addr_t addr;
  bd_addr_type_t addr_type;
  uint8_t status;

  switch (hci_event_packet_get_type(packet)) {
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
          printf("Connection complete\n");
          con_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          UNUSED(con_handle);

          // for testing, choose one of the following actions

          // manually start pairing
          // sm_request_pairing(con_handle);

          // gatt client request to authenticated characteristic
in sm_pairing_central (short cut, uses hard-coded value handle)
          //
gatt_client_read_value_of_characteristic_using_value_handle(&packet
con_handle, 0x0009);

          // general gatt client request to trigger mandatory
authentication
          //
gatt_client_discover_primary_services(&packet_handler,
con_handle);
          break;
        default:
          break;
      }
      break;
    case SM_EVENT_JUST_WORKS_REQUEST:
      printf("Just Works requested\n");
```

```
sm_just_works_confirm(sm_event_just_works_request_get_handle(packet
      break;
    case SM_EVENT_NUMERIC_COMPARISON_REQUEST:
      printf("Confirming numeric comparison: %"PRIu32"\n",
sm_event_numeric_comparison_request_get_passkey(packet));

sm_numeric_comparison_confirm(sm_event_passkey_display_number_get_h
      break;
    case SM_EVENT_PASSKEY_DISPLAY_NUMBER:
      printf("Display Passkey: %"PRIu32"\n",
sm_event_passkey_display_number_get_passkey(packet));
      break;
    case SM_EVENT_IDENTITY_CREATED:
      sm_event_identity_created_get_identity_address(packet,
addr);
      printf("Identity created: type %u address %s\n",
sm_event_identity_created_get_identity_addr_type(packet),
bd_addr_to_str(addr));
      break;
    case SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED:

sm_event_identity_resolving_succeeded_get_identity_address(packet,
addr);
      printf("Identity resolved: type %u address %s\n",
sm_event_identity_resolving_succeeded_get_identity_addr_type(packet
bd_addr_to_str(addr));
      break;
    case SM_EVENT_IDENTITY_RESOLVING_FAILED:
      sm_event_identity_created_get_address(packet, addr);
      printf("Identity resolving failed\n");
      break;
    case SM_EVENT_PAIRING_STARTED:
      printf("Pairing started\n");
      break;
    case SM_EVENT_PAIRING_COMPLETE:
      switch (sm_event_pairing_complete_get_status(packet)){
        case ERROR_CODE_SUCCESS:
          printf("Pairing complete, success. CTKD active %u\n",
sm_event_pairing_complete_get_ctkd_active(packet));
          break;
```

```
        case ERROR_CODE_CONNECTION_TIMEOUT:
          printf("Pairing failed, timeout\n");
          break;
        case ERROR_CODE_REMOTE_USER_TERMINATED_CONNECTION:
          printf("Pairing failed, disconnected\n");
          break;
        case ERROR_CODE_AUTHENTICATION_FAILURE:
          printf("Pairing failed, authentication failure with
reason = %u\n", sm_event_pairing_complete_get_reason(packet));
          break;
        default:
          break;
      }
      break;
    case SM_EVENT_REENCRYPTION_STARTED:
      sm_event_reencryption_complete_get_address(packet, addr);
      printf("Bonding information exists for addr type %u,
identity addr %s -> re-encryption started\n",
           sm_event_reencryption_started_get_addr_type(packet),
bd_addr_to_str(addr));
      break;
    case SM_EVENT_REENCRYPTION_COMPLETE:
      switch (sm_event_reencryption_complete_get_status(packet)){
        case ERROR_CODE_SUCCESS:
          printf("Re-encryption complete, success\n");
          break;
        case ERROR_CODE_CONNECTION_TIMEOUT:
          printf("Re-encryption failed, timeout\n");
          break;
        case ERROR_CODE_REMOTE_USER_TERMINATED_CONNECTION:
          printf("Re-encryption failed, disconnected\n");
          break;
        case ERROR_CODE_PIN_OR_KEY_MISSING:
          printf("Re-encryption failed, bonding information
missing\n\n");
          printf("Assuming remote lost bonding information\n");
          printf("Deleting local bonding information to allow for
new pairing...\n");
          sm_event_reencryption_complete_get_address(packet,
addr);
```

```
          addr_type =
sm_event_reencryption_started_get_addr_type(packet);
          gap_delete_bonding(addr_type, addr);
          break;
        default:
          break;
      }
      break;
    case GATT_EVENT_QUERY_COMPLETE:
      status = gatt_event_query_complete_get_att_status(packet);
      switch (status){
        case ATT_ERROR_INSUFFICIENT_ENCRYPTION:
          printf("GATT Query failed, Insufficient Encryption\n");
          break;
        case ATT_ERROR_INSUFFICIENT_AUTHENTICATION:
          printf("GATT Query failed, Insufficient
Authentication\n");
          break;
        case ATT_ERROR_BONDING_INFORMATION_MISSING:
          printf("GATT Query failed, Bonding Information
Missing\n");
          break;
        case ATT_ERROR_SUCCESS:
          printf("GATT Query successful\n");
          break;
        default:
          printf("GATT Query failed, status 0x%02x\n",
gatt_event_query_complete_get_att_status(packet));
          break;
      }
      break;
    default:
      break;
  }
}

/*
 *
static void hci_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;

  hci_con_handle_t con_handle;

  switch (hci_event_packet_get_type(packet)) {
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
          printf("Connection complete\n");
          con_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          UNUSED(con_handle);

          // for testing, choose one of the following actions

          // manually start pairing
          // sm_request_pairing(con_handle);

          // gatt client request to authenticated characteristic
in sm_pairing_central (short cut, uses hard-coded value handle)
          //
gatt_client_read_value_of_characteristic_using_value_handle(&packet
con_handle, 0x0009);

          // general gatt client request to trigger mandatory
authentication
          //
gatt_client_discover_primary_services(&packet_handler,
con_handle);
          break;
        default:
          break;
      }
      break;
    default:
      break;
```

```
  }
}
```

The packet handler is used to handle new connections, can trigger Security Request

LE Credit-Based Flow-Control Mode Client - Send Data over L2CAP

Source Code: le_credit_based_flow_control_mode_client.c

Connects to ‘LE CBM Server’ and streams data via L2CAP Channel in LE Credit-Based Flow-Control Mode (CBM)

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.

```
#define REPORT_INTERVAL_MS 3000

// support for multiple clients
typedef struct {
  char name;
  hci_con_handle_t connection_handle;
  uint16_t cid;
  int  counter;
  char test_data[TEST_PACKET_SIZE];
  int  test_data_len;
  uint32_t test_data_sent;
  uint32_t test_data_start;
} le_cbm_connection_t;

static le_cbm_connection_t le_cbm_connection;

static void test_reset(le_cbm_connection_t * context){
  context->test_data_start = btstack_run_loop_get_time_ms();
```

```
  context->test_data_sent = 0;
}

static void test_track_data(le_cbm_connection_t * context, int
bytes_transferred){
  context->test_data_sent += bytes_transferred;
  // 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: %"PRIu32" bytes -> %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;
}
```

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){

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

  // send
  l2cap_send(le_cbm_connection.cid, (uint8_t *)
le_cbm_connection.test_data, le_cbm_connection.test_data_len);
```

```
  // track
  test_track_data(&le_cbm_connection,
le_cbm_connection.test_data_len);

  // request another packet
  l2cap_request_can_send_now_event(le_cbm_connection.cid);
}
```

SM Packet Handler

The packet handler is used to handle pairing requests

LE Credit-Based Flow-Control Mode Server - Receive data over L2CAP

Source Code: le_credit_based_flow_control_mode_server.c

iOS 11 and newer supports L2CAP channels in LE Credit-Based Flow-Control Mode for fast transfer over LE https://github.com/bluekitchen/CBL2CAPChannel-Demo

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_cbm_connection_t * context){
  context->test_data_start = btstack_run_loop_get_time_ms();
  context->test_data_sent = 0;
}

static void test_track_data(le_cbm_connection_t * context, int
bytes_transferred){
  context->test_data_sent += bytes_transferred;
  // 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: %"PRIu32" 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;
}
```

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){

  if (le_cbm_connection.cid == 0) return;

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

  // send
  l2cap_send(le_cbm_connection.cid, (uint8_t *)
le_cbm_connection.test_data, le_cbm_connection.test_data_len);

  // track
  test_track_data(&le_cbm_connection,
le_cbm_connection.test_data_len);

  // request another packet
```

```
  l2cap_request_can_send_now_event(le_cbm_connection.cid);
}
```

HCI + L2CAP 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.

SM Packet Handler

Main Application Setup

LE Peripheral - Delayed Response

Source Code: att_delayed_response.c

The packet handler is used to handle pairing requests

Listing here shows main application code. It initializes L2CAP, the Security Manager, and configures the ATT Server with the pre-compiled ATT Database generated from $le_credit_based_flow_control_mode_server.gatt$. Finally, it configures the advertisements and boots the Bluetooth stack.

If the value for a GATT Chararacteristic isn't availabl for read, the value ATT_READ_RESPONSE_PENDING can be returned. When the value is available, att_server_response_ready is then called to complete the ATT request.

Similarly, the error code ATT_ERROR_WRITE_RESPONSE_PENING can be returned when it is unclear if a write can be performed or not. When the decision was made, att_server_response_ready is is then called to complete the ATT request.

Main Application Setup

Listing here shows main application code. It initializes L2CAP, the Security Manager and configures the ATT Server with the pre-compiled ATT Database generated from *att_delayed_response.gatt*. Additionally, it enables the Battery Service Server with the current battery level. Finally, it configures the advertisements 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.

```
#ifdef ENABLE_ATT_DELAYED_RESPONSE
static btstack_timer_source_t att_timer;
static hci_con_handle_t con_handle;
static int value_ready;
#endif

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 connection_handle,
uint16_t att_handle, uint16_t transaction_mode, uint16_t offset,
uint8_t *buffer, uint16_t buffer_size);

const uint8_t adv_data[] = {
  // Flags general discoverable, BR/EDR not supported
  0x02, 0x01, 0x06,
  // Name
  0x08, 0x09, 'D', 'e', 'l', 'a', 'y', 'e', 'd',
};
const uint8_t adv_data_len = sizeof(adv_data);

const char * test_string = "Delayed response";

static void example_setup(void){

  l2cap_init();

  // setup SM: Display only
  sm_init();
```

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

  // 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);
}
```

att_invalidate_value Handler

The att_invalidate_value handler ‘invalidates’ the value of the single Characteristic provided in this example

att_update_value Handler

The att_update_value handler ‘updates’ the value of the single Characteristic provided in this example

```
#ifdef ENABLE_ATT_DELAYED_RESPONSE
static void att_update_value(struct btstack_timer_source *ts){
  UNUSED(ts);
  value_ready = 1;

  // trigger ATT Server to try request again
  int status = att_server_response_ready(con_handle);

  printf("Value updated -> complete ATT request - status
0x%02x\n", status);

  // simulated value becoming stale again
  att_timer.process = &att_invalidate_value;
```

```
  btstack_run_loop_set_timer(&att_timer, ATT_VALUE_DELAY_MS);
  btstack_run_loop_add_timer(&att_timer);
}
#endif
```

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 here.

ATT Write

```
// 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){

#ifdef ENABLE_ATT_DELAYED_RESPONSE
  switch (att_handle){
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
      if (value_ready){
        return att_read_callback_handle_blob((const uint8_t
*)test_string, strlen(test_string), offset, buffer, buffer_size);
      } else {
        printf("Read callback for handle 0x%02x, but value not
ready -> report response pending\n", att_handle);
        con_handle = connection_handle;
        return ATT_READ_RESPONSE_PENDING;
      }
      break;
    case ATT_READ_RESPONSE_PENDING:
```

```
      // virtual handle indicating all attributes have been
queried
      printf("Read callback for virtual handle 0x%02x - all
attributes have been queried (important for read multiple or read
by type) -> start updating values\n", att_handle);
      // simulated delayed response for example
      att_timer.process = &att_update_value;
      btstack_run_loop_set_timer(&att_timer, ATT_VALUE_DELAY_MS);
      btstack_run_loop_add_timer(&att_timer);
      return 0;
    default:
      break;
  }
#else
  UNUSED(connection_handle);
  // useless code when ENABLE_ATT_DELAYED_RESPONSE is not defined
- but avoids built errors
  if (att_handle ==
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
{
    printf("ENABLE_ATT_DELAYED_RESPONSE not defined in
btstack_config.h, responding right away");
    return att_read_callback_handle_blob((const uint8_t
*)test_string, (uint16_t) strlen(test_string), offset, buffer,
buffer_size);
  }
#endif

  return 0;
}

/*

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);
  UNUSED(connection_handle);
```

```
  if (att_handle ==
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
{
    printf("Write request, value: ");
    printf_hexdump(buffer, buffer_size);
#ifdef ENABLE_ATT_DELAYED_RESPONSE
    if (value_ready){
      printf("Write callback, value ready\n");
      return 0;
    } else {
      printf("Write callback for handle 0x%02x, but not ready ->
return response pending\n", att_handle);
    }
    // simulated delayed response for example
    att_timer.process = &att_update_value;
    btstack_run_loop_set_timer(&att_timer, ATT_VALUE_DELAY_MS);
    btstack_run_loop_add_timer(&att_timer);
    return ATT_ERROR_WRITE_RESPONSE_PENDING;
#else
    printf("ENABLE_ATT_DELAYED_RESPONSE not defined in
btstack_config.h, responding right away");
    return 0;
#endif
  }
  return 0;
}
```

LE ANCS Client - Apple Notification Service

Source Code: ancs_client_demo.c

LE Man-in-the-Middle Tool

Source Code: le_mitm.c

The example first does an LE scan and allows the user to select a Peripheral device. Then, it connects to the Peripheral and starts advertising with the same data as the Peripheral device. ATT Requests and responses are forwarded between the peripheral and the central Security requests are handled locally.

@note A Bluetooth Controller that supports Central and Peripheral Role at the same time is required for this example. See chipset/README.md

Performance - Stream Data over GATT (Client)

Source Code: le_streamer_client.c

Connects to 'LE Streamer' and subscribes to test characteristic

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.

```
#define TEST_MODE_WRITE_WITHOUT_RESPONSE 1
#define TEST_MODE_DUPLEX         3

// configure test mode: send only, receive only, full duplex
#define TEST_MODE TEST_MODE_DUPLEX

#define REPORT_INTERVAL_MS 1000

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

static void test_track_data(le_streamer_client_connection_t *
```

```
context, uint32_t bytes_sent, uint32_t bytes_received){
  context->test_data_sent   += bytes_sent;
  context->test_data_received += bytes_received;

  // report
  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 throughput
  int bytes_per_second_sent   = context->test_data_sent   * 1000 /
time_passed;
  int bytes_per_second_received = context->test_data_received *
1000 / time_passed;
  int bytes_per_second_total = bytes_per_second_sent +
bytes_per_second_received;
  printf("%c: Sent %u.%03u kB/s, Received %u.%03u kB/s => Total
%u.%03u kB/s\n", context->name,
    bytes_per_second_sent   / 1000, bytes_per_second_sent   %
1000,
    bytes_per_second_received / 1000, bytes_per_second_received %
1000,
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);

  // restart
  test_reset(context);
}
```

Performance - Stream Data over GATT (Server)

Source Code: gatt_streamer_server.c

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.

  In theory, we should also update the connection parameters, but we already get a connection interval of 30 ms and there's no public way to use a shorter interval with iOS (if we're not implementing an HID device).

  Note: To start the streaming, run the example. On remote device use some GATT Explorer, e.g. LightBlue, BLExplr to enable notifications.

Main Application Setup

Listing here shows main application 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){

  l2cap_init();

  // setup SM: Display only
  sm_init();

#ifdef ENABLE_GATT_OVER_CLASSIC
  // init SDP, create record for GATT and register with SDP
  sdp_init();
  memset(gatt_service_buffer, 0, sizeof(gatt_service_buffer));
  gatt_create_sdp_record(gatt_service_buffer, 0x10001,
ATT_SERVICE_GATT_SERVICE_START_HANDLE,
ATT_SERVICE_GATT_SERVICE_END_HANDLE);
  sdp_register_service(gatt_service_buffer);
  printf("SDP service record size: %u\n",
de_get_len(gatt_service_buffer));

  // configure Classic GAP
  gap_set_local_name("GATT Streamer BR/EDR 00:00:00:00:00:00");
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_discoverable_control(1);
```

```
#endif

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

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

  // register for ATT events
  att_server_register_packet_handler(att_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();
}
```

Track throughput

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

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

static void test_track_evaluate(le_streamer_connection_t *
```

```
context) {
  // 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_sent = context->test_data_sent * 1000 /
time_passed;
  int bytes_per_second_received = context->test_data_received *
1000 / time_passed;
  int bytes_per_second_total = bytes_per_second_sent +
bytes_per_second_received;
  printf("%c: Sent %u.%03u kB/s, Received %u.%03u kB/s => Total
%u.%03u kB/s\n", context->name,
    bytes_per_second_sent   / 1000, bytes_per_second_sent   %
1000,
    bytes_per_second_received / 1000, bytes_per_second_received %
1000,
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);
  // store into test data buffer for use with GATT Browser
  btstack_snprintf_best_effort(context->test_data, sizeof(context-
>test_data),
    "Notification Throughput %u.%03u kB/s",
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);

  // restart
  test_reset(context);
}

static void test_track_data(le_streamer_connection_t * context,
uint32_t bytes_sent, uint32_t bytes_received){
  context->test_data_sent   += bytes_sent;
  context->test_data_received += bytes_received;
  test_track_evaluate(context);
}
```

HCI Packet Handler

The packet handler is used track incoming connections and to stop notifications on disconnect It is also a good place to request the connection parameter update as indicated in the commented code block.

```
static void hci_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;

  uint16_t conn_interval;
  hci_con_handle_t con_handle;
  static const char * const phy_names[] = {
    "Reserved", "1 M", "2 M", "Codec"
  };

  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("To start the streaming, please run the
le_streamer_client example on other device, or use some GATT
Explorer, e.g. LightBlue, BLExplr.\n");
      }
      break;
    case HCI_EVENT_DISCONNECTION_COMPLETE:
      con_handle =
hci_event_disconnection_complete_get_connection_handle(packet);
      printf("- LE Connection 0x%04x: disconnect, reason %02x\n",
con_handle,
hci_event_disconnection_complete_get_reason(packet));
      break;
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
```

```
          // print connection parameters (without using float
operations)
          con_handle  =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          conn_interval =
gap_subevent_le_connection_complete_get_conn_interval(packet);
          printf("- LE Connection 0x%04x: connected - connection
interval %u.%02u ms, latency %u\n", con_handle, conn_interval *
125 / 100,
            25 * (conn_interval & 3),
gap_subevent_le_connection_complete_get_conn_latency(packet));

          // request min con interval 15 ms for iOS 11+
          printf("- LE Connection 0x%04x: request 15 ms connection
interval\n", con_handle);
          gap_request_connection_parameter_update(con_handle, 12,
12, 4, 0x0048);
          break;
        default:
          break;
      }
      break;
    case HCI_EVENT_LE_META:
      switch (hci_event_le_meta_get_subevent_code(packet)) {
        case HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
hci_subevent_le_connection_update_complete_get_connection_handle(pa
          conn_interval =
hci_subevent_le_connection_update_complete_get_conn_interval(packet
          printf("- LE Connection 0x%04x: connection update -
connection interval %u.%02u ms, latency %u\n", con_handle,
conn_interval * 125 / 100,
            25 * (conn_interval & 3),
hci_subevent_le_connection_update_complete_get_conn_latency(packet)
          break;
        case HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE:
          con_handle =
hci_subevent_le_data_length_change_get_connection_handle(packet);
```

```
          printf("- LE Connection 0x%04x: data length change - max
%u bytes per packet\n", con_handle,

hci_subevent_le_data_length_change_get_max_tx_octets(packet));
          break;
        case HCI_SUBEVENT_LE_PHY_UPDATE_COMPLETE:
          con_handle =
hci_subevent_le_phy_update_complete_get_connection_handle(packet);
          printf("- LE Connection 0x%04x: PHY update - using LE %s
PHY now\n", con_handle,

phy_names[hci_subevent_le_phy_update_complete_get_tx_phy(packet)]);
          break;
        default:
          break;
      }
      break;

    default:
      break;
  }
}
```

ATT Packet Handler

The packet handler is used to track the ATT MTU Exchange and trigger ATT send

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

  int mtu;
  le_streamer_connection_t * context;
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (hci_event_packet_get_type(packet)) {
        case ATT_EVENT_CONNECTED:
          // setup new
```

```
            context =
connection_for_conn_handle(HCI_CON_HANDLE_INVALID);
            if (!context) break;
            context->connection_handle =
att_event_connected_get_handle(packet);
            context->test_data_len =
btstack_min(att_server_get_mtu(context->connection_handle) - 3,
sizeof(context->test_data));
            // store into test data buffer for use with GATT Browser
            btstack_snprintf_best_effort(context->test_data,
context  ->test_data_len, "GATT Streamer starting up...");
            printf("%c: ATT connected, handle 0x%04x, test data len
%u\n", context->name, context->connection_handle, context-
>test_data_len);
            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_hanc
            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;
          case ATT_EVENT_DISCONNECTED:
            context =
connection_for_conn_handle(att_event_disconnected_get_handle(packet
            if (!context) break;
            // free connection
            printf("%c: ATT disconnected, handle 0x%04x\n", context-
>name, context->connection_handle);
            context->le_notification_enabled = 0;
            context->connection_handle = HCI_CON_HANDLE_INVALID;
            break;
          default:
```

```
          break;
      }
      break;
    default:
      break;
  }
}
```

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];

  // send
  att_server_notify(context->connection_handle, context-
>value_handle, (uint8_t*) context->test_data, context-
```

```
>test_data_len);

  // track
  test_track_data(context, context->test_data_len, 0);

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

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

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 here.

```
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);

  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_C
      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){
        switch (att_handle){
```

```
          case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_C
            context->value_handle =
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
            break;
          default:
            break;
        }
        att_server_request_can_send_now_event(context-
>connection_handle);
      }
      test_reset(context);
      break;
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
      test_track_data(context, 0, buffer_size);
      break;
    default:
      printf("Write to 0x%04x, len %u\n", att_handle,
buffer_size);
      break;
  }
  return 0;
}
```

LE Credit-Based Flow-Control Mode Client - Send Data over L2CAP

Source Code: le_credit_based_flow_control_mode_client.c

Connects to ‘LE CBM Server’ and streams data via L2CAP Channel in LE Credit-Based Flow-Control Mode (CBM)

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.

```
#define REPORT_INTERVAL_MS 3000

// support for multiple clients
typedef struct {
  char name;
  hci_con_handle_t connection_handle;
  uint16_t cid;
  int  counter;
  char test_data[TEST_PACKET_SIZE];
  int  test_data_len;
  uint32_t test_data_sent;
  uint32_t test_data_start;
} le_cbm_connection_t;

static le_cbm_connection_t le_cbm_connection;

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

static void test_track_data(le_cbm_connection_t * context, int
bytes_transferred){
  context->test_data_sent += bytes_transferred;
  // 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: %"PRIu32" bytes -> %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;
}
```

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){

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

  // send
  l2cap_send(le_cbm_connection.cid, (uint8_t *)
le_cbm_connection.test_data, le_cbm_connection.test_data_len);

  // track
  test_track_data(&le_cbm_connection,
le_cbm_connection.test_data_len);

  // request another packet
  l2cap_request_can_send_now_event(le_cbm_connection.cid);
}
```

SM Packet Handler

The packet handler is used to handle pairing requests

LE Credit-Based Flow-Control Mode Server - Receive data over L2CAP

Source Code: le_credit_based_flow_control_mode_server.c

iOS 11 and newer supports L2CAP channels in LE Credit-Based Flow-Control Mode for fast transfer over LE https://github.com/bluekitchen/CBL2CAPChannel-Demo

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_cbm_connection_t * context){
  context->test_data_start = btstack_run_loop_get_time_ms();
  context->test_data_sent = 0;
}

static void test_track_data(le_cbm_connection_t * context, int
bytes_transferred){
  context->test_data_sent += bytes_transferred;
  // 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: %"PRIu32" 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;
}
```

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){

  if (le_cbm_connection.cid == 0) return;

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

  // send
  l2cap_send(le_cbm_connection.cid, (uint8_t *)
le_cbm_connection.test_data, le_cbm_connection.test_data_len);

  // track
  test_track_data(&le_cbm_connection,
le_cbm_connection.test_data_len);

  // request another packet
  l2cap_request_can_send_now_event(le_cbm_connection.cid);
}
```

HCI + L2CAP 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.

SM Packet Handler

Main Application Setup

Performance - Stream Data over SPP (Client)

Source Code: spp_streamer_client.c

The packet handler is used to handle pairing requests

Listing here shows main application code. It initializes L2CAP, the Security Manager, and configures the ATT Server with the pre-compiled ATT Database generated from $le_{c}redit_{b}ased_{f}low_{c}ontrol_{m}ode_{s}erver.gatt$. Finally, it configures the advertisements and boots the Bluetooth stack.

Note: The SPP Streamer Client scans for and connects to SPP Streamer, and measures the throughput.

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.

```
#define REPORT_INTERVAL_MS 3000
static uint32_t test_data_transferred;
static uint32_t test_data_start;

static void test_reset(void){
  test_data_start = btstack_run_loop_get_time_ms();
  test_data_transferred = 0;
}

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

  // restart
  test_data_start = now;
```

```
  test_data_transferred  = 0;
}
```

SDP Query Packet Handler

Store RFCOMM Channel for SPP service and initiates RFCOMM connection

Gerenal Packet Handler

Handles startup (BTSTACK_EVENT_STATE), inquiry, pairing, starts SDP query for SPP service, and RFCOMM connection

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

  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  rfcomm_init();

#ifdef ENABLE_L2CAP_ENHANCED_RETRANSMISSION_MODE_FOR_RFCOMM
  // setup ERTM management
  rfcomm_enable_l2cap_ertm(&rfcomm_ertm_request_handler,
&rfcomm_ertm_released_handler);
#endif

  // register for HCI events
```

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

  // init SDP
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);

  // turn on!
    hci_power_control(HCI_POWER_ON);

  return 0;
}
```

Performance - Stream Data over SPP (Server)

Source Code: spp_streamer.c

After RFCOMM connections gets open, request a RFCOMM_EVENT_CAN_SEND_NOW via rfcomm_request_can_send_now_event().

When we get the RFCOMM_EVENT_CAN_SEND_NOW, send data and request another one.

Note: To test, run the example, pair from a remote device, and open the Virtual Serial Port.

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.

```
#define REPORT_INTERVAL_MS 3000
static uint32_t test_data_transferred;
static uint32_t test_data_start;

static void test_reset(void){
  test_data_start = btstack_run_loop_get_time_ms();
```

```
  test_data_transferred = 0;
}

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

  // restart
  test_data_start = now;
  test_data_transferred  = 0;
}
```

Packet Handler

The packet handler of the combined example is just the combination of the individual packet handlers.

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

  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
```

```
key derivation
  sm_init();
#endif

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

#ifdef ENABLE_L2CAP_ENHANCED_RETRANSMISSION_MODE_FOR_RFCOMM
  // setup ERTM management
  rfcomm_enable_l2cap_ertm(&rfcomm_ertm_request_handler,
&rfcomm_ertm_released_handler);
#endif

  // 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,
sdp_create_service_record_handle(), RFCOMM_SERVER_CHANNEL, "SPP
Streamer");
  btstack_assert(de_get_len( spp_service_buffer) <=
sizeof(spp_service_buffer));
  sdp_register_service(spp_service_buffer);

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

  // short-cut to find other SPP Streamer
  gap_set_class_of_device(TEST_COD);

  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_set_local_name("SPP Streamer 00:00:00:00:00:00");
  gap_discoverable_control(1);

  spp_create_test_data();

  // turn on!
    hci_power_control(HCI_POWER_ON);
```

```
  return 0;
}
```

A2DP Sink - Receive Audio Stream and Control Playback

Source Code: a2dp_sink_demo.c

This A2DP Sink example demonstrates how to use the A2DP Sink service to receive an audio data stream from a remote A2DP Source device. In addition, the AVRCP Controller is used to get information on currently played media, such are title, artist and album, as well as to control the playback, i.e. to play, stop, repeat, etc. If HAVE_BTSTACK_STDIN is set, press SPACE on the console to show the available AVDTP and AVRCP commands.

To test with a remote device, e.g. a mobile phone, pair from the remote device with the demo, then start playing music on the remote device. Alternatively, set the device_addr_string to the Bluetooth address of your remote device in the code, and call connect from the UI.

For more info on BTstack audio, see our blog post A2DP Sink and Source on STM32 F4 Discovery Board.

Main Application Setup

The Listing here shows how to set up AD2P Sink and AVRCP services. Besides calling init() method for each service, you'll also need to register several packet handlers:

- hci_packet_handler - handles legacy pairing, here by using fixed '0000' pin code.
- a2dp_sink_packet_handler - handles events on stream connection status (established, released), the media codec configuration, and, the status of the stream itself (opened, paused, stopped).

- handle_l2cap_media_data_packet - used to receive streaming data. If STORE_TO_WAV_FILE directive (check btstack_config.h) is used, the SBC decoder will be used to decode the SBC data into PCM frames. The resulting PCM frames are then processed in the SBC Decoder callback.
- avrcp_packet_handler - receives AVRCP connect/disconnect event.
- avrcp_controller_packet_handler - receives answers for sent AVRCP commands.
- avrcp_target_packet_handler - receives AVRCP commands, and registered notifications.
- stdin_process - used to trigger AVRCP commands to the A2DP Source device, such are get now playing info, start, stop, volume control. Requires HAVE_BTSTACK_STDIN.

  To announce A2DP Sink and AVRCP services, you need to create corresponding SDP records and register them with the SDP service.

  Note, currently only the SBC codec is supported. If you want to store the audio data in a file, you'll need to define STORE_TO_WAV_FILE. If STORE_TO_WAV_FILE directive is defined, the SBC decoder needs to get initialized when a2dp_sink_packet_handler receives event A2DP_SUBEVENT_STREAM_STARTED. The initialization of the SBC decoder requires a callback that handles PCM data:

- handle_pcm_data - handles PCM audio frames. Here, they are stored in a wav file if STORE_TO_WAV_FILE is defined, and/or played using the audio library.

```
static void hci_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);
static void a2dp_sink_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t * packet, uint16_t event_size);
static void handle_l2cap_media_data_packet(uint8_t seid, uint8_t
*packet, uint16_t size);
static void avrcp_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);
static void avrcp_controller_packet_handler(uint8_t packet_type,
```

```
uint16_t channel, uint8_t *packet, uint16_t size);
static void avrcp_target_packet_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size);
#ifdef HAVE_BTSTACK_STDIN
static void stdin_process(char cmd);
#endif

static int setup_demo(void){

  // init protocols
  l2cap_init();
  sdp_init();
#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif
#ifdef ENABLE_AVRCP_COVER_ART
  goep_client_init();
  avrcp_cover_art_client_init();
#endif

  // Init profiles
  a2dp_sink_init();
  avrcp_init();
  avrcp_controller_init();
  avrcp_target_init();



  // Configure A2DP Sink
  a2dp_sink_register_packet_handler(&a2dp_sink_packet_handler);

a2dp_sink_register_media_handler(&handle_l2cap_media_data_packet);
  a2dp_sink_demo_stream_endpoint_t * stream_endpoint =
&a2dp_sink_demo_stream_endpoint;
  avdtp_stream_endpoint_t * local_stream_endpoint =
a2dp_sink_create_stream_endpoint(AVDTP_AUDIO,
                                                AVDTP_CODEC_SBC,
media_sbc_codec_capabilities,
sizeof(media_sbc_codec_capabilities),
```

```
                                                stream_endpoint-
>media_sbc_codec_configuration, sizeof(stream_endpoint-
>media_sbc_codec_configuration));
  btstack_assert(local_stream_endpoint != NULL);
  // - Store stream enpoint's SEP ID, as it is used by A2DP API to
identify the stream endpoint
  stream_endpoint->a2dp_local_seid =
avdtp_local_seid(local_stream_endpoint);


  // Configure AVRCP Controller + Target
  avrcp_register_packet_handler(&avrcp_packet_handler);

avrcp_controller_register_packet_handler(&avrcp_controller_packet_h

avrcp_target_register_packet_handler(&avrcp_target_packet_handler);


  // Configure SDP

  // - Create and register A2DP Sink service record
  memset(sdp_avdtp_sink_service_buffer, 0,
sizeof(sdp_avdtp_sink_service_buffer));
  a2dp_sink_create_sdp_record(sdp_avdtp_sink_service_buffer,
sdp_create_service_record_handle(),
                AVDTP_SINK_FEATURE_MASK_HEADPHONE, NULL, NULL);
  btstack_assert(de_get_len( sdp_avdtp_sink_service_buffer) <=
sizeof(sdp_avdtp_sink_service_buffer));
  sdp_register_service(sdp_avdtp_sink_service_buffer);

  // - Create AVRCP Controller service record and register it with
SDP. We send Category 1 commands to the media player, e.g. play/
pause
  memset(sdp_avrcp_controller_service_buffer, 0,
sizeof(sdp_avrcp_controller_service_buffer));
  uint16_t controller_supported_features = 1 <<
AVRCP_CONTROLLER_SUPPORTED_FEATURE_CATEGORY_PLAYER_OR_RECORDER;
#ifdef AVRCP_BROWSING_ENABLED
  controller_supported_features |= 1 <<
AVRCP_CONTROLLER_SUPPORTED_FEATURE_BROWSING;
```

```
#endif
#ifdef ENABLE_AVRCP_COVER_ART
  controller_supported_features |= 1 <<
AVRCP_CONTROLLER_SUPPORTED_FEATURE_COVER_ART_GET_LINKED_THUMBNAIL;
#endif

avrcp_controller_create_sdp_record(sdp_avrcp_controller_service_buf
sdp_create_service_record_handle(),
                     controller_supported_features, NULL, NULL);
  btstack_assert(de_get_len( sdp_avrcp_controller_service_buffer)
<= sizeof(sdp_avrcp_controller_service_buffer));
  sdp_register_service(sdp_avrcp_controller_service_buffer);

  // - Create and register A2DP Sink service record
  //   -  We receive Category 2 commands from the media player,
e.g. volume up/down
  memset(sdp_avrcp_target_service_buffer, 0,
sizeof(sdp_avrcp_target_service_buffer));
  uint16_t target_supported_features = 1 <<
AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_MONITOR_OR_AMPLIFIER;
  avrcp_target_create_sdp_record(sdp_avrcp_target_service_buffer,
                   sdp_create_service_record_handle(),
target_supported_features, NULL, NULL);
  btstack_assert(de_get_len( sdp_avrcp_target_service_buffer) <=
sizeof(sdp_avrcp_target_service_buffer));
  sdp_register_service(sdp_avrcp_target_service_buffer);

  // - Create and register Device ID (PnP) service record
  memset(device_id_sdp_service_buffer, 0,
sizeof(device_id_sdp_service_buffer));
  device_id_create_sdp_record(device_id_sdp_service_buffer,
                sdp_create_service_record_handle(),
DEVICE_ID_VENDOR_ID_SOURCE_BLUETOOTH,
BLUETOOTH_COMPANY_ID_BLUEKITCHEN_GMBH, 1, 1);
  btstack_assert(de_get_len( device_id_sdp_service_buffer) <=
sizeof(device_id_sdp_service_buffer));
  sdp_register_service(device_id_sdp_service_buffer);


  // Configure GAP - discovery / connection
```

```
  // - Set local name with a template Bluetooth address, that will
be automatically
  //   replaced with an actual address once it is available, i.e.
when BTstack boots
  //   up and starts talking to a Bluetooth module.
  gap_set_local_name("A2DP Sink Demo 00:00:00:00:00:00");

  // - Allow to show up in Bluetooth inquiry
  gap_discoverable_control(1);

  // - Set Class of Device - Service Class: Audio, Major Device
Class: Audio, Minor: Headphone
  gap_set_class_of_device(0x200404);

  // - Allow for role switch in general and sniff mode

gap_set_default_link_policy_settings( LM_LINK_POLICY_ENABLE_ROLE_SW
| LM_LINK_POLICY_ENABLE_SNIFF_MODE );

  // - Allow for role switch on outgoing connections
  //   - This allows A2DP Source, e.g. smartphone, to become
master when we re-connect to it.
  gap_set_allow_role_switch(true);



  // Register for HCI events
  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

  // Inform about audio playback / test options
#ifdef HAVE_POSIX_FILE_IO
  if (!btstack_audio_sink_get_instance()){
    printf("No audio playback.\n");
  } else {
    printf("Audio playback supported.\n");
  }
#ifdef STORE_TO_WAV_FILE
   printf("Audio will be stored to \'%s\' file.\n",
wav_filename);
```

```
#endif
#endif
  return 0;
}
```

Handle Media Data Packet

Here the audio data, are received through the handle_l2cap_media_data_packet callback. Currently, only the SBC media codec is supported. Hence, the media data consists of the media packet header and the SBC packet. The SBC frame will be stored in a ring buffer for later processing (instead of decoding it to PCM right away which would require a much larger buffer). If the audio stream wasn't started already and there are enough SBC frames in the ring buffer, start playback.

A2DP Source - Stream Audio and Control Volume

Source Code: a2dp_source_demo.c

This A2DP Source example demonstrates how to send an audio data stream to a remote A2DP Sink device and how to switch between two audio data sources. In addition, the AVRCP Target is used to answer queries on currently played media, as well as to handle remote playback control, i.e. play, stop, repeat, etc. If HAVE_BTSTACK_STDIN is set, press SPACE on the console to show the available AVDTP and AVRCP commands.

To test with a remote device, e.g. a Bluetooth speaker, set the device_addr_string to the Bluetooth address of your remote device in the code, and use the UI to connect and start playback.

For more info on BTstack audio, see our blog post A2DP Sink and Source on STM32 F4 Discovery Board.

Main Application Setup

The Listing here shows how to setup AD2P Source and AVRCP services. Besides calling init() method for each service, you'll also need to register several packet handlers:

- hci_packet_handler - handles legacy pairing, here by using fixed '0000' pin code.
- a2dp_source_packet_handler - handles events on stream connection status (established, released), the media codec configuration, and, the commands on stream itself (open, pause, stopp).
- avrcp_packet_handler - receives connect/disconnect event.
- avrcp_controller_packet_handler - receives answers for sent AVRCP commands.
- avrcp_target_packet_handler - receives AVRCP commands, and registered notifications.
- stdin_process - used to trigger AVRCP commands to the A2DP Source device, such are get now playing info, start, stop, volume control. Requires HAVE_BTSTACK_STDIN.

  To announce A2DP Source and AVRCP services, you need to create corresponding SDP records and register them with the SDP service.

```
static void hci_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);
static void a2dp_source_packet_handler(uint8_t packet_type,
uint16_t channel, uint8_t * event, uint16_t event_size);
static void avrcp_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);
static void avrcp_target_packet_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size);
static void avrcp_controller_packet_handler(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size);
#ifdef HAVE_BTSTACK_STDIN
static void stdin_process(char cmd);
#endif
```

```
static int a2dp_source_and_avrcp_services_init(void){

  // Request role change on reconnecting headset to always use
them in slave mode
  hci_set_master_slave_policy(0);
  // enabled EIR
  hci_set_inquiry_mode(INQUIRY_MODE_RSSI_AND_EIR);

  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  // Initialize  A2DP Source
  a2dp_source_init();

a2dp_source_register_packet_handler(&a2dp_source_packet_handler);

  // Create stream endpoint
  avdtp_stream_endpoint_t * local_stream_endpoint =
a2dp_source_create_stream_endpoint(AVDTP_AUDIO, AVDTP_CODEC_SBC,
media_sbc_codec_capabilities,
sizeof(media_sbc_codec_capabilities),
media_sbc_codec_configuration,
sizeof(media_sbc_codec_configuration));
  if (!local_stream_endpoint){
    printf("A2DP Source: not enough memory to create local stream
endpoint\n");
    return 1;
  }

  avdtp_set_preferred_sampling_frequency(local_stream_endpoint,
A2DP_SOURCE_DEMO_PREFERRED_SAMPLING_RATE);

  // Store stream enpoint's SEP ID, as it is used by A2DP API to
indentify the stream endpoint
```

```
  media_tracker.local_seid =
avdtp_local_seid(local_stream_endpoint);

avdtp_source_register_delay_reporting_category(media_tracker.local_

  // Initialize AVRCP Service
  avrcp_init();
  avrcp_register_packet_handler(&avrcp_packet_handler);
  // Initialize AVRCP Target
  avrcp_target_init();

avrcp_target_register_packet_handler(&avrcp_target_packet_handler);

  // Initialize AVRCP Controller
  avrcp_controller_init();

avrcp_controller_register_packet_handler(&avrcp_controller_packet_h

  // Initialize SDP,
  sdp_init();

  // Create A2DP Source service record and register it with SDP
  memset(sdp_a2dp_source_service_buffer, 0,
sizeof(sdp_a2dp_source_service_buffer));
  a2dp_source_create_sdp_record(sdp_a2dp_source_service_buffer,
sdp_create_service_record_handle(),
AVDTP_SOURCE_FEATURE_MASK_PLAYER, NULL, NULL);
  btstack_assert(de_get_len( sdp_a2dp_source_service_buffer) <=
sizeof(sdp_a2dp_source_service_buffer));
  sdp_register_service(sdp_a2dp_source_service_buffer);

  // Create AVRCP Target service record and register it with SDP.
We receive Category 1 commands from the headphone, e.g. play/pause
  memset(sdp_avrcp_target_service_buffer, 0,
sizeof(sdp_avrcp_target_service_buffer));
  uint16_t supported_features =
AVRCP_FEATURE_MASK_CATEGORY_PLAYER_OR_RECORDER;
#ifdef AVRCP_BROWSING_ENABLED
  supported_features |= AVRCP_FEATURE_MASK_BROWSING;
#endif
```

```
  avrcp_target_create_sdp_record(sdp_avrcp_target_service_buffer,
sdp_create_service_record_handle(), supported_features, NULL,
NULL);
  btstack_assert(de_get_len( sdp_avrcp_target_service_buffer) <=
sizeof(sdp_avrcp_target_service_buffer));
  sdp_register_service(sdp_avrcp_target_service_buffer);

  // Create AVRCP Controller service record and register it with
SDP. We send Category 2 commands to the headphone, e.g. volume up/
down
  memset(sdp_avrcp_controller_service_buffer, 0,
sizeof(sdp_avrcp_controller_service_buffer));
  uint16_t controller_supported_features =
AVRCP_FEATURE_MASK_CATEGORY_MONITOR_OR_AMPLIFIER;

avrcp_controller_create_sdp_record(sdp_avrcp_controller_service_buf
sdp_create_service_record_handle(), controller_supported_features,
NULL, NULL);
  btstack_assert(de_get_len( sdp_avrcp_controller_service_buffer)
<= sizeof(sdp_avrcp_controller_service_buffer));
  sdp_register_service(sdp_avrcp_controller_service_buffer);

  // Register Device ID (PnP) service SDP record
  memset(device_id_sdp_service_buffer, 0,
sizeof(device_id_sdp_service_buffer));
  device_id_create_sdp_record(device_id_sdp_service_buffer,
sdp_create_service_record_handle(),
DEVICE_ID_VENDOR_ID_SOURCE_BLUETOOTH,
BLUETOOTH_COMPANY_ID_BLUEKITCHEN_GMBH, 1, 1);
  btstack_assert(de_get_len( device_id_sdp_service_buffer) <=
sizeof(device_id_sdp_service_buffer));
  sdp_register_service(device_id_sdp_service_buffer);

  // Set local name with a template Bluetooth address, that will
be automatically
  // replaced with a actual address once it is available, i.e.
when BTstack boots
  // up and starts talking to a Bluetooth module.
  gap_set_local_name("A2DP Source 00:00:00:00:00:00");
  gap_discoverable_control(1);
```

```
  gap_set_class_of_device(0x200408);

  // Register for HCI events.
  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);

#ifdef ENABLE_MODPLAYER
  data_source = STREAM_MOD;
#else
  data_source = STREAM_SINE;
#endif

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

#ifdef HAVE_BTSTACK_STDIN
  btstack_stdin_setup(stdin_process);
#endif
  return 0;
}
```

AVRCP Browsing - Browse Media Players and Media Information

Source Code: avrcp_browsing_client.c

This example demonstrates how to use the AVRCP Controller Browsing service to browse madia players and media information on a remote AVRCP Source device.

To test with a remote device, e.g. a mobile phone, pair from the remote device with the demo, then use the UI for browsing. If HAVE_BTSTACK_STDIN is set, press SPACE on the console to show the available AVDTP and AVRCP commands.

Main Application Setup

The Listing here shows how to setup AVRCP Controller Browsing service. To announce AVRCP Controller Browsing service, you need to create corresponding SDP record and register it with the SDP service. You'll also need to register several packet handlers:

- stdin_process callback - used to trigger AVRCP commands, such are get media players, playlists, albums, etc. Requires HAVE_BTSTACK_STDIN.
- avrcp_browsing_controller_packet_handler - used to receive answers for AVRCP commands.

```
static void avrcp_browsing_controller_packet_handler(uint8_t
packet_type, uint16_t channel, uint8_t *packet, uint16_t size);
static void avrcp_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);
static void a2dp_sink_packet_handler(uint8_t packet_type, uint16_t
channel, uint8_t *packet, uint16_t size);

#ifdef HAVE_BTSTACK_STDIN
static void stdin_process(char cmd);
#endif


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

  // Initialize L2CAP.
  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  a2dp_sink_init();
```

```
  a2dp_sink_register_packet_handler(&a2dp_sink_packet_handler);

  avdtp_stream_endpoint_t * local_stream_endpoint =
a2dp_sink_create_stream_endpoint(AVDTP_AUDIO,
    AVDTP_CODEC_SBC, media_sbc_codec_capabilities,
sizeof(media_sbc_codec_capabilities),
    media_sbc_codec_configuration,
sizeof(media_sbc_codec_configuration));
  if (!local_stream_endpoint){
    printf("A2DP Sink: not enough memory to create local stream
endpoint\n");
    return 1;
  }
  a2dp_local_seid = avdtp_local_seid(local_stream_endpoint);

  // Initialize AVRCP service.
  avrcp_init();
  // Initialize AVRCP Controller & Target Service.
  avrcp_controller_init();
  avrcp_target_init();

  avrcp_register_packet_handler(&avrcp_packet_handler);
  avrcp_controller_register_packet_handler(&avrcp_packet_handler);
  avrcp_target_register_packet_handler(&avrcp_packet_handler);

  // Initialize AVRCP Browsing Service.
  avrcp_browsing_init();
  avrcp_browsing_controller_init();
  avrcp_browsing_target_init();

  // Register for HCI events.

avrcp_browsing_controller_register_packet_handler(&avrcp_browsing_c

avrcp_browsing_target_register_packet_handler(&avrcp_browsing_contr

avrcp_browsing_register_packet_handler(&avrcp_browsing_controller_p

  // Initialize SDP.
  sdp_init();
```

```
  // setup AVDTP sink
  memset(sdp_avdtp_sink_service_buffer, 0,
sizeof(sdp_avdtp_sink_service_buffer));
  a2dp_sink_create_sdp_record(sdp_avdtp_sink_service_buffer,
sdp_create_service_record_handle(),
AVDTP_SINK_FEATURE_MASK_HEADPHONE, NULL, NULL);
  btstack_assert(de_get_len( sdp_avdtp_sink_service_buffer) <=
sizeof(sdp_avdtp_sink_service_buffer));
  sdp_register_service(sdp_avdtp_sink_service_buffer);

  // Create AVRCP service record and register it with SDP.
  memset(sdp_avrcp_browsing_controller_service_buffer, 0,
sizeof(sdp_avrcp_browsing_controller_service_buffer));

  uint16_t supported_features =
AVRCP_FEATURE_MASK_CATEGORY_PLAYER_OR_RECORDER;
#ifdef AVRCP_BROWSING_ENABLED
  supported_features |= AVRCP_FEATURE_MASK_BROWSING;
#endif

avrcp_controller_create_sdp_record(sdp_avrcp_browsing_controller_se
sdp_create_service_record_handle(), supported_features, NULL,
NULL);

btstack_assert(de_get_len( sdp_avrcp_browsing_controller_service_bu
<= sizeof(sdp_avrcp_browsing_controller_service_buffer));

sdp_register_service(sdp_avrcp_browsing_controller_service_buffer);

  // Set local name with a template Bluetooth address, that will
be automatically
  // replaced with a actual address once it is available, i.e.
when BTstack boots
  // up and starts talking to a Bluetooth module.
  gap_set_local_name("AVRCP Browsing Client 00:00:00:00:00:00");
  gap_discoverable_control(1);
  gap_set_class_of_device(0x200408);

  // Register for HCI events.
  hci_event_callback_registration.callback =
```

```
&avrcp_browsing_controller_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);


#ifdef HAVE_BTSTACK_STDIN
  // Parse human readable Bluetooth address.
  sscanf_bd_addr(device_addr_string, device_addr);
  btstack_stdin_setup(stdin_process);
#endif
  printf("Starting BTstack ...\n");
  hci_power_control(HCI_POWER_ON);
  return 0;
}
```

HFP AG - Audio Gateway

Source Code: hfp_ag_demo.c

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.

Main Application Setup

Listing here shows main application 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();

  // Request role change on reconnecting headset to always use
```

```
them in slave mode
  hci_set_master_slave_policy(0);

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

  // L2CAP
  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  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_ENHANCED_VOICE_RECOGNITION_STATUS) |
    (1<<HFP_AGSF_VOICE_RECOGNITION_TEXT) |
    (1<<HFP_AGSF_EC_NR_FUNCTION) |
    (1<<HFP_AGSF_THREE_WAY_CALLING);

  // 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_init_apple_identification("BTstack", 0);
```

```
  hfp_ag_set_subscriber_number_information(&subscriber_number, 1);

  // SDP Server
  sdp_init();
  memset(hfp_service_buffer, 0, sizeof(hfp_service_buffer));
  hfp_ag_create_sdp_record_with_codecs( hfp_service_buffer,
sdp_create_service_record_handle(),
                      rfcomm_channel_nr, hfp_ag_service_name, 0,
supported_features, sizeof(codecs), codecs);
  btstack_assert(de_get_len( hfp_service_buffer) <=
sizeof(hfp_service_buffer));
  sdp_register_service(hfp_service_buffer);

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

  // register for HFP events
  hfp_ag_register_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
  // turn on!
  hci_power_control(HCI_POWER_ON);
  return 0;
}
```

HFP HF - Hands-Free

Source Code: hfp_hs_demo.c

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.

Main Application Setup

Listing here shows main application 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;

  // Init protocols
  // init L2CAP
  l2cap_init();
  rfcomm_init();
  sdp_init();
#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  // Init profiles
  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_VOICE_RECOGNITION_FUNCTION)  |
    (1<<HFP_HFSF_ENHANCED_VOICE_RECOGNITION_STATUS) |
    (1<<HFP_HFSF_VOICE_RECOGNITION_TEXT) |
    (1<<HFP_HFSF_EC_NR_FUNCTION) |
    (1<<HFP_HFSF_REMOTE_VOLUME_CONTROL);

  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(hfp_hf_packet_handler);
  // Set Apple Accessory Information: features = battery reporting
& docked/powered, battery=9, docked=1

hfp_hf_apple_set_identification(BLUETOOTH_COMPANY_ID_BLUEKITCHEN_GM
0x0001, "0001", 3);
  hfp_hf_apple_set_battery_level(9);
  hfp_hf_apple_set_docked_state(1);

  // Configure SDP

  // - Create and register HFP HF service record
  memset(hfp_service_buffer, 0, sizeof(hfp_service_buffer));
  hfp_hf_create_sdp_record_with_codecs(hfp_service_buffer,
sdp_create_service_record_handle(),
               rfcomm_channel_nr, hfp_hf_service_name,
hf_supported_features, sizeof(codecs), codecs);
  btstack_assert(de_get_len( hfp_service_buffer) <=
sizeof(hfp_service_buffer));
  sdp_register_service(hfp_service_buffer);

  // Configure GAP - discovery / connection

  // - Set local name with a template Bluetooth address, that will
be automatically
  //   replaced with an actual address once it is available, i.e.
when BTstack boots
  //   up and starts talking to a Bluetooth module.
  gap_set_local_name("HFP HF Demo 00:00:00:00:00:00");

  // - Allow to show up in Bluetooth inquiry
  gap_discoverable_control(1);

  // - Set Class of Device - Service Class: Audio, Major Device
Class: Audio, Minor: Hands-Free device
  gap_set_class_of_device(0x200408);
```

```
  // - Allow for role switch in general and sniff mode

gap_set_default_link_policy_settings( LM_LINK_POLICY_ENABLE_ROLE_SW
| LM_LINK_POLICY_ENABLE_SNIFF_MODE );

  // - Allow for role switch on outgoing connections - this allows
HFP AG, e.g. smartphone, to become master when we re-connect to it
  gap_set_allow_role_switch(true);

  // Register for HCI events and SCO packets
  hci_event_callback_registration.callback = &hci_packet_handler;
  hci_add_event_handler(&hci_event_callback_registration);
  hci_register_sco_packet_handler(&hci_packet_handler);



  // Init SCO / HFP audio processing
  sco_demo_init();

#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;
}
```

HSP AG - Audio Gateway

Source Code: hsp_ag_demo.c

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.

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

Main Application Setup

Listing here shows main application 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();
  sco_demo_set_codec(HFP_CODEC_CVSD);

  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif
```

```
  sdp_init();

  memset((uint8_t *)hsp_service_buffer, 0,
sizeof(hsp_service_buffer));
  hsp_ag_create_sdp_record(hsp_service_buffer,
sdp_create_service_record_handle(), rfcomm_channel_nr,
hsp_ag_service_name);
  btstack_assert(de_get_len( hsp_service_buffer) <=
sizeof(hsp_service_buffer));
  sdp_register_service(hsp_service_buffer);

  rfcomm_init();

  hsp_ag_init(rfcomm_channel_nr);
  hsp_ag_register_packet_handler(&packet_handler);

  // register for SCO packets
  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;
}
```

HSP HS - Headset

Source Code: hsp_hs_demo.c

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.

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

Main Application Setup

Listing here shows main application 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();
```

```
  sco_demo_set_codec(HFP_CODEC_CVSD);

  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  sdp_init();
  memset(hsp_service_buffer, 0, sizeof(hsp_service_buffer));
  hsp_hs_create_sdp_record(hsp_service_buffer,
sdp_create_service_record_handle(), rfcomm_channel_nr,
hsp_hs_service_name, 0);
  btstack_assert(de_get_len( hsp_service_buffer) <=
sizeof(hsp_service_buffer));
  sdp_register_service(hsp_service_buffer);

  rfcomm_init();

  hsp_hs_init(rfcomm_channel_nr);

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

  // register for HSP events
  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);
```

```
  // Parse human readable Bluetooth address.
  sscanf_bd_addr(device_addr_string, device_addr);

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

Audio Driver - Play Sine

Source Code: audio_duplex.c

Play sine to test and validate audio output with simple wave form.

Audio Driver - Play 80's MOD Song

Source Code: mod_player.c

Audio Driver - Forward Audio from Source to Sink

Source Code: audio_duplex.c

SPP Server - Heartbeat Counter over RFCOMM

Source Code: spp_counter.c

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.

Note: To test, please run the spp_counter example, and then pair from a remote device, and open the Virtual Serial Port.

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 here. The SDP record created by 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 . 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();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  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,
sdp_create_service_record_handle(), RFCOMM_SERVER_CHANNEL, "SPP
Counter");
  btstack_assert(de_get_len( spp_service_buffer) <=
sizeof(spp_service_buffer));
```

```
  sdp_register_service(spp_service_buffer);
}
```

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 here.

```
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){
    btstack_snprintf_assert_complete(lineBuffer,
sizeof(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);
}
```

Bluetooth Logic

The Bluetooth logic is implemented within the packet handler, see Listing here. In this example, 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:
            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:
            if (rfcomm_event_channel_opened_get_status(packet)) {
              printf("RFCOMM channel open failed, status 0x%02x\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,
(uint16_t) strlen(lineBuffer));
            break;

...
}
```

SPP Server - RFCOMM Flow Control

Source Code: spp_flowcontrol.c

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.

SPP Service Setup

Listing here shows how to 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_SERVER_CHANNEL, 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,
sdp_create_service_record_handle(), 1, "SPP Counter");
  btstack_assert(de_get_len( spp_service_buffer) <=
sizeof(spp_service_buffer));
  sdp_register_service(spp_service_buffer);
}
```

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 here. If the flag is set, a new credit will be granted by the heartbeat handler, introducing a delay of up to 1 second. The heartbeat handler code is shown in Listing here.

```
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);
}


// 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;
...
}
```

PAN - lwIP HTTP and DHCP Server

Source Code: pan_lwip_http_server.c

Bluetooth PAN is mainly used for Internet Tethering, where e.g. a mobile phone provides internet connection to a laptop or a tablet.

Instead of regular internet access, it's also possible to provide a Web app on a Bluetooth device, e.g. for configuration or maintenance. For some device, this can be a more effective way to provide an interface compared to dedicated smartphone applications (for Android and iOS).

Before iOS 11, accessing an HTTP server via Bluetooth PAN was not supported on the iPhone, but on iPod and iPad. With iOS 11, this works as expected.

After pairing your device, please open the URL http://192.168.7.1 in your web browser.

Packet Handler

All BNEP events are handled in the platform/bnep_lwip.c BNEP-LWIP Adapter. Here, we only print status information and handle pairing requests.

PAN BNEP Setup

DHCP Server Configuration

Large File Download

DHCP Server Setup

Main

BNEP/PANU (Linux only)

Source Code: panu_demo.c

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.

BNEP_EVENT_CHANNEL_CLOSED is received when the connection gets closed.

Listing here shows the setup of the PAN setup

Listing here shows the DHCP Server configuration for network 192.168.7.0/8

Listing here Shows how a configurable test file for performance tests is generated on the fly. The filename is the number of bytes to generate, e.g. /1048576.txt results in a 1MB file.

Listing here shows the setup of the lwIP network stack and starts the DHCP Server

Setup the lwIP network and PAN NAP

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.

Note: currently supported only on Linux and provides a TAP network interface which you can configure yourself.

To enable client mode, uncomment ENABLE_PANU_CLIENT below.

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 here.

```
static void packet_handler (uint8_t packet_type, uint16_t channel,
uint8_t *packet, uint16_t size);
#ifdef ENABLE_PANU_CLIENT
static void handle_sdp_client_query_result(uint8_t packet_type,
uint16_t channel, uint8_t *packet, uint16_t size);
#endif
static void network_send_packet_callback(const uint8_t * packet,
uint16_t size);

static void panu_setup(void){

  // Initialize L2CAP
  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  // init SDP, create record for PANU and register with SDP
  sdp_init();
  memset(panu_sdp_record, 0, sizeof(panu_sdp_record));
  uint16_t network_packet_types[] = { NETWORK_TYPE_IPv4,
NETWORK_TYPE_ARP, 0};  // 0 as end of list

  // Initialise BNEP
  bnep_init();
  // Minimum L2CAP MTU for bnep is 1691 bytes
```

```
#ifdef ENABLE_PANU_CLIENT
  bnep_register_service(packet_handler,
BLUETOOTH_SERVICE_CLASS_PANU, 1691);
  // PANU
  pan_create_panu_sdp_record(panu_sdp_record,
sdp_create_service_record_handle(), network_packet_types, NULL,
NULL, BNEP_SECURITY_NONE);
#else
  bnep_register_service(packet_handler,
BLUETOOTH_SERVICE_CLASS_NAP, 1691);
  // NAP Network Access Type: Other, 1 MB/s
  pan_create_nap_sdp_record(panu_sdp_record,
sdp_create_service_record_handle(), network_packet_types, NULL,
NULL, BNEP_SECURITY_NONE, PAN_NET_ACCESS_TYPE_OTHER, 1000000,
NULL, NULL);
#endif
  btstack_assert(de_get_len( panu_sdp_record) <=
sizeof(panu_sdp_record));
  sdp_register_service(panu_sdp_record);

  // Initialize network interface
  btstack_network_init(&network_send_packet_callback);

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

SDP parser callback

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

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) {
#ifdef ENABLE_PANU_CLIENT
        case BTSTACK_EVENT_STATE:
          if (btstack_event_state_get_state(packet) ==
HCI_STATE_WORKING){
            printf("Start SDP BNEP query for remote PAN Network
Access Point (NAP).\n");

sdp_client_query_uuid16(&handle_sdp_client_query_result,
remote_addr, BLUETOOTH_SERVICE_CLASS_NAP);
          }
          break;
#endif
...

                case BNEP_EVENT_CHANNEL_OPENED:
          if (bnep_event_channel_opened_get_status(packet)) {
            printf("BNEP channel open failed, status 0x%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);
            bnep_event_channel_opened_get_remote_address(packet,
event_addr);
            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);
            btstack_network_up(local_addr);
            printf("Network Interface %s activated\n",
```

```
btstack_network_get_name());
          }
                    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_network_down();
          break;

        case BNEP_EVENT_CAN_SEND_NOW:
          if (network_buffer_len > 0) {
            bnep_send(bnep_cid, (uint8_t*) network_buffer,
network_buffer_len);
            network_buffer_len = 0;
            btstack_network_packet_sent();
          }
          break;

        default:
          break;
      }
      break;

    case BNEP_DATA_PACKET:
      // Write out the ethernet frame to the network interface
      btstack_network_process_packet(packet, size);
      break;

    default:
      break;
  }
}
```

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 send of a stored network packet. The tap datas source can be enabled 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.

Network packet handler

A pointer to the network packet is stored and a BNEP_EVENT_CAN_SEND_NOW requested

```
static void network_send_packet_callback(const uint8_t * packet,
uint16_t size){
  network_buffer = packet;
  network_buffer_len = size;
  bnep_request_can_send_now_event(bnep_cid);
}
```

HID Keyboard Classic

Source Code: hid_keyboard_demo.c

This HID Device example demonstrates how to implement an HID keyboard. Without a HAVE_BTSTACK_STDIN, a fixed demo text is sent If HAVE_BTSTACK_STDIN is defined, you can type from the terminal

Main Application Setup

Listing here shows main application code. To run a HID Device service you need to initialize the SDP, and to create and register HID Device record with it. 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;

  // allow to get found by inquiry
  gap_discoverable_control(1);
  // use Limited Discoverable Mode; Peripheral; Keyboard as CoD
  gap_set_class_of_device(0x2540);
  // set local name to be identified - zeroes will be replaced by
actual BD ADDR
  gap_set_local_name("HID Keyboard Demo 00:00:00:00:00:00");
  // allow for role switch in general and sniff mode

gap_set_default_link_policy_settings( LM_LINK_POLICY_ENABLE_ROLE_SW
| LM_LINK_POLICY_ENABLE_SNIFF_MODE );
  // allow for role switch on outgoing connections - this allow
HID Host to become master when we re-connect to it
  gap_set_allow_role_switch(true);

  // L2CAP
  l2cap_init();

#ifdef ENABLE_BLE
```

```
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  // SDP Server
  sdp_init();
  memset(hid_service_buffer, 0, sizeof(hid_service_buffer));

  uint8_t hid_virtual_cable = 0;
  uint8_t hid_remote_wake = 1;
  uint8_t hid_reconnect_initiate = 1;
  uint8_t hid_normally_connectable = 1;

  hid_sdp_record_t hid_params = {
    // hid sevice subclass 2540 Keyboard, hid counntry code 33 US
    0x2540, 33,
    hid_virtual_cable, hid_remote_wake,
    hid_reconnect_initiate, hid_normally_connectable,
    hid_boot_device,
    host_max_latency, host_min_timeout,
    3200,
    hid_descriptor_keyboard,
    sizeof(hid_descriptor_keyboard),
    hid_device_name
  };

  hid_create_sdp_record(hid_service_buffer,
sdp_create_service_record_handle(), &hid_params);
  btstack_assert(de_get_len( hid_service_buffer) <=
sizeof(hid_service_buffer));
  sdp_register_service(hid_service_buffer);

  // See https://www.bluetooth.com/specifications/assigned-
numbers/company-identifiers if you don't have a USB Vendor ID and
need a Bluetooth Vendor ID
  // device info: BlueKitchen GmbH, product 1, version 1
  device_id_create_sdp_record(device_id_sdp_service_buffer,
sdp_create_service_record_handle(),
DEVICE_ID_VENDOR_ID_SOURCE_BLUETOOTH,
```

```
BLUETOOTH_COMPANY_ID_BLUEKITCHEN_GMBH, 1, 1);
  btstack_assert(de_get_len( device_id_sdp_service_buffer) <=
sizeof(device_id_sdp_service_buffer));
  sdp_register_service(device_id_sdp_service_buffer);

  // HID Device
  hid_device_init(hid_boot_device,
sizeof(hid_descriptor_keyboard), hid_descriptor_keyboard);

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

  // register for HID events
  hid_device_register_packet_handler(&packet_handler);

#ifdef HAVE_BTSTACK_STDIN
  sscanf_bd_addr(device_addr_string, device_addr);
  btstack_stdin_setup(stdin_process);
#endif

  btstack_ring_buffer_init(&send_buffer, send_buffer_storage,
sizeof(send_buffer_storage));

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

HID Mouse Classic

Source Code: hid_mouse_demo.c

This HID Device example demonstrates how to implement an HID keyboard. Without a HAVE_BTSTACK_STDIN, a fixed demo text is sent If HAVE_BTSTACK_STDIN is defined, you can type from the terminal

Main Application Setup

Listing here shows main application code. To run a HID Device service you need to initialize the SDP, and to create and register HID Device record with it. 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;

  // allow to get found by inquiry
  gap_discoverable_control(1);
  // use Limited Discoverable Mode; Peripheral; Pointing Device as
CoD
  gap_set_class_of_device(0x2580);
  // set local name to be identified - zeroes will be replaced by
actual BD ADDR
  gap_set_local_name("HID Mouse Demo 00:00:00:00:00:00");
  // allow for role switch in general and sniff mode

gap_set_default_link_policy_settings( LM_LINK_POLICY_ENABLE_ROLE_SW
| LM_LINK_POLICY_ENABLE_SNIFF_MODE );
  // allow for role switch on outgoing connections - this allow
HID Host to become master when we re-connect to it
  gap_set_allow_role_switch(true);

  // L2CAP
  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
  sm_init();
#endif

  // SDP Server
  sdp_init();

  uint8_t hid_virtual_cable = 0;
```

```
  uint8_t hid_remote_wake = 1;
  uint8_t hid_reconnect_initiate = 1;
  uint8_t hid_normally_connectable = 1;

  hid_sdp_record_t hid_params = {
    // hid sevice subclass 2580 Mouse, hid counntry code 33 US
    0x2580, 33,
    hid_virtual_cable, hid_remote_wake,
    hid_reconnect_initiate, hid_normally_connectable,
    hid_boot_device,
    0xFFFF, 0xFFFF, 3200,
    hid_descriptor_mouse_boot_mode,
    sizeof(hid_descriptor_mouse_boot_mode),
    hid_device_name
  };

  memset(hid_service_buffer, 0, sizeof(hid_service_buffer));
  hid_create_sdp_record(hid_service_buffer,
sdp_create_service_record_handle(), &hid_params);
  btstack_assert(de_get_len( hid_service_buffer) <=
sizeof(hid_service_buffer));
  sdp_register_service(hid_service_buffer);

  // See https://www.bluetooth.com/specifications/assigned-
numbers/company-identifiers if you don't have a USB Vendor ID and
need a Bluetooth Vendor ID
  // device info: BlueKitchen GmbH, product 2, version 1
  device_id_create_sdp_record(device_id_sdp_service_buffer,
sdp_create_service_record_handle(),
DEVICE_ID_VENDOR_ID_SOURCE_BLUETOOTH,
BLUETOOTH_COMPANY_ID_BLUEKITCHEN_GMBH, 2, 1);
  btstack_assert(de_get_len( device_id_sdp_service_buffer) <=
sizeof(device_id_sdp_service_buffer));
  sdp_register_service(device_id_sdp_service_buffer);

  // HID Device
  hid_device_init(hid_boot_device,
sizeof(hid_descriptor_mouse_boot_mode),
hid_descriptor_mouse_boot_mode);
  // register for HCI events
```

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

  // register for HID
  hid_device_register_packet_handler(&packet_handler);

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

HID Host Classic

Source Code: hid_host_demo.c

This example implements a HID Host. For now, it connects to a fixed device. It will connect in Report protocol mode if this mode is supported by the HID Device, otherwise it will fall back to BOOT protocol mode.

Main application configuration

In the application configuration, L2CAP and HID host are initialized, and the link policies are set to allow sniff mode and role change.

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

static void hid_host_setup(void){

  // Initialize L2CAP
  l2cap_init();

#ifdef ENABLE_BLE
  // Initialize LE Security Manager. Needed for cross-transport
key derivation
```

```
  sm_init();
#endif

  // Initialize HID Host
  hid_host_init(hid_descriptor_storage,
sizeof(hid_descriptor_storage));
  hid_host_register_packet_handler(packet_handler);

  // Allow sniff mode requests by HID device and support role
switch

gap_set_default_link_policy_settings(LM_LINK_POLICY_ENABLE_SNIFF_MC
| LM_LINK_POLICY_ENABLE_ROLE_SWITCH);

  // try to become master on incoming connections
  hci_set_master_slave_policy(HCI_ROLE_MASTER);

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

  // make discoverable to allow HID device to initiate connection
  gap_discoverable_control(1);

  // Disable stdout buffering
  setvbuf(stdin, NULL, _IONBF, 0);
}
```

HID Report Handler

Use BTstack's compact HID Parser to process incoming HID Report in Report protocol mode. Iterate over all fields and process fields with usage page = 0x07 / Keyboard Check if SHIFT is down and process first character (don't handle multiple key presses)

Packet Handler

The packet handler responds to various HID 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) {
#ifndef HAVE_BTSTACK_STDIN
         case BTSTACK_EVENT_STATE:
           if (btstack_event_state_get_state(packet) ==
HCI_STATE_WORKING){
             status = hid_host_connect(remote_addr,
hid_host_report_mode, &hid_host_cid);
             if (status != ERROR_CODE_SUCCESS){
               printf("HID host connect failed, status 0x%02x.\n",
status);
             }
           }
           break;
#endif
...
         case HCI_EVENT_HID_META:
           switch (hci_event_hid_meta_get_subevent_code(packet)){

             case HID_SUBEVENT_INCOMING_CONNECTION:
               // There is an incoming connection: we can accept it
or decline it

               // If there are no resources to handle the
connection,
               // the status will be
ERROR_CODE_MEMORY_CAPACITY_EXCEEDED

               // The hid_host_report_mode in the
hid_host_accept_connection function
               // allows the application to request a protocol
mode.
```

```
              // For available protocol modes, see
hid_protocol_mode_t in btstack_hid.h file.
              if
(hid_subevent_incoming_connection_get_status(packet) ==
ERROR_CODE_SUCCESS) {

hid_host_accept_connection(hid_subevent_incoming_connection_get_hic
hid_host_report_mode);
              }
              break;

            case HID_SUBEVENT_CONNECTION_OPENED:
              // The status field of this event indicates if the
control and interrupt
              // connections were opened successfully.
              status =
hid_subevent_connection_opened_get_status(packet);
              if (status != ERROR_CODE_SUCCESS) {
                printf("Connection failed, status 0x%02x\n",
status);
                app_state = APP_IDLE;
                hid_host_cid = 0;
                return;
              }
              app_state = APP_CONNECTED;
              hid_host_descriptor_available = false;
              hid_host_cid =
hid_subevent_connection_opened_get_hid_cid(packet);
              hid_host_caps_lock = false;
              hid_host_led_report_len = 0;
              printf("HID Host connected.\n");
              break;

            case HID_SUBEVENT_DESCRIPTOR_AVAILABLE:
              // This event will follows
HID_SUBEVENT_CONNECTION_OPENED event.
              // For incoming connections, i.e. HID Device
initiating the connection,
              // the HID_SUBEVENT_DESCRIPTOR_AVAILABLE is delayed,
and some HID
```

```
              // reports may be received via HID_SUBEVENT_REPORT
event. It is up to
              // the application if these reports should be
buffered or ignored until
              // the HID descriptor is available.
              status =
hid_subevent_descriptor_available_get_status(packet);
              if (status == ERROR_CODE_SUCCESS){
                hid_host_descriptor_available = true;
                printf("HID Descriptor available, please start
typing.\n");
                hid_host_demo_lookup_caps_lock_led();
              } else {
                printf("Cannot handle input report, HID Descriptor
is not available, status 0x%02x\n", status);
              }
              break;

            case HID_SUBEVENT_REPORT:
              // Handle input report.
              if (hid_host_descriptor_available){

hid_host_handle_interrupt_report(hid_subevent_report_get_report(pac
hid_subevent_report_get_report_len(packet));
              } else {

printf_hexdump(hid_subevent_report_get_report(packet),
hid_subevent_report_get_report_len(packet));
              }
              break;

            case HID_SUBEVENT_SET_PROTOCOL_RESPONSE:
              // For incoming connections, the library will set
the protocol mode of the
              // HID Device as requested in the call to
hid_host_accept_connection. The event
              // reports the result. For connections initiated by
calling hid_host_connect,
              // this event will occur only if the established
report mode is boot mode.
```

```
                status =
hid_subevent_set_protocol_response_get_handshake_status(packet);
                if (status != HID_HANDSHAKE_PARAM_TYPE_SUCCESSFUL){
                  printf("Error set protocol, status 0x%02x\n",
status);
                  break;
                }
                switch
((hid_protocol_mode_t)hid_subevent_set_protocol_response_get_protoc
{
                  case HID_PROTOCOL_MODE_BOOT:
                    printf("Protocol mode set: BOOT.\n");
                    break;
                  case HID_PROTOCOL_MODE_REPORT:
                    printf("Protocol mode set: REPORT.\n");
                    break;
                  default:
                    printf("Unknown protocol mode.\n");
                    break;
                }
                break;

              case HID_SUBEVENT_CONNECTION_CLOSED:
                // The connection was closed.
                hid_host_cid = 0;
                hid_host_descriptor_available = false;
                printf("HID Host disconnected.\n");
                break;

              default:
                break;
            }
            break;
          default:
            break;
        }
        break;
      default:
        break;
```

```
    }
}
```

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

HID Keyboard LE

Source Code: hog_keyboard_demo.c

HID Mouse LE

Source Code: hog_mouse_demo.c

HID Boot Host LE

Source Code: hog_boot_host_demo.c

This example implements a minimal HID-over-GATT Boot Host. It scans for LE HID devices, connects to it, discovers the Characteristics relevant for the HID Service and enables Notifications on them. It then dumps all Boot Keyboard and Mouse Input Reports

HOG Boot Keyboard Handler

Boot Keyboard Input Report contains a report of format [modifier, reserved, 6 x usage for key 1..6 from keyboard usage] Track new usages, map key usage to actual character and simulate terminal

HOG Boot Mouse Handler

Boot Mouse Input Report contains a report of format [buttons, dx, dy, dz = scroll wheel] Decode packet and print on stdout

@param packet_type @param channel @param packet @param size

Test if advertisement contains HID UUID

```
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) break;
          btstack_assert(app_state == W4_WORKING);
          hog_start_connect();
          break;
        case GAP_EVENT_ADVERTISING_REPORT:
          if (app_state != W4_HID_DEVICE_FOUND) break;
          if (adv_event_contains_hid_service(packet) == false)
break;
          // stop scan
          gap_stop_scan();
          // store remote device address and type
          gap_event_advertising_report_get_address(packet,
remote_device.addr);
          remote_device.addr_type =
gap_event_advertising_report_get_address_type(packet);
          // connect
          printf("Found, connect to device with %s address %s ...
\n", remote_device.addr_type == 0 ? "public" : "random" ,
bd_addr_to_str(remote_device.addr));
          hog_connect();
          break;
        case HCI_EVENT_DISCONNECTION_COMPLETE:
          if (app_state != READY) break;

          connection_handle = HCI_CON_HANDLE_INVALID;
          switch (app_state){
            case READY:
              printf("\nDisconnected, try to reconnect...\n");
```

```
              app_state = W4_TIMEOUT_THEN_RECONNECT;
              break;
            default:
              printf("\nDisconnected, start over...\n");
              app_state = W4_TIMEOUT_THEN_SCAN;
              break;
          }
          // set timer
          btstack_run_loop_set_timer(&connection_timer, 100);
          btstack_run_loop_set_timer_handler(&connection_timer,
&hog_reconnect_timeout);
          btstack_run_loop_add_timer(&connection_timer);
          break;
        case HCI_EVENT_META_GAP:
          // wait for connection complete
          if (hci_event_gap_meta_get_subevent_code(packet) !=
GAP_SUBEVENT_LE_CONNECTION_COMPLETE) break;
          if (app_state != W4_CONNECTED) return;
          btstack_run_loop_remove_timer(&connection_timer);
          connection_handle =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          // request security
          app_state = W4_ENCRYPTED;
          sm_request_pairing(connection_handle);
          break;
        default:
          break;
      }
      break;
    default:
      break;
  }
}
```

HCI packet handler

The SM packet handler receives Security Manager Events required for pairing. It also receives events generated during Identity Resolving see Listing here.

```
static void sm_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;

  bool connect_to_service = false;

  switch (hci_event_packet_get_type(packet)) {
    case SM_EVENT_JUST_WORKS_REQUEST:
      printf("Just works requested\n");

sm_just_works_confirm(sm_event_just_works_request_get_handle(packet
      break;
    case SM_EVENT_NUMERIC_COMPARISON_REQUEST:
      printf("Confirming numeric comparison: %"PRIu32"\n",
sm_event_numeric_comparison_request_get_passkey(packet));

sm_numeric_comparison_confirm(sm_event_passkey_display_number_get_h
      break;
    case SM_EVENT_PASSKEY_DISPLAY_NUMBER:
      printf("Display Passkey: %"PRIu32"\n",
sm_event_passkey_display_number_get_passkey(packet));
      break;
    case SM_EVENT_PAIRING_COMPLETE:
      switch (sm_event_pairing_complete_get_status(packet)){
        case ERROR_CODE_SUCCESS:
          printf("Pairing complete, success\n");
          connect_to_service = true;
          break;
        case ERROR_CODE_CONNECTION_TIMEOUT:
          printf("Pairing failed, timeout\n");
          break;
        case ERROR_CODE_REMOTE_USER_TERMINATED_CONNECTION:
          printf("Pairing failed, disconnected\n");
          break;
        case ERROR_CODE_AUTHENTICATION_FAILURE:
          printf("Pairing failed, reason = %u\n",
sm_event_pairing_complete_get_reason(packet));
```

```
            break;
          default:
            break;
        }
        break;
      case SM_EVENT_REENCRYPTION_COMPLETE:
        printf("Re-encryption complete, success\n");
        connect_to_service = true;
        break;
      default:
        break;
    }

    if (connect_to_service){
      // continue - query primary services
      app_state = W2_QUERY_HID_SERVICE;
      hog_host_request_to_send();
    }
  }


    l2cap_init();

    // setup SM: Display only
    sm_init();
    sm_set_io_capabilities(IO_CAPABILITY_DISPLAY_ONLY);
    sm_set_authentication_requirements(SM_AUTHREQ_SECURE_CONNECTION
  | SM_AUTHREQ_BONDING);

    //
    gatt_client_init();

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

    // register for events from Security Manager
    sm_event_callback_registration.callback = &sm_packet_handler;
    sm_add_event_handler(&sm_event_callback_registration);
```

Dual Mode - SPP and LE Counter

Source Code: spp_and_le_counter.c

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 the stack is configured.

Note: To test, please run the example, and then:

- for SPP pair from a remote device, and open the Virtual Serial Port,
- for LE use some GATT Explorer, e.g. LightBlue, BLExplr, to enable notifications.

Advertisements

The Flags attribute in the Advertisement Data indicates if a device is dual-mode or le-only.

```
const uint8_t adv_data[] = {
  // Flags general discoverable
  0x02, BLUETOOTH_DATA_TYPE_FLAGS, 0x02,
  // 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,
};
```

Packet Handler

The packet handler of the combined example is just the combination of the individual packet handlers.

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);
}
```

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)
{
  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,
sdp_create_service_record_handle(), RFCOMM_SERVER_CHANNEL, "SPP
Counter");
  btstack_assert(de_get_len( spp_service_buffer) <=
sizeof(spp_service_buffer));
  sdp_register_service(spp_service_buffer);

#ifdef ENABLE_GATT_OVER_CLASSIC
  // init SDP, create record for GATT and register with SDP
  memset(gatt_service_buffer, 0, sizeof(gatt_service_buffer));
  gatt_create_sdp_record(gatt_service_buffer, 0x10001,
ATT_SERVICE_GATT_SERVICE_START_HANDLE,
ATT_SERVICE_GATT_SERVICE_END_HANDLE);
  sdp_register_service(gatt_service_buffer);
  printf("SDP service record size: %u\n",
de_get_len(gatt_service_buffer));
#endif

  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 SM: Display only
  sm_init();

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

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

  // register for ATT events
  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);

  // 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();

  // turn on!
    hci_power_control(HCI_POWER_ON);

  return 0;
}
```

Performance - Stream Data over GATT (Server)

Source Code: gatt_streamer_server.c

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.

> In theory, we should also update the connection parameters, but we already get a connection interval of 30 ms and there's no public way to use a shorter interval with iOS (if we're not implementing an HID device).
>
> Note: To start the streaming, run the example. On remote device use some GATT Explorer, e.g. LightBlue, BLExplr to enable notifications.

Main Application Setup

Listing here shows main application 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){

  l2cap_init();

  // setup SM: Display only
  sm_init();

#ifdef ENABLE_GATT_OVER_CLASSIC
  // init SDP, create record for GATT and register with SDP
  sdp_init();
  memset(gatt_service_buffer, 0, sizeof(gatt_service_buffer));
  gatt_create_sdp_record(gatt_service_buffer, 0x10001,
ATT_SERVICE_GATT_SERVICE_START_HANDLE,
ATT_SERVICE_GATT_SERVICE_END_HANDLE);
  sdp_register_service(gatt_service_buffer);
  printf("SDP service record size: %u\n",
de_get_len(gatt_service_buffer));

  // configure Classic GAP
  gap_set_local_name("GATT Streamer BR/EDR 00:00:00:00:00:00");
  gap_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);
  gap_discoverable_control(1);
#endif
```

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

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

  // register for ATT events
  att_server_register_packet_handler(att_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();
}
```

Track throughput

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

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

static void test_track_evaluate(le_streamer_connection_t *
context) {
  // 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_sent = context->test_data_sent * 1000 /
time_passed;
  int bytes_per_second_received = context->test_data_received *
1000 / time_passed;
  int bytes_per_second_total = bytes_per_second_sent +
bytes_per_second_received;
  printf("%c: Sent %u.%03u kB/s, Received %u.%03u kB/s => Total
%u.%03u kB/s\n", context->name,
    bytes_per_second_sent   / 1000, bytes_per_second_sent   %
1000,
    bytes_per_second_received / 1000, bytes_per_second_received %
1000,
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);
  // store into test data buffer for use with GATT Browser
  btstack_snprintf_best_effort(context->test_data, sizeof(context-
>test_data),
    "Notification Throughput %u.%03u kB/s",
    bytes_per_second_total  / 1000, bytes_per_second_total  %
1000);

  // restart
  test_reset(context);
}

static void test_track_data(le_streamer_connection_t * context,
uint32_t bytes_sent, uint32_t bytes_received){
  context->test_data_sent   += bytes_sent;
  context->test_data_received += bytes_received;
  test_track_evaluate(context);
}
```

HCI Packet Handler

The packet handler is used track incoming connections and to stop notifications on disconnect It is also a good place to request the connection parameter update as indicated in the commented code block.

```
static void hci_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;

  uint16_t conn_interval;
  hci_con_handle_t con_handle;
  static const char * const phy_names[] = {
    "Reserved", "1 M", "2 M", "Codec"
  };

  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("To start the streaming, please run the
le_streamer_client example on other device, or use some GATT
Explorer, e.g. LightBlue, BLExplr.\n");
      }
      break;
    case HCI_EVENT_DISCONNECTION_COMPLETE:
      con_handle =
hci_event_disconnection_complete_get_connection_handle(packet);
      printf("- LE Connection 0x%04x: disconnect, reason %02x\n",
con_handle,
hci_event_disconnection_complete_get_reason(packet));
      break;
    case HCI_EVENT_META_GAP:
      switch (hci_event_gap_meta_get_subevent_code(packet)) {
        case GAP_SUBEVENT_LE_CONNECTION_COMPLETE:
```

```
          // print connection parameters (without using float
operations)
          con_handle  =
gap_subevent_le_connection_complete_get_connection_handle(packet);
          conn_interval =
gap_subevent_le_connection_complete_get_conn_interval(packet);
          printf("- LE Connection 0x%04x: connected - connection
interval %u.%02u ms, latency %u\n", con_handle, conn_interval *
125 / 100,
            25 * (conn_interval & 3),
gap_subevent_le_connection_complete_get_conn_latency(packet));

          // request min con interval 15 ms for iOS 11+
          printf("- LE Connection 0x%04x: request 15 ms connection
interval\n", con_handle);
          gap_request_connection_parameter_update(con_handle, 12,
12, 4, 0x0048);
          break;
        default:
          break;
      }
      break;
    case HCI_EVENT_LE_META:
      switch (hci_event_le_meta_get_subevent_code(packet)) {
        case HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE:
          // print connection parameters (without using float
operations)
          con_handle  =
hci_subevent_le_connection_update_complete_get_connection_handle(pa
          conn_interval =
hci_subevent_le_connection_update_complete_get_conn_interval(packet
          printf("- LE Connection 0x%04x: connection update -
connection interval %u.%02u ms, latency %u\n", con_handle,
conn_interval * 125 / 100,
            25 * (conn_interval & 3),
hci_subevent_le_connection_update_complete_get_conn_latency(packet)
          break;
        case HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE:
          con_handle =
hci_subevent_le_data_length_change_get_connection_handle(packet);
```

```
          printf("- LE Connection 0x%04x: data length change - max
%u bytes per packet\n", con_handle,

hci_subevent_le_data_length_change_get_max_tx_octets(packet));
          break;
        case HCI_SUBEVENT_LE_PHY_UPDATE_COMPLETE:
          con_handle =
hci_subevent_le_phy_update_complete_get_connection_handle(packet);
          printf("- LE Connection 0x%04x: PHY update - using LE %s
PHY now\n", con_handle,

phy_names[hci_subevent_le_phy_update_complete_get_tx_phy(packet)]);
          break;
        default:
          break;
      }
      break;

    default:
      break;
  }
}
```

ATT Packet Handler

The packet handler is used to track the ATT MTU Exchange and trigger ATT send

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

  int mtu;
  le_streamer_connection_t * context;
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (hci_event_packet_get_type(packet)) {
        case ATT_EVENT_CONNECTED:
          // setup new
```

```
            context =
connection_for_conn_handle(HCI_CON_HANDLE_INVALID);
            if (!context) break;
            context->connection_handle =
att_event_connected_get_handle(packet);
            context->test_data_len =
btstack_min(att_server_get_mtu(context->connection_handle) - 3,
sizeof(context->test_data));
            // store into test data buffer for use with GATT Browser
            btstack_snprintf_best_effort(context->test_data,
context  ->test_data_len, "GATT Streamer starting up...");
            printf("%c: ATT connected, handle 0x%04x, test data len
%u\n", context->name, context->connection_handle, context-
>test_data_len);
            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_hanc
            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;
          case ATT_EVENT_DISCONNECTED:
            context =
connection_for_conn_handle(att_event_disconnected_get_handle(packet
            if (!context) break;
            // free connection
            printf("%c: ATT disconnected, handle 0x%04x\n", context-
>name, context->connection_handle);
            context->le_notification_enabled = 0;
            context->connection_handle = HCI_CON_HANDLE_INVALID;
            break;
          default:
```

```
          break;
      }
      break;
    default:
      break;
  }
}
```

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];

  // send
  att_server_notify(context->connection_handle, context-
>value_handle, (uint8_t*) context->test_data, context-
```

```
>test_data_len);

  // track
  test_track_data(context, context->test_data_len, 0);

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

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

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 here.

```
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);

  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_C
      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){
        switch (att_handle){
```

```
          case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_C
            context->value_handle =
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
            break;
          default:
            break;
        }
        att_server_request_can_send_now_event(context-
>connection_handle);
      }
      test_reset(context);
      break;
    case
ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HA
      test_track_data(context, 0, buffer_size);
      break;
    default:
      printf("Write to 0x%04x, len %u\n", att_handle,
buffer_size);
      break;
  }
  return 0;
}
```

SDP Client - Query Remote SDP Records

Source Code: sdp_general_query.c

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

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 here. Via this handler, the SDP 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_general_query_init(void){
  // init L2CAP
  l2cap_init();

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

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 here. In this example we used fixed address of the remote device shown in Listing here. Please update it with the address of a device in your vicinity, e.g., one reported by the GAP Inquiry example in the previous section.

```
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("Connecting to %s\n", bd_addr_to_str(remote_addr));
        sdp_client_query_uuid16(&handle_sdp_client_query_result,
remote_addr, BLUETOOTH_PROTOCOL_L2CAP);
      }
      break;
    default:
      break;
  }
}
```

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 here.

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 (hci_event_packet_get_type(packet)){
    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(pa


attribute_value[sdp_event_query_attribute_byte_get_data_offset(pack
= 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;
    default:
      break;
  }
}
```

SDP Client - Query RFCOMM SDP record

Source Code: sdp_rfcomm_query.c

The example shows how the SDP Client is used to get all RFCOMM service records from a remote device. It extracts the remote RFCOMM Server Channel, which are needed to connect to a remote RFCOMM service.

SDP Client - Query BNEP SDP record

Source Code: sdp_bnep_query.c

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.

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 here. Via this handler, the SDP 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_bnep_qeury_init(void){
  // init L2CAP
  l2cap_init();

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

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;
  }
}
```

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 here. Here, we show how to 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), (int)
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);
                des_iterator_next(&prot_it);
                switch (uuid){
                  case BLUETOOTH_PROTOCOL_L2CAP:
                    if (!des_iterator_has_more(&prot_it))
continue;

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;

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;
...
}
```

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.

PBAP Client - Get Contacts from Phonebook Server

Source Code: pbap_client_demo.c

Note: The Bluetooth address of the remote Phonbook server is hardcoded. Change it before running example, then use the UI to connect to it, to set and query contacts.

Testing - Enable Device Under Test (DUT) Mode for Classic

Source Code: dut_mode_classic.c

DUT mode can be used for production testing. This example just configures the Bluetooth Controller for DUT mode.

Bluetooth Logic

When BTstack is up and running, send Enable Device Under Test Mode Command and print its result.

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

Main Application Setup

Listing here shows main application 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[]) {
```

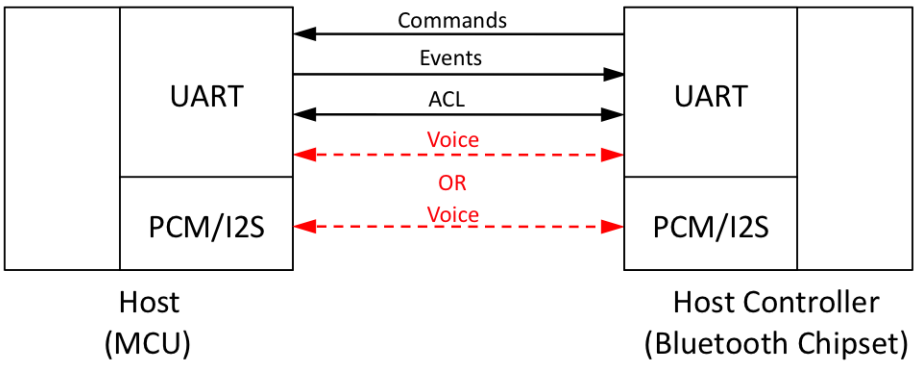
```
  (void)argc;
  (void)argv;

  // disable Secure Simple Pairinng
  gap_ssp_set_enable(0);

  // make device connectable
  // @note: gap_connectable_control will be enabled when an L2CAP
service
  // (e.g. RFCOMM) is initialized). Therefore, it's not needed in
regular applications
  gap_connectable_control(1);

  // make device discoverable
  gap_discoverable_control(1);

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

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

#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.

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 below. 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.



Host Controller to Host connection

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.

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.

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.

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*

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.

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.

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.

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 documentation on the nRF5 series is complete and very informative. TI and Nordic also provide excellent support via their respective web forum.

Infineon acquired Cypress Semiconductor Corporation in 2020, which acquired the Bluetooth + Wifi division of Broadcom in 2016 provides support via their Community Forum. In addition, firmware updates (PatchRAM files) for Bluetooth + Wifi controllers are available via Murata's Cypress GitHub.

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

Chipset Overview

As the following Table has too many columns, we split it in two parts.

Part 1.

Chipset	Type	HCI Transport	BD_ADDR (1)	SCO over HCI (2)	LE DLE	Multiple LE Roles (3)
Atmel ATWILC3000	LE	H4	Yes	n.a	No	No
Broadcom UART	Dual mode	H4, H5	Rarely	Partially (2)	No	Maybe (3)
Broadcom USB Dongles	Dual mode	USB	Yes	Yes	No	No
CSR UART	Dual mode	H4, H5, BCSP	Rarely	Partially (2)	No	No
CSR USB Dongles	Dual mode	USB	Mostly	Yes	No	No
Infineon CYW207xx	Dual mode	H4, H5, USB	Don’t know	Partially (2)	Yes	Yes
Infineon CYW208xx	Dual mode	H4, H5, USB	Don’t know	Partially (2)	Yes	Yes
Infineon CYW43xxx	Dual mode + Wifi	H4, H5	Don’t know	Partially (2)	Don’t know	On newer versions
Infineon CYW5310	Dual mode	H4	Don’t know	Yes	Yes	Yes
Infineon CYW5551x, CYW5557x	Dual mode + Wifi	H4, H5	Mostly	Yes	Yes	Yes
Cypress PSoC 4	LE	H4	Don’t know	n.a.	Yes	Don’t know
Dialog DA14531	LE	H4	No	n.a.	Yes	Yes
Dialog DA14581	LE	H4, SPI	No	n.a.	No	No
Dialog DA14585	LE	H4, SPI	No	n.a.	Yes	Yes
Dialog DA1469x	LE	H4, SPI	No	n.a.	Yes	Yes

Chipset	Type	HCI Transport	BD_ADDR (1)	SCO over HCI (2)	LE DLE	Multiple LE Roles (3)
Espressif ESP32	Dual mode + Wifi	VHCI, H4	Yes	Yes	Yes	Yes
Espressif ESP32-S3, C2, C3, C5, C6,H2	LE + Wifi	VHCI, H4	Yes	No	Yes	Yes
EM 9301	LE	SPI, H4	No	n.a.	No	No
EM 9304	LE	SPI, H4	Yes	n.a.	Yes	Yes
EM 9305	LE	SPI, H4	Yes	n.a.	Yes	Yes
Intel Dual Wireless 3165, 8260, 8265	Dual mode	USB	Yes	Probably	Don’t know	Don’t know
Nordic nRF	LE	H4	Fixed Random	n.a.	Yes	Yes
NXP 88W8997	Dual mode	H4	Yes	Partially(2)	Yes	Yes
NXP IW416	Dual mode	H4	Yes	No	Yes	Yes
NXP IW61x	Dual mode	H4	Yes	Partially(2)	Yes	Yes
STM STLC2500D	Classic	H4	No	Don’t know	n.a	n.a.
STM32WB	LE	VHCI	Yes	n.a.	Yes	Yes
STM32WB0	LE	VHCI, H4	Yes	n.a.	Yes	Yes
Renesas RX23W	LE	H4	No	n.a.	Yes	Yes
Realtek UART	Dual mode + Wifi	H5	Yes	Yes	Don’t know	Don’t know
		USB	Yes	Yes		

Chipset	Type	HCI Transport	BD_ADDR (1)	SCO over HCI (2)	LE DLE	Multiple LE Roles (3)
Realtek USB Dongles	Dual mode + Wifi				Don’t know	Don’t know
Toshiba TC35661	Dual mode	H4	No	No	No	No
TI CC256x, WL183x	Dual mode	H4, H5, eHCILL	Yes	Yes	No	Yes for CC256XC

Part 2.

Chipset	Classic SC (4)	LE Addr Resolution	BTstack folder	Comment
Atmel ATWILC3000	n.a.	Don’t know	atwilc3000	BLE Firmware size: 60 kB
Broadcom UART	43438: Yes		bcm	Max UART baudrate 2 mbps
Broadcom USB Dongles	BCM20702: No		bcm	
CSR UART	CSR8811: No		csr	
CSR USB Dongles	CSR8510: No		csr	
Infineon CYW207xx	Yes	Yes	bcm	
Infineon CYW208xx	Yes	Don’t know	bcm	Keep CTS high during power cycle
Infineon CYW43xxx	On newer versions	On newer versions	bcm	Bluetooth + Wifi Combo Controller
Infineon CYW55310	Yes	Yes	bcm	

Chipset	Classic SC (4)	LE Addr Resolution	BTstack folder	Comment
				autobaud-mode needed, see posix-h4-airoc
Infineon CYW5551x, CYW5557x	Yes	Yes	bcm	autobaud-mode needed, see posix-h4-airoc
Cypress PSoC 4	n.a.	Don’t know		HCI Firmware part of PSoC Creator kits examples
Dialog DA14531	n.a.	Don’t know	da145xx	Official HCI firmware included in BTstack
Dialog DA14581	n.a.	Don’t know	da145xx	Official HCI firmware included in BTstack
Dialog DA14585	n.a.	Yes	da145xx	Official HCI firmware included in BTstack
Dialog DA1469x	n.a.	Yes	da145xx	HCI Firmware part of DA1469x SDK
Espressif ESP32	Yes	Don’t know		SoC with Bluetooth and Wifi
Espressif ESP32-S3, C2, C3, C5,C6,H2	Yes	Yes		SoC with Bluetooth and Wifi
EM 9301	n.a.	Don’t know	em9301	Custom HCI SPI implementation
EM 9304	n.a.		em9301	

Chipset	Classic SC (4)	LE Addr Resolution	BTstack folder	Comment
		Don’t know		Custom HCI SPI implementation
EM 9305	n.a.	Yes	em9301	Custom HCI SPI implementation
Intel Dual Wireless 3165, 8260, 8265	Don’t know	Don’t know	intel	Firmware size: 400 kB
Nordic nRF	n.a.	Yes		Requires HCI firmware
NXP 88W8997	No	Yes	nxp	Requires initial firmware
NXP IW416	No	Yes	nxp	Requires initial firmware
NXP IW61x	No	Yes	nxp	Requires initial firmware
STM STLC2500D	No	n.a.	stlc2500d	Custom deep sleep management not supported
STM32WB	n.a.	Yes		See port/ stm32wb55x-nucleo-freertos
STM32WB0	n.a.	Yes		HCI Firmware part of STM32WB0 Cube Package
Renesas RX23W	n.a .	Don’t know		HCI Firmware part of BTTS
Realtek UART	Don’t know	Don’t know	realtek	Requires initial firmware + config
Realtek USB Dongles	Don’t know	Don’t know	realtek	

Chipset	Classic SC (4)	LE Addr Resolution	BTstack folder	Comment
				Requires initial firmware + config
Toshiba TC35661	No	No	tc3566	Only -007/009 models provide full HCI. See below
TI CC256x, WL183x	No	No	cc256x	Also WL185x, WL187x, and WL189x

Notes:

1. BD_ADDR: Indicates if Bluetooth chipset comes with its own valid MAC Address. 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 in general. BTstack can receive SCO packets without problems. However, only TI CC256x has support for using SCO buffers in the Controller and a useful flow control. On CSR/Broadcom/Cypress Controllers, BTstack cannot queue multiple SCO packets in the Controller. Instead, the SCO packet must be sent periodically at the right time - without a clear indication about when this time is. The current implementation observes the timestamps of the received SCO packets to schedule sending packets. With full control over the system and no other Bluetooth data, this can be flawless, but it's rather fragile in general. For these, it's necessary to use the I2S/PCM interface for stable operation. , 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.

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. The BLE Controller is qualified as QDID 99659. Please note: the BLE firmware is around 60 kB. It might need a separate Wifi firmware as well.

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.

Broadcom/Cypress/Infineon Semiconductor

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 CYW20704 A2 controller supports both DLE as well as multiple LE roles and is available e.g. from LairdTech as UART module (BT860), USB module (BT850), and USB dongle.

Interestingly, the CYW20704 exhibits the same UART flow control bug as the CC2564. You can add ENABLE_CYPRESS_BAUDRATE_CHANGE_FLOWCONTROL_BUG_WORKAROUND to activate a workaround and/or read the bug & workardound description in the TI section below.

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, but with the takeover by Cypress, documentation is directly available.

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

The PSoC 4 SoCs can be programmed with the “BLE DTM” HCI Firmware from the PSoC Creator Kit Examples. The UART baudrate is set to 115200. For higher baud rates, the clocks probably need to be configured differently, as the IDE gives a warning about this.

The CYW20819 can be used as a SoC with Cypress’ Bluetooth stack. To use it as a regular Bluetooth Controller over HCI H4, CTS must be asserted during Power-Up / Reset.

The CYW43xxx series contains a Wifi and Bluetooth Controller. The Bluetooth Controller can be used independent from the Wifi part.

Newer Controller likes the CYW5557x series requires to enter a so-called autobaud mode by asserting CTS (low) during reset/power-up. In this mode, only a subset of HCI commands are available. Please see posix-h4-bcm port to get started.

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 only can send audio correctly over UART with newer Controllers that support SCO Flow Control. HSP and HFP Narrow Band Speech is supported via I2C/PCM pins. Newer Controllers provide an mSBC codec that allows to use HSP/HFP incl. WBS over PCM/I2S with ENABLE_BCM_PCM_WBS.

CSR / Qualcomm Incorporated

CSR plc has been acquired by Qualcomm Incorporated in August 2015.

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.

Dialog Semiconductor / Renesas

Daialo Semiconductor has been aquired by Renesas in February 2021.

They offers the DA145xx and DA1469xx series of LE-only SoCs 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.

The 581 does not implement the Data Length Extension or supports multiple concurrent roles, while later versions support it.

The mechanism to boot a firmware via UART/SPI has mostly stayed the same, while the set of supported interfaces and baudrates have slightly changed.

The DA1469x uses an external flash. The DA 1469x SDK contains a HCI firmware that can be compiled and downloaded into flash using the SmartSnippets Studio.

Unexpected issues: - DA14585 cannot scan for other devices if advertising is enabled alhtough it supports multiple Peripheral/Central roles (last check: 6.0.14.1114)

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.

BTstack integration: *btstack_chipset_da145xx.c* contains the code to download the provided HCI firmware into the SRAM of the DA145xx. After that, it can be used as any other HCI chipset. No special support needed for DA1469x after compiling and flashing the HCI firmware.

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. Since ESP-IDF v4.3, SCO-over-HCI is usable for HSP/HFP.

The newer ESP32-S3 and ESP32-C3 SoCs have a newer LE Controller that also supports 2M-PHY, but does support Classic (BR/EDR) anymore.

ALl can either be used as an SoC with the application running on the ESP32 itself or can be configured as a regular Bluetooth HCI Controller. BTstack can work either on the SoC itself or on another MCU with the ESP32 connected via 4-wire UART.

See Espressif's ESP-Hosted firmware for use as Bluetooth/Wifi Ccontroller.

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-l073rz-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 for EM9301 since it does not have a stored fix address. The EM9304 comes with an valid address stored in OTP.

SCO data is not supported since it is LE only.

Baud rate can 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 (EM9304 only).

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.

Intel Dual Wireless 8260, 8265

Wifi/Bluetooth combo cards mainly used in mobile computers. The Bluetooth part requires the upload of a firmware file and a configuration file. SCO, DLE, Multiple roles not tested.

Nordic nRF5 series

The Single-Mode LE chipsets from the Nordic nRF5 series chipsets usually 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. 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 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: Support for a nRF5 chipset with the Zephyr Controller is provided by *btstack_chipset_zephyr.c*. It queries the static random address during init.

To use these chipsets with BTstack, you need to flash the HCI UART example from NCS or Zephyr onto the chipset. Please see the instructions in chipset/zephyr/nrf5340_dongle for nRF5340 DK and ADK.

NXP Semiconductors

NXP Semiconductors acquired the Bluetooth + Wifi division of Marvel in 2019 and continues their products with new names. As the Controllers contain no Bluetooth firmware, the firmware needs to be uploaded on start. BTstack supports firmware upload for older Controllers with bootloader version v1, like the NXP 88W8997.

BD ADDR is stored in Controller.

SCO data is routed over HCI by default but does not support flow control.

Baud rate is currently kept at 115200

Init script is required.

BTstack integration: firmware update required and implemented by *btstack_chipset_nxp.c* See port/posix-h4-nxp for details on how to use it.

Realtek

Realtek provides Dual-Mode Bluetooth Controllers with USB and UART (H4/H5) interfaces as well as combined Bluetooth/WiFi Controllers, which are also available as M.2 modules for laptops. They commonly require to download a patch and a configuration file. Patch and configuration file can be found as part of their Linux drivers.

BD ADDR is stored in Controller.

SCO data can either be routed over HCI with working flow control or over I2S/PCM. The 8822CS supports mSBC codec internally.

Baud rate is set by the config file.

Init script is required.

BTstack integration: H4/H5 Controller require firmware upload. 'rtk_attach' can be used for this on Embedded Linux. For H4 and USB Controllers, *btstack_chipset_realtek.c* implements the patch and config upload mechanism. See port/libusb for details on how to use it.

Renesas Electronics

Please see Dialog Semiconductor for DA14xxx Bluetooth SoCs above.

Renesas currently has 3 LE-only SoCs: the older 16-bit RL78 and the newer RX23W and the RA4W1. For the newer SoCs, Renesas provides a pre-compiled HCI firmware as well as an HCI project for their e2 Studio IDE. The HCI firmware needs to be programmed into the SoC Flash.

Both newer SoC provide the newer Bluetooth 5.0 features like DLE, 2M-PHY, Long Range, and Multiple Roles.

To install the HCI Firmware on the Target Board for RX23W:

- Download Bluetooth Test Tool Suite
- Install Renesas Flash Programmer
- Follow instruction in Target Board for RX23W Quick Start Guide to flash `rx23w_uart_hci_sci8_br2000k_v1.00.mot`

BD Addr fixed to 74:90:50:FF:FF:FF. A Windows tool in the BTTS suite allows to set a public Bluetooth Address.

Baud rate: The baud rate is fixed at 115200 resp. 2000000 with the provided firmware images. With 2 mbps, there's no need to update the baudrate at run-time.

BTstack integration: No special support needed.

STMicroelectronics

STMicroelectronics has several different Bluetooth series.

STLC2500D

It 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.

BlueNRG

The BlueNRG series is an LE-only SoC which can be used with an HCI Firmware over a custom SPI interface.

STM32-WB

The STM32-WB5x series microcontroller is an SoC with a multi-protocol 2.4 Ghz radio co-processor. It provides a virtual HCI interface.

STM32-WB0

The STM32-WB50 series microcontroller is an SoC with a multi-protocol 2.4 Ghz radio co-processor. The STM32Cube MCU Package for STM32WB0 series provides the HCI firmware.

To flash this firmware, you can: - Download and Install STM32CubeIDE - Download and Unzip the STM32WB0 MCU Package - Open STM32CubeIDE - Import projects`Projects/{Your DevKit}/Applications/BLE/BLE_TransparentMode/STM32CubeIDE` and make sure the option 'copy into workspace' is unchecked - In `ble_conf.h`, add the line: - `#define BLESTACK_CONTROLLER_ONLY 1` - Replace the controller+stack library with

the controller only library: - Go to Properties->C/C++ Build->Settings->Tool Settings->MCU GCC Linker->Libraries in STM32CubeIDE - Replace `:stm32wb0x_ble_stack.a` with `:stm32wb0x_ble_stack_controller_only.a` - Connect your Nucleo board and press the “Run” button

The HCI firmware is configured for 921600 baud and no hardware flow control. You can modify these settings in `MX_USART1_UART_Init()` in `main.c`. Use `UART_HWCONTROL_RTS_CTS` to enable hardware flowcontrol.

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. Some of the latter support ANT as well.

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.

TI provides an alternative firmware that integrates an SBC Codec in the Bluetooth Controller itself for Assisted A2DP (A3DP) and Assisted HFP (Wide-band speech support). While this can save computation and code size on the main host, it cannot be used together with BLE, making it useless in most projects.

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, CC2567	0x1B0F	6.6.15
CC256xB	0x1B90	6.7.16

Chipset	LMP Subversion	AKA
CC256xC	0x9a1a	6.12.26
WL18xx	0xac20	11.8.32

SCO data: Routing of SCO data 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 `ENABLE_CC256X_BAUDRATE_CHANGE_FLOWCONTROL_BUG_WORKAROUND`. 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 is be routed over HCI with `ENABLE_SCO_OVER_HCI` or to PCM/I2S with `ENABLE_SCO_OVER_PCM`. Wide-band speech is supported in both cases. For SCO-over-HCI, BTstack implements the mSBC Codec. For SCO-over-I2S, Assisted HFP can be used. - Assisted HFP: BTstack provides support for Assisted HFP mode if enabled with `ENABLE_CC256X_ASSISTED_HFP` and SCO is routed over PCM/I2S with `ENABLE_SCO_OVER_PCM`. During startup, the PCM/I2S is configured and the HFP implementation will enable/disable the mSBC Codec for Wide-band-speech when needed.

Known issues: - The CC2564C v1.5 may loose the connection in Peripheral role with a Slave Latency > 0 when the Central updates Connection Parameters. See https://github.com/bluekitchen/btstack/issues/429

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 resp TC35561-009 for the newer silicon.

We first tried their USB Evaluation Stick that contains an USB-to-UART adapter and the PAN1026 module that contains the TC35661 -501. While it does support the HCI interface and Bluetooth Classic operations worked as expected, LE HCI Commands are not supported. With the -007 and the -009 models, everything works as expected.

SCO data does not seem to be supported.

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.

#Porting to Other Platforms

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

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.

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.

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);
```

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_in-stance* function returns a pointer to a control struct suitable for the CC256x chipset.

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).

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.

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);
```

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.

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.

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 below.

```
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;
```

#Existing Ports

Existing ports

- BTstack Port for Ambiq Apollo2 with EM9304
- Archive of earlier ports
- BTstack Port for the Espressif ESP32 Platform
- BTstack Port for FreeBSD Systems
- BTstack Port for POSIX Systems with libusb Library
- BTstack Port for POSIX Systems with Intel Wireless 8260/8265 Controllers
- BTstack Port for Linux Systems
- BTstack Port for the Maxim MAX32630FTHR ARM Cortex-M4F
- BTstack Port for MSP432P401 Launchpad with CC256x

- BTstack Port with Cinnamon for Nordic nRF5 Series
- BTstack Port for POSIX Systems with H4 Bluetooth Controller
- BTstack Port for POSIX Systems with modern Infineon (CYW) H4 Bluetooth Controller
- BTstack Port for POSIX Systems with Atmel ATWILC3000 Controller
- BTstack Port for POSIX Systems with Dialog Semiconductor DA14531 Controller
- BTstack Port for POSIX Systems with Dialog Semiconductor DA14581 Controller
- BTstack Port for POSIX Systems with Dialog Semiconductor DA14585 Controller
- BTstack Port for POSIX Systems with NXP/Marvel H4 Bluetooth Controller
- BTstack Port for POSIX Systems with Zephyr-based Controller
- BTstack Port for QT with H4 Bluetooth Controller
- BTstack Port for QT with USB Bluetooth Dongle
- BTstack Port for Raspberry Pi 3 with BCM4343 Bluetooth/Wifi Controller
- BTstack Port for Renesas Eval Kit EK-RA6M4 with DA14531
- BTstack Port for Renesas Target Board TB-S1JA with CC256x
- BTstack Port for Ezurio Vela IF820 Series
- BTstack Port for SAMV71 Ultra Xplained with ATWILC3000 SHIELD
- BTstack Port for STM32 F4 Discovery Board with CC256x
- BTstack Port for STM32 F4 Discovery Board with USB Bluetooth Controller
- BTstack Port for STM32 Nucleo L073RZ Board with EM9304 Controller
- BTstack Port with Cinnamon for Semtech SX1280 Controller on Miromico FMLR-80
- BTstack Port with Cinnamon for Semtech SX1280 Controller on STM32L476 Nucleo
- BTstack Port for STM32WB55 Nucleo Boards using FreeRTOS
- BTstack on Web using WebSerial
- BTstack Port for WICED platform
- BTstack Port for Windows Systems with Bluetooth Controller connected via Serial Port
- BTstack Port for Windows Systems with DA14585 Controller connected via Serial Port
- BTstack Port for Windows Systems with Zephyr-based Controller
- BTstack Port for Windows Systems using the WinUSB Driver

- BTstack Port for Windows Systems with Intel Wireless 8260/8265 Controllers
- BTstack Port for Zephyr RTOS

BTstack Port for Ambiq Apollo2 with EM9304

This port uses the Ambiq Apollo2 EVB and the Ambiq EM9304 (AM BLE) shield. HAL and BSP from Ambiq Suite 1.2.11 were used together with the regular ARM GCC toolchain. Firmware upload is possible via the internal J-Link interface or the 10-pin Mini ARM-JTAG Interface.

Hardware

Ambiq Apollo2 EVB + AM_BLE Shield - http://ambiqmicro.com/apollo-ultra-low-power-mcus/apollo2-mcu/

Software

AmbiqSuite: - http://ambiqmicro.com/apollo-ultra-low-power-mcus/apollo2-mcu/

Please clone BTstack as AmbiqSuite/third-party/bstack folder into the AmbiqSuite.

Create Example Projects

To create example GCC projects, go to the Apollo2-EM9304 folder

```
$ cd port/apollo2-em9304
```

and run make

```
$ ./create_examples.py
```

All examples are placed in the boards/apollo2_evb_am_ble/examples folder with btstack_ prefix.

Compile & Run Example Project

Go to to the gcc folder of one of the example folders and run make

```
$ make
```

To upload, please follow the instructions in the Apollo Getting Started documents.

Debug output

printf is routed over the USB connector of the EVB at 115200.

In port/apollo2-em9304/btstack_config.h additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/btstack_port.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

TODO

- BTstack's TLV persisten storage via Flash memory is not implemented yet.
- SPI Fullduplex: Newer Apollo 2 revisions supports SPI Full Duplex. The Ambiq Suite 1.2.11 does not cover Full Duplex with IRQ callback. It could be emulated by setting the Full Duplex mode and doing a regular write operation. When the write is complete, the received data can be read from the IOM FIFO.
- During MCU sleep without an ongoing SPI operation, the SPI could be fully disabled, which would reduce enrgey consumption.

Archive of earlier ports

The ports in this folder are not recommended for new designs. This does not mean that the target hardware is not suitable for new designs, just that the ports would need to reworked/refreshed.

List of ports with reason for move to this archive:

- MSP430 ports: The ports used the not-maintained version of the community MSP430 gcc port, which does not support more than 64 kB of FLASH RAM. A current port should use the official GCC version sponsored by TI. In addition, the MSP430 UART does not

support hardware RTS/CTS. For a new port, our ringbuffer approach should be used. Individual ports:
    - EZ430-RF256x Bluetooth Evaluation Tool for MSP430
    - MSP430F5438 Experimenter Board for MSP430 with Bluetooth CC2564 Module Evaluation Board
    - MSP-EXP430F5529LP LaunchPad with Bluetooth CC2564 Module Evaluation Board and EM Adapter BoosterPack with additional 32768Hz quartz oscillator
- Ports for Broadcom/Cypress Controllers using HCI Transport H5: The PatchRAM cannot be uploaded via H5. This requires to upload the PatchRAM using H4 mode and then to start the stack using H5 transport. In addition, several bugs have been observed in H5 mode, e.g. LE Encrypt not wokring. Unless absolutely neccessary, it's better to use H4 mode.
    - Unix-based system connected to Broadcom/Cypress Bluetooth module via H5 over serial port
    - Broadcom platforms that support the WICED SDK via H5 UART, see wiced-h4
- Port for MicroChip PIC32 with Harmony Framework: the original port was for Harmony v1, while there's a Harmony V3 out since 2019:
    - Microchip's PIC32 Bluetooth Audio Development Kit
- iOS port:
    - BTstack on iOS is not supported anymore.

BTstack Port for the Espressif ESP32 Platform

Setup

- Follow Espressif IoT Development Framework (ESP-IDF) setup to install XTensa toolchain and the ESP-IDF.
- Currently used for testing: ESP-IDF v5.4

Usage

Please note: Since BTstack v1.7, BTstack sources are not and should not be copied into `${ESP_IDF}/components` folder. Please remove `${ESP_IDF}/components/btstack`

The current build system allows to build all provided examples for all supported ESP32 versions. By default, the 'gatt_counter' example will be build for the original ESP32.

To select example 'gatt_streamer_server' and the ESP32-C3:

EXAMPLE='gatt_streamer_server' idf.py set-target esp32c3

To get a list of valid targets:

idf.py set-target

To get a list of BTstack examples:

```
idf.py list-examples
```

To change the example:

```
idf.py fullclean
```

To compile the example, run:

```
idf.py build
```

To upload the binary to your device, run:

```
idf.py -p PATH-TO-DEVICE flash
```

To get debug output, run:

```
idf.py monitor
```

You can quit the monitor with CTRL-].

Configuration

The configuration file 'btstack_config.h' is provided by the main project via the `btstack_config` component in `components/btstack_config/btstack_config.h`

Integration into custom projects

The esp32 port (this folder) contains the components ‘btstack’ and ‘btstack_config’. It can be compiled out-of-tree by setting the environment variable BTSTACK_ROOT, example:

```
BTSTACK_ROOT=/path/to/btstack EXAMPLE='gatt_streamer_server'
idf.py build
```

Limitations

Issues with the Bluetooth Controller Implementation

There are different issues in the Bluetooth Controller of the ESP32 that is provided in binary. We’ve submitted appropriate issues on the GitHub Issues page here: https://github.com/espressif/esp-idf/issues/created_by/mringwal

Audio playback

Audio playback is implemented by `btstack_audio_esp32_v4.c` resp. `btstack_audio_esp32_v5.c` and supports generic I2S codecs as well as the ES8388 on the ESP32 LyraT v4.3 devkit.

Due to the various ESP32 variants, please double check the I2S Configuration in the mentioned audio file and verify that the correct menuconfig options are set.

For a generic ESP32, the first I2S interface is used with with the following pin out:

ESP32 pin	I2S Pin
GPIO0	MCK
GPIO5	BCLK
GPIO25	LRCK
GPIO26	DOUT
GPIO35	DIN

If support for the LyraT v4.3 is enabled via menuconfig - Example Board Configuration –> ESP32 board –> ESP32-LyraT V4.3, CONFIG_ESP_LYRAT_V4_3_BOARD gets defined and the ES8388 will be configured as well.

We've also used the MAX98357A on the Adafruit breakout board. The simplest audio example is the `mod_player`, which plays back an 8 kB sound file and the a2dp_sink_demo that implements a basic Bluetooth loudspeaker. You can test the audio input with the `audio_duplex` example.

ESP32 printf/log

The BTstack port setups a buffered output for printf/ESP log macros. However, if that gets full, the main thread will get blocked. If you're playing audio, e.g. a2dp_sink_demo, and a lot of text is sent over the UART, this will cause audio glitches as the I2S DMA driver isn't re-filled on time. If this happens, you can increase the UART baudrate, reduce the debug output or increase the output buffer size - or any combination of these. :)

Multi-Threading

BTstack is not thread-safe, but you're using a multi-threading OS. Any function that is called from BTstack, e.g. packet handlers, can directly call into BTstack without issues. For other situations, you need to provide some general 'do BTstack tasks' function and trigger BTstack to execute it on its own thread.

To call a function from the BTstack thread, you can use *btstack_run_loop_execute_on_main_thread* allows to directly schedule a function callback, i.e. 'do BTstack tasks' function, from the BTstack thread. The called function then checks for pending BTstack tasks and executes them.

BTstack Port for FreeBSD Systems

Overview

This port assumes that FreeBSD provides an ng_hci netgraph node for a connected Bluetooth Controller. In most cases, these are Bluetooth USB dongles or built-in Bluetooth Controller connected via USB.

For Bluetooth Controllers connected via UART, the POSX-H4 port might be a better option als

Implementation details

In FreeBSD 13.2, the hci node is connected to a l2cap node and a btsock_hci_raw node. In order to take control, this port create a custom netgraph ng_socket node and connect to the ‘acl’ and ‘raw’ hooks of the hci node. The OS Bluetooth functionality will be interrupted.

Compilation

BTstack’s FeeeBSD port does not have additional dependencies. To compile the cmake project with Make

```
mkdir build
cd build
cmake ..
make
```

or using Ninja:

```
mkdir build
cd build
cmake -G Ninja ..
ninja
```

Running the examples

As the port needs to reconfigure the Bluetooth netgraph node, it needs to run with root privileges. It tries to connect to ‘ubt0hci’ by default. If your Bluetooth Controller is different, you can select it with ‘-u node’ On start, BTstack prints the path to the packet log and prints the information on the detected Buetooth Controller.

```
$ sudo gatt_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
BTstack up and running on 00:1A:7D:DA:71:13.
```

ToDO

- drop privileges after startup
- auto-detect ng_hci node
- support for profiles that require SCO: HFP & HSP

BTstack Port for POSIX Systems with libusb Library

Compilation

The quickest way to try BTstack is on a Linux or OS X system with an additional USB Bluetooth dongle. It requires pkg-config and libusb-1.0 or higher to be installed.

On a recent Debian-based system, all you need is:

```
sudo apt-get install gcc git cmake ninja-build pkg-config
libusb-1.0 portaudio19-dev
```

When everything is ready, you compile all examples with make:

```
make
```

or using CMake + Ninja:

```
mkdir build
cd build
```

```
cmake -G Ninja ..
ninja
```

Environment Setup

Linux

On Linux, the USB Bluetooth dongle is usually not accessible to a regular user.

You can add a udev rule for your dongle to extend access rights to user processes.

For this, create `/etc/udev/rules.d/btstack.rules` and add this

```
# Match all devices from CSR
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a12", MODE="0666"

# Match all devices from Realtek
SUBSYSTEM=="usb", ATTRS{idVendor}=="0bda", MODE="0666"

# Match Cypress Semiconductor / Broadcom BCM20702A, e.g. DeLOCK
Bluetooth 4.0 dongle
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a5c", ATTRS{idProduct}
=="21e8", MODE="0666"

# Match Asus BT400
SUBSYSTEM=="usb", ATTRS{idVendor}=="0b05", ATTRS{idProduct}
=="17cb", MODE="0666"

# Match Laird BT860 / Cypress Semiconductor CYW20704A2
SUBSYSTEM=="usb", ATTRS{idVendor}=="04b4", ATTRS{idProduct}
=="f901", MODE="0666"
```

macOS

On macOS, the OS will try to use a plugged-in Bluetooth Controller if one is available. It's best to to tell the OS to always use the internal Bluetooth Contoller.

For this, execute:

```
sudo nvram bluetoothHostControllerSwitchBehavior=never
```

and then reboot to activate the change.

Note: if you get this error,

```
libusb: warning [darwin_open] USBDeviceOpen: another process has
device opened for exclusive access
libusb: error [darwin_reset_device] ResetDevice: device not opened
for exclusive access
```

and you didn't start another instance and you didn't assign the USB Controller to a virtual machine, macOS uses the plugged-in Bluetooth Controller. Please configure NVRAM as explained and try again after a reboot.

Broadcom/Cypress/Infineon Controllers

During startup BTstack queries the Controlle for the Local Name, which is set to the Controller type (e.g. 'BCM20702A). The chipset support uses this information to look for a local PatchRAM file of that name and uploads it.

Realtek Controllers

During startup, the libusb HCI transport implementations reports the USB Vendor/Product ID, which is then forwarded to the Realtek chipset support. The chipset support contains a mapping between USB Product ID and (Patch, Configuration) files. If found, these are uploaded.

Running the examples

BTstack's HCI USB transport will try to find a suitable Bluetooth module and use it.

On start, BTstack will try to find a suitable Bluetooth module. It will also print the path to the packet log as well as the USB path.

```
$ ./le_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
Packet Log: /tmp/hci_dump_6.pklg
```

```
USB device 0x0a12/0x0001, path: 06
Local version information:
- HCI Version    0x0006
- HCI Revision   0x22bb
- LMP Version    0x0006
- LMP Subversion 0x22bb
- Manufacturer 0x000a
BTstack up and running on 00:1A:7D:DA:71:01.
```

If you want to run multiple examples at the same time, it helps to fix the path to the used Bluetooth module by passing -u usb-path to the executable.

Example running le_streamer and le_streamer_client in two processes, using CSR Bluetooth dongles at USB path 6 and 4:

```
./le_streamer -u 6
Specified USB Path: 06
Packet Log: /tmp/hci_dump_6.pklg
USB device 0x0a12/0x0001, path: 06
Local version information:
- HCI Version    0x0006
- HCI Revision   0x22bb
- LMP Version    0x0006
- LMP Subversion 0x22bb
- Manufacturer 0x000a
BTstack up and running on 00:1A:7D:DA:71:01.
To start the streaming, please run the le_streamer_client example
on other device, or use some GATT Explorer, e.g. LightBlue,
BLExplr.

$ ./le_streamer_client -u 4
Specified USB Path: 04
Packet Log: /tmp/hci_dump_4.pklg
USB device 0x0a12/0x0001, path: 04
Local version information:
- HCI Version    0x0006
- HCI Revision   0x22bb
- LMP Version    0x0006
- LMP Subversion 0x22bb
- Manufacturer 0x000a
```

```
BTstack up and running on 00:1A:7D:DA:71:02.
Start scanning!
```

BTstack Port for POSIX Systems with Intel Wireless 8260/8265 Controllers

Same as port/libusb, but customized for Intel Wireless 8260 and 8265 Controllers. These controller require firmware upload and configuration to work. Firmware and config is downloaded from the Linux firmware repository.

Compilation

Requirements: - pkg-config - libusb-1.0

On a recent Debian-based system, all you need is:

```
apt-get install gcc git libusb-1.0 pkg-config
```

When everything is ready, you compile all examples with:

```
make
```

Environment

On Linux, the USB Bluetooth dongle is usually not accessible to a regular user. You can either: - run the examples as root - add a udev rule for your dongle to extend access rights to user processes

To add an udev rule, please create `/etc/udev/rules.d/btstack.rules` and add this

```
# Match all devices from CSR
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a12", MODE="0666"

# Match DeLOCK Bluetooth 4.0 dongle
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a5c", ATTRS{device}=="21e8",
MODE="0666"

# Match Asus BT400
```

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="0b05", ATTRS{device}=="17cb",
MODE="0666"

# Match Laird BT860 / Cypress Semiconductor CYW20704A2
SUBSYSTEM=="usb", ATTRS{idVendor}=="04b4", ATTRS{device}=="f901",
MODE="0666"

# Match Intel Wireless 8260 8265
SUBSYSTEM=="usb", ATTRS{idVendor}=="8027", ATTRS{device}=="0a2b,
MODE="0666"
```

On macOS, the OS will try to use a plugged-in Bluetooth Controller if one is available. It's best to to tell the OS to always use the internal Bluetooth Contoller.

For this, execute:

```
sudo nvram bluetoothHostControllerSwitchBehavior=never
```

and then reboot to activate the change.

Running the examples

BTstack's HCI USB transport will try to find a suitable Bluetooth module and use it.

On start, BTstack will try to find a suitable Bluetooth module. It will also print the path to the packet log as well as the USB path.

```
$ ./le_counter
Packet Log: /tmp/hci_dump.pklg
USB Path: 03-01-04-03
Firwmare ./ibt-12-16.sfi
Firmware upload complete
Firmware operational
Done 1
BTstack counter 0001
USB Path: 03-01-04-03
```

BTstack up and running on F8:34:41:D5:BE:6F.

If you want to run multiple examples at the same time, it helps to fix the path to the used Bluetooth module by passing -u usb-path to the executable.

Example running le_streamer and le_streamer_client in two processes, using Bluetooth dongles at USB path 6 and 4:

```
./le_streamer -u 6
Specified USB Path: 06
Packet Log: /tmp/hci_dump_6.pklg
USB Path: 06
BTstack up and running on 00:1A:7D:DA:71:13.
To start the streaming, please run the le_streamer_client example
on other device, or use some GATT Explorer, e.g. LightBlue,
BLExplr.

$ ./le_streamer_client -u 4
Specified USB Path: 04
Packet Log: /tmp/hci_dump_4.pklg
USB Path: 04
BTstack up and running on 00:1A:7D:DA:71:13.
Start scanning!
```

BTstack Port for Linux Systems

While BTstack can directly work on Linux with most Bluetooth Controllers that are connected via UART (port/posix-h4) or USB (port/libusb), it might be convenient to use the Linux Bluetooth Subsystem in some cases, e.g. if the Bluetooth Controller uses other transports (such as SDIO) or the Controller is already fully configured by the distributions.

Compilation

In addition to regular C build tools, you also need the Bluetooth development package installed.

```
sudo apt install libbluetooth-dev
```

Now you can compile it as usual with CMake

```
mkdir build
cd build
cmake ..
make
```

Running the examples

Please make sure that BlueZ is not installed or at least disabled

```
sudo systemctl stop bluetooth
sudo systemctl disable bluetooth
sudo systemctl mask bluetooth
```

Also make sure that the chosen device (here, hci0) is down

```
sudo hciconfig hci0 down
```

To check

```
hciconfig hci0


hci0:   Type: Primary  Bus: USB
        BD Address: 00:1A:7D:DA:71:13  ACL MTU: 1021:8  SCO MTU:
64:1
        DOWN
        RX bytes:566359 acl:0 sco:0 events:40 errors:0
        TX bytes:2174059 acl:2694 sco:0 commands:329 errors:0
```

To access the Bluetooth Controller, you can either run the examples as root, or, set the necessary permissions for a compiled example, e.g.

```
sudo setcap 'cap_net_raw,cap_net_admin+eip' gatt_counter
```

Now, you can run this example as a regular user

```
$ ./gatt_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
BTstack up and running on 00:1A:7D:DA:71:13.
```

Status

When running gatt_counter, a basic LE Peripheral with a GATT Service, the first two connections fail, while the third and later work as expected. There's no difference when looking at the HCI Trace with btmon, just that no ACL packets are received in the failing attempts.

BTstack Port for the Maxim MAX32630FTHR ARM Cortex-M4F

This port uses the MAX32630FTHR ARM Cortex M4F Board with the onboard TI CC2564B Bluetooth controller. It usually comes with the DAPLINK Programming Adapter. The DAPLINK allows to upload firmware via a virtual mass storage device (like mbed), provides a virtual COM port for a console, and enables debugging via the SWD interface via OpenOCD.

The port uses non-blocking polling UART communication with hardware flow control for Bluetooth controller. It was tested and achieved up to 1.8 Mbps bandwidth between two Max32630FTHR boards.

Prerequisites

Maxim SDK

The Maxim has dropped the support for MAX32630 and is not recommended for new designs and it is also dropped from the latest SDK.

Maxim SDK Mirror

The required minimum set of files from the SDK are included in the port files port/max32630-fthr/maxim for convenience and to support linux build.

> **(For reference) ARM Cortex Toolchain - a.k.a. the old Maxim SDK**
>
> The ARM Cortex Toolchain is free software that provides peripheral libraries, linker files, initial code and some board files. It also provides Eclipse Neon and Maxim modified OpenOCD to program the microcontroller together with various examples for Maxim Cortex M4F ARM processors.

> For debugging, OpenOCD can be used. The regular OpenOCD does not support Maxim ARM microcontrollers yet, but a modified OpenOCD for use with Maxim devices can be found in the Maxim ARM Toolchain.

> Toolchain and Eclipse guide can be found in README.pdf file where the Maxim Toolchain is installed. Please note that this port was done using Makefiles.

ARM Toolchain Setup

Download and extract arm toolchain. Record extract folder as `TOOLCHAIN_PATH`.

Windows msys: arm-gnu-toolchain-14.2.rel1-mingw-w64-x86_64-arm-none-eabi.zip

WSL Linux: arm-gnu-toolchain-14.2.rel1-x86_64-arm-none-eabi.tar.xz

Python - 3.12 or newer

Download and install python and make sure the alias `python` is accessible in the build terminal.

MSYS2 setup - (windows only)

The ARM Cortex Toolchain contains msys1.0, but it is complete (lacking unzip) and it is recommended that to use a more recent msys2.

Download and install msys2 - https://github.com/msys2/msys2-installer/releases/ - Install with option to Run msys2 after completion. - Record install folder as `MSYS_PATH` - Install required packages make and unzip. `pacman -S make unzip` - close msys2

Environment Variables

The `BTSTACK_ROOT`, `MAXIM_PATH`, and `PATH` must point to the previously setup tools as absolute paths.

The `BTSTACK_ROOT` is provided as relative path for examples in `port/max32630-fthr/example/project/Makefile`, but must be provided otherwise.

If not provided the `MAXIM_PATH` is set based on `BTSTACK_ROOT`.

Windows msys:

Run scripts/setenv.bat or set manually

```
<a name="sec:max32630-fthrPort"></a>

## TOOLCHAIN_PATH=%EXTRACT_PATH%
<a name="sec:max32630-fthrPort"></a>

## MSYS_BIN=%MSYS_PATH%\usr\bin
set PATH=%TOOLCHAIN_PATH%/bin;%MSYS_BIN%;%PATH%
```

WSL Linux:

```
PATH="${TOOLCHAIN_PATH}/bin:${PATH}"
```

Build

The examples can be compiled using `make`. The build is implemented as a combination of several make and python scripts. For details see the scripts `port/max32630-fthr`, `port/max323630-fthr/scripts`, and `port/max32630-fthr/example/template` folders.

Running the make script `port/max32630-fthr/Makefile` will create Makefiles for the projects. And the projects are built running make scripts `port/max32630-fthr/example/Makefile` or `port/max32630-fthr/example/project/Makefile`.

```
cd %BTSTACK_ROOT%/port/max32630-fthr
.../port/max32630-fthr$ make
> Creating example projects:
> project - a2dp_sink_demo
```

```
> project - a2dp_source_demo
> project - ancs_client_demo
> ...
.../port/max32630-fthr$ cd example
<a name="sec:max32630-fthrPort"></a>

## build a make target default(all), dual, general, classic, or
        ble
.../port/max32630-fthr/example$ make
<a name="sec:max32630-fthrPort"></a>

## or build individual project e.g. gatt_counter
.../port/max32630-fthr/example$ cd gatt_counter
.../port/max32630-fthr/example/gatt_counter$ make
```

In each example/project folder the make target is to build .elf file in the build folder which is convenient for debugging using Eclipse or GDB.

For flashing via the virtual USB drive, the make also builds .bin file relative to the project ../bin/project.bin e.g. which is for the examples `example/bin`.

Flashing Max32630 ARM Processor

A firmware binary can be flashed either by copying the .bin file to the DAPLINK mass storage drive, or by using OpenOCD on the command line, or from Eclipse CDT.

The simplest way is drag and drop the generated .bin file to the DAPLINK mass storage drive. Once the file is copied to the mass storage device, the DAPLINK should program and then run the new firmware.

Alternatively, OpenOCD can be used to flash and debug the device. A suitable programming script can be found in the scripts folder.

Debugging

OpenOCD can also be used for developing and especially for debugging. Eclipse or GDB via OpenOCD could be used for step by step debugging.

Debug output

printf messages are redirected to UART2. UART2 is accessible via the DAPLINK Programming Adapter as a virtual COM port at 115200 baud with no flow control. If this doesn't work for you, you can connect P3_1 (UART TX) of the MAX32630FTHR board to a USB-to-UART adapter.

Additional debug information can be enabled by uncommenting ENABLE_LOG_INFO in the src/btstack_config.h header file and a clean rebuild.

TODOs

- Support for BTSTACK_STDIN
- Add flash-openocd to Makefile template
- Add Eclipse CDT projects for max32630fthr
- Implement hal_led.h to control LED on board

BTstack Port for MSP432P401 Launchpad with CC256x

This port is for the the TI MSP432P401R Launchpad with TI's CC256x Bluetooth Controller using TI's DriverLib (without RTOS). For easy development, Ozone project files are generated as well.

As the MSP432P401 does not have support for hardware RTS/CTS, this port makes use of Ping Pong DMA transfer mode (similar to circular DMA on other MCUs) to use two adjacent receive buffers and raise RTS until a completed buffer is processed.

Hardware

TI MSP432P401R LaunchPad

As Bluetooth Controller, there are two BoosterPacks that can be use: 1. BOOST-CC2564MODA CC2564B BoosterPack (USD 20) 2. Evaluation Module (EM) Adapto (USD 20) with one of the CC256x modules: -

CC2564B Dual-mode Bluetooth® Controller Evaluation Module (USD 20) - CC2564C Dual-mode Bluetooth® Controller Evaluation Module (USD 60)

The CC2564B Booster pack is around USD 20 while thhe EM Adapter with the CC2564C module is around USD 80.

The project in the BTstack repo `port/msp432p401lp-cc256x' is configured for the EM Adapter + newer CC2564C module.

When using the CC2564B (either as BOOST-CC2564MODA or CC2564B Dual-mode Bluetooth® Controller Evaluation Module), the *bluetooth_init_cc2564B_1.8_BT_Spec_4.1.c* must be used as cc256x_init_script. See Makefile variable INIT_SCRIPT.

When using the CC2564B Booster Pack, please use uncomment the defines for the GPIO definition (search for `BOOST-CC2564MODA`)

When using the EM Adapter Booster Pack, please make sure to solder a 32.768 kHz quarz oscillator as explained in 4.7 of the EM Wireless Booster Pack User Guide. If you don't have an oscillator of that size, you might solder one upside done (turtle-on-back style) to the unused upper right pad and wire GCC, VCC, and clock with thin wires.

Software

To build all examples, you need the regular ARM GCC toolcahin installed. Run make

```
$ make
```

All examples and the .jdebug Ozone project files are placed in the 'gcc' folder.

Flash And Run The Examples

The Makefile builds different versions: - example.elf: .elf file with all debug information - example.bin: .bin file that can be used for flashing

There are different options to flash and debug the MSP432P401R LaunchPad. If all but the jumpers for power (the left three) are removed on J101, an external JTAG like SEGGER's J-Link can be connected via J8 'MSP432 IN'.

Run Example Project using Ozone

When using an external J-Link programmer, you can flash and debug using the cross-platform SEGGER Ozone Debugger. It is included in some J-Link programmers or can be used for free for evaluation usage.

Just start Ozone and open the .jdebug file in the build folder. When compiled with `ENABLE_SEGGER_RTT`, the debug output shows up in the Terminal window of Ozone.

Debug output

All debug output is send via SEGGER RTT or via USART2. To get the console from USART2, remove `ENABLE_SEGGER_RTT` from btstack_config.h and open a terminal to the virtual serial port of the Launchpad at 115200.

In btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be disabled/enabled via ENABLE_LOG_INFO.

Also, the full packet log can be enabled in main.c by uncommenting the hci_dump_init(..) line. The output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/ create_packet_log.py

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. The Makefile contains rules to update the .h file when the .gatt was modified.

BTstack Port with Cinnamon for Nordic nRF5 Series

Cinnamon is BlueKitchen's minimal, yet robust Controller/Link Layer implementation for use with BTstack.

In contrast to common Link Layer implementations, our focus is on a robust and compact implementation for production use, where code size matters (e.g. current code size about 8 kB).

Status

The current implementation supports a single Peripheral role, or, passive scanning in Observer role. In the Peripheral role, channel map updates, as well as connection param updates are supported.

Support for LE Central Role as well as Encryption is planned but not supported yet.

Requirements

- arm-none-eabi toolchain
- Nordic's nRF5-SDK

Supported Hardware

All nNRF5x SOCs. Built files are provided for PCA10040 (52832 DK), but others can be supported with minimal changes.

Use

- Provide path to nRF5-SDK either in `NRF5_SDK_ROOT` environment variable or directly in `pca10040/armgcc/Makefile`.
- run make
- All supported examples are built in the `build` folder.
- You can use Segger's OZONE with the provided `EXAMPLE.jdebug` project file to flash and run the examples.

BTstack Port for POSIX Systems with H4 Bluetooth Controller

Configuration

Most Bluetooth Bluetooth Controllers connected via UART/H4 require some special configuration, e.g. to set the UART baud rate, and/or require firmware patches during startup. In this port, we've tried to do most of these automatically based on information gathered from the Bluetooth Controller. Here's some Controller specific details:

TI CC256x

The CC2564x needs the correct init script to start up. The Makfile already has entries for most silicon revisions:

- CC2560: bluetooth_init_cc2564_2.14.c
- CC2564B: bluetooth_init_cc2564B_1.8_BT_Spec_4.1.c
- CC2564C: bluetooth_init_cc2564C_1.5.c

Please pick the correct one. The main.c verifies that the correct script is loaded, but the init script is linked to the executable.

Broadcom BCM/CYW 43430

The correct firmware file needs to be provided in the current working directory. The Makefile downloads the one for the BCM43430 e.g. found on later Raspberry Pi editions. Please see the separate port/raspi, too.

Compilation

BTstack's POSIX-H4 does not have additional dependencies. You can directly run make.

```
make
```

Running the examples

On start, BTstack prints the path to the packet log and prints the information on the detected Buetooth Controller.

```
$ ./le_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
BTstack up and running on 00:1A:7D:DA:71:13.
```

Please note that BTstack will increase the baudrate. Before starting again, you should reset or power-cycle the Bluetooth Controller.

BTstack Port for POSIX Systems with modern Infineon (CYW) H4 Bluetooth Controller

Configuration

Newer Infineon Airoc (tm) Controllers like the CYW5557x series accept PatchRAM upload only in a so-called 'auto-baud mode' which is entered by asserting CTS (low) and starting/resetting the controller via BT_REG_EN. This port currently only supports the CYW5557x Controllers.

Compilation

BTstack's posix-h4-bcm port does not have additional dependencies. You can directly run cmake and then your default build system. E.g. with Ninja:

```
mkdir build
cd build
cmake -G Ninja ..
ninja
```

Running the examples

On start, BTstack opens the serial port, which asserts CTS, and requests you to reset the Controller with a countdown.

```
$ ./gatt_counter
Packet Log: /tmp/hci_dump.pklg
Phase 1: Download firmware
```

```
Please reset Bluetooth Controller, e.g. via RESET button. Firmware
download starts in:
3
2
1
Firmware download started
Local version information:
- HCI Version    0x000a
- HCI Revision   0x0b73
- LMP Version    0x000a
- LMP Subversion 0x2257
- Manufacturer 0x0009
Phase 2: Main app
BTstack up and running at 55:56:0A:0A:76:93
...
```

ToDo

- select PatchRAM file based on HCI Read Local Version Information

BTstack Port for POSIX Systems with Atmel ATWILC3000 Controller

This port allows to use the ATWILC3000 connected via UART with BTstack running on a POSIX host system, see test setup below (which lacks a proper RESET button).

Compilation

$ make

The Makefile downloads the wilc3000_bt_firmware.bin firmware from the GitHub atwilc3000/firmware repo.

Usage

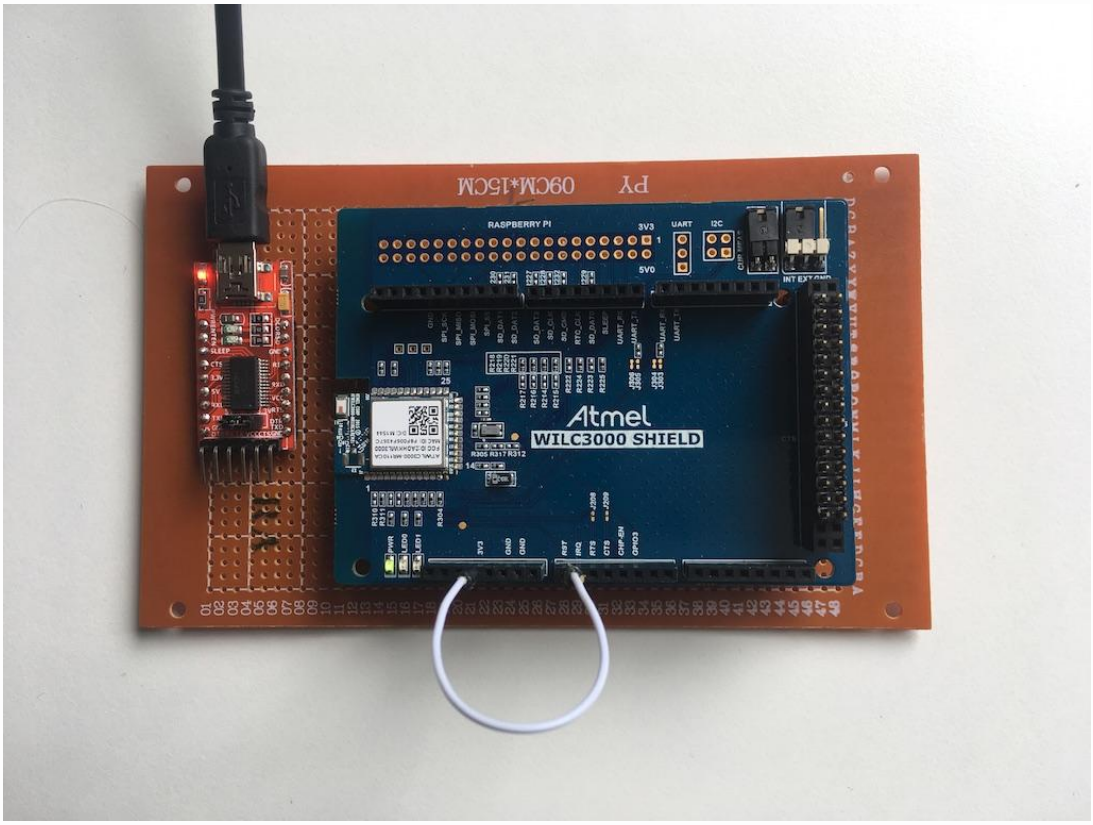
Just run any of the provided examples, e.g.

$ ./le_counter

At start, the firmware file is first uploaded to the ATWILC3000, before BTstack start up.

Please note that it configures the ATWILC3000 for a higher baud rate it does not detect if the firmware has already been downloaded, so you need to reset the ATWILC3000 before starting an example.

Tested with the official ATWILC3000 SHIELD on OS X.



BTstack Port for POSIX Systems with Dialog Semiconductor DA14531 Controller

This port allows to use the DA14531 connected via UART with BTstack running on a POSIX host system.

Instead of storing the HCI firmware in the OTP, it first downloads the hci_531_active_uart_460800.hex firmware from the 6.0.16.1144 SDK, before BTstack starts up.

After Power Cycle, please start one of the test applications and press the Reset button to trigger firmware download.

Please note that it does not detect if the firmware has already been downloaded, so you need to Power Cycle the DA14531 before starting an example again.

Alternatively, after running one of the examples once to upload the firmware, you can use the regular 'posix-h4' port and change the initial UART baud rate to 460800 as long as you don't power cycle the dev kit.

For production use, the DA14531 could be power cycled from the main CPU during startup, e.g. after the call to btstack_chipset_da145xx_download_firmware_with_uart, or, the HCI firmware could be burned into the OTP.

Software Setup / Firmware

On the DA14531 USB Development Kit, the UART is configured via DIP switched. By this, the mapping to the DA14531 GPIOs is fixed. In SDK 6.0.6.1144, the GPIO mapping of RTS and CTS is flipped. In order to be able to us the same HCI firmware on both dev kits, we've used the following configuration in `user_perip_setup.h`

```
    #define UART1_TX_PORT           GPIO_PORT_0
    #define UART1_TX_PIN            GPIO_PIN_0

    #define UART1_RX_PORT           GPIO_PORT_0
    #define UART1_RX_PIN            GPIO_PIN_1

    #define UART1_RTSN_PORT         GPIO_PORT_0
    #define UART1_RTSN_PIN          GPIO_PIN_4

    #define UART1_CTSN_PORT         GPIO_PORT_0
    #define UART1_CTSN_PIN          GPIO_PIN_3
```

We also increased the UART baudrate to 460800

```
#define UART1_BAUDRATE              UART_BAUDRATE_460800
```

We also disabled the SLEEP mode in user_config.h:

```
static const sleep_state_t app_default_sleep_mode =
        ARCH_SLEEP_OFF;
```

After compilation with Keil uVision 5, the generated .hex file is copied into btstack/chipset/da145xx as `hci_531_active_uart_460800.hex`, and then `convert_hex_files" is used to convert it into a C data array.

Hardware Setup - Dev Kit Pro

To use the DA14531 Dev Kit Pro with BTstack, please make the following modifications: - Follow Chapter 4.1 and Figure 4 in the DA14531 Development Kit Pro Hardware User Manual UM-B-114 and set SW1 of the 14531 daughter board into position "BUCK" position marked with an "H" on the left side. - configure the dev kit for Full UART (4-wire) Configuration by adding jumper wires between J1 and J2

Hardware Setup - Dev Kit USB

To use the Dev Kit USB with BTstack, please make the following modifications: - Follow Chapter 5.6 in the DA14531 USB Development Kit Hardware UM-B-125 and set the DIP switches as described.

Example Run

```
$ ./gatt_counter
Packet Log: /tmp/hci_dump.pklg
Phase 1: Download firmware
Phase 2: Main app
BTstack counter 0001
BTstack up and running on 80:EA:CA:70:00:08.
```

BTstack Port for POSIX Systems with Dialog Semiconductor DA14581 Controller

This port allows to use the DA14581 connected via UART with BTstack running on a POSIX host system.

It first downloads the hci_581_active_uart.hex firmware from the DA14581_HCI_3.110.2.12 SDK packet, before BTstack starts up.

Please note that it does not detect if the firmware has already been downloaded, so you need to reset the DA14581 before starting an example.

For production use, the HCI firmware could be flashed into the OTP and the firmware download could be skipped.

Tested with the official DA14581 Dev Kit on OS X.

BTstack Port for POSIX Systems with Dialog Semiconductor DA14585 Controller

This port allows to use the DA14585 connected via UART with BTstack running on a POSIX host system.

It first downloads the hci_581.hex firmware from the 6.0.8.509 SDK, before BTstack starts up.

Please note that it does not detect if the firmware has already been downloaded, so you need to reset the DA14585 before starting an example.

For production use, the HCI firmware could be flashed into the OTP and the firmware download could be skipped.

Tested with the official DA14585 Dev Kit Basic on OS X.

BTstack Port for POSIX Systems with NXP/ Marvel H4 Bluetooth Controller

Configuration

Most Bluetooth Bluetooth Controllers connected via UART/H4 require some special configuration, e.g. to set the UART baud rate, and/or require firmware patches during startup. In this port, we've show how a NXP/Marvell Controller can be configured for use with BTstack. It's unclear if the required firmware file for older Controllers,

e.g. 88W8997, can be detected during the firmware upload. This port selects the firmware for NXP 88W8997. For newer Controllers, e.g. IW416 or IW612, the firmware can be selected automatically.

Compilation

BTstack's posix-h4-nxp port does not have additional dependencies. You can directly run cmake and then your default build system. E.g. with Ninja:

```
mkdir build
cd build
cmake -G Ninja ..
ninja
```

Running the examples

Please reset the Controller first. On start, BTstack prints the path to the packet log.

```
$ ./gatt_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
BTstack up and running on 00:1A:7D:DA:71:13.
```

Issues

- NXP 88W8997 does not support SCO Flow Control which causes glitches when sending audio

ToDo

- increase baud rate for firmware upload
- skip firmware upload if firmware already present
- increase baud rate for application

BTstack Port for POSIX Systems with Zephyr-based Controller

The main difference to the regular posix-h4 port is that that the Zephyr Controller uses 1000000 as baud rate. In addition, the port defaults to use the fixed static address stored during production.

Prepare Zephyr Controller

Please follow this blog post about how to compile and flash `samples/bluetooth/hci_uart` to a connected nRF5 dev kit.

In short: 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, and flash an example project 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.
- In *samples/bluetooth/hci_uart*, compile the firmware for nRF52 Dev Kit

  $ make BOARD=nrf52_pca10040

- Upload the firmware

  $ make flash

- For the nRF51 Dev Kit, use `make BOARD=nrf51_pca10028`.

Configure serial port

To set the serial port of your Zephyr Controller, you can either update config.device_name in main.c or always start the examples with the `-u /path/to/serialport` option.

Compile Examples

```
$ make
```

Run example

Just run any of the created binaries, e.g.

```
$ ./le_counter
```

The packet log will be written to /tmp/hci_dump.pklg

BTstack Port for QT with H4 Bluetooth Controller

Windows is supported with the MinGW Kit.

Windows with MSVC or Embedded (bare metal) platforms not supported yet.

Configuration

Most Bluetooth Bluetooth Controllers connected via UART/H4 require some special configuration, e.g. to set the UART baud rate, and/or require firmware patches during startup. In this port, we've tried to do most of these automatically based on information gathered from the Bluetooth Controller. Here's some Controller specific details:

TI CC256x

The CC2564x needs the correct init script to start up. The Makfile already has entries for most silicon revisions:

- CC2560: bluetooth_init_cc2564_2.14.c
- CC2564B: bluetooth_init_cc2564B_1.8_BT_Spec_4.1.c
- CC2564C: bluetooth_init_cc2564C_1.5.c

Please pick the correct one. The main.c verifies that the correct script is loaded, but the init script is linked to the executable.

Broadcom BCM/CYW 43430

The correct firmware file needs to be provided in the current working directory. The Makefile downloads the one for the BCM43430 e.g. found on later Raspberry Pi editions. Please see the separate port/raspi, too.

Compilation

On all platforms, you'll need Qt Python 3 installed. On macOS/Linux libusb-1.0 or higher is required, too.

When everything is ready, you can open the provided CMakelists.txt project in Qt Creator and run any of the provided examples. See Qt documentation on how to compile on the command line or with other IDEs

You can also compile the project in a shell if you provide the path to the cmake folder of your Qt installation, e.g. on macOS with brew providing QT 5.15.16

```
cmake -DCMAKE_PREFIX_PATH=/opt/homebrew/Cellar/qt@5/5.15.16/lib/
cmake/Qt5 ..
```

Running the examples

BTstack's HCI USB transport will try to find a suitable Bluetooth module and use it.

On start, BTstack will try to find a suitable Bluetooth module. It will also print the path to the packet log as well as the USB path.

```
$ ./gattcounter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
USB Path: 06
BTstack up and running on 00:1A:7D:DA:71:13.
```

BTstack Port for QT with USB Bluetooth Dongle

Uses libusb Library on macOS and Linux and WinUSB on Windows. Windows is supported with the MinGW Kit.

Windows with MSVC or Embedded (bare metal) platforms not supported yet.

Compilation

On all platforms, you'll need Qt Python 3 installed. On macOS/Linux libusb-1.0 or higher is required, too.

When everything is ready, you can open the provided CMakelists.txt project in Qt Creator and run any of the provided examples. See Qt documentation on how to compile on the command line or with other IDEs

You can also compile the project in a shell if you provide the path to the cmake folder of your Qt installation, e.g. on macOS with brew providing QT 5.15.16

```
cmake -DCMAKE_PREFIX_PATH=/opt/homebrew/Cellar/qt@5/5.15.16/lib/
cmake/Qt5 ..
```

Environment Setup

Windows

To allow WinUSB to access an USB Bluetooth dongle, you need to install a special device driver to make it accessible to user space processes.

It works like this:

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

Linux

On Linux, the USB Bluetooth dongle is usually not accessible to a regular user. You can either: - run the examples as root - add a udev rule for your dongle to extend access rights to user processes

To add an udev rule, please create `/etc/udev/rules.d/btstack.rules` and add this

```
# Match all devices from CSR
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a12", MODE="0666"

# Match DeLOCK Bluetooth 4.0 dongle
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a5c", ATTRS{device}=="21e8",
MODE="0666"

# Match Asus BT400
SUBSYSTEM=="usb", ATTRS{idVendor}=="0b05", ATTRS{device}=="17cb",
MODE="0666"

# Match Laird BT860 / Cypress Semiconductor CYW20704A2
SUBSYSTEM=="usb", ATTRS{idVendor}=="04b4", ATTRS{device}=="f901",
MODE="0666"
```

macOS

On macOS, the OS will try to use a plugged-in Bluetooth Controller if one is available. It's best to to tell the OS to always use the internal Bluetooth Contoller.

For this, execute:

```
sudo nvram bluetoothHostControllerSwitchBehavior=never
```

and then reboot to activate the change.

Note: if you get this error,

```
libusb: warning [darwin_open] USBDeviceOpen: another process has
device opened for exclusive access
libusb: error [darwin_reset_device] ResetDevice: device not opened
for exclusive access
```

and you didn't start another instance and you didn't assign the USB Controller to a virtual machine, macOS uses the plugged-in Bluetooth Controller. Please configure NVRAM as explained and try again after a reboot.

Running the examples

BTstack's HCI USB transport will try to find a suitable Bluetooth module and use it.

On start, BTstack will try to find a suitable Bluetooth module. It will also print the path to the packet log as well as the USB path.

```
$ ./gatt_counter
Packet Log: /tmp/hci_dump.pklg
BTstack counter 0001
USB Path: 06
BTstack up and running on 00:1A:7D:DA:71:13.
```

If you want to run multiple examples at the same time, it helps to fix the path to the used Bluetooth module by passing -u usb-path to the executable.

Example running le_streamer and le_streamer_client in two processes, using Bluetooth dongles at USB path 6 and 4:

```
./gatt_streamer_server -u 6
Specified USB Path: 06
Packet Log: /tmp/hci_dump_6.pklg
USB Path: 06
BTstack up and running on 00:1A:7D:DA:71:13.
To start the streaming, please run the le_streamer_client example
on other device, or use some GATT Explorer, e.g. LightBlue,
BLExplr.

$ ./le_streamer_client -u 4
Specified USB Path: 04
Packet Log: /tmp/hci_dump_4.pklg
USB Path: 04
BTstack up and running on 00:1A:7D:DA:71:13.
Start scanning!
```

BTstack Port for Raspberry Pi 3 with BCM4343 Bluetooth/Wifi Controller

Tested with Raspberry Pi 3 Model B V1.2, Raspberry Pi 3 Model B+, and Raspberry Pi Zero W V1.1.

With minor fixes, the port should also work with older Raspberry Pi models that use the RedBear pHAT. See TODO at the end.

Raspberry Pi 3 / Zero W Setup

There are various options for setting up the Raspberry Pi, have a look at the Internet. Here's what we did:

Install Raspian Stretch Lite:

- Insert empty SD Card
- Download Raspian Stretch Lite from https://www.raspberrypi.org/downloads/raspbian/
- Install Etcher from https://etcher.io
- Run Etcher:
  - Select the image you've download before
  - Select your SD card
  - Flash!

Configure Wifi

Create the file wpa_supplicant.conf in the root folder of the SD Card with your Wifi credentials:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
network={
    ssid="YOUR_NETWORK_NAME"
    psk="YOUR_PASSWORD"
    key_mgmt=WPA-PSK
}
```

Alternatively, just plug-it in via Ethernet - unless you have a Raspberry Pi Zero W.

Enable SSH

Create an empty file called ‘ssh’ in the root folder of the SD Card to enable SSH.

Boot

If everything was setup correctly, it should now boot up and join your Wifi network. You can reach it via mDSN as ‘raspberrypi.local’ and log in with user: pi, password: raspberry.

Disable bluez

By default, bluez will start up using the the BCM4343. To make it available to BTstack, you can disable its system services:

```
$ sudo systemctl disable hciuart
$ sudo systemctl disable bthelper
$ sudo systemctl disable bluetooth
```

and if you don’t want to restart, you can stop them right away. Otherwise, please reboot here.

```
$ sudo systemctl stop hciuart
$ sudo systemctl stop bthelper
$ sudo systemctl stop bluetooth
```

If needed, they can be re-enabled later as well.

Compilation

The Makefile assumes cross-compilation using the regular GCC Cross Toolchain for gnueabihf: arm-linux-gnueabihf-gcc. This should be available in your package manager. Read on for a heavy, but easy-to-use approach.

Compile using Docker

For non-Linux users, we recommend to use a Raspberry Pi Cross-Compiler in a Docker Container. Please follow the installation instructions in the README.

Then, setup a shared folder in Docker that contains the BTstack repository. Now, go to the BTstack repository and 'switch' to the Raspberry Pi Cross-Compiler container:

```
$ rpxc bash
```

The default images doesn't have a Python installation, so we manually install it:

```
$ sudo apt-get install python
```

Change to the port/raspi folder inside the BTstack repo:

```
$ cd btstack/port/raspi
```

and compile as usual:

```
$ make
```

For regular use, it makes sense to add Python permanently to the Docker container. See documentation at GitHub.

Running the examples

Copy one of the examples to the Rasperry Pi and just run them. BTstack will power cycle the Bluetooth Controller on models without hardware flowcontrol, i.e., Pi 3 A/B. With flowcontrol, e.g Pi Zero W and Pi 3 A+/B+, the firmware will only uploaded on first start.

```
pi@raspberrypi:~ $ ./le_counter
Packet Log: /tmp/hci_dump.pklg
Hardware UART without flowcontrol
Phase 1: Download firmware
Phase 2: Main app
BTstack counter 0001
BTstack up and running at B8:27:EB:27:AF:56
```

Bluetooth Hardware Overview

Model	Bluetooth Controller	UART Type	UART Flowcontrol	BT_REG_EN	HCI Transport	Baudrate
Older	None					

Model	Bluetooth Controller	UART Type	UART Flowcontrol	BT_REG_EN	HCI Transport	Baudrate
Pi 3 Model A, Model B	CYW43438	Hardware	No	128	H5	921600
Pi 3 Model A+, Model B +	CYW43455	Hardware	Yes	129 (See 1)	H4	921600 (See 2)
Pi Zero W	CYW43438	Hardware	Yes	45	H4	921600

1. Model A+/B+ have BT_REG_EN AND WL_REG_EN on the same (virtual) GPIO 129. A Bluetooth Controller power cycle also shuts down Wifi (temporarily). BTstack avoids a power cycle on A+/B+.
2. Model A+/B+ support 3 mbps baudrate. Not enabled/activated yet.

TODO

- Raspberry Pi Zero W: Check if higher baud rate can be used, 3 mbps does not work.
- Raspberry + RedBear IoT pHAT (AP6212A = BCM4343) port: IoT pHAT need to get detected and the UART configured appropriately.

BTstack Port for Renesas Eval Kit EK-RA6M4 with DA14531

This port uses the Renesas EK-RA6M4 and a Renesas DA14531 Controller on the MikroeE BLE Tiny Click board

Renesas e2 Studio (Eclise-based) was used with the FSP HAL and without an RTOS to generate project sources. Then, a new CMake buildfile was created to allow for cross-platform development and compilation of all examples. For easy debugging, Ozone project files are generated as well.

Hardware

Renesas Eval Kit EK-RA6M4:

- The RA6 contains a build in J-Link programmer which supports debut output via SEGGER RTT.
- It uses the MikroBus port for the DA1451

MikroBus	MCU	Function
J21/2	P115	RESET (active high)
P21/3	P205	RTS
J21/4	P204	CTS
J22/4	P613	TX
J22/3	P614	RX

- UART RTS: Manual RTS control in UART callback handler. MikroBus slot with UART 7 does not have RTSCTS7 on the pin used by BLE Tiny Click module.
- BSP

```
// 0x1000 main stack
#define BSP_CFG_STACK_MAIN_BYTES (0x1000)

// printf allocates memory from the heap
#define BSP_CFG_HEAP_BYTES (0x800)
```

Renesas DA14531 Module on MikroE BLE Tiny Click board with

- The board comes with some demo application and needs to be programmed with an HCI firmware to use it with a regular Bluetooth stack.

- Firmware details:
  - Keil uVision project `DA145xx_SDK/x.x.xx.xxxx/projects/target_apps/hci` on Windows

```
// Config: user_periph_setup.h
#define UART1_TX_PORT   GPIO_PORT_0
#define UART1_TX_PIN    GPIO_PIN_6
#define UART1_RX_PORT   GPIO_PORT_0
#define UART1_RX_PIN    GPIO_PIN_5
#define UART1_RTSN_PORT GPIO_PORT_0
#define UART1_RTSN_PIN  GPIO_PIN_7
#define UART1_CTSN_PORT GPIO_PORT_0
#define UART1_CTSN_PIN  GPIO_PIN_8
#define UART1_BAUDRATE  UART_BAUDRATE_460800
#define UART1_DATABITS  UART_DATABITS_8

// Config: user_config.h
static const sleep_state_t app_default_sleep_mode =
ARCH_SLEEP_OFF;
```

- Firmware installation:
  - Connect GND (pin 5) and VCC (pin 6) with jumper wires to the RA6 dev board.
  - Connect it with a ARM-Cortex 10-pin connector to a J-Link device
  - Start SmartBond Flash Programmer
  - The Programmer should auto-detect the DA14531 via the J-Link.
  - Select `firmware/hci_531_rx05_tx06_rts07_cts08_468000.hex` as firmware file and click `Program`

Software

The port provides a CMake project file that uses the installed Arm Gnu Toolchain.

- Install Arm GNU Toolchain
- Install CMake
- Install Ninja
- To compile, go to the port folder:

```
cd btstack/port/renesas-ek-ra6me4a-da14531
```

- Create a build folder and go to build folder

```
mkdir build && cd build
```

- Create Ninja build files

```
cmake -G Ninja ..
```

- Build all examples

```
ninja
```

This will build all examples as .elf files as well as .jdebug Ozone debug project files Alternatively, the CMakeLists.txt can be used to compile using Make (`cmake -G "Unix Makefiles" ..` and `make`) or or use the project in most modern IDEs (CLion, Visual Studio, Visual Studio Code, ...)

Run Example Project using Ozone

After building the examples, the generated .elf file can be used with Ozone. Start Ozone and open the provided .jdebug file. The debug output is readily available in the RTT Terminal.

Debug output

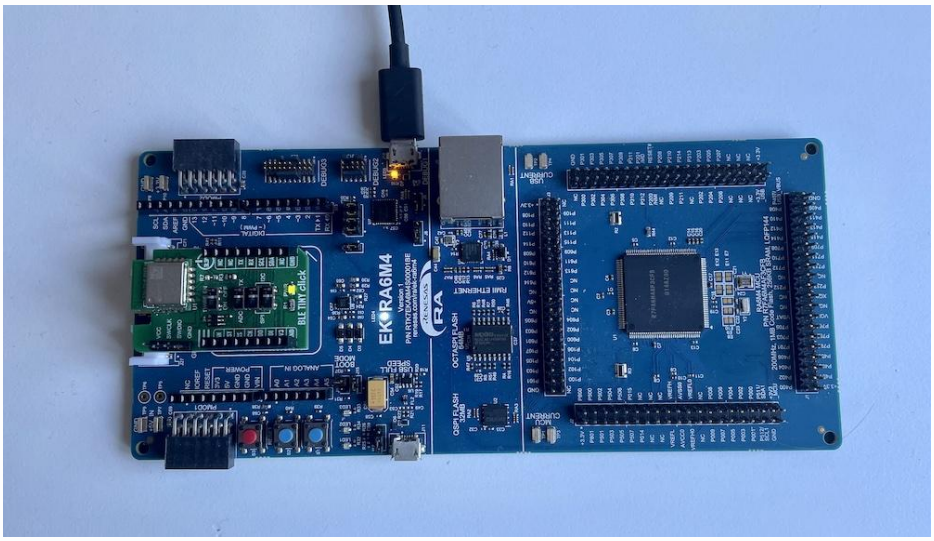
All debug output is send via SEGGER RTT.

In src/btstack_config.h resp. in example/btstack_config.h of the generated projects.

Also, the full packet log with addtional log information can be enabled in src/hal_entry.c by uncommenting the hci_dump_init(...) call.

The console output can then be converted into .pklg files by running tool/create_packet_log.py. The .pklg file can be analyzed with the macOS X PacketLogger or WireShark.

Setup



Renesas Eval Kit EK-RA6M4 with DA14531

Updating HAL Configuration

- Start Renesas RA v3.7.0/e2-studio on Windows and open `e2-project`
- Open `configuration.xml` to get to “FSP Configuration” perspective
  - to add modules, click “New Stack”
  - module is configured in “Properties” view (usually below next to ‘Problems’ etc)
- Press button re-generates sources
- Copy folder `e2-project` into this port
- Check diff for unexpected changes
- If needed:
  - Update CMakeLists.txt to add new modules
  - Add code to enable (‘open’) new module in `R_BSP_WarmStart` of `port/hal_entry.c`

BTstack Port for Renesas Target Board TB-S1JA with CC256x

This port uses the Renesas TB-S1JA with TI’s CC256XEM ST Adapter Kit that allows to plug in a CC256xB or CC256xC Bluetooth module. Renesas e2 Studio (Eclise-based) was used with the SSP HAL and without an RTOS. For easy debugging, Ozone project files are generated as well.

Hardware

Renesas Target Board TB-S1JA: - TB-S1JA Target Board Kit

- CC2564B Bluetooth Controller:
  1. The best option is to get it as a BoostPack
     - Info: BOOST-CC2564MODA: http://www.ti.com/tool/BOOST-CC2564MODA
  2. Alternatively, get the evaluation module together with the EM Wireless Booster pack and a 32.768 kHz oscillator
     - EM Wireless Booster Pack:
       - Info
       - User Guide
  - CC256x Bluetooth module:
    - CC2564B Dual-mode Bluetooth® Controller Evaluation Module
    - CC2564C Dual-mode Bluetooth® Controller Evaluation Module
    - The module with the older CC2564B is around USD 20, while the one with the new CC2564C costs around USD 60

The projects are configured for the CC2564C. When using the CC2564B, *bluetooth_init_cc2564B_1.8_BT_Spec_4.1.c* should be used as cc256x_init_script. You can update this in the create_examples.py script.

Connct the Target Board to the TI Boosterpack, see Booster Pack Pinout: http://www.ti.com/ww/en/launchpad/dl/boosterpack-pinout-v2.pdf

J2 PIN	S1JA PORT	S1JA Signal	Boosterpack
2	P301	RXD0	3 (LP1)
4	P302	TXD0	4 (LP1)
6	P304	CTS0	36 (LP2)
8	P303	RTS0	37 (LP2)
10	VCC	VCC	1 (LP1)
12	VSS	GND	20 (LP2)
14	P112	nShutdown	19 (LP1)

Software

Generate example projects

```
$ python create_examples.py
```

This will generate an e2 Studio project for each example.

Excluded Examples

The a2dp examples (a2dp_source_demo and a2dp_sink_demo) were disabled as the C open-source SBC codec with compile option -O2 wasn't fast enough to provide real-time encoding/decoding.

Build, Flash And Run The Examples in e2 Studio

Open the e2 Studio project and press the 'Debug' button. Debug output is only available via SEGGER RTT. You can run SEGGER's JLinkRTTViewer or use Ozone as described below.

Run Example Project using Ozone

After compiling the project with e2 Studio, the genereated .elf file can be used with Ozone (also e.g. on macOS). In Ozone, the debug output is readily available in the terminal. A .jdebug file is provided in the project folder.

Debug output

All debug output is send via SEGGER RTT.

In src/btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/hal_entry.c by uncommenting the hci_dump_init(...) call. The console output can then be converted into .pklg files by running tool/create_packet_log.py. The .pklg file can be analyzed with the macOS X PacketLogger or WireShark.

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. The create_examples.py script converts the .gatt files into a corresponding .h for the project. After updating a .gatt file, the .h can be updated manually by running the provided update_gatt_db.sh or update_gatt_db.bat scripts.

Note: In theory, this can be integrated into the e2 Studio/Eclipse project.

Notes

- HCI UART is set to 2 mbps. Using 3 or 4 mbps causes hang during startup

Nice to have

- Allow compilation using Makefile/CMake without the e2 Studio, e.g. on the Mac.

BTstack Port for Ezurio Vela IF820 Series

The Vela IF820 Development kit and USB dongle use an Raspberry Pi RP2040 MCU as a combined USB-to-UART adapter and SWD programmer.

This port uses the RP2040 of the development kit as the main target with a connected Vela IF820 / CYW20820 Bluetooth Controller.

In addition, the console is routed over USB CDC for easy integration and an USB audio interface is provided for sound output.

Hardware

As this port targets the RP2040 on the development kit or the USB dongle, the Vela IF820 in module format cannot be used.

More info on the Vela IF820 series

Loading HCI Firmware into the Vela IF820/CYW20820

To use the Bluetooth Controller with BTstack, it needs to be flashed with the HCI Firmware provided by Ezurio. Please use their Flasher tool with the provided HCI firmware.

To flash the HCI firmware, hold the “Module Recovery” button while pressing and releasing “Module Reset”. Then start the Windows GUI and select the correct HCI firmware.

Software

The port for the RP2040 requires the official pico-sdk. Please follow the Getting Started Guide and make sure to point `PICO_SDK_PATH` to the root folder of the SDK.

The current `pico-sdk` v2.2.0 uses TinyUSB v0.18, which has an issue with opening the virtual soundcard more than once. Please update TinyUSB in `lib/tinyusb` to the current master branch or at least commit 5b200c4:

```
$ cd ${PICO_SDK_PATH}/lib/tinyusb
$ git checkout 5b200c4
```

After pointing `PICO_SDK_PATH` to the SDK, all examples can be compiled like this:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make
```

All examples are placed in the `build` folder. Please make sure to specify the Release build type. By default, pico-sdk uses Debug built type and the generated code isn’t fast enough for the SBC Codec in the A2DP demos.

You can compile a single example by calling make with the name of the target .elf file, e.g. `make gatt_counter.elf`

Flash And Run The Examples

By default, different versions of each example are build including: - example.elf: .elf file with all debug information - example.uf2: binary file that can be used to flash via the mass storage interface.

Flash using MSC Interface

Hold the BOOTSEL button then press and release the “Debug Reset” button to enter bootloader mode. The RP2040 should get mounted by the OS. You can now copy any .uf2 into the root folder of the virtual RP2040 device. Press “Debug Reset” to start it.

Flash using Picotool

If you still have the picoprobe firmware installed, hold the BOOTSEL button then press and release the “Debug Reset” button to enter bootloader mode. You can now comfortably load any .elf file with: `picotool load gatt_streamer_server.elf`. After that press the “Debug Reset” button once or run `picotool reboot`.

The port contains the custom Reset logic to enter bootloader mode and you can directly load a new target with `picotool load -f a2dp_sink_demo.elf`

Flash and Debug using SWD Interface

As the Vela IF820 has a TagConnect 2030 interface, the RP2040 can be programmed and debugged with a suitable adapter and any SWD programmer, e.g. SEGGER J-Link or Raspberry Pi’s Picoprobe.

A2DP Sink Demo Example

The most interesting example for this port is the A2DP Sink Demo as it allows to stream music in Hifi quality to your computer.

After flashing the demo onto the RP2040, you can connect from a mobile phone to the `A2DP Sink Demo xx:xx:xx:xx:xx:xx` device and play music. The music is available via the USB soundcard implemented on the RP2040 as part of this port.

On macOS, you can start `Quicktime Player.app` and select `New Audio Recording` from the File menu. The RP2040 shows up as "BTstack Audio" interface. Another option is to use `Audacity`, enable `Software playthrough of interface` in `Settings->Recording` and start recording.

Console / Debug Output

All debug output is sent over the USB CDC. You can interact with the BTstack examples with a regular terminal program like picocom

GATT Database

In BTstack, the GATT Database is defined via the `.gatt` file in the example folder. The `CMakeLists.txt` contains rules to update the `.h` file when the `.gatt` was modified.

HCI Traces

To enable HCI traces, you can uncomment the `WANT_HCI_DUMP=1` line in `CMakeLists.txt`. If you capture the output into a `log.txt` file, you can use `tool/create_packet_log.py log.txt` to convert it into `log.pklg` which can be opened and analyzed with Wireshark.

Uninstall

To restore the Vela IF820 dev kit to its original form, or to update the firmware on the CYW20820 module, you can re-flash `picoprobe` either with `picotool` or by entering the recovery mode (hold BOOSEL, press and release Debug Reset).

BTstack Port for SAMV71 Ultra Xplained with ATWILC3000 SHIELD

This port uses the SAMV71 Ultra Xplained Ultra evaluation kit with an ATWILC3000 SHIELD. The code is based on the Advanced Software Framework (ASF) (previously know as Atmel Software Framework). It uses the GCC Makefiles provided by the ASF. OpenOCD is used to upload the firmware to the device.

Create Example Projects

To create all example projects in the example folder, you can run:

```
$ make
```

Compile Example

In one of the example folders:

```
$ make
```

To upload the firmware:

```
$ make flash
```

You need to connect the the Debug USB to your computer.

Debug output

printf is routed to USART1, which is connected to the virtual serial port. To get the console output, open a terminal at 115200.

In btstack_config.h, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in the main() function on main.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

TODOs

```
- Implement hal_flash_sector.h to persist link keys
```

Issues

```
- Bluetooth UART driver uses per-byte interrupts and doesn't work
reliable at higher baud rates (921600 seems ok, 2 mbps already
causes problems).
- An older XDMA-based implementation only sends 0x00 bytes over
the UART. It might be better to just read incoming data into two
```

`buffers, (e.g. using a two element linked list with XDMA), and raising RTS when one buffer is full.`

BTstack Port for STM32 F4 Discovery Board with CC256x

This port uses the STM32 F4 Discovery Board with TI's CC256XEM ST Adapter Kit that allows to plug in a CC256xB or CC256xC Bluetooth module. STCubeMX was used to provide the HAL, initialize the device, and create an initial CMakeLists.txt. For easy development, Ozone project files are generated as well.

Hardware

STM32 Development kit and adapter for CC256x module: - STM32 F4 Discovery Board - CC256xEM Bluetooth Adatper Kit for ST

CC256x Bluetooth module: - CC2564B Dual-mode Bluetooth® Controller Evaluation Module - CC2564C Dual-mode Bluetooth® Controller Evaluation Module

The module with the older CC2564B is around USD 20, while the one with the new CC2564C costs around USD 60. The projects are configured for the CC2564C. When using the CC2564B, *bluetooth_init_cc2564B_1.8_BT_Spec_4.1.c* should be used as cc256x_init_script.

Software

The build system uses CMake. To build all examples, run `cmake` and `make` in the build folder. The examples are built with

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

All examples and the .jedbug Ozone project files are placed in the 'build' folder.

Flash And Run The Examples

Cmake builds different versions: - example.elf: .elf file with all debug information - example.bin: .bin file that can be used for flashing

There are different options to flash and debug the F4 Discovery board. The F4 Discovery boards comes with an on-board ST-Link programmer and debugger. As an alternative, the ST-Link programmer can be replaced by an SEGGER J-Link OB. Finally, the STM32 can be programmed with any ARM Cortex JTAG or SWD programmer via the SWD jumper.

Run Example Project using Ozone

When using an external J-Link programmer or after installing J-Link OB on the F4 Discovery board, you can flash and debug using the cross-platform SEGGER Ozone Debugger. It is included in some J-Link programmers or can be used for free for evaluation usage.

Just start Ozone and open the .jdebug file in the build folder. When compiled with “ENABLE_SEGGER_RTT”, the debug output shows up in the Terminal window of Ozone.

Debug output

All debug output can be either send via SEGGER RTT or via USART2. To get the console from USART2, connect PA2 (USART2 TX) of the Discovery board to an USB-2-UART adapter and open a terminal at 115200.

In src/btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/port.c resp. btstack/port/stm32-f4discovery-cc256x/src/port.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. The CMakeLists.txt contains rules to update the .h file when the .gatt was modified.

Maintainer Notes - Updating The Port

The Audio BSP is from the STM32F4Cube V1.16 firmware and not generated from STM32CubeMX. To update the HAL, run 'generate code' in CubeMX. After that, make sure to re-apply the patches to the UART and check if the hal config was changed.

BTstack Port for STM32 F4 Discovery Board with USB Bluetooth Controller

This port uses the STM32 F4 Discovery Board with an USB Bluetooth Controller plugged into its USB UTG port. See blog post for details.

STCubeMX was used to provide the HAL, initialize the device, and the Makefile. For easy development, Ozone project files are generated as well.

Hardware

STM32 Development kit with USB OTG adapter and USB CSR8510 Bluetooth Controller - STM32 F4 Discovery Board

Software

To build all examples, run make

```
$ make
```

All examples and the .jedbug Ozone project files are placed in the 'build' folder.

Flash And Run The Examples

The Makefile builds different versions: - example.elf: .elf file with all debug information - example.bin: .bin file that can be used for flashing

There are different options to flash and debug the F4 Discovery board. The F4 Discovery boards comes with an on-board ST-Link programmer and debugger. As an alternative, the ST-Link programmer can be replaced by an SEGGER J-Link OB. Finally, the STM32 can be programmed with any ARM Cortex JTAG or SWD programmer via the SWD jumper.

Run Example Project using Ozone

When using an external J-Link programmer or after installing J-Link OB on the F4 Discovery board, you can flash and debug using the cross-platform SEGGER Ozone Debugger. It is included in some J-Link programmers or can be used for free for evaluation usage.

Just start Ozone and open the .jdebug file in the build folder. When compiled with “ENABLE_SEGGER_RTT”, the debug output shows up in the Terminal window of Ozone.

Debug output

The debug output can send via SEGGER RTT.

In src/btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/port.c resp. btstack/port/stm32-f4discovery-cc256x/src/port.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. The Makefile contains rules to update the .h file when the .gatt was modified.

Maintainer Notes - Updating The Port

The Audio BSP is from the STM32F4Cube V1.16 firmware and not generated from STM32CubeMX. To update the HAL, run 'generate code' in CubeMX. After that, make sure to re-apply the patches to the UART and check if the hal config was changed.

BTstack Port for STM32 Nucleo L073RZ Board with EM9304 Controller

This port uses the STM32 Nucleo-L073RZ Board with EM's EM9304 Shield.

The STM32CubeMX tool was used to provide the HAL, initialize the device, and create a basic Makefile. The Makefile has been exteneded to compile all BTstack LE examples. For easy development, Ozone project files are generated as well.

Hardware

In this port, the EM9304 is conencted via the SPI1 interface and configured for 8 Mhz. Datasheet for the EM9304

It assumes that the EM9304 does not contain any patches and uploads the latest Metapatch during startup.

Software

To build all examples, run make

```
$ make
```

All examples and the .jedbug Ozone project files are placed in the ‘build’ folder.

Flash And Run The Examples

The Makefile builds different versions: - example.elf: .elf file with all debug information - example.bin: .bin file that can be used for flashing

There are different options to flash and debug the F4 Discovery board. The F4 Discovery boards comes with an on-board ST-Link programmer and debugger. As an alternative, the ST-Link programmer can be replaced by an SEGGER J-Link OB. Finally, the STM32 can be programmed with any ARM Cortex JTAG or SWD programmer via the SWD jumper.

Run Example Project using Ozone

When using an external J-Link programmer or after installing J-Link OB on the F4 Discovery board, you can flash and debug using the cross-platform SEGGER Ozone Debugger. It is included in some J-Link programmers or can be used for free for evaluation usage.

Just start Ozone and open the .jdebug file in the build folder. When compiled with “ENABLE_SEGGER_RTT”, the debug output shows up in the Terminal window of Ozone.

Debug output

All debug output can be either send via SEGGER RTT or via USART2. To get the console from USART2, connect PA2 (USART2 TX) of the Discovery board to an USB-2-UART adapter and open a terminal at 115200.

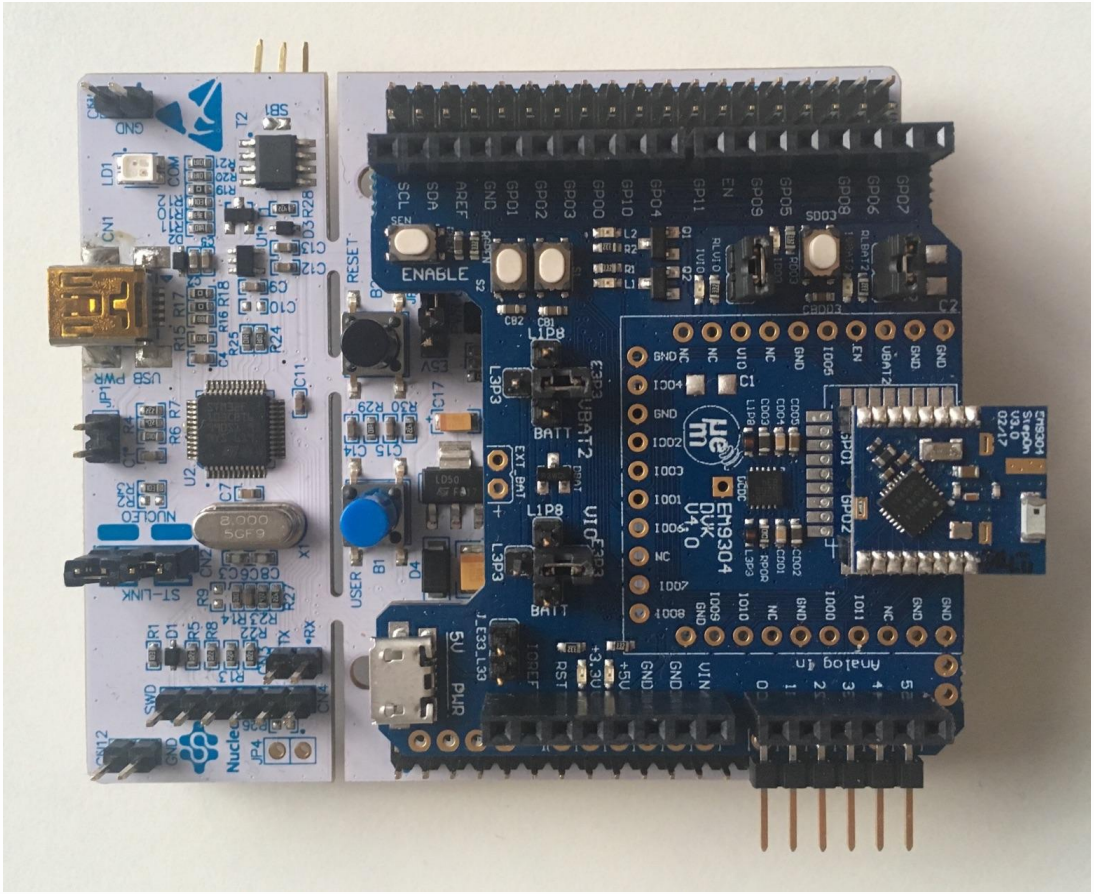
In src/btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/port.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. During the build, the .gatt file is converted into a .h file with a binary representation of the GATT Database and useful defines for the application.

Image



BTstack Port with Cinnamon for Semtech SX1280 Controller on Miromico FMLR-80

Cinnamon is BlueKitchen's minimal, yet robust Controller/Link Layer implementation for use with BTstack.

In contrast to common Link Layer implementations, our focus is on a robust and compact implementation for production use, where code size matters (e.g. current code size about 8 kB).

Overview

This port targets the Semtech SX1280 radio controller. The Host Stack and the Controller (incl. Link Layer) run on a STM32 MCU, with the SX1280 connected via SPI.

It uses the SX1280 C-Driver from Semtech to communicate with the SX1280. The main modification was to the SPI driver that uses DMA for full packets.

Status

Tested with the Miromico FMLR-80-P-STL4E module and our SX1280 Shield - see (port/stm32-l476rg-nucleo-sx1280). On the FMLR-80-P-STL4E module, the 52 Mhz clock for the SX1280 is controlled by the MCU.

SEGGER RTT is used for debug output, so a Segger J-Link programmer is required.

Uses 32.768 kHz crytstal as LSE for timing

Support for Broadcast and Peripheral roles.

The Makefile project compiles gatt_counter, gatt_streamer_server, hog_mouse and hog_keyboard examples.

Limitation

Advertising State:

- Only Connectable Advertising supported
- Only fixed public BD_ADDR of 33:33:33:33:33:33 is used

Connection State:

- Encryption not implemented
- Some LL PDUs not supported

Central Role:

- Not implemented

Observer Role:

- Not implemented

Low power mode - basically not implemented:

- MCU does not sleep
- SPI could be disabled during sleep
- 1 ms SysTick is used for host stack although 16-bit tick time is provided by LPTIM1 based on 32768 Hz LSE.

Getting Started

For the FMLR-80-P-STL4E module, just run make. You can upload the EXAMPLE.elf file created in build folder, e.g. with Ozone using the provided EXAMPLE.jdebug, and run it.

TODO

General

- indicate random address in advertising pdus
- allow to set random BD_ADDR via HCI command and use in Advertisements
- support other regular adv types
- handle Encryption

Low Power

- enter STANDY_RC mode when idle
- implement MCU sleep (if next wakeup isn't immediate)
- sleep after connection request until first packet from Central
- replace SysTick with tick counter based on LPTIM1
- disable SPI on MCU sleep

BTstack Port with Cinnamon for Semtech SX1280 Controller on STM32L476 Nucleo

Cinnamon is BlueKitchen's minimal, yet robust Controller/Link Layer implementation for use with BTstack.

In contrast to common Link Layer implementations, our focus is on a robust and compact implementation for production use, where code size matters (e.g. current code size about 8 kB).

Overview

This port targets the Semtech SX1280 radio controller. The Host Stack and the Controller (incl. Link Layer) run on a STM32 MCU, with the SX1280 connected via SPI.

It uses the SX1280 C-Driver from Semtech to communicate with the SX1280. The main modification was to the SPI driver that uses DMA for full packets.

Status

Tested with theMiromico FMLR-80-P-STL4E module (see port/stm32-l451-miromico-sx1280) and the STM32 L476 Nucleo dev kit with our SX1280 Shield.

SEGGER RTT is used for debug output, so a Segger J-Link programmer is required, but the on-board ST-Link programmer and debugger can be replaced by an SEGGER J-Link OB.

Uses 32.768 kHz crytstal as LSE for timing

Support for Broadcast and Peripheral roles.

The Makefile project compiles gatt_counter, gatt_streamer_server, hog_mouse and hog_keyboard examples.

Limitation

Advertising State:

- Only Connectable Advertising supported
- Only fixed public BD_ADDR of 33:33:33:33:33:33 is used

Connection State:

- Encryption not implemented
- Some LL PDUs not supported

Central Role:

- Not implemented

Observer Role:

- Not implemented

Low power mode - basically not implemented:

- MCU does not sleep
- SPI could be disabled during sleep
- 1 ms SysTick is used for host stack although 16-bit tick time is provided by LPTIM1 based on 32768 Hz LSE.

Getting Started

Just run make. You can upload the EXAMPLE.elf file created in build folder, e.g. with Ozone using the provided EXAMPLE.jdebug, and run it.

TODO

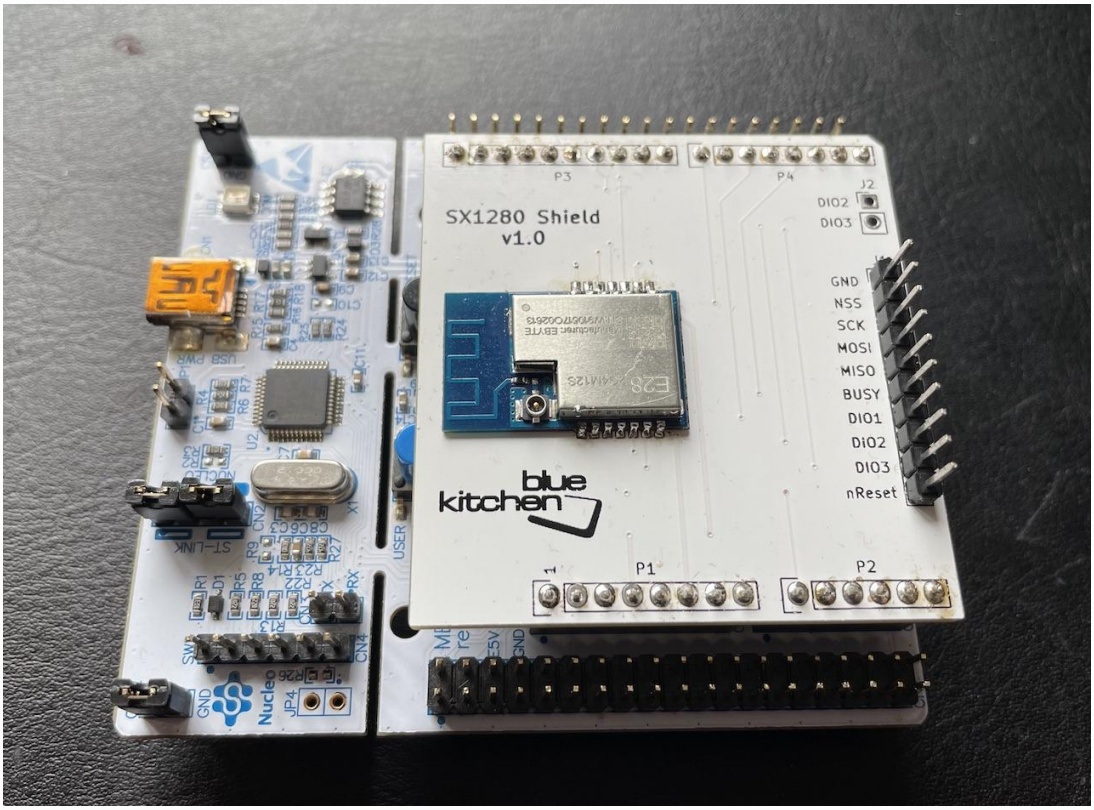
General

- indicate random address in advertising pdus
- allow to set random BD_ADDR via HCI command and use in Advertisements

- support other regular adv types
- handle Encryption

Low Power

- enter STANDY_RC mode when idle
- implement MCU sleep (if next wakeup isn't immediate)
- sleep after connection request until first packet from Central
- replace SysTick with tick counter based on LPTIM1
- disable SPI on MCU sleep



BTstack Port for STM32WB55 Nucleo Boards using FreeRTOS

This port supports the Nucleo68 and the USB dongle of the P-NUCLEO-WB55 kit. Both have 1 MB of Flash memory.

The STM32Cube_FW_WB_V1.3.0 provides the HAL and WPAN, and initializes the device and the initial Makefile. For easy development, Ozone project files are generated as well.

Hardware

In this port, the Nucelo68 or the USB Dongle from the P-NUCLEO-WB55 can be used.

Last test was done using FUS v1.2 and HCI BLE Firmware v1.13 on Nucleo68 See STM32Cube_FW_WB_V1.13.0/Projects/STM32WB_Copro_Wireless_Binaries/Release_Notes.html for firmware install instructions.

Note: Segger RTT is currently not really usable. When sleep modes ared disabled for debuggin (see port_thread()) RTT works, but the output buffer quickly overflows. In Block mode, radio stops working.

Nucleo68

The debug output is sent over USART1 and is available via the ST-Link v2.

USB Dongle

To flash the dongle, SWD can be used via the lower 6 pins on CN1: - 3V3 - PB3 - SWO (semi hosting not used) - PA14 - SCLK - PA13 - SWDIO - NRST - GND

The debug output is sent over USART1 and is available via PB6.

Software

FreeRTOS V10.2.0 is used to run stack, you can get this example version by checking out official repo:

```
$ cd Middlewares
$ git submodule add https://github.com/aws/amazon-freertos.git
$ git submodule update
& cd amazon-freertos && git checkout v1.4.8
```

Or by specifying path to FreeRTOS

```
$ make FREERTOS_ROOT=path_to_freertos
```

To build all examples, run make

```
$ make
```

All examples and the .jedbug Ozone project files are placed in the ‘build’ folder.

Flash And Run The Examples

The Makefile builds different versions: - example.elf: .elf file with all debug information - example.bin: .bin file that can be used for flashing

Nucleo68

There are different options to flash and debug the Nucleo68 board. The Nucleo68 boards comes with an on-board ST-Link programmer and debugger. As an alternative, the ST-Link programmer can be replaced by an SEGGER J-Link OB. Finally, the STM32 can be programmed with any ARM Cortex JTAG or SWD programmer via the SWD jumper.

USB Dongle

Please use any ARM Cortex SWD programmer via the SWD interface desribed in the hardware section.

Run Example Project using Ozone

When using an external J-Link programmer or after installing J-Link OB on the Nucleo68 board, you can flash and debug using the cross-platform SEGGER Ozone Debugger. It is included in some J-Link programmers or can be used for free for evaluation usage.

Just start Ozone and open the .jdebug file in the build folder. When compiled with “ENABLE_SEGGER_RTT”, the debug output shows up in the Terminal window of Ozone. Note: as mentioned before, Segger RTT currently stops working when CPU2 has started up.

Debug output

All debug output can be either send via SEGGER RTT or via USART1. To get the console from USART1, simply connect your board under STLink-v2 to your PC or connect PB6 (USART1 TX) of the Nucleo board to an USB-2-UART adapter and open a terminal at 115200.

In src/btstack_config.h resp. in example/btstack_config.h of the generated projects, additional debug information can be enabled by uncommenting ENABLE_LOG_INFO.

Also, the full packet log can be enabled in src/btstack_port.c by uncommenting the hci_dump_init(..) line. The console output can then be converted into .pklg files for OS X PacketLogger or WireShark by running tool/create_packet_log.py

GATT Database

In BTstack, the GATT Database is defined via the .gatt file in the example folder. During the build, the .gatt file is converted into a .h file with a binary representation of the GATT Database and useful defines for the application.

BTstack on Web using WebSerial

This ports compiles BTstack into WebAssembly and allows to run it in a Chrome-based Browser. It communicates to the Bluetooth Controller via the WebSerial API.

Requirements

- Emscripten from https://emscripten.org

Build instructions

- Get Emscripten repository somewhere:
  - `git clone https://github.com/emscripten-core/emsdk.git`
- make sure EMSDK points to the root of your Emscripten installation:
  - `export EMSDK=<path/to/your/emscripten/repository>`

- create build folder and change to it:
  - `mkdir ninja`
  - `cd ninja`
- configure:
  - `cmake .. \ -DCMAKE_TOOLCHAIN_FILE=$EMSDK/upstream/emscripten/cmake/Modules/Platform/Emscripten.cmake \ -DCMAKE_BUILD_TYPE=Release`
- build using your default CMake generator:
  - `ninja` or `make`

Run Local Web Server

- `python3 -m http.server 8080`

Test

- Open local URL
- Force refresh of .js files: CMD+Alt+R on Safari, CMD+Shift+R on Chrome

Supported Bluetooth Controllers

The baudrate is currently set to 115200 and there's no chipset detection. Suitable Controllers are therefore: - Controllers with USB CDC and no need for firmware upload: - e.g. BL654 with Bluetooth SIG PTS Firmware - Controllers with regular UART that start up at 115200 and no need for firmware upload: - e.g. Ezurio IF820/CYW20820 with HCI Firmware.

Status

Supported Controller start up and all examples work as expected. The 115200 baud are to low for A2DP. There's no audio driver yet as well as no support for STDIO. In addition, TLV is not supported, so bonding information cannot be stored. A better UI will come, too. :)

BTstack Port for WICED platform

Tested with: - WICED SDK 3.4-6.2.1 - RedBear Duo: Please install RedBear WICED Add-On - Inventek Systems ISM4334x - Please contact Inventek Systems for WICED platform files - Inventek Systems ISM4343 (BCM43438 A1) - Please contact Inventek Systems for WICED platform files

To integrate BTstack into the WICED SDK, please move the BTstack project into WICED-SDK-6.2.1/libraries.

Then create projects for BTstack examples in WICED/apps/btstack by running:

```
./create_examples.py
```

Now, the BTstack examples can be build from the WICED root in the same way as other examples, e.g.:

```
./make btstack.spp_and_le_counter-RB_DUO
```

to build the SPP-and-LE-Counter example for the RedBear Duo (or use ISM43340_M4G_L44 / ISM4343_WBM_L151 for the Inventek Systems devices).

See WICED documentation about how to upload the firmware.

It should work with all WICED platforms that contain a Broadcom Bluetooth chipset.

The maximal baud rate is currenty limited to 1 mbps.

The port uses the generated WIFI address plus 1 as Bluetooth MAC address. It persists the LE Device DB and Classic Link Keys via the DCT mechanism.

All examples that rovide a GATT Server use the GATT DB in the .gatt file. Therefore you need to run ./update_gatt_db.sh in the apps/btstack/$(EXAMPLE) folder after modifying the .gatt file.

BTstack Port for Windows Systems with Bluetooth Controller connected via Serial Port

The Windows-H4 port uses the native run loop and allows to use Bluetooth Controllers connected via Serial Port.

Make sure to manually reset the Bluetooth Controller before starting any of the examples.

The port provides both a regular Makefile as well as a CMake build file. It uses native Win32 APIs for file access and does not require the Cygwin or mingw64 build/runtine. All examples can also be build with Visual Studio 2022 (e.g. Community Edition).

Visual Studio 2022

Visual Studio can directly open the provided `port/windows-windows-h4/CMakeLists.txt` and allows to compile and run all examples.

mingw64

It can also be compiles with a regular Unix-style toolchain like mingw-w64. mingw64-w64 is based on MinGW, which '…provides a complete Open Source programming tool set which is suitable for the development of native MS-Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs.'

We've used the Msys2 package available from the downloads page on Windows 10, 64-bit and use the MSYS2 MinGW 64-bit start menu item to compile 64-bit binaries.

In the MSYS2 shell, you can install everything with pacman:

```
$ pacman -S git
$ pacman -S cmake
$ pacman -S make
$ pacman -S mingw-w64-x86_64-toolchain
$ pacman -S mingw-w64-x86_64-portaudio
```

```
$ pacman -S python
$ pacman -S winpty
```

Compilation with CMake

With mingw64-w64 installed, just go to the port/windows-h4 directory and use CMake as usual

```
$ cd port/windows-h4
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note: When compiling with msys2-32 bit and/or the 32-bit toolchain, compilation fails as `conio.h` seems to be mission. Please use msys2-64 bit with the 64-bit toolchain for now.

Console Output

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 ./gatt_counter.exe
```

The packet log will be written to hci_dump.pklg

BTstack Port for Windows Systems with DA14585 Controller connected via Serial Port

This port allows to use the DA14585 connected via Serial Port with BTstack running on a Win32 host system.

It first downloads the hci_585.hex firmware from the 6.0.8.509 SDK, before BTstack starts up.

Please note that it does not detect if the firmware has already been downloaded, so you need to reset the DA14585 before starting an example.

For production use, the HCI firmware could be flashed into the OTP and the firmware download could be skipped.

Tested with the official DA14585 Dev Kit Basic on OS X and Windows 10.

The port provides both a regular Makefile as well as a CMake build file. It uses native Win32 APIs for file access and does not require the Cygwin or mingw64 build/runtine. All examples can also be build with Visual Studio 2022 (e.g. Community Edition).

Visual Studio 2022

Visual Studio can directly open the provided `port/windows-windows-h4-da14585/CMakeLists.txt` and allows to compile and run all examples.

mingw64

It can also be compiles with a regular Unix-style toolchain like mingw-w64. mingw64-w64 is based on MinGW, which '…provides a complete Open Source programming tool set which is suitable for the development of native MS-Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs.'

We've used the Msys2 package available from the downloads page on Windows 10, 64-bit and use the MSYS2 MinGW 64-bit start menu item to compile 64-bit binaries.

In the MSYS2 shell, you can install everything with pacman:

```
$ pacman -S git
$ pacman -S cmake
$ pacman -S make
$ pacman -S mingw-w64-x86_64-toolchain
$ pacman -S mingw-w64-x86_64-portaudio
$ pacman -S python
$ pacman -S winpty
```

Compilation with CMake

With mingw64-w64 installed, just go to the port/windows-h4 directory and use CMake as usual

```
$ cd port/windows-h4
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note: When compiling with msys2-32 bit and/or the 32-bit toolchain, compilation fails as `conio.h` seems to be mission. Please use msys2-64 bit with the 64-bit toolchain for now.

Console Output

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 ./gatt_counter.exe
```

The packet log will be written to hci_dump.pklg

BTstack Port for Windows Systems with Zephyr-based Controller

The main difference to the regular windows-h4 port is that that the Zephyr Contoller uses 1000000 as baud rate. In addition, the port defaults to use the fixed static address stored during production.

The port provides both a regular Makefile as well as a CMake build file. It uses native Win32 APIs for file access and does not require the Cygwin or mingw64 build/runtine. All examples can also be build with Visual Studio 2022 (e.g. Community Edition).

Prepare Zephyr Controller

Please follow this blog post about how to compile and flash `samples/bluetooth/hci_uart` to a connected nRF5 dev kit.

In short: 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, and flash an example project 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.
- In *samples/bluetooth/hci_uart*, compile the firmware for nRF52 Dev Kit

  $ make BOARD=nrf52_pca10040

- Upload the firmware

  $ make flash

- For the nRF51 Dev Kit, use `make BOARD=nrf51_pca10028`.

Configure serial port

To set the serial port of your Zephyr Controller, you can either update config.device_name in main.c or always start the examples with the correct `-u COMx` option.

Visual Studio 2022

Visual Studio can directly open the provided `port/windows-windows-h4-zephyr/CMakeLists.txt` and allows to compile and run all examples.

mingw64

It can also be compiles with a regular Unix-style toolchain like mingw-w64. mingw64-w64 is based on MinGW, which '...provides a complete Open Source programming tool set which is suitable for the development of native MS-Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs.'

In the MSYS2 shell, you can install everything with pacman:

```
$ pacman -S git
$ pacman -S cmake
$ pacman -S make
$ pacman -S mingw-w64-x86_64-toolchain
$ pacman -S mingw-w64-x86_64-portaudio
$ pacman -S python
$ pacman -S winpty
```

Compilation with CMake

With mingw64-w64 installed, just go to the port/windows-h4 directory and use CMake as usual

```
$ cd port/windows-h4
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note: When compiling with msys2-32 bit and/or the 32-bit toolchain, compilation fails as `conio.h` seems to be mission. Please use msys2-64 bit with the 64-bit toolchain for now.

Console Output

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 ./gatt_counter.exe
```

The packet log will be written to hci_dump.pklg

BTstack Port for Windows Systems using the WinUSB Driver

The Windows-WinUSB port uses the native run loop and WinUSB API to access a USB Bluetooth dongle.

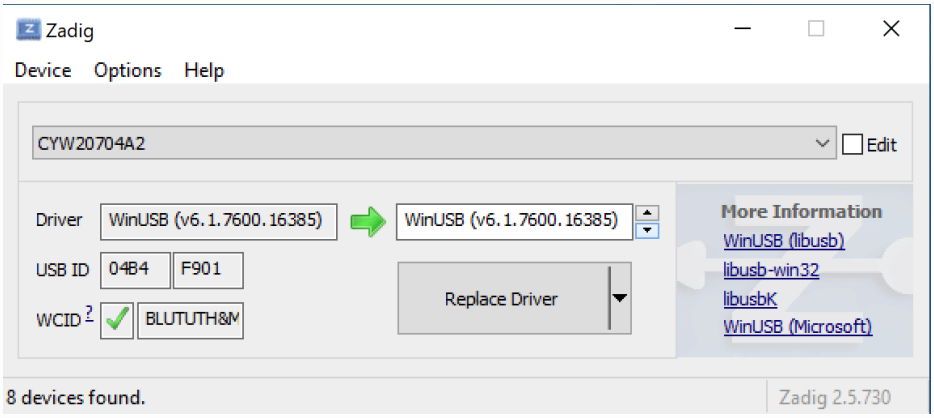
The port provides both a regular Makefile as well as a CMake build file. It uses native Win32 APIs for file access and does not require the Cygwin or mingw64 build/runtine. All examples can also be build with Visual Studio 2022 (e.g. Community Edition).

Access to Bluetooth USB Dongle with Zadig

To allow WinUSB to access an USB Bluetooth dongle, you need to install a special device driver to make it accessible to user space processes.

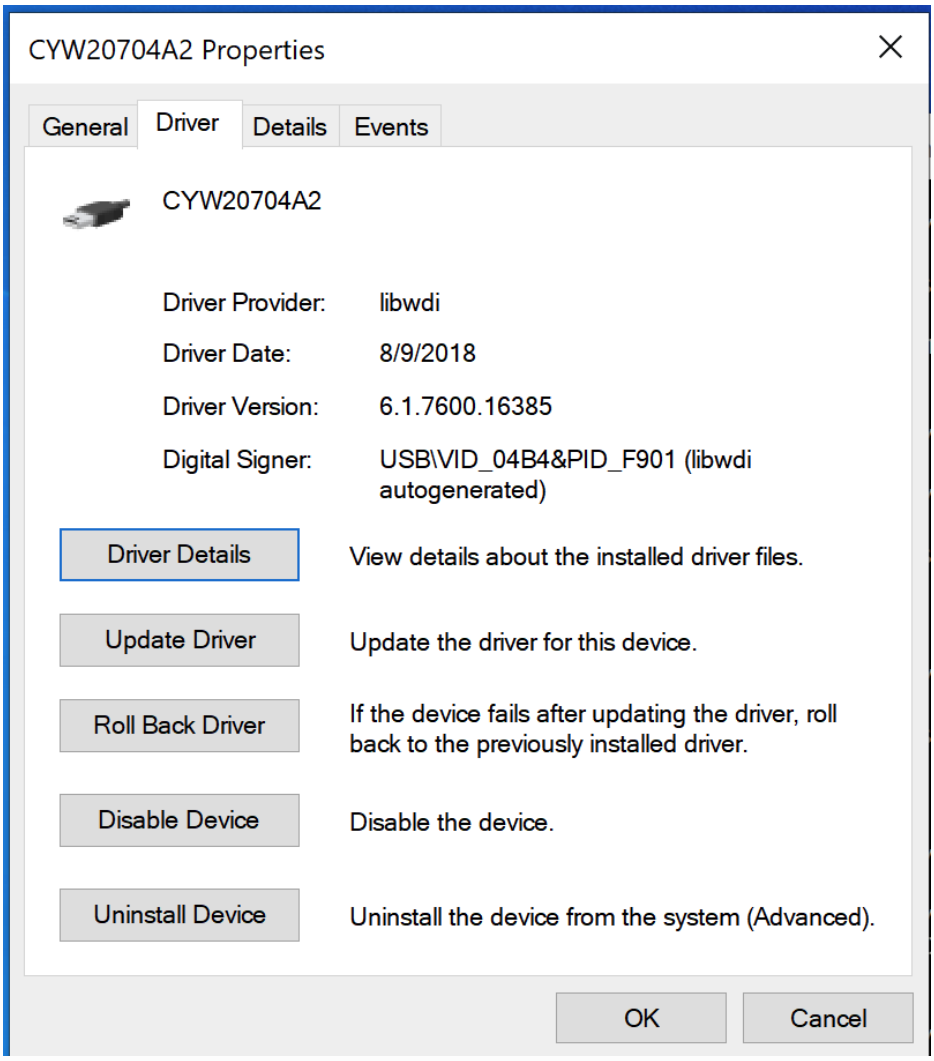
It works like this:

- Download Zadig
- Start Zadig
- Select Options -> “List all devices”
- Select USB Bluetooth dongle in the big pull down list
- Select WinUSB in the right pull down list
- Select “Replace Driver”



Zadig showing CYW20704A2

After the new driver was installed, your device is shown in the Device Manager with Device Provider ‘libwdi’



Device Manager showing CYW20704A2

Visual Studio 2022

Visual Studio can directly open the provided `port/windows-winusb/CMakeLists.txt` and allows to compile and run all examples. For this, the C++ CMake tools for Windows is required. They are part of the Desktop development with C++ workloads.

mingw64

It can also be compiles with a regular Unix-style toolchain like mingw-w64. mingw64-w64 is based on MinGW, which '...provides a complete Open Source programming tool set which is suitable for the development of native MS-Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs.'

In the MSYS2 shell, you can install everything with pacman:

```
$ pacman -S git
$ pacman -S cmake
$ pacman -S make
$ pacman -S mingw-w64-x86_64-toolchain
$ pacman -S mingw-w64-x86_64-portaudio
$ pacman -S python
$ pacman -S winpty
```

Compilation with CMake

With mingw64-w64 installed, just go to the port/windows-h4 directory and use CMake as usual

```
$ cd port/windows-h4
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note: When compiling with msys2-32 bit and/or the 32-bit toolchain, compilation fails as `conio.h` seems to be mission. Please use msys2-64 bit with the 64-bit toolchain for now.

Console Output

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 ./spp_and_le_counter.exe
```

BTstack Port for Windows Systems with Intel Wireless 8260/8265 Controllers

Same as port/windows-winusb, but customized for Intel Wireless 8260 and 8265 Controllers. These controller require firmware upload and configuration to work. Firmware and config is downloaded from the Linux firmware repository.

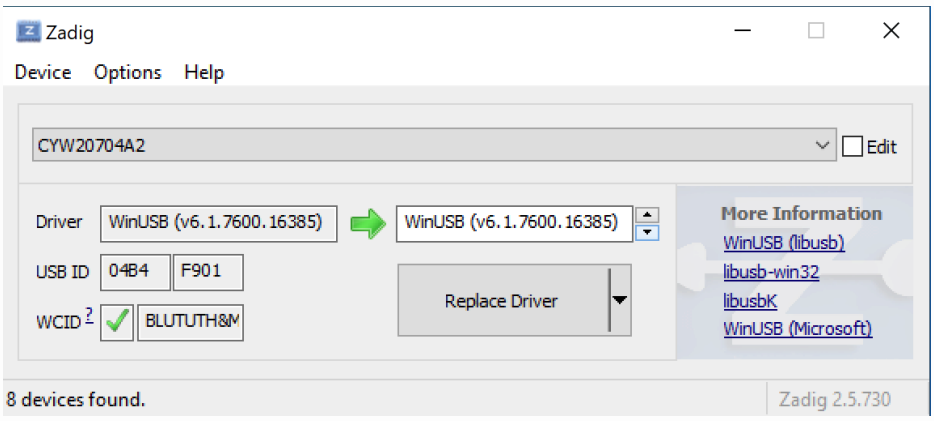
The port provides both a regular Makefile as well as a CMake build file. It uses native Win32 APIs for file access and does not require the Cygwin or mingw64 build/runtine. All examples can also be build with Visual Studio 2022 (e.g. Community Edition).

Access to Bluetooth USB Dongle with Zadig

To allow WinUSB to access an USB Bluetooth dongle, you need to install a special device driver to make it accessible to user space processes.

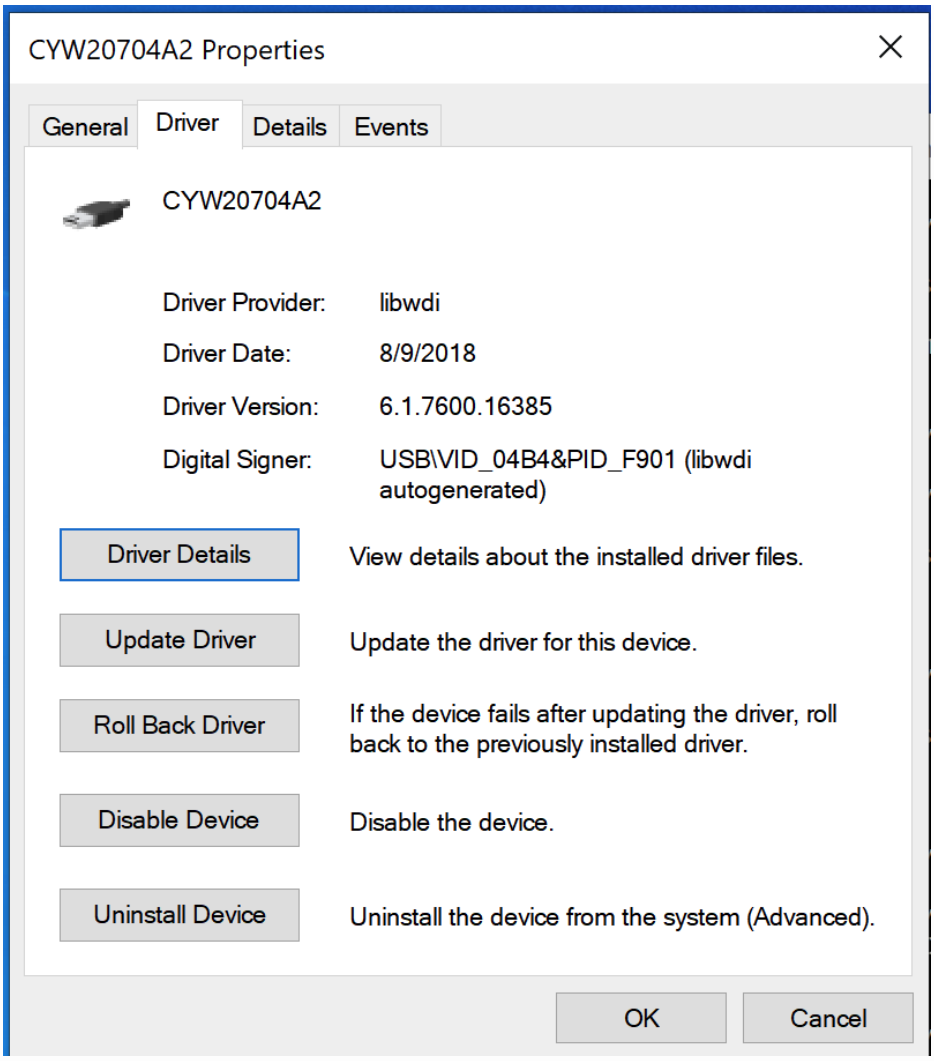
It works like this:

- Download Zadig
- Start Zadig
- Select Options -> “List all devices”
- Select USB Bluetooth dongle in the big pull down list
- Select WinUSB in the right pull down list
- Select “Replace Driver”



Zadig showing CYW20704A2

After the new driver was installed, your device is shown in the Device Manager with Device Provider ‘libwdi’



Device Manager showing CYW20704A2

Visual Studio 2022

Visual Studio can directly open the provided `port/windows-winusb-intel/CMakeLists.txt` and allows to compile and run all examples.

mingw64

It can also be compiles with a regular Unix-style toolchain like mingw-w64. mingw64-w64 is based on MinGW, which '...provides a complete Open Source programming tool set which is suitable for the development of native MS-Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs.'

In the MSYS2 shell, you can install everything with pacman:

```
$ pacman -S git
$ pacman -S cmake
$ pacman -S make
$ pacman -S mingw-w64-x86_64-toolchain
$ pacman -S mingw-w64-x86_64-portaudio
$ pacman -S python
$ pacman -S winpty
```

Compilation with CMake

With mingw64-w64 installed, just go to the port/windows-h4 directory and use CMake as usual

```
$ cd port/windows-h4
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note: When compiling with msys2-32 bit and/or the 32-bit toolchain, compilation fails as `conio.h` seems to be mission. Please use msys2-64 bit with the 64-bit toolchain for now.

Console Output

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 ./gatt_counter.exe
```

The packet log will be written to hci_dump.pklg

BTstack Port for Zephyr RTOS

Overview

This port targets any platform supported by Zephyr that either contains a built-in Bluetooth Controller or is connected to an external Controller via one of the supported Zephyr HCI Transports drivers (see `zephyr/drivers/bluetooth/hci`)

Status

Tested with nRF52 DK (PCA10040), nRF52840 DK (PC10056) and nRF5340 DK (PCA10095) boards only. It uses the fixed static random BD ADDR stored in NRF_FICR/NRF_FICR_S, which will not compile on non nRF SoCs.

Build Environment

The first step needs to done once. Step two is needed every time to setup the environment.

1. Build Environment Preconditions

Follow the getting started guide until you are able to build an example.

Then update the `ZEPHYR_ROOT` variable in `env.sh` to point to your `zephyrproject`. Defaults to `~/zephyrproject`

2. Prepare the build environment

To setup your environment to build a BTstack example, run the provided setup in `env.sh`.

```
source env.sh
```

Building and Running on nRF52840

1. Build Example

You can build an example using:

```
west build -b nrf52840dk/nrf52840
```

`nrf52840dk/nrf52840` selected the Nordic nRF52840 DK. For the older nRF52 DK with nRF52832, you can specify nrf52dk/nrf52832. To get a list of all supported Zephyr targets, run `west boards`

To change zephyr platform settings use:

```
west build -b nrf52840dk/nrf52840 -t menuconfig
```

To build a different example, e.g. the `gatt_streamer_server`, set the EXAMPLE environment variable:

```
EXAMPLE=gatt_streamer_server west build -b nrf52840dk/nrf52840
```

2. Flash Example

To flash a connected board:

```
west flash
```

Building and Running on nRF5340

The nrf5340 is a dual core SoC, where one core is used for Bluetooth HCI functionality and the other for the high level application logic. So both cores need to be programmed separately. Using the nRF5340-DK for example allows debug output on both cores to separate UART ports. For the nRF5340 a network core firmware is required, which one depends on the use-case. With 2. an options is given.

1. Building the application

build using:

```
west build -b nrf5340dk/nrf5340/cpuapp
```

with debug:

```
west build -b nrf5340dk/nrf5340/cpuapp -- -
        DOVERLAY_CONFIG=debug_overlay.conf
```

2. Using zephyr network core image

the `hci_rgmsg` application needs to be loaded first to the network core. Configure network core by selecting the appropriate config file, for example `nrf5340_cpunet_iso-bt_ll_sw_split.conf`. additionally it's required to increase the main stack size from

```
CONFIG_MAIN_STACK_SIZE=512
```

to

```
CONFIG_MAIN_STACK_SIZE=4096
```

then the network core image can be compiled and flashed

```
west build -b nrf5340dk/nrf5340/cpunet -- -
        DCONF_FILE=nrf5340_cpunet_iso-bt_ll_sw_split.conf
west flash
```

or with debugging

```
west build -b nrf5340dk/nrf5340/cpunet -- -
        DCONF_FILE=nrf5340_cpunet_iso-bt_ll_sw_split.conf -
        DOVERLAY_CONFIG=debug_overlay.conf
west flash
```

Build and Running on STM32 Nucleo 144 board with Ezurio M.2 Adapter and M.2 Bluetooth Controller

Ezurio's Wi-Fi M.2 2230 to STM32 Nucleo Adapter Card allows to connect any Bluetooth/Wifi M.2 Card to most STM32 Nucleo-144 boards. M.2 modules are available from multiple module vendors, e.g. Ezurio, u-blox, and Embedded Artists.

The adapter board takes care of 3.3V and 1.8V translation and routes SDIO for Wifi and UART for Bluetooth to the Nucleo board. On most Nucleo-144 boards, the Bluetooth UART ends up on the same STM32, namely: PD3-PD6. See `port/zephyr/boards/shields/ezurio_m2_nucleo_144_adapter.overlay` for details.

We tested CYW55513 and CYW55573 M.2 modules with the following boards:

Nucleo board	Working
nucleo_f439zi	x
nucleo_h753zi	x
nucleo_h563zi	x
nucleo_h755zi	

The PatchRAM .hcd files can be placed in the project folder and specified by the `CONFIG_AIROC_CUSTOM_FIRMWARE_HCD_BLOB` option in the corresponding `boards/$BOARD.conf` file.

1. Build Example

```
west build -b nucleo_f439zi --shield ezurio_m2_nucleo_144_adapter
west build -b nucleo_h755zi_q/stm32h755xx/m7 --shield
        ezurio_m2_nucleo_144_adapter
```

TODO

- Allow/document use of Zephyr HCI Drivers

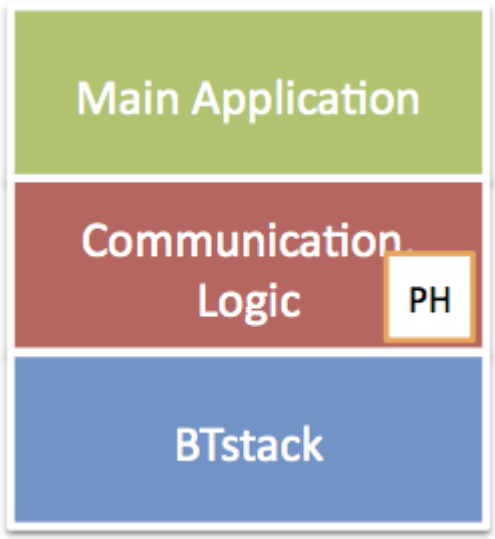
#Integrating with Existing Systems

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.

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 below.



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:

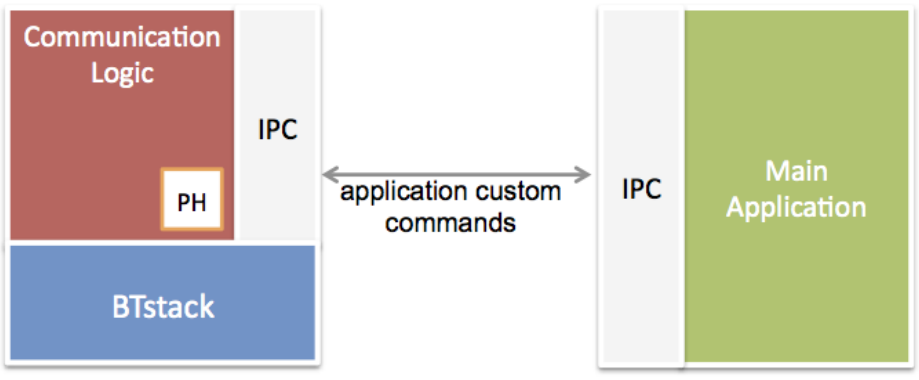
- *btstack_run_loop_posix.c* is an implementation for POSIX compliant systems. The data sources are modeled as file descriptors and managed in a linked list. Then, the *select* function is used to wait for the next file descriptor to become ready or timer to expire.
- *btstack_run_loop_cocoa.c* is an integration 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.
- *btstack_run_loop_qt* is an integration for the Qt run loop. The data sources on Windows systems use Event objects via Qt's QEventNotifier adapter, while a QSocketNotifier is used for Mac/Linux to handle file descriptors. With these in place, the Windows/POSIX implemenations for HCI USB/H2 and HCI H4 can be used.

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:

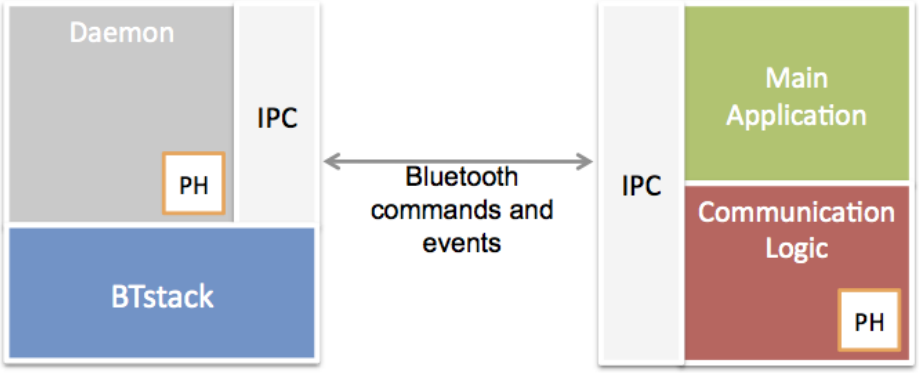
- The Communication Logic is implemented on a dedicated BTstack thread, and the Main Application communicates with the BTstack thread via application-specific messages over an Interprocess Communication (IPC) as depicted in Figure below. This option results in less code and quick adaption.



BTstack in multi-threaded environment - monolithic solution.

- 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 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 below. This option requires more code but provides more flexibility.



BTstack in multi-threaded environment - solution with daemon.

APIs

AD Data Parser API

ad_parser.h : AD Data (Advertisements and EIR) Parser

```
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);
bool 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
bool ad_data_contains_uuid16(uint8_t ad_len, const uint8_t *
ad_data, uint16_t uuid16);
bool ad_data_contains_uuid128(uint8_t ad_len, const uint8_t *
ad_data, const uint8_t * uuid128);
```

ATT Database Engine API

att_db.h

```
// map ATT ERROR CODES on to att_read_callback length
#define ATT_READ_ERROR_CODE_OFFSET       0xfe00u

// custom BTstack ATT Response Pending for att_read_callback
#define ATT_READ_RESPONSE_PENDING                0xffffu

// internally used to signal write response pending
// To ask ATT Server to defer the write response, you can return
ATT_ERROR_WRITE_RESPONSE_PENDING in your att_write_callback
#define ATT_INTERNAL_WRITE_RESPONSE_PENDING      0xfffeu

/**
 * @brief 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
 *
 * If ENABLE_ATT_DELAYED_READ_RESPONSE is defined, you may return
ATT_READ_RESPONSE_PENDING if data isn't available yet
 * and call att_server_response_ready to re-trigger the callback.
 *
 * @param con_handle of hci le connection
 * @param attribute_handle to be read
 * @param offset defines start of attribute value
 * @param buffer
 * @param buffer_size
```

```
 * @return size of value if buffer is NULL, otherwise number of
bytes copied
 */
typedef uint16_t (*att_read_callback_t)(hci_con_handle_t
con_handle, uint16_t attribute_handle, uint16_t offset, uint8_t *
buffer, uint16_t buffer_size);

/**
 * @brief ATT Client Write Callback for Dynamic Data
 * Each Prepared Write Request triggers a callback with
transaction mode ATT_TRANSACTION_MODE_ACTIVE.
 * On Execute Write, the callback will be called with
ATT_TRANSACTION_MODE_VALIDATE and allows to validate all queued
writes and return an application error.
 * If none of the registered callbacks return an error for
ATT_TRANSACTION_MODE_VALIDATE and the callback will be called with
ATT_TRANSACTION_MODE_EXECUTE.
 * Otherwise, all callbacks will be called with
ATT_TRANSACTION_MODE_CANCEL.
 *
 * If the additional validation step is not needed, just return 0
for all callbacks with transaction mode
ATT_TRANSACTION_MODE_VALIDATE.
 *
 * If ENABLE_ATT_DELAYED_READ_RESPONSE is defined, you may return
ATT_ERROR_WRITE_RESPONSE_PENDING if data isn't available yet
 * and call att_server_response_ready to re-trigger the callback.
 *
 * @param con_handle of hci le connection
 * @param attribute_handle to be written
 * @param transaction - ATT_TRANSACTION_MODE_NONE for regular
writes. For prepared writes: ATT_TRANSACTION_MODE_ACTIVE,
ATT_TRANSACTION_MODE_VALIDATE, ATT_TRANSACTION_MODE_EXECUTE,
ATT_TRANSACTION_MODE_CANCEL
 * @param offset into the value - used for queued writes and long
attributes
 * @param buffer
 * @param buffer_size
 * @param signature used for signed write commmands
 * @return 0 if write was ok, ATT_ERROR_PREPARE_QUEUE_FULL if no
```

```
space in queue, ATT_ERROR_INVALID_OFFSET if offset is larger than
max buffer
 */
typedef int (*att_write_callback_t)(hci_con_handle_t con_handle,
uint16_t attribute_handle, uint16_t transaction_mode, uint16_t
offset, uint8_t *buffer, uint16_t buffer_size);

// Read & Write Callbacks for handle range
typedef struct att_service_handler {
    btstack_linked_item_t * item;
    att_read_callback_t read_callback;
    att_write_callback_t write_callback;
    btstack_packet_handler_t packet_handler;
    uint16_t start_handle;
    uint16_t end_handle;
    uint8_t flags;
} att_service_handler_t;

// MARK: ATT Operations

/**
 * @brief setup ATT database
 * @param db
 */
void att_set_db(uint8_t const * db);

/*
 * @brief set callback for read of dynamic attributes
 * @param callback
 */
void att_set_read_callback(att_read_callback_t callback);

/**
 * @brief set callback for write of dynamic attributes
 * @param callback
 */
void att_set_write_callback(att_write_callback_t callback);

/**
 * @brief debug helper, dump ATT database to stdout using log_info
```

```
 */
void att_dump_attributes(void);

/**
 * @brief process ATT request against database and put response
into response buffer
 * @param att_connection used for mtu and security properties
 * @param request_buffer, request_len: ATT request from clinet
 * @param response_buffer for result
 * @return len of data in response buffer. 0 = no response,
 *          ATT_READ_RESPONSE_PENDING if it was returned at least
once for dynamic data (requires ENABLE_ATT_DELAYED_READ_RESPONSE)
 */
uint16_t att_handle_request(att_connection_t * att_connection,
                            uint8_t * request_buffer,
                            uint16_t request_len,
                            uint8_t * response_buffer);

/**
 * @brief setup value notification in response buffer for a given
handle and value
 * @param att_connection
 * @param attribute_handle
 * @param value
 * @param value_len
 * @param response_buffer for notification
 */
uint16_t att_prepare_handle_value_notification(att_connection_t *
att_connection,
                                               uint16_t
attribute_handle,
                                               const uint8_t
*value,
                                               uint16_t
value_len,
                                               uint8_t *
response_buffer);

/**
 * @brief setup value notification in response buffer for multiple
```

```
handles and values
 * @param att_connection
 * @param attribute_handle
 * @param value
 * @param value_len
 * @param response_buffer for notification
 */
uint16_t
att_prepare_handle_value_multiple_notification(att_connection_t *
att_connection,
                                               uint8_t
num_attributes,
                                               const
uint16_t * attribute_handles,
                                               const
uint8_t ** values_data,
                                               const
uint16_t * values_len,
                                               uint8_t *
response_buffer);

/**
 * @brief setup value indication in response buffer for a given
handle and value
 * @param att_connection
 * @param attribute_handle
 * @param value
 * @param value_len
 * @param response_buffer for indication
 */
uint16_t att_prepare_handle_value_indication(att_connection_t *
att_connection,
                                             uint16_t
attribute_handle,
                                             const uint8_t *value,
                                             uint16_t value_len,
                                             uint8_t *
response_buffer);

/**
```

```
 * @brief transcation queue of prepared writes, e.g., after
disconnect
 * @return att_connection
 */
void att_clear_transaction_queue(att_connection_t *
att_connection);

// att_read_callback helpers for a various data types

/**
 * @brief Handle read of blob like data for att_read_callback
 * @param blob of data
 * @param blob_size of blob
 * @param offset from att_read_callback
 * @param buffer from att_read_callback
 * @param buffer_size from att_read_callback
 * @return value size for buffer == 0 and num bytes copied
otherwise
 */
uint16_t att_read_callback_handle_blob(const uint8_t * blob,
uint16_t blob_size, uint16_t offset, uint8_t * buffer, uint16_t
buffer_size);

/**
 * @brief Handle read of little endian unsigned 32 bit value for
att_read_callback
 * @param value
 * @param offset from att_read_callback
 * @param buffer from att_read_callback
 * @param buffer_size from att_read_callback
 * @return value size for buffer == 0 and num bytes copied
otherwise
 */
uint16_t att_read_callback_handle_little_endian_32(uint32_t value,
uint16_t offset, uint8_t * buffer, uint16_t buffer_size);

/**
 * @brief Handle read of little endian unsigned 16 bit value for
att_read_callback
 * @param value
```

```
 * @param offset from att_read_callback
 * @param buffer from att_read_callback
 * @param buffer_size from att_read_callback
 * @return value size for buffer == 0 and num bytes copied
otherwise
 */
uint16_t att_read_callback_handle_little_endian_16(uint16_t value,
uint16_t offset, uint8_t * buffer, uint16_t buffer_size);

/**
 * @brief Handle read of single byte for att_read_callback
 * @param blob of data
 * @param blob_size of blob
 * @param offset from att_read_callback
 * @param buffer from att_read_callback
 * @param buffer_size from att_read_callback
 * @return value size for buffer == 0 and num bytes copied
otherwise
 */
uint16_t att_read_callback_handle_byte(uint8_t value, uint16_t
offset, uint8_t * buffer, uint16_t buffer_size);


// experimental client API
/**
 * @brief Get UUID for handle
 * @param attribute_handle
 * @return 0 if not found
 */
uint16_t att_uuid_for_handle(uint16_t attribute_handle);

/**
 * @brief Get const value for handle
 * @param attribute_handle
 * @param out_value_len  output variable that hold value len
 * @return value
 */

const uint8_t * gatt_server_get_const_value_for_handle(uint16_t
attribute_handle, uint16_t * out_value_len);
```

```
// experimental GATT Server API

/**
 * @brief Get handle range for primary or secondary service.
 * @param uuid16
 * @param start_handle
 * @param end_handle
 * @return false if not found
 */
bool gatt_server_get_handle_range_for_service_with_uuid16(uint16_t
uuid16, uint16_t * start_handle, uint16_t * end_handle);

/**
 * @brief Get handle range for included service.
 * @param start_handle
 * @param end_handle
 * @param uuid16
 * @param out_included_service_handle
 * @param out_included_service_start_handle
 * @param out_included_service_end_handle
 * @return false if not found
 */
bool gatt_server_get_included_service_with_uuid16(uint16_t
start_handle, uint16_t end_handle, uint16_t uuid16,
    uint16_t * out_included_service_handle, uint16_t *
out_included_service_start_handle, uint16_t *
out_included_service_end_handle);

/**
 * @brief Get value handle for characteristic.
 * @param start_handle
 * @param end_handle
 * @param uuid16
 * @return 0 if not found
 */
uint16_t
gatt_server_get_value_handle_for_characteristic_with_uuid16(uint16_
start_handle, uint16_t end_handle, uint16_t uuid16);
```

```
/**
 * @brief Get descriptor handle for characteristic.
 * @param start_handle
 * @param end_handle
 * @param characteristic_uuid16
 * @param descriptor_uuid16
 * @return 0 if not found
 */
uint16_t
gatt_server_get_descriptor_handle_for_characteristic_with_uuid16(ui
start_handle, uint16_t end_handle, uint16_t characteristic_uuid16,
uint16_t descriptor_uuid16);

/**
 * @brief Get client configuration handle for characteristic.
 * @param start_handle
 * @param end_handle
 * @param characteristic_uuid16
 * @return 0 if not found
 */
uint16_t
gatt_server_get_client_configuration_handle_for_characteristic_with
start_handle, uint16_t end_handle, uint16_t
characteristic_uuid16);

/**
 * @brief Get server configuration handle for characteristic.
 * @param start_handle
 * @param end_handle
 * @param characteristic_uuid16
 * @param descriptor_uuid16
 * @return 0 if not found
 */
uint16_t
gatt_server_get_server_configuration_handle_for_characteristic_with
start_handle, uint16_t end_handle, uint16_t
characteristic_uuid16);


/**
```

```
 * @brief Get handle range for primary service.
 * @param uuid128
 * @param start_handle
 * @param end_handle
 * @return false if not found
 */
bool gatt_server_get_handle_range_for_service_with_uuid128(const
uint8_t * uuid128, uint16_t * start_handle, uint16_t *
end_handle);

/**
 * @brief Get value handle.
 * @param start_handle
 * @param end_handle
 * @param uuid128
 * @return 0 if not found
 */
uint16_t
gatt_server_get_value_handle_for_characteristic_with_uuid128(uint16
start_handle, uint16_t end_handle, const uint8_t * uuid128);

/**
 * @brief Get client configuration handle.
 * @param start_handle
 * @param end_handle
 * @param uuid128
 * @return 0 if not found
 */
uint16_t
gatt_server_get_client_configuration_handle_for_characteristic_with
start_handle, uint16_t end_handle, const uint8_t * uuid128);

/**
 * @brief Get Database Hash provided via GATT Database Hash
characteristic
 * @note Used by att_server to discard stored CCCD values for
bonded devices if hash has changed
 * @param hash
 * @return true if hash is available
```

```
 */
bool gatt_server_get_database_hash(uint8_t * hash);
```

Runtine ATT Database Setup API

att_db_util.h : Helper to construct ATT DB at runtime (BTstack GATT Compiler is not used).

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

/**
 * @brief Set handle value for next table entry
 * @param handle
 * @note new handle must be >= previous to avoid reusing assigned handles
 */
void att_db_util_set_next_handle(uint16_t handle);

/**
 * @brief Add primary service for 16-bit UUID
 * @param uuid16
 * @return attribute handle for the new service definition
 */
uint16_t att_db_util_add_service_uuid16(uint16_t uuid16);

/**
 * @brief Add primary service for 128-bit UUID
 * @param uuid128
 * @return attribute handle for the new service definition
 */
uint16_t att_db_util_add_service_uuid128(const uint8_t * uuid128);

/**
 * @brief Add secondary service for 16-bit UUID
 * @param uuid16
 * @return attribute handle for the new service definition
 */
```

```
uint16_t att_db_util_add_secondary_service_uuid16(uint16_t
uuid16);

/**
 * @brief Add secondary service for 128-bit UUID
 * @param uuid128
 * @return attribute handle for the new service definition
 */
uint16_t att_db_util_add_secondary_service_uuid128(const uint8_t *
uuid128);

/**
 * @brief Add included service with 16-bit UUID
 * @param start_group_handle
 * @param end_group_handle
 * @param uuid16
 * @return attribute handle for the new service definition
 */
uint16_t att_db_util_add_included_service_uuid16(uint16_t
start_group_handle, uint16_t  end_group_handle, uint16_t uuid16);

/**
 * @brief Add Characteristic with 16-bit UUID, properties, and
data
 * @param uuid16
 * @param properties        - see ATT_PROPERTY_* in src/
bluetooth.h
 * @param read_permissions  - see ATT_SECURITY_* in src/
bluetooth.h
 * @param write_permissions - see ATT_SECURITY_* in src/
bluetooth.h
 * @param data returned in read operations if ATT_PROPERTY_DYNAMIC
is not specified
 * @param data_len
 * @return attribute handle of the new characteristic value
declaration
 * @note If properties contains ATT_PROPERTY_NOTIFY or
ATT_PROPERTY_INDICATE flags, a Client Configuration Characteristic
Descriptor (CCCD)
 *       is created as well. The attribute value handle of the
```

```
CCCD is the attribute value handle plus 1
 */
uint16_t att_db_util_add_characteristic_uuid16(uint16_t uuid16,
uint16_t properties, uint8_t read_permission, uint8_t
write_permission, uint8_t * data, uint16_t data_len);

/**
 * @brief Add Characteristic with 128-bit UUID, properties, and
data
 * @param uuid128
 * @param properties        - see ATT_PROPERTY_* in src/
bluetooth.h
 * @param read_permissions  - see ATT_SECURITY_* in src/
bluetooth.h
 * @param write_permissions - see ATT_SECURITY_* in src/
bluetooth.h
 * @param data returned in read operations if ATT_PROPERTY_DYNAMIC
is not specified
 * @param data_len
 * @return attribute handle of the new characteristic value
declaration
 * @note If properties contains ATT_PROPERTY_NOTIFY or
ATT_PROPERTY_INDICATE flags, a Client Configuration Characteristic
Descriptor (CCCD)
 *       is created as well. The attribute value handle of the
CCCD is the attribute value handle plus 1
 */
uint16_t att_db_util_add_characteristic_uuid128(const uint8_t *
uuid128, uint16_t properties, uint8_t read_permission, uint8_t
write_permission, uint8_t * data, uint16_t data_len);

/**
* @brief Add descriptor with 16-bit UUID, properties, and data
* @param uuid16
* @param properties        - see ATT_PROPERTY_* in src/bluetooth.h
* @param read_permissions  - see ATT_SECURITY_* in src/bluetooth.h
* @param write_permissions - see ATT_SECURITY_* in src/bluetooth.h
* @param data returned in read operations if ATT_PROPERTY_DYNAMIC
is not specified
* @param data_len
```

```
* @return attribute handle of the new characteristic descriptor
declaration
*/
uint16_t att_db_util_add_descriptor_uuid16(uint16_t uuid16,
uint16_t properties, uint8_t read_permission, uint8_t
write_permission, uint8_t * data, uint16_t data_len);

/**
* @brief Add descriptor with 128-bit UUID, properties, and data
* @param uuid128
* @param properties        - see ATT_PROPERTY_* in src/bluetooth.h
* @param read_permissions  - see ATT_SECURITY_* in src/bluetooth.h
* @param write_permissions - see ATT_SECURITY_* in src/bluetooth.h
* @param data returned in read operations if ATT_PROPERTY_DYNAMIC
is not specified
* @param data_len
* @return attribute handle of the new characteristic descriptor
declaration
*/
uint16_t att_db_util_add_descriptor_uuid128(const uint8_t *
uuid128, uint16_t properties, uint8_t read_permission, uint8_t
write_permission, 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);

/**
 * @brief Get number of bytes that are included in GATT Database
Hash
 */
uint16_t att_db_util_hash_len(void);

/**
```

```
 * @brief init generator for GATT Database Hash
 */
void att_db_util_hash_init(void);

/**
 * @brief get next byte from generator for GATT Database Hash
 */
uint8_t att_db_util_hash_get_next(void);

/**
 * @brief Calculate GATT Database Hash using crypto engine
 * @param request
 * @param db_hash
 * @param callback
 * @param callback_arg
 */
void att_db_util_hash_calc(btstack_crypto_aes128_cmac_t * request,
uint8_t * db_hash, void (* callback)(void * arg), void *
callback_arg);
```

ATT Dispatch API

att_dispatch.h : Dispatcher for independent implementation of ATT client and server.

```
/**
 * @brief Register packet handler for ATT Client PDUs
 * @param packet_handler
 */
void att_dispatch_register_client(btstack_packet_handler_t
packet_handler);

/**
 * @brief Register packet handler for ATT Server PDUs
 * @param packet_handler
 */
void att_dispatch_register_server(btstack_packet_handler_t
packet_handler);

/**
```

```
 * @brief Register L2CAP ATT Service with fixed PSM
 * @note requires ENABLE_GATT_OVER_CLASSIC
 */
void att_dispatch_classic_register_service(void);

/**
 * @brief Create L2CAP ATT Connection
 * @note ENABLE_GATT_OVER_CLASSIC
 * @param address
 * @param l2cap_psm
 */
uint8_t att_dispatch_classic_connect(bd_addr_t address, uint16_t
l2cap_psm, uint16_t *out_cid);

/**
 * @brief can send packet for client
 * @param con_handle
 */
bool att_dispatch_client_can_send_now(hci_con_handle_t
con_handle);

/**
 * @brief can send packet for server
 * @param con_handle
 */
bool att_dispatch_server_can_send_now(hci_con_handle_t
con_handle);

/**
 * @brief Request emission of L2CAP_EVENT_CAN_SEND_NOW as soon as
possible for client
 * @note L2CAP_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_dispatch_client_request_can_send_now_event(hci_con_handle_t
con_handle);
```

```
/**
 * @brief Request emission of L2CAP_EVENT_CAN_SEND_NOW as soon as
possible for server
 * @note L2CAP_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_dispatch_server_request_can_send_now_event(hci_con_handle_t
con_handle);

/**
* @brief Used for propogating a updated MTU from att_server to
gatt_client
* @param con_handle
* @param mtu
*/
void att_dispatch_server_mtu_exchanged(hci_con_handle_t
con_handle, uint16_t new_mtu);

/**
* @brief Used for propogating a updated MTU from gatt_client to
att_server
* @param con_handle
* @param mtu
*/
void att_dispatch_client_mtu_exchanged(hci_con_handle_t
con_handle, uint16_t new_mtu);
```

ATT Server API

att_server.h

```
/**
 * @title ATT Server
 *
 * @TLV 'BTC' x
 */
```

```
/*
 * @brief setup ATT server
 * @param db attribute database created by compile-gatt.py
 * @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 Enable support for Enhanced ATT bearer
 * @note Requires ENABLE_GATT_OVER_EATT
 * @param num_eatt_bearers
 * @param storage_buffer
 * @param storage_size must be >= num_eatt_bearers *
sizeof(att_server_eatt_bearer_t)
 * @return status   ERROR_CODE_SUCCESS
 *                  ERROR_CODE_MEMORY_CAPACITY_EXCEEDED if buffer
too small or no entry in l2cap service pool
 *                  L2CAP_SERVICE_ALREADY_REGISTERED if called
twice
 */
uint8_t att_server_eatt_init(uint8_t num_eatt_bearers, uint8_t *
storage_buffer, uint16_t storage_size);

/*
 * @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 register read/write callbacks for specific handle range
 * @param att_service_handler_t
 */
void att_server_register_service_handler(att_service_handler_t *
```

```
handler);

/**
 * @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
 * @return 0 if ok, error otherwise
 */
uint8_t
att_server_register_can_send_now_callback(btstack_context_callback_
* callback_registration, hci_con_handle_t con_handle);

/**
 * @brief Return ATT MTU
 * @param con_handle
 * @return mtu if ok, 0 otherwise
 */
uint16_t att_server_get_mtu(hci_con_handle_t con_handle);

/**
 * @brief Request callback when sending notifcation 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
 * @return ERROR_CODE_SUCCESS if ok,
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if handle unknown, and
ERROR_CODE_COMMAND_DISALLOWED if callback already registered
 */
uint8_t
att_server_request_to_send_notification(btstack_context_callback_re
* callback_registration, hci_con_handle_t con_handle);

/**
 * @brief Request callback when sending indication 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
 * @return ERROR_CODE_SUCCESS if ok,
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if handle unknown, and
ERROR_CODE_COMMAND_DISALLOWED if callback already registered
 */
uint8_t
att_server_request_to_send_indication(btstack_context_callback_regi
* 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
 */
uint8_t att_server_notify(hci_con_handle_t con_handle, uint16_t
attribute_handle, const uint8_t *value, uint16_t value_len);

/**
 * @brief notify client about multiple attribute value changes
 * @param con_handle
 * @param num_attributes
 * @param attribute_handles[]
 * @param values_data[]
 * @param values_len[]
 * @return 0 if ok, error otherwise
 */
uint8_t att_server_multiple_notify(hci_con_handle_t con_handle,
uint8_t num_attributes,
                                   const uint16_t *
attribute_handles, const uint8_t ** values_data, const uint16_t *
values_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
 */
uint8_t att_server_indicate(hci_con_handle_t con_handle, uint16_t
attribute_handle, const uint8_t *value, uint16_t value_len);

#ifdef ENABLE_ATT_DELAYED_RESPONSE
/**
 * @brief response ready - called after returning
ATT_READ__RESPONSE_PENDING in an att_read_callback or
 * ATT_ERROR_WRITE_REQUEST_PENDING IN att_write_callback before to
trigger callback again and complete the transaction
 * @nore The ATT Server will retry handling the current ATT
request
 * @param con_handle
 * @return 0 if ok, error otherwise
 */
uint8_t att_server_response_ready(hci_con_handle_t con_handle);
#endif

/**
 * De-Init ATT Server
 */
void att_server_deinit(void);

// the following functions will be removed soon

/**
 * @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);
// end of deprecated functions
```

ANCS Client API

ancs_client.h

```
void ancs_client_init(void);
void ancs_client_register_callback(btstack_packet_handler_t
callback);
const char * ancs_client_attribute_name_for_id(uint16_t id);
```

Battery Service Client API

battery_service_client.h

```
/**
 * @brief Initialize Battery Service.
 */
void battery_service_client_init(void);

/**
 * @brief Connect to Battery Services of remote device. The client
will try to register for notifications.
 * If notifications are not supported by remote Battery Service,
the client will poll battery level
 * If poll_interval_ms is 0, polling is disabled, and only
notifications will be received.
 * In either case, the battery level is received via
GATTSERVICE_SUBEVENT_BATTERY_SERVICE_LEVEL event.
 * The battery level is reported as percentage, i.e. 100 = full
and it is valid if the ATT status is equal to ATT_ERROR_SUCCESS,
 * see ATT errors (see bluetooth.h) for other values.
 *
```

```
 * For manual polling, see
battery_service_client_read_battery_level below.
 *
 * Event GATTSERVICE_SUBEVENT_BATTERY_SERVICE_CONNECTED is emitted
with status ERROR_CODE_SUCCESS on success, otherwise
 * GATT_CLIENT_IN_WRONG_STATE,
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no battery
service is found, or ATT errors (see bluetooth.h).
 * This event event also returns number of battery instances found
on remote server, as well as poll bitmap that indicates which
indexes
 * of services require polling, i.e. they do not support
notification on battery level change,
 *
 * @param con_handle
 * @param packet_handler
 * @param poll_interval_ms or 0 to disable polling
 * @param battery_service_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED if there is already a client
associated with con_handle, or BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t battery_service_client_connect(hci_con_handle_t
con_handle, btstack_packet_handler_t packet_handler, uint32_t
poll_interval_ms, uint16_t * battery_service_cid);

/**
 * @brief Read battery level for service with given index. Event
GATTSERVICE_SUBEVENT_BATTERY_SERVICE_LEVEL is
 * received with battery level (unit is in percentage, i.e. 100 =
full). The battery level is valid if the ATTT status
 * is equal to ATT_ERROR_SUCCESS, see ATT errors (see bluetooth.h)
for other values.
 * @param battery_service_cid
 * @param service_index
 * @return status
 */
uint8_t battery_service_client_read_battery_level(uint16_t
battery_service_cid, uint8_t service_index);
```

```
/**
 * @brief Disconnect from Battery Service.
 * @param battery_service_cid
 * @return status
 */
uint8_t battery_service_client_disconnect(uint16_t
battery_service_cid);

/**
 * @brief De-initialize Battery Service.
 */
void battery_service_client_deinit(void);
```

Battery Service Server API

battery_service_server.h

```
/**
 * @brief Init Battery Service Server with ATT DB
 * @param battery_value in range 0-100
 */
void battery_service_server_init(uint8_t battery_value);

/**
 * @brief Update battery value
 * @note triggers notifications if subscribed
 * @param battery_value in range 0-100
 */
void battery_service_server_set_battery_value(uint8_t
battery_value);
```

Battery Service Server v1.1 API

battery_service_v1_server.h

```
/**
 * @brief Init Battery Service Server with ATT DB
 */
void battery_service_v1_server_init(void);
```

```
/**
 * @brief Register Battery Service
 * @param service
 * @param connections array of
battery_service_v1_server_connection_t for storage
 * @param connection_max_num
 * @param out_service_id
 */
void battery_service_v1_server_register(battery_service_v1_t *
service, battery_service_v1_server_connection_t * connections,
uint8_t connection_max_num,  uint16_t * out_service_id);

/**
 * @brief Deregister Battery Service Server
 * @param service
 */
void battery_service_v1_server_deregister(battery_service_v1_t *
service);

/**
 * @brief Set callback for broadcast updates.
 * @param callback
 */
void
battery_service_v1_server_set_packet_handler(btstack_packet_handler
callback);

/**
 * @brief Update battery level
 * @note Triggers notification if subscribed
 * @param service
 * @param battery_level in range 0-100
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_battery_level(battery_service_v1_t *
service,uint8_t battery_level);
```

```
/**
 * @brief Update battery level status
 * @note Triggers notification or indication if subscribed
 * @param service
 * @param battery_level_status in range 0-100
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_battery_level_status(battery_service_
* service, const battery_level_status_t * battery_level_status);

/**
 * @brief Update battery estimated service date as days elapsed
since the Epoch (Jan 1, 1970)
 * @note Triggers notification or indication if subscribed
 * @param service
 * @param estimated_service_date_days
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_estimated_service_date_days(battery_s
* service, uint32_t estimated_service_date_days);

/**
 * @brief Update battery critcal status flags
 * @note Triggers indication if subscribed
 * @param service
 * @param critcal_status_flags
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_critcal_status_flags(battery_service_
* service, uint8_t critcal_status_flags);

/**
 * @brief Update battery energy status
 * @note Triggers notification or indication if subscribed
```

```
 * @param service
 * @param energy_status
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_energy_status(battery_service_v1_t *
service, const battery_energy_status_t * energy_status);

/**
 * @brief Update battery time status
 * @note Triggers notification or indication if subscribed
 * @param service
 * @param time_status
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_time_status(battery_service_v1_t *
service, const battery_time_status_t * time_status);

/**
 * @brief Update battery health status
 * @note Triggers notification or indication if subscribed
 * @param service
 * @param health_status
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_health_status(battery_service_v1_t *
service, const battery_health_status_t * health_status);

/**
 * @brief Update battery health information
 * @note Triggers indication if subscribed
 * @param service
 * @param health_information
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
```

```
 */
uint8_t
battery_service_v1_server_set_health_information(battery_service_v1
* service, const battery_health_information_t *
health_information);

/**
 * @brief Update battery information
 * @note Triggers indication if subscribed
 * @param service
 * @param information
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_information(battery_service_v1_t *
service, const battery_information_t * information);

/**
 * @brief Update manufacturer name
 * @note Triggers indication if subscribed
 * @param service
 * @param manufacturer_name
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_manufacturer_name(battery_service_v1_
* service, const char * manufacturer_name);

/**
 * @brief Update model_number
 * @note Triggers indication if subscribed
 * @param service
 * @param model number
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_model_number(battery_service_v1_t *
```

```
service, const char * model_number);

/**
 * @brief Update serial_number
 * @note Triggers indication if subscribed
 * @param service
 * @param serial number
 * @return ERROR_CODE_SUCCESS if value is valid, otherwise
ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE
 */
uint8_t
battery_service_v1_server_set_serial_number(battery_service_v1_t *
service, const char * serial_number);

/**
 * @brief Get Advertisement Data for all active Characteristic
Broadcasts
 * @param adv_interval
 * @param adv_buffer
 * @param adv_size
 * @return
 */
uint16_t
battery_service_v1_server_get_broadcast_advertisement(uint16_t
adv_interval, uint8_t * adv_buffer, uint16_t adv_size);

/**
 * @brief Get Advertisement Data for single active Characteristic
Broadcast
 * @param adv_interval
 * @param adv_buffer
 * @param adv_size
 * @return
 */
uint16_t
battery_service_v1_server_get_broadcast_advertisement_single(batter
* service, uint16_t adv_interval, uint8_t * adv_buffer, uint16_t
adv_size);

void battery_service_v1_server_deinit(void);
```

Bond Management Service Server API

bond_management_service_server.h

```
#define BOND_MANAGEMENT_CONTROL_POINT_OPCODE_NOT_SUPPORTED 0x80
#define BOND_MANAGEMENT_OPERATION_FAILED 0x81

#define BMF_DELETE_ACTIVE_BOND_CLASSIC_AND_LE
0x00001
#define BMF_DELETE_ACTIVE_BOND_CLASSIC_AND_LE_WITH_AUTH
0x00002
#define BMF_DELETE_ACTIVE_BOND_CLASSIC
0x00004
#define BMF_DELETE_ACTIVE_BOND_CLASSIC_WITH_AUTH
0x00008
#define BMF_DELETE_ACTIVE_BOND_LE
0x00010
#define BMF_DELETE_ACTIVE_BOND_LE_WITH_AUTH
0x00020
#define BMF_DELETE_ALL_BONDS_CLASSIC_AND_LE
0x00040
#define BMF_DELETE_ALL_BONDS_CLASSIC_AND_LE_WITH_AUTH
0x00080
#define BMF_DELETE_ALL_BONDS_CLASSIC
0x00100
#define BMF_DELETE_ALL_BONDS_CLASSIC_WITH_AUTH
0x00200
#define BMF_DELETE_ALL_BONDS_LE
0x00400
#define BMF_DELETE_ALL_BONDS_LE_WITH_AUTH
0x00800
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC_AND_LE
0x01000
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC_AND_LE_WITH_AUTH
0x02000
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC
0x04000
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC_WITH_AUTH
0x08000
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_LE
```

```
0x10000
#define BMF_DELETE_ALL_BUT_ACTIVE_BOND_LE_WITH_AUT
0x20000

typedef enum {
    BOND_MANAGEMENT_CMD_DELETE_ACTIVE_BOND_CLASSIC_AND_LE = 0x01,
    BOND_MANAGEMENT_CMD_DELETE_ACTIVE_BOND_CLASSIC,
    BOND_MANAGEMENT_CMD_DELETE_ACTIVE_BOND_LE,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BONDS_CLASSIC_AND_LE,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BONDS_CLASSIC,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BONDS_LE,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC_AND_LE,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BUT_ACTIVE_BOND_CLASSIC,
    BOND_MANAGEMENT_CMD_DELETE_ALL_BUT_ACTIVE_BOND_LE
} bond_management_cmd_t;

/**
 * @brief Init Bond Management Service with ATT DB
 * @param supported_features
 */
void bond_management_service_server_init(uint32_t
supported_features);

/**
 * @brief Set authorisation string
 * @note String is not copied
 * @param authorisation_string
 */
void bond_management_service_server_set_authorisation_string(const
char * authorisation_string);
```

Cycling Power Service Server API

cycling_power_service_server.h

```
#define CYCLING_POWER_MANUFACTURER_SPECIFIC_DATA_MAX_SIZE   16

typedef enum {
    CP_PEDAL_POWER_BALANCE_REFERENCE_UNKNOWN = 0,
    CP_PEDAL_POWER_BALANCE_REFERENCE_LEFT,
```

```
    CP_PEDAL_POWER_BALANCE_REFERENCE_NOT_SUPPORTED
} cycling_power_pedal_power_balance_reference_t;

typedef enum {
    CP_TORQUE_SOURCE_WHEEL = 0,
    CP_TORQUE_SOURCE_CRANK,
    CP_TORQUE_SOURCE_NOT_SUPPORTED
} cycling_power_torque_source_t;

typedef enum {
    CP_SENSOR_MEASUREMENT_CONTEXT_FORCE = 0,
    CP_SENSOR_MEASUREMENT_CONTEXT_TORQUE
} cycling_power_sensor_measurement_context_t;

typedef enum {
    CP_DISTRIBUTED_SYSTEM_UNSPECIFIED = 0,
    CP_DISTRIBUTED_SYSTEM_NOT_SUPPORTED,
    CP_DISTRIBUTED_SYSTEM_SUPPORTED
} cycling_power_distributed_system_t;

typedef enum {
    CP_MEASUREMENT_FLAG_PEDAL_POWER_BALANCE_PRESENT = 0,
    CP_MEASUREMENT_FLAG_PEDAL_POWER_BALANCE_REFERENCE, // 0 -
unknown, 1 - left
    CP_MEASUREMENT_FLAG_ACCUMULATED_TORQUE_PRESENT,
    CP_MEASUREMENT_FLAG_ACCUMULATED_TORQUE_SOURCE, // 0 - wheel
based, 1 - crank based
    CP_MEASUREMENT_FLAG_WHEEL_REVOLUTION_DATA_PRESENT,
    CP_MEASUREMENT_FLAG_CRANK_REVOLUTION_DATA_PRESENT,
    CP_MEASUREMENT_FLAG_EXTREME_FORCE_MAGNITUDES_PRESENT,
    CP_MEASUREMENT_FLAG_EXTREME_TORQUE_MAGNITUDES_PRESENT,
    CP_MEASUREMENT_FLAG_EXTREME_ANGLES_PRESENT,
    CP_MEASUREMENT_FLAG_TOP_DEAD_SPOT_ANGLE_PRESENT,
    CP_MEASUREMENT_FLAG_BOTTOM_DEAD_SPOT_ANGLE_PRESENT,
    CP_MEASUREMENT_FLAG_ACCUMULATED_ENERGY_PRESENT,
    CP_MEASUREMENT_FLAG_OFFSET_COMPENSATION_INDICATOR,
    CP_MEASUREMENT_FLAG_RESERVED
} cycling_power_measurement_flag_t;

typedef enum {
```

```
    CP_INSTANTANEOUS_MEASUREMENT_DIRECTION_UNKNOWN = 0,
    CP_INSTANTANEOUS_MEASUREMENT_DIRECTION_TANGENTIAL_COMPONENT,
    CP_INSTANTANEOUS_MEASUREMENT_DIRECTION_RADIAL_COMPONENT,
    CP_INSTANTANEOUS_MEASUREMENT_DIRECTION_LATERAL_COMPONENT
} cycling_power_instantaneous_measurement_direction_t;

typedef enum {
    CP_VECTOR_FLAG_CRANK_REVOLUTION_DATA_PRESENT = 0,
    CP_VECTOR_FLAG_FIRST_CRANK_MEASUREMENT_ANGLE_PRESENT,
    CP_VECTOR_FLAG_INSTANTANEOUS_FORCE_MAGNITUDE_ARRAY_PRESENT,
    CP_VECTOR_FLAG_INSTANTANEOUS_TORQUE_MAGNITUDE_ARRAY_PRESENT,
    CP_VECTOR_FLAG_INSTANTANEOUS_MEASUREMENT_DIRECTION = 4, // 2
bit
    CP_VECTOR_FLAG_RESERVED = 6
} cycling_power_vector_flag_t;

typedef enum {
    CP_SENSOR_LOCATION_OTHER,
    CP_SENSOR_LOCATION_TOP_OF_SHOE,
    CP_SENSOR_LOCATION_IN_SHOE,
    CP_SENSOR_LOCATION_HIP,
    CP_SENSOR_LOCATION_FRONT_WHEEL,
    CP_SENSOR_LOCATION_LEFT_CRANK,
    CP_SENSOR_LOCATION_RIGHT_CRANK,
    CP_SENSOR_LOCATION_LEFT_PEDAL,
    CP_SENSOR_LOCATION_RIGHT_PEDAL,
    CP_SENSOR_LOCATION_FRONT_HUB,
    CP_SENSOR_LOCATION_REAR_DROPOUT,
    CP_SENSOR_LOCATION_CHAINSTAY,
    CP_SENSOR_LOCATION_REAR_WHEEL,
    CP_SENSOR_LOCATION_REAR_HUB,
    CP_SENSOR_LOCATION_CHEST,
    CP_SENSOR_LOCATION_SPIDER,
    CP_SENSOR_LOCATION_CHAIN_RING,
    CP_SENSOR_LOCATION_RESERVED
} cycling_power_sensor_location_t;

typedef enum {
    CP_FEATURE_FLAG_PEDAL_POWER_BALANCE_SUPPORTED = 0,
    CP_FEATURE_FLAG_ACCUMULATED_TORQUE_SUPPORTED,
```

```
    CP_FEATURE_FLAG_WHEEL_REVOLUTION_DATA_SUPPORTED,
    CP_FEATURE_FLAG_CRANK_REVOLUTION_DATA_SUPPORTED,
    CP_FEATURE_FLAG_EXTREME_MAGNITUDES_SUPPORTED,
    CP_FEATURE_FLAG_EXTREME_ANGLES_SUPPORTED,
    CP_FEATURE_FLAG_TOP_AND_BOTTOM_DEAD_SPOT_ANGLE_SUPPORTED,
    CP_FEATURE_FLAG_ACCUMULATED_ENERGY_SUPPORTED,
    CP_FEATURE_FLAG_OFFSET_COMPENSATION_INDICATOR_SUPPORTED,
    CP_FEATURE_FLAG_OFFSET_COMPENSATION_SUPPORTED,

CP_FEATURE_FLAG_CYCLING_POWER_MEASUREMENT_CHARACTERISTIC_CONTENT_MA
    CP_FEATURE_FLAG_MULTIPLE_SENSOR_LOCATIONS_SUPPORTED,
    CP_FEATURE_FLAG_CRANK_LENGTH_ADJUSTMENT_SUPPORTED,
    CP_FEATURE_FLAG_CHAIN_LENGTH_ADJUSTMENT_SUPPORTED,
    CP_FEATURE_FLAG_CHAIN_WEIGHT_ADJUSTMENT_SUPPORTED,
    CP_FEATURE_FLAG_SPAN_LENGTH_ADJUSTMENT_SUPPORTED,
    CP_FEATURE_FLAG_SENSOR_MEASUREMENT_CONTEXT, // 0-force based,
1-torque based
    CP_FEATURE_FLAG_INSTANTANEOUS_MEASUREMENT_DIRECTION_SUPPORTED,
    CP_FEATURE_FLAG_FACTORY_CALIBRATION_DATE_SUPPORTED,
    CP_FEATURE_FLAG_ENHANCED_OFFSET_COMPENSATION_SUPPORTED,
    CP_FEATURE_FLAG_DISTRIBUTED_SYSTEM_SUPPORT = 20,  // 0-
unspecified, 1-not for use in distr. system, 2-used in distr.
system, 3-reserved
    CP_FEATURE_FLAG_RESERVED = 22
} cycling_power_feature_flag_t;

typedef enum {
    CP_CALIBRATION_STATUS_INCORRECT_CALIBRATION_POSITION = 0x01,
    CP_CALIBRATION_STATUS_MANUFACTURER_SPECIFIC_ERROR_FOLLOWS =
0xFF
} cycling_power_calibration_status_t;


/**
 * @brief Init Server with ATT DB
 *
 * @param feature_flags
 * @param pedal_power_balance_reference
 * @param torque_source
 * @param supported_sensor_locations
```

```
 * @param num_supported_sensor_locations
 * @param current_sensor_location
 */
void cycling_power_service_server_init(uint32_t feature_flags,

cycling_power_pedal_power_balance_reference_t
pedal_power_balance_reference, cycling_power_torque_source_t
torque_source,

cycling_power_sensor_location_t * supported_sensor_locations,
uint16_t num_supported_sensor_locations,

cycling_power_sensor_location_t current_sensor_location);

/**
 * @brief Setup measurement as advertisement data
 * @param adv_interval
 * @param adv_buffer
 * @param adv_size
 * @return
 */
int cycling_power_get_measurement_adv(uint16_t adv_interval,
uint8_t * adv_buffer, uint16_t adv_size);

/**
 * @brief Register callback for the calibration and broadcast
updates.
 *
 * @param callback
 */
void
cycling_power_service_server_packet_handler(btstack_packet_handler_
callback);

/**
 * @brief Report calibration done.
 * @param measurement_type
 * @param calibrated_value
 */
void
```

```
cycling_power_server_calibration_done(cycling_power_sensor_measurem
measurement_type, uint16_t calibrated_value);

/**
 * @brief Report enhanced calibration done.
 *
 * @param measurement_type
 * @param calibrated_value
 * @param manufacturer_company_id
 * @param num_manufacturer_specific_data
 * @param manufacturer_specific_data
 */
void
cycling_power_server_enhanced_calibration_done(cycling_power_sensor
measurement_type,
                                               uint16_t
calibrated_value, uint16_t manufacturer_company_id,
                                               uint8_t
num_manufacturer_specific_data, uint8_t *
manufacturer_specific_data);

/**
 * @brief Set factory calibration date.
 * @param date
 * @return 1 if successful
 */
int
cycling_power_service_server_set_factory_calibration_date(gatt_date
date);

/**
 * @brief Set sampling rate.
 * @param sampling_rate_Hz
 */
void cycling_power_service_server_set_sampling_rate(uint8_t
sampling_rate_Hz);

/**
 * @brief Accumulate torque value.
 * @param torque_Nm
```

```
 */
void cycling_power_service_server_add_torque(int16_t torque_Nm);

/**
 * @brief Accumulate wheel revolution value and set the time of
the last measurement.
 * @param wheel_revolution
 * @param wheel_event_time_s  unit: seconds
 */
void cycling_power_service_server_add_wheel_revolution(int32_t
wheel_revolution, uint16_t wheel_event_time_s);

/**
 * @brief Accumulate crank revolution value and set the time of
the last measurement.
 * @param crank_revolution
 * @param crank_event_time_s  unit: seconds
 */
void cycling_power_service_server_add_crank_revolution(uint16_t
crank_revolution, uint16_t crank_event_time_s);

/**
 * @brief Accumulate energy.
 * @param energy_kJ  unit: kilo Joule
 */
void cycling_power_service_add_energy(uint16_t energy_kJ);

/**
 * @brief Set the value of the power. The Instantaneous Power
field represents either
 * the total power the user is producing or a part of the total
power depending on the
 * type of sensor (e.g., single sensor or distributed power sensor
system).
 *
 * @param instantaneous_power_W  unit: watt
 */
void cycling_power_service_server_set_instantaneous_power(int16_t
instantaneous_power_W);
```

```
/**
 * @brief Set the pedal power balance value. The Pedal Power
Balance field represents the ratio between
 * the total amount of power measured by the sensor and a
reference (either unknown or left).
 *
 * @param pedal_power_balance_percentage
 */
void cycling_power_service_server_set_pedal_power_balance(uint8_t
pedal_power_balance_percentage);

/**
 * @brief Set the minimum and maximum force value measured in a
single crank revolution.
 *
 * This, so called, Extreme Force Magnitude field pair will be
included in the Cycling Power Measurement
 * characteristic only if the device supports the Extreme
Magnitudes feature and
 * the Sensor Measurement Context of the Cycling Power Feature
characteristic is set to 0 (Force-based).
 *
 * @param min_force_magnitude_N  unit: newton
 * @param max_force_magnitude_N  unit: newton
 */
void cycling_power_service_server_set_force_magnitude(int16_t
min_force_magnitude_N, int16_t max_force_magnitude_N);

/**
 * @brief Set the minimum and maximum torque value measured in a
single crank revolution.
 *
 * This, so called, Extreme Torque Magnitude field pair will be
included in the Cycling Power Measurement
 * characteristic only if the device supports the Extreme
Magnitudes feature and
 * the Sensor Measurement Context of the Cycling Power Feature
characteristic is set to 1 (Torque-based).
 *
 * @param min_torque_magnitude_Nm  unit: newton meter
```

```
 * @param max_torque_magnitude_Nm  unit: newton meter
 */
void cycling_power_service_server_set_torque_magnitude(int16_t
min_torque_magnitude_Nm, int16_t max_torque_magnitude_Nm);

/**
 * @brief Set the minimum and maximum angle of the crank measured
in a single crank revolution (unit: degree).
 *
 * This, so called, Extreme Angles Magnitude field pair will be
included in the Cycling Power Measurement
 * characteristic only if the device supports the Extreme Angles
feature.
 *
 * @param min_angle_degree
 * @param max_angle_degree
 */
void cycling_power_service_server_set_angle(uint16_t
min_angle_degree, uint16_t max_angle_degree);

/**
 * @brief Set the value of the crank angle measured when the value
of the Instantaneous Power value becomes positive.
 *
 * This field will be included in the Cycling Power Measurement
characteristic
 * only if the device supports the Top and Bottom Dead Spot Angles
feature.
 *
 * @param top_dead_spot_angle_degree
 */
void cycling_power_service_server_set_top_dead_spot_angle(uint16_t
top_dead_spot_angle_degree);

/**
 * @brief Set the value of the crank angle measured when the value
of the Instantaneous Power value becomes negative.
 *
 * This field will be included in the Cycling Power Measurement
characteristic
```

```
 * only if the device supports the Top and Bottom Dead Spot Angles
feature.
 *
 * @param bottom_dead_spot_angle_degree
 */
void
cycling_power_service_server_set_bottom_dead_spot_angle(uint16_t
bottom_dead_spot_angle_degree);

/**
 * @brief Set the raw sensor force magnitude measurements.
 *
 * @param force_magnitude_count
 * @param force_magnitude_N_array  unit: newton
 */
void cycling_power_service_server_set_force_magnitude_values(int
force_magnitude_count, int16_t * force_magnitude_N_array);

/**
 * @brief Set the raw sensor torque magnitude measurements.
 *
 * @param force_magnitude_count
 * @param force_magnitude_N_array  unit: newton meter
 */
void cycling_power_service_server_set_torque_magnitude_values(int
torque_magnitude_count, int16_t * torque_magnitude_Nm_array);

/**
 * @brief Set the instantaneous measurement direction. The field
describes
 * the direction of the Instantaneous Magnitude values the Server
measures
 * (e.g., Unknown, Tangential Component, Radial Component, or
Lateral Component).
 *
 * @param direction
 */
void
cycling_power_service_server_set_instantaneous_measurement_directic
direction);
```

```
/**
 * Set first crank measurement angle. The value will be present
only in the first packet of
 * the Cycling Power Vector notification.
 *
 * @param first_crank_measurement_angle_degree
 */
void
cycling_power_service_server_set_first_crank_measurement_angle(uint
first_crank_measurement_angle_degree);

/**
 * Get measurement flags bitmask.
 * @return measurement_flags_bitmask, see
cycling_power_measurement_flag_t
 */
uint16_t cycling_power_service_measurement_flags(void);

/**
 * Get vector flags bitmask.
 * @return vector_flags_bitmask, see cycling_power_vector_flag_t
 */
uint8_t  cycling_power_service_vector_flags(void);

/**
 * @brief Trigger notification if subscribed
 */
void cycling_power_service_server_update_values(void);
```

Cycling Speed and Cadence Service Server API

cycling_speed_and_cadence_service_server.h

```
typedef enum {
    CSC_SERVICE_SENSOR_LOCATION_OTHER = 0,
    CSC_SERVICE_SENSOR_LOCATION_TOP_OF_SHOE,
    CSC_SERVICE_SENSOR_LOCATION_IN_SHOE,
```

```
    CSC_SERVICE_SENSOR_LOCATION_HIP,
    CSC_SERVICE_SENSOR_LOCATION_FRONT_WHEEL,
    CSC_SERVICE_SENSOR_LOCATION_LEFT_CRANK,
    CSC_SERVICE_SENSOR_LOCATION_RIGHT_CRANK,
    CSC_SERVICE_SENSOR_LOCATION_LEFT_PEDAL,
    CSC_SERVICE_SENSOR_LOCATION_RIGHT_PEDAL,
    CSC_SERVICE_SENSOR_LOCATION_FRONT_HUB,
    CSC_SERVICE_SENSOR_LOCATION_REAR_DROPOUT,
    CSC_SERVICE_SENSOR_LOCATION_CHAINSTAY,
    CSC_SERVICE_SENSOR_LOCATION_REAR_WHEEL,
    CSC_SERVICE_SENSOR_LOCATION_REAR_HUB,
    CSC_SERVICE_SENSOR_LOCATION_CHEST,
    CSC_SERVICE_SENSOR_LOCATION_SPIDER,
    CSC_SERVICE_SENSOR_LOCATION_CHAIN_RING,
    CSC_SERVICE_SENSOR_LOCATION_RESERVED
} cycling_speed_and_cadence_sensor_location_t;

typedef enum {
    CSC_FLAG_WHEEL_REVOLUTION_DATA_SUPPORTED = 0,
    CSC_FLAG_CRANK_REVOLUTION_DATA_SUPPORTED,
    CSC_FLAG_MULTIPLE_SENSOR_LOCATIONS_SUPPORTED
} csc_feature_flag_bit_t;

typedef enum {
    CSC_OPCODE_IDLE = 0,
    CSC_OPCODE_SET_CUMULATIVE_VALUE = 1,
    CSC_OPCODE_START_SENSOR_CALIBRATION,
    CSC_OPCODE_UPDATE_SENSOR_LOCATION,
    CSC_OPCODE_REQUEST_SUPPORTED_SENSOR_LOCATIONS,
    CSC_OPCODE_RESPONSE_CODE = 16
} csc_opcode_t;

/**
 * @brief Init Server with ATT DB
 */
void cycling_speed_and_cadence_service_server_init(uint32_t
supported_sensor_locations,
    uint8_t multiple_sensor_locations_supported, uint8_t
wheel_revolution_data_supported, uint8_t
crank_revolution_data_supported);
```

```
/**
 * @brief Update heart rate (unit: beats per minute)
 * @note triggers notifications if subscribed
 */
void
cycling_speed_and_cadence_service_server_update_values(int32_t
wheel_revolutions, uint16_t last_wheel_event_time, uint16_t
crank_revolutions, uint16_t last_crank_event_time);
```

Device Information Service Client API

device_information_service_client.h

```
/**
 * @brief Initialize Device Information Service.
 */
void device_information_service_client_init(void);

/**
 * @brief Query Device Information Service. The client will query
the remote service and emit events:
 *
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_MANUFACTURER_NAME
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_MODEL_NUMBER
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_SERIAL_NUMBER
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_HARDWARE_REVISION
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_FIRMWARE_REVISION
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_SOFTWARE_REVISION
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_SYSTEM_ID
 * -
GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_IEEE_REGULATORY_CERTIFICATI
 * - GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_PNP_ID
 * -
GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_UDI_FOR_MEDICAL_DEVICES
 *
 * Event GATTSERVICE_SUBEVENT_DEVICE_INFORMATION_DONE is received
when all queries are done, of if service was not found.
 * The status field of this event indicated ATT errors (see
bluetooth.h).
```

```
 *
 * @param con_handle
 * @param packet_handler
 * @return status ERROR_CODE_SUCCESS on success, otherwise
GATT_CLIENT_IN_WRONG_STATE if query is already in progress
 */
uint8_t device_information_service_client_query(hci_con_handle_t
con_handle, btstack_packet_handler_t packet_handler);


/**
 * @brief De-initialize Device Information Service.
 */
void device_information_service_client_deinit(void);
```

Device Information Service Server API

device_information_service_server.h

```
/**
 * @text Th eDevice Information Service allows to query
manufacturer and/or
 * vendor information about a device.
 *
 * To use with your application, add `#import
<device_information_service.gatt>` to your .gatt file.
 *
 * *Note*: instead of calling all setters, you can create a local
copy of the .gatt file and remove
 * all Characteristics that are not relevant for your application
and define all fixed values in the .gatt file.
 */

/**
 * @brief Init Device Information Service Server with ATT DB
 * @param battery_value in range 0-100
 */
void device_information_service_server_init(void);

/**
```

```
 * @brief Set Manufacturer Name
 * @param manufacturer_name
 */
void device_information_service_server_set_manufacturer_name(const
char * manufacturer_name);

/**
 * @brief Set Model Number
 * @param model_number
 */
void device_information_service_server_set_model_number(const char
* model_number);

/**
 * @brief Set Serial Number
 * @param serial_number
 */
void device_information_service_server_set_serial_number(const
char * serial_number);

/**
 * @brief Set Hardware Revision
 * @param hardware_revision
 */
void device_information_service_server_set_hardware_revision(const
char * hardware_revision);

/**
 * @brief Set Firmware Revision
 * @param firmware_revision
 */
void device_information_service_server_set_firmware_revision(const
char * firmware_revision);

/**
 * @brief Set Software Revision
 * @param software_revision
 */
void device_information_service_server_set_software_revision(const
char * software_revision);
```

```
/**
 * @brief Set System ID
 * @param manufacturer_identifier uint40
 * @param organizationally_unique_identifier uint24
 */
void device_information_service_server_set_system_id(uint64_t
manufacturer_identifier, uint32_t
organizationally_unique_identifier);

/**
 * @brief Set IEEE 11073-20601 regulatory certification data list
 * @note: format duint16. duint16 is two uint16 values
concatenated together.
 * @param value_a
 * @param value_b
 */
void
device_information_service_server_set_ieee_regulatory_certificatior
value_a, uint16_t value_b);

/**
 * @brief Set PnP ID
 * @param vendor_source_id
 * @param vendor_id
 * @param product_id
 * @param product_version
 */
void device_information_service_server_set_pnp_id(uint8_t
vendor_source_id, uint16_t vendor_id, uint16_t product_id,
uint16_t product_version);

/**
 * @brief Set UDI for medical devices
 * @param label
 * @param device_id
 * @param issuer
 * @param authority
 */
void
```

```
device_information_service_server_set_udi_for_medical_devices(const
char * label, const char * device_id, const char * issuer, const
char * authority);
```

Heart Rate Service Server API

heart_rate_service_server.h

```
typedef enum {
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_OTHER = 0,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_CHEST,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_WRIST,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_FINGER,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_HAND,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_EAR_LOBE,
    HEART_RATE_SERVICE_BODY_SENSOR_LOCATION_FOOT
} heart_rate_service_body_sensor_location_t;

typedef enum {
    HEART_RATE_SERVICE_SENSOR_CONTACT_UNKNOWN = 0,
    HEART_RATE_SERVICE_SENSOR_CONTACT_UNSUPPORTED,
    HEART_RATE_SERVICE_SENSOR_CONTACT_NO_CONTACT,
    HEART_RATE_SERVICE_SENSOR_CONTACT_HAVE_CONTACT
} heart_rate_service_sensor_contact_status_t;



/**
 * @brief Init Battery Service Server with ATT DB
 * @param body_sensor_location
 * @param energy_expended_supported
 */
void
heart_rate_service_server_init(heart_rate_service_body_sensor_locat
body_sensor_location, int energy_expended_supported);



/**
 * @brief Add Energy Expended to the internal accumulator.
 * @param energy_expended_kJ   energy expended in kilo Joules
since the last update
```

```
 */
void heart_rate_service_add_energy_expended(uint16_t
energy_expended_kJ);

/**
 * @brief Update heart rate (unit: beats per minute)
 * @note triggers notifications if subscribed
 * @param heart_rate_bpm        beats per minute
 * @param contact
 * @param rr_interval_count
 * @param rr_intervals      resolution in 1/1024 seconds
 */
void heart_rate_service_server_update_heart_rate_values(uint16_t
heart_rate_bpm,
    heart_rate_service_sensor_contact_status_t contact, int
rr_interval_count, uint16_t * rr_intervals);
```

HID Service Server API

hids_device.h

```
typedef struct {
    uint16_t value_handle;
    uint16_t client_configuration_handle;
    uint16_t client_configuration_value;

    hid_report_type_t type;
    uint8_t id;
    uint8_t size;
} hids_device_report_t;

/**
 * @text Implementation of the GATT HIDS Device
 * To use with your application, add '#import <hids.gatt>' to
your .gatt file
 */

/**
 * @brief Set up HIDS Device with single INPUT, OUTPUT and FEATURE
report
```

```
 */
void hids_device_init(uint8_t hid_country_code, const uint8_t *
hid_descriptor, uint16_t hid_descriptor_size);

/**
 * @brief Set up HIDS Device for multiple instances of INPUT,
OUTPUT and FEATURE reports
 */
void hids_device_init_with_storage(uint8_t hid_country_code, const
uint8_t * hid_descriptor, uint16_t hid_descriptor_size,
    uint16_t num_reports, hids_device_report_t * report_storage);

/**
 * @brief Register callback for the HIDS Device client.
 * @param callback
 */
void hids_device_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Register report callback for Get Report operation
 * @param callback
 */
void hids_device_register_get_report_callback(void (*callback)
(hci_con_handle_t con_handle, hid_report_type_t report_type,
uint16_t report_id, uint16_t max_report_size, uint8_t *
out_report));

/**
 * @brief Request can send now event to send HID Report
 * Generates an HIDS_SUBEVENT_CAN_SEND_NOW subevent
 * @param hid_cid
 * @return ERROR_CODE_SUCCESS if ok,
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if handle unknown, and
ERROR_CODE_COMMAND_DISALLOWED if callback already registered
 */
uint8_t hids_device_request_can_send_now_event(hci_con_handle_t
con_handle);

/**
```

```
 * @brief Send HID Input Report for Report ID
 * @param con_handle
 * @param report_id
 * @param report
 * @param report_len
 * @returns status
 */
uint8_t hids_device_send_input_report_for_id(hci_con_handle_t
con_handle, uint16_t report_id, const uint8_t * report, uint16_t
report_len);

/**
 * @brief Send HID Input Report for first Input Report
 * @param con_handle
 * @param report
 * @param report_len
 * @returns status
 */
uint8_t hids_device_send_input_report(hci_con_handle_t con_handle,
const uint8_t * report, uint16_t report_len);

/**
 * @brief Send HID Boot Mouse Input Report
 * @param con_handle
 * @param report
 * @param report_len
 * @returns status
 */
uint8_t hids_device_send_boot_mouse_input_report(hci_con_handle_t
con_handle, const uint8_t * report, uint16_t report_len);

/**
 * @brief Send HID Boot Mouse Input Report
 * @param con_handle
 * @param report
 * @param report_len
 * @returns status
 */
uint8_t
```

```
hids_device_send_boot_keyboard_input_report(hci_con_handle_t
con_handle, const uint8_t * report, uint16_t report_len);
```

HID Service Client - HIDS Host API

hids_host.h

```
/**
 * @brief Initialize HID Service Host. The HID Descriptor storage
is shared between all connections.
 *
 * @param hid_descriptor_storage
 * @param hid_descriptor_storage_len
 */
void hids_host_init(uint8_t * hid_descriptor_storage, uint16_t
hid_descriptor_storage_len);

/**
 * @brief Connect to HID Services of remote device. Event
GATTSERVICE_SUBEVENT_HID_SERVICE_CONNECTED will be emitted
 * after all remote HID services and characteristics are found,
and notifications for all input reports are enabled.
 * Status code can be ERROR_CODE_SUCCES if at least one HID
service is found, otherwise either ATT errors or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no service
or report map or report are found.
 * It also contains the number of individual HIDS Services.
 *
 * Connection state is stored in hids_host_t objects provided via
memory pool
 * Max number of connections is limited by MAX_NR_HIDS_HOSTS
unless HAVE_MALLOC is used
 *
 * @param con_handle
 * @param packet_handler
 * @param protocol_mode see hid_protocol_mode_t in btstack_hid.h
 * @param hids_cid (out) to use for other functions
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED if there is already a client
 *         associated with con_handle, or
```

```
BTSTACK_MEMORY_ALLOC_FAILED.
 */
uint8_t hids_host_connect(hci_con_handle_t con_handle,
btstack_packet_handler_t packet_handler, hid_protocol_mode_t
protocol_mode, uint16_t * hids_cid);

/**
 * @brief Send HID report. Event
GATTSERVICE_SUBEVENT_HID_REPORT_WRITTEN is emitted.
 *
 * @param hids_cid
 * @param report_id
 * @param report_type
 * @param report
 * @param report_len
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state,
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found, or
 * ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE if report length
exceeds MTU.
 */
uint8_t hids_host_send_write_report(uint16_t hids_cid, uint8_t
report_id, hid_report_type_t report_type, const uint8_t * report,
uint8_t report_len);

/**
 * @brief Get HID report. Event GATTSERVICE_SUBEVENT_HID_REPORT is
emitted.
 *
 * @param hids_cid
 * @param report_id
 * @param report_type
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state,
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found, or
 * ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE if report length
```

```
exceeds MTU.
 */
uint8_t hids_host_send_get_report(uint16_t hids_cid, uint8_t
report_id, hid_report_type_t report_type);

/**
 * @brief Get HID Information. Event
GATTSERVICE_SUBEVENT_HID_INFORMATION is emitted.
 *
 * @param hids_cid
 * @param service_index
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state, or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found.
 */
uint8_t hids_host_get_hid_information(uint16_t hids_cid, uint8_t
service_index);

/**
 * @brief Get Protocol Mode. Event
GATTSERVICE_SUBEVENT_HID_PROTOCOL_MODE is emitted.
 *
 * @param hids_cid
 * @param service_index
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state, or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found.
 */
uint8_t hids_host_get_protocol_mode(uint16_t hids_cid, uint8_t
service_index);

/**
 * @brief Set Protocol Mode.
 *
 * @param hids_cid
 * @param service_index
```

```
 * @param protocol_mode
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state, or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found.
 */
uint8_t hids_host_send_set_protocol_mode(uint16_t hids_cid,
uint8_t service_index, hid_protocol_mode_t protocol_mode);

/**
 * @brief Send Suspend to remote HID service.
 *
 * @param hids_cid
 * @param service_index
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state, or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found.
 */
uint8_t hids_host_send_suspend(uint16_t hids_cid, uint8_t
service_index);

/**
 * @brief Send Exit Suspend to remote HID service.
 *
 * @param hids_cid
 * @param service_index
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state, or
 * ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no report
with given type and ID is found.
 */
uint8_t hids_host_send_exit_suspend(uint16_t hids_cid, uint8_t
service_index);

/**
 * @brief Enable all notifications. Event
```

```
GATTSERVICE_SUBEVENT_HID_SERVICE_REPORTS_NOTIFICATION reports
current configuration.
 *
 * @param hids_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER, or
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state.
 */
uint8_t hids_host_enable_notifications(uint16_t hids_cid);

/**
 * @brief Disable all notifications. Event
GATTSERVICE_SUBEVENT_HID_SERVICE_REPORTS_NOTIFICATION reports
current configuration.
 *
 * @param hids_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER, or
 * ERROR_CODE_COMMAND_DISALLOWED if client is in wrong state.
 */
uint8_t hids_host_disable_notifications(uint16_t hids_cid);

/**
 * @brief Disconnect from HID Service.
 *
 * @param hids_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER
 */
uint8_t hids_host_disconnect(uint16_t hids_cid);

/*
 * @brief Get descriptor data. For services in boot mode without a
Report Map, a default HID Descriptor for Keyboard/Mouse is
provided.
 *
 * @param hid_cid
 * @return data
 */
const uint8_t *
```

```
hids_host_descriptor_storage_get_descriptor_data(uint16_t
hids_cid, uint8_t service_index);

/*
 * @brief Get descriptor length
 *
 * @param hid_cid
 * @return length
 */
uint16_t hids_host_descriptor_storage_get_descriptor_len(uint16_t
hids_cid, uint8_t service_index);

/**
 * @brief De-initialize HID Service Client.
 *
 */
void hids_host_deinit(void);
```

Immediate Alert Service Client API

immediate_alert_service_client.h

```
/**
 * @brief Initialize Immediate Alert Service.
 */
void immediate_alert_service_client_init(void);

 /**
 * @brief Connect to a Immediate Alert Service instance of remote
device. The client will automatically register for notifications.
 * The mute state is received via
GATTSERVICE_SUBEVENT_LLS_CLIENT_MUTE event.
 * The mute state can be 0 - off, 1 - on, 2 - disabled and it is
valid if the ATT status is equal to ATT_ERROR_SUCCESS,
 * see ATT errors (see bluetooth.h) for other values.
 *
 * Event GATTSERVICE_SUBEVENT_LLS_CLIENT_CONNECTED is emitted with
status ERROR_CODE_SUCCESS on success, otherwise
 * GATT_CLIENT_IN_WRONG_STATE,
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no audio
```

```
input control service is found, or ATT errors (see bluetooth.h).
 *
 * @param con_handle
 * @param packet_handler
 * @param iac_connection
 * @param iac_characteristics_storage           storage for
characteristics
 * @param iac_characteristics_num               ==
IMMEDIATE_ALERT_SERVICE_CLIENT_NUM_CHARACTERISTICS
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED if there is already a client
associated with con_handle, or BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t immediate_alert_service_client_connect(
         hci_con_handle_t con_handle,
         btstack_packet_handler_t packet_handler,
         ias_client_connection_t * iac_connection,
         uint16_t * ias_cid
);

uint8_t immediate_alert_service_client_write_alert_level(uint16_t
ias_cid, ias_alert_level_t alert_level);



/**
 * @brief Disconnect.
 * @param ias_cid
 * @return status
 */
uint8_t immediate_alert_service_client_disconnect(uint16_t
ias_cid);

/**
 * @brief De-initialize Immediate Alert Service.
 */
void immediate_alert_service_client_deinit(void);
```

Immediate Alert Service Server API

immediate_alert_service_server.h

```
/**
 * @brief Init Immediate Alert Service Server with ATT DB
 * @param mute_state
 */
void immediate_alert_service_server_init(void);

/**
 * @brief Register packet handler
 */
void
immediate_alert_service_server_register_packet_handler(btstack_pack
packet_handler);

/**
 * @brief Set alert level.
 * @param alert_level
 */
uint8_t immediate_alert_service_server_set_alert_level(const
ias_alert_level_t alert_level);

/**
 * @brief Set alert timeout. Default is 0 that equals to no
timeout.
 * @param alert_timeout_ms
 */
uint8_t immediate_alert_service_server_set_alert_timeout(uint32_t
alert_timeout_ms);


void immediate_alert_service_server_stop_alerting(void);
```

Immediate Alert Service Util API

immediate_alert_service_util.h

Link Loss Service Client API

link_loss_service_client.h

```
/**
 * @brief Initialize Link Loss Service.
 */
void link_loss_service_client_init(void);

 /**
 * @brief Connect to a Link Loss Service instance of remote
device. The client will automatically register for notifications.
 * The mute state is received via
GATTSERVICE_SUBEVENT_LLS_CLIENT_MUTE event.
 * The mute state can be 0 - off, 1 - on, 2 - disabeled and it is
valid if the ATT status is equal to ATT_ERROR_SUCCESS,
 * see ATT errors (see bluetooth.h) for other values.
 *
 * Event GATTSERVICE_SUBEVENT_LLS_CLIENT_CONNECTED is emitted with
status ERROR_CODE_SUCCESS on success, otherwise
 * GATT_CLIENT_IN_WRONG_STATE,
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no audio
input control service is found, or ATT errors (see bluetooth.h).
 *
 * @param con_handle
 * @param packet_handler
 * @param llc_connection
 * @param llc_characteristics_storage           storage for
characteristics
 * @param llc_characteristics_num               ==
LINK_LOSS_SERVICE_CLIENT_NUM_CHARACTERISTICS
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED if there is already a client
associated with con_handle, or BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t link_loss_service_client_connect(
         hci_con_handle_t con_handle,
         btstack_packet_handler_t packet_handler,
         lls_client_connection_t * llc_connection,
         gatt_service_client_characteristic_t *
```

```
llc_characteristics_storage, uint8_t llc_characteristics_num,
         uint16_t * lls_cid
);

uint8_t link_loss_service_client_read_alert_level(uint16_t
lls_cid);
uint8_t link_loss_service_client_write_alert_level(uint16_t
lls_cid, lls_alert_level_t alert_level);



/**
 * @brief Disconnect.
 * @param lls_cid
 * @return status
 */
uint8_t link_loss_service_client_disconnect(uint16_t lls_cid);

/**
 * @brief De-initialize Link Loss Service.
 */
void link_loss_service_client_deinit(void);
```

Link Loss Service Server API

link_loss_service_server.h

```
/**
 * @brief Init Link Loss Service Server with ATT DB
 * @param mute_state
 */
void link_loss_service_server_init(void);

/**
 * @brief Register packet handler
 */
void
link_loss_service_server_register_packet_handler(btstack_packet_har
packet_handler);

/**
```

```
 * @brief Set alert level. Default is no alert.
 * @param alert_level
 */
uint8_t link_loss_service_server_set_alert_level(lls_alert_level_t
alert_level);

/**
 * @brief Set alert timeout. Default is 0 that equals to no
timeout.
 * @param alert_timeout_ms
 */
uint8_t link_loss_service_server_set_alert_timeout(uint32_t
alert_timeout_ms);


void link_loss_service_server_stop_alerting(void);
```

Link Loss Service Util API

link_loss_service_util.h

Nordic SPP Service Server API

nordic_spp_service_server.h

```
/**
 * @text The Nordic SPP Service is implementation of the Nordic
SPP-like profile.
 *
 * To use with your application, add `#import
<nordic_spp_service.gatt>` to your .gatt file
 * and call all functions below. All strings and blobs need to
stay valid after calling the functions.
 */

/**
 * @brief Init Nordic SPP Service Server with ATT DB
 * @param packet_handler for events and tx data from peer as
RFCOMM_DATA_PACKET
```

```
 */
void nordic_spp_service_server_init(btstack_packet_handler_t
packet_handler);

/**
 * @brief Queue send request. When called, one packet can be send
via nordic_spp_service_send below
 * @param request
 * @param con_handle
 */
void
nordic_spp_service_server_request_can_send_now(btstack_context_call
* request, hci_con_handle_t con_handle);

/**
 * @brief Send data
 * @param con_handle
 * @param data
 * @param size
 */
int nordic_spp_service_server_send(hci_con_handle_t con_handle,
const uint8_t * data, uint16_t size);
```

Scan Parameters Service Client API

scan_parameters_service_client.h

```
/**
 * @brief Initialize Scan Parameters Service.
 */
void scan_parameters_service_client_init(void);

/**
 * @brief Set Scan Parameters Service. It will update all
connected devices.
 * @param scan_interval
 * @param scan_window
 */
void scan_parameters_service_client_set(uint16_t scan_interval,
uint16_t scan_window);
```

```
/**
 * @brief Connect to Scan Parameters Service of remote device.
 *
 * The GATTSERVICE_SUBEVENT_SCAN_PARAMETERS_SERVICE_CONNECTED
event completes the request.
 * Its status is set to ERROR_CODE_SUCCESS if remote service and
SCAN_INTERVAL_WINDOW characteristic are found.
 * *
 * Other status codes of this event:
 * - GATT_CLIENT_IN_WRONG_STATE: client in wrong state
 * - ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE: service or
characteristic not found
 * - ATT errors, see bluetooth.h
 *
 * @param con_handle
 * @param packet_handler
 * @param sps_connection
 * @param sps_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if client with con_handle
not found
 */
uint8_t scan_parameters_service_client_connect(
        hci_con_handle_t con_handle, btstack_packet_handler_t
packet_handler,
        sps_client_connection_t * sps_connection,
        uint16_t * sps_cid);

/**
 * @brief Disconnect from Scan Parameters Service.
 * @param scan_parameters_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if client with con_handle
is not found
 */
uint8_t scan_parameters_service_client_disconnect(uint16_t
scan_parameters_cid);
```

```
/**
 * @brief De-initialize Scan Parameters Service.
 */
void scan_parameters_service_client_deinit(void);
```

Scan Parameters Service Server API

scan_parameters_service_server.h

```
/**
 * @text The Scan Parameters Service enables a remote GATT Client
to store the LE scan parameters it is using locally. These
parameters can be utilized by the application to optimize power
consumption and/or reconnection latency.
 *
 * To use with your application, add `#import
<scan_parameters_service.gatt>` to your .gatt file
 * and call all functions below. All strings and blobs need to
stay valid after calling the functions.
 */

/**
 * @brief Init Scan Parameters Service Server with ATT DB
 * @param packet_handler
 */
void scan_parameters_service_server_init(btstack_packet_handler_t
packet_handler);

/**
 * @brief Request scan parameters from Scan Parameters Client.
 */
void scan_parameters_service_server_request_scan_parameters(void);
```

Immediate Alert Service Client API

tx_power_service_client.h

```
/**
 * @brief Initialize Immediate Alert Service.
```

```
 */
void tx_power_service_client_init(void);

 /**
 * @brief Connect to a TX Power Service instance of remote device.
The client will automatically register for notifications.
 * The TX Power level state is received via
GATTSERVICE_SUBEVENT_TXPS_CLIENT_TX_POWER_LEVEL event.
 *
 * Event GATTSERVICE_SUBEVENT_TXPS_CLIENT_CONNECTED is emitted
with status ERROR_CODE_SUCCESS on success, otherwise
 * GATT_CLIENT_IN_WRONG_STATE,
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if no audio
input control service is found, or ATT errors (see bluetooth.h).
 *
 * @param con_handle
 * @param packet_handler
 * @param iac_connection
 * @param iac_characteristics_storage           storage for
characteristics
 * @param iac_characteristics_num               ==
TX_POWER_SERVICE_CLIENT_NUM_CHARACTERISTICS
 * @return status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED if there is already a client
associated with con_handle, or BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t tx_power_service_client_connect(
         hci_con_handle_t con_handle,
         btstack_packet_handler_t packet_handler,
         txps_client_connection_t * iac_connection,
         gatt_service_client_characteristic_t *
iac_characteristics_storage, uint8_t iac_characteristics_num,
         uint16_t * txps_cid
);

/**
 * @note Trigger read of TX Power Level characteristic.
 * @param txps_cid
 * @return status
 */
```

```
uint8_t tx_power_service_client_read_tx_power_level(uint16_t
txps_cid);

/**
 * @brief Disconnect.
 * @param txps_cid
 * @return status
 */
uint8_t tx_power_service_client_disconnect(uint16_t txps_cid);

/**
 * @brief De-i  nitialize Immediate Alert Service.
 */
void tx_power_service_client_deinit(void);
```

TX Power Service Server API

tx_power_service_server.h

```
/**
 * @brief Init TX Power Service Server with ATT DB
 * @param tx_power_level
 */
void tx_power_service_server_init(int8_t tx_power_level);

/**
 * @brief Update TX power level
 * @param tx_power_level_dBm range [-100,20]
 */
void tx_power_service_server_set_level(int8_t tx_power_level_dBm);
```

u-blox SPP Service Server API

ublox_spp_service_server.h

```
/**
 * @text The u-blox SPP Service is implementation of the u-Blox
SPP-like profile.
 *
```

```
 * To use with your application, add `#import
<ublox_spp_service.gatt>` to your .gatt file
 * and call all functions below. All strings and blobs need to
stay valid after calling the functions.
 */

/**
 * @brief Init ublox SPP Service Server with ATT DB
 * @param packet_handler for events and tx data from peer as
RFCOMM_DATA_PACKET
 */
void ublox_spp_service_server_init(btstack_packet_handler_t
packet_handler);

/**
 * @brief Queue send request. When called, one packet can be send
via ublox_spp_service_send below
 * @param request
 * @param con_handle
 */
void
ublox_spp_service_server_request_can_send_now(btstack_context_callb
* request, hci_con_handle_t con_handle);

/**
 * @brief Send data
 * @param con_handle
 * @param data
 * @param size
 */
int ublox_spp_service_server_send(hci_con_handle_t con_handle,
const uint8_t * data, uint16_t size);
```

GATT Client API

gatt_client.h

```
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 Set minimum required security level for GATT Client
 * @note  The Bluetooth specification makes the GATT Server
responsible to check for security.
 *        This allows an attacker to spoof an existing device with
a GATT Servers, but skip the authentication part.
 *        If your application is exchanging sensitive data with a
remote device, you would need to manually check
 *        the security level before sending/receive such data.
 *        With level > 0, the GATT Client triggers authentication
for all GATT Requests and defers any exchange
 *        until the required security level is established.
 *        gatt_client_request_can_write_without_response_event
does not trigger authentication
 *        gatt_client_request_to_write_without_response does not
trigger authentication
```

```
 *  @pram level, default LEVEL_0 (no encryption required)
 */
void gatt_client_set_required_security_level(gap_security_level_t
level);

/**
 * @brief Connect to remote GATT Server over Classic (BR/EDR)
Connection
 *        GATT_EVENT_CONNECTED with status and con_handle for
other API functions
 *        is emitted on connection complete.
 * @note requires ENABLE_GATT_OVER_CLASSIC.
 * @param addr
 * @return status
 */
uint8_t gatt_client_classic_connect(btstack_packet_handler_t
callback, bd_addr_t addr);

/**
 * @brief Disconnect o Classic (BR/EDR) connection to a remote
GATT Server
 * @note requires ENABLE_GATT_OVER_CLASSIC
 * @param con_handle
 * @return status
 */
uint8_t gatt_client_classic_disconnect(btstack_packet_handler_t
callback, hci_con_handle_t con_handle);

/**
 * @brief Setup Enhanced LE Bearer with up to 5 channels on
existing LE connection
 * @param callback for GATT_EVENT_CONNECTED and
GATT_EVENT_DISCONNECTED events
 * @param con_handle
 * @param num_channels
 * @param storage_buffer for L2CAP connection
 * @param storage_size - each channel requires (2 * ATT MTU) + 10
bytes
 * @return
 */
```

```
uint8_t gatt_client_le_enhanced_connect(btstack_packet_handler_t
callback, hci_con_handle_t con_handle, uint8_t num_channels,
uint8_t * storage_buffer, uint16_t storage_size);

/**
 * @brief MTU is available after the first query has completed. If
status is equal to ERROR_CODE_SUCCESS, it returns the real value,
 * otherwise the default value ATT_DEFAULT_MTU (see bluetooth.h).
 * @param  con_handle
 * @param  mtu or 0 in case of error
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if MTU
is not exchanged and MTU auto-exchange is disabled
 *
ERROR_CODE_SUCCESS                                        on
success
 */
uint8_t gatt_client_get_mtu(hci_con_handle_t con_handle, uint16_t
* mtu);

/**
 * @brief Sets whether a MTU Exchange Request shall be
automatically send before the
 * first attribute read request is send. Default is enabled.
 * @param enabled
 */
void gatt_client_mtu_enable_auto_negotiation(uint8_t enabled);

/**
 * @brief Sends a MTU Exchange Request, this allows for the client
to exchange MTU
 * when gatt_client_mtu_enable_auto_negotiation is disabled.
 * @param  callback
 * @param  con_handle
```

```
 */
void gatt_client_send_mtu_negotiation(btstack_packet_handler_t
callback, hci_con_handle_t con_handle);

/**
 * @brief Returns 1 if the GATT client is ready to receive a
query. It is used with daemon.
 * @param  con_handle
 * @return is_ready_status     0 - if no GATT client for
con_handle is found, or is not ready, otherwise 1
 */
int gatt_client_is_ready(hci_con_handle_t con_handle);

/**
 * @brief Discovers all primary services.
 * For each found service a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of
discovery.
 * @param  callback
 * @param  con_handle
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_primary_services(btstack_packet_handler_t
callback, hci_con_handle_t con_handle);

/**
 * @brief Discovers all secondary services.
 * For each found service a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of
discovery.
 * @param  callback
 * @param  con_handle
```

```
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_secondary_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 a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of
discovery.
 * @param callback
 * @param con_handle
 * @param uuid16
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_primary_services_by_uuid16(btstack_packet_hanc
callback, hci_con_handle_t con_handle, uint16_t uuid16);

/**
 * @brief Discovers a specific primary service given its UUID.
This service may exist multiple times.
 * For each found service a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param callback
 * @param con_handle
```

```
 * @param uuid16
 * @param service_id    - context provided to callback in events
 * @param connection_id - contest provided to callback in events
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_primary_services_by_uuid16_with_context(btstac
callback, hci_con_handle_t con_handle,

uint16_t uuid16, uint16_t service_id, uint16_t connection_id);

/**
 * @brief Discovers a specific primary service given its UUID.
This service may exist multiple times.
 * For each found service a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of
discovery.
 * @param  callback
 * @param  con_handle
 * @param  uuid128
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_primary_services_by_uuid128(btstack_packet_har
callback, hci_con_handle_t con_handle, const uint8_t * uuid128);

/**
 * @brief Discovers a specific primary service given its UUID.
This service may exist multiple times.
```

```
 * For each found service a GATT_EVENT_SERVICE_QUERY_RESULT event
will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of
discovery.
 * @param callback
 * @param con_handle
 * @param uuid128
 * @param service_id    - context provided to callback in events
 * @param connection_id - context provided to callback in events
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_primary_services_by_uuid128_with_context(btsta
callback, hci_con_handle_t con_handle,

const uint8_t * uuid128, uint16_t service_id, uint16_t
connection_id);

/**
 * @brief Finds included services within the specified service.
 * For each found included service a
GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT event will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event 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.
 * @param  callback
 * @param  con_handle
 * @param  service
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
```

```
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_find_included_services_for_service(btstack_packet_handl
callback, hci_con_handle_t con_handle, gatt_client_service_t *
service);

/**
 * @brief Finds included services within the specified service.
 * For each found included service a
GATT_EVENT_INCLUDED_SERVICE_QUERY_RESULT event will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event 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.
 * @param  callback
 * @param  con_handle
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @param  service_id
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_find_included_services_for_service_with_context(btstack
callback, hci_con_handle_t con_handle,

gatt_client_service_t * service, uint16_t service_id, uint16_t
```

```
connection_id);

/**
 * @brief Discovers all characteristics within the specified
service.
 * For each found characteristic a
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will be emited.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  service
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristics_for_service(btstack_packet_har
callback, hci_con_handle_t con_handle, gatt_client_service_t *
service);

/**
 * @brief Discovers all characteristics within the specified
service.
 * For each found characteristic a
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will be emited.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  service
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
```

```
 */
uint8_t
gatt_client_discover_characteristics_for_service_with_context(btsta
callback, hci_con_handle_t con_handle, gatt_client_service_t *
service,

uint16_t service_id, uint16_t connection_id);

/**
 * @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 a
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  start_handle
 * @param  end_handle
 * @param  uuid16
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristics_for_handle_range_by_uuid16(bts
callback, hci_con_handle_t con_handle, uint16_t start_handle,
uint16_t end_handle, uint16_t uuid16);

/**
 * @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 a
```

```
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  start_handle
 * @param  end_handle
 * @param  uuid128
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristics_for_handle_range_by_uuid128(bt
callback, hci_con_handle_t con_handle, uint16_t start_handle,
uint16_t end_handle, const uint8_t * uuid128);

/**
 * @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 a
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  service
 * @param  uuid16
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristics_for_service_by_uuid16(btstack_
```

```
callback, hci_con_handle_t con_handle, gatt_client_service_t *
service, uint16_t uuid16);

/**
 * @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 a
GATT_EVENT_CHARACTERISTIC_QUERY_RESULT event will emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  service
 * @param  uuid128
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristics_for_service_by_uuid128(btstack
callback, hci_con_handle_t con_handle, gatt_client_service_t *
service, const uint8_t * uuid128);

/**
 * @brief Discovers attribute handle and UUID of a characteristic
descriptor within the specified characteristic.
 * For each found descriptor a
GATT_EVENT_ALL_CHARACTERISTIC_DESCRIPTORS_QUERY_RESULT event will
be emitted.
 *
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
```

```
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristic_descriptors(btstack_packet_hanc
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_t * characteristic);


/**
 * @brief Discovers attribute handle and UUID of a characteristic
descriptor within the specified characteristic.
 * For each found descriptor a
GATT_EVENT_ALL_CHARACTERISTIC_DESCRIPTORS_QUERY_RESULT event will
be emitted.
 *
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of discovery.
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_discover_characteristic_descriptors_with_context(btstac
callback, hci_con_handle_t con_handle,

gatt_client_characteristic_t * characteristic, uint16_t
service_id, uint16_t connection_it);

/**
 * @brief Reads the characteristic value using the
```

```
characteristic's value handle.
 * If the characteristic value is found a
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_value_of_characteristic(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 a
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_value_of_characteristic_using_value_handle(btstack
callback, hci_con_handle_t con_handle, uint16_t value_handle);
```

```
/**
 * @brief Reads the characteristic value using the
characteristic's value handle.
 * If the characteristic value is found a
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_value_of_characteristic_using_value_handle_with_cc
callback,

hci_con_handle_t con_handle,

uint16_t value_handle,

uint16_t service_id,

uint16_t connection_id);

/**
 * @brief Reads the characteric value of all characteristics with
the uuid.
 * For each characteristic value found a
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  start_handle
```

```
 * @param  end_handle
 * @param  uuid16
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_value_of_characteristics_by_uuid16(btstack_packet_
callback, hci_con_handle_t con_handle, uint16_t start_handle,
uint16_t end_handle, uint16_t uuid16);

/**
 * @brief Reads the characteric value of all characteristics with
the uuid.
 * For each characteristic value found a
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  start_handle
 * @param  end_handle
 * @param  uuid128
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_value_of_characteristics_by_uuid128(btstack_packet
callback, hci_con_handle_t con_handle, uint16_t start_handle,
uint16_t end_handle, const 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, a
GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT event with
updated value offset will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_long_value_of_characteristic(btstack_packet_handle
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_t * characteristic);

/**
 * @brief Reads the long characteristic value using the
characteristic's value handle.
 * The value will be returned in several blobs.
 * For each blob, a
GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT event with
updated value offset will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
```

```
gatt_client_read_long_value_of_characteristic_using_value_handle(bt
callback, hci_con_handle_t con_handle, uint16_t value_handle);

/**
 * @brief Reads the long characteristic value using the
characteristic's value handle.
 * The value will be returned in several blobs.
 * For each blob, a
GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT event with
updated value offset will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_long_value_of_characteristic_using_value_handle_wi
callback,

hci_con_handle_t con_handle, uint16_t value_handle,

uint16_t service_id, uint16_t connection_id);

/**
 * @brief Reads the long characteristic value using the
characteristic's value handle.
 * The value will be returned in several blobs.
 * For each blob, a
GATT_EVENT_LONG_CHARACTERISTIC_VALUE_QUERY_RESULT event with
updated value offset will be emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
```

```
 * @param  con_handle
 * @param  value_handle
 * @param  offset
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_long_value_of_characteristic_using_value_handle_wi
callback, hci_con_handle_t con_handle, uint16_t value_handle,
uint16_t offset);

/*
 * @brief Read multiple characteristic values.
 * The all results are emitted via single
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event,
 * followed by the GATT_EVENT_QUERY_COMPLETE event, which marks
the end of read.
 * @param  callback
 * @param  con_handle
 * @param  num_value_handles
 * @param  value_handles list of handles
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_multiple_characteristic_values(btstack_packet_hanc
callback, hci_con_handle_t con_handle, int num_value_handles,
uint16_t * value_handles);

/*
 * @brief Read multiple varaible characteristic values. Only
supported over LE Enhanced Bearer
```

```
 * The all results are emitted via single
GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT event,
 * followed by the GATT_EVENT_QUERY_COMPLETE event, which marks
the end of read.
 * @param  callback
 * @param  con_handle
 * @param  num_value_handles
 * @param  value_handles list of handles
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_read_multiple_variable_characteristic_values(btstack_pa
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.
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is copied on success and does not need to be
retained
 * @return status BTSTACK_MEMORY_ALLOC_FAILED, if no GATT client
for con_handle is found
 *                GATT_CLIENT_IN_WRONG_STATE , if GATT client is
not ready
 *                BTSTACK_ACL_BUFFERS_FULL   , if L2CAP cannot
send, there are no free ACL slots
 *                ERROR_CODE_SUCCESS         , if query is
successfully registered
 */
uint8_t
gatt_client_write_value_of_characteristic_without_response(hci_con_
```

```
con_handle, uint16_t value_handle, uint16_t value_length, uint8_t
* value);

/**
 * @brief Writes the authenticated characteristic value using the
characteristic's value handle
 * without an acknowledgment that the write was successfully
performed.
 * @note GATT_EVENT_QUERY_COMPLETE is emitted with
ATT_ERROR_SUCCESS for success,
 * or ATT_ERROR_BONDING_INFORMATION_MISSING if there is no bonding
information stored.
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  message_len
 * @param  message is not copied, make sure memory is accessible
until write is done
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                 if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                              if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                               if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                       if query
is successfully registered
 */
uint8_t
gatt_client_signed_write_without_response(btstack_packet_handler_t
callback, hci_con_handle_t con_handle, uint16_t value_handle,
                                                  uint16_t
message_len, uint8_t * message);

/**
 * @brief Writes the characteristic value using the
characteristic's value handle.
```

```
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed, if the event's att_status
field is set to
 * ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_write_value_of_characteristic(btstack_packet_handler_t
callback, hci_con_handle_t con_handle, uint16_t value_handle,
uint16_t value_length, uint8_t * value);

/**
 * @brief Writes the characteristic value using the
characteristic's value handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed, if the event's att_status
field is set to
 * ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
```

```
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_write_value_of_characteristic_with_context(btstack_pack
callback, hci_con_handle_t con_handle, uint16_t value_handle,

uint16_t value_length, uint8_t * value, uint16_t service_id,
uint16_t connection_id);

/**
 * @brief Writes the characteristic value using the
characteristic's value handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
```

```
BTSTACK_MEMORY_ALLOC_FAILED                                     if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                      if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                              if query
is successfully registered
 */
uint8_t
gatt_client_write_long_value_of_characteristic(btstack_packet_handl
callback, hci_con_handle_t con_handle, uint16_t value_handle,
uint16_t value_length, uint8_t * value);

/**
 * @brief Writes the characteristic value using the
characteristic's value handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                        if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                                     if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                      if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                              if query
```

```
is successfully registered
 */
uint8_t
gatt_client_write_long_value_of_characteristic_with_context(btstack
callback, hci_con_handle_t con_handle, uint16_t value_handle,

uint16_t value_length, uint8_t * value, uint16_t service_id,
uint16_t connection_id);

/**
 * @brief Writes the characteristic value using the
characteristic's value handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  offset of value
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_write_long_value_of_characteristic_with_offset(btstack_
callback, hci_con_handle_t con_handle, uint16_t value_handle,
uint16_t offset, uint16_t value_length, uint8_t * value);
```

```
/**
 * @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_EVENT_QUERY_COMPLETE EVENT marks the end of write.
 * The write is successfully performed, if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  value_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_reliable_write_long_value_of_characteristic(btstack_pac
callback, hci_con_handle_t con_handle, uint16_t value_handle,
uint16_t value_length, uint8_t * value);

/**
 * @brief Reads the characteristic descriptor using its handle.
 * If the characteristic descriptor is found, a
GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
```

```
 * @param  callback
 * @param  con_handle
 * @param  descriptor
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                 if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                              if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                               if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                       if query
is successfully registered
 */
uint8_t
gatt_client_read_characteristic_descriptor(btstack_packet_handler_t
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_descriptor_t * descriptor);

/**
 * @brief Reads the characteristic descriptor using its handle.
 * If the characteristic descriptor is found, a
GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  descriptor
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                 if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                              if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                               if GATT
client is not ready
 *
```

```
ERROR_CODE_SUCCESS                                      if query
is successfully registered
 */
uint8_t
gatt_client_read_characteristic_descriptor_using_descriptor_handle(
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, a
GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  descriptor
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                             if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                              if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                      if query
is successfully registered
 */
uint8_t
gatt_client_read_long_characteristic_descriptor(btstack_packet_hanc
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_descriptor_t * descriptor);

/**
 * @brief Reads the long characteristic descriptor using its
handle. It will be returned in several blobs.
 * For each blob, a
```

```
GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  descriptor_handle
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_read_long_characteristic_descriptor_using_descriptor_ha
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, a
GATT_EVENT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT event will be
emitted.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of read.
 * @param  callback
 * @param  con_handle
 * @param  descriptor_handle
 * @param  offset
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
```

```
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                    if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                            if query
is successfully registered
 */
uint8_t
gatt_client_read_long_characteristic_descriptor_using_descriptor_ha
callback, hci_con_handle_t con_handle, uint16_t descriptor_handle,
uint16_t offset);

/**
 * @brief Writes the characteristic descriptor using its handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  descriptor
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                      if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                                   if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                    if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                            if query
is successfully registered
 */
uint8_t
gatt_client_write_characteristic_descriptor(btstack_packet_handler_
```

```
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_descriptor_t * descriptor, uint16_t
value_length, uint8_t * value);

/**
 * @brief Writes the characteristic descriptor using its handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  descriptor_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                 if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                              if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                               if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                       if query
is successfully registered
 */
uint8_t
gatt_client_write_characteristic_descriptor_using_descriptor_handle
callback, hci_con_handle_t con_handle, uint16_t descriptor_handle,
uint16_t value_length, uint8_t * value);

/**
 * @brief Writes the characteristic descriptor using its handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
```

```
 * @param  callback
 * @param  con_handle
 * @param  descriptor
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_write_long_characteristic_descriptor(btstack_packet_har
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_descriptor_t * descriptor, uint16_t
value_length, uint8_t * value);

/**
 * @brief Writes the characteristic descriptor using its handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  descriptor_handle
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
```

```
 *
BTSTACK_MEMORY_ALLOC_FAILED                                 if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                  if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                          if query
is successfully registered
 */
uint8_t
gatt_client_write_long_characteristic_descriptor_using_descriptor_h
callback, hci_con_handle_t con_handle, uint16_t descriptor_handle,
uint16_t value_length, uint8_t * value);

/**
 * @brief Writes the characteristic descriptor using its handle.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  descriptor_handle
 * @param  offset of value
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                    if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                                 if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                  if GATT
client is not ready
 *
ERROR_CODE_SUCCESS                                          if query
is successfully registered
```

```
 */
uint8_t
gatt_client_write_long_characteristic_descriptor_using_descriptor_h
callback, hci_con_handle_t con_handle, uint16_t descriptor_handle,
uint16_t offset, uint16_t value_length, uint8_t * value);

/**
 * @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.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @param
configuration
GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION,
GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                               if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                if GATT
client is not ready
 *
GATT_CLIENT_CHARACTERISTIC_NOTIFICATION_NOT_SUPPORTED     if
configuring notification, but characteristic has no notification
property set
 *
```

```
GATT_CLIENT_CHARACTERISTIC_INDICATION_NOT_SUPPORTED       if
configuring indication, but characteristic has no indication
property set
 *
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE         if
configuration is invalid
 *
ERROR_CODE_SUCCESS                                        if query
is successfully registered
 */
uint8_t
gatt_client_write_client_characteristic_configuration(btstack_packe
callback, hci_con_handle_t con_handle,
gatt_client_characteristic_t * characteristic, uint16_t
configuration);

/**
 * @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.
 * The GATT_EVENT_QUERY_COMPLETE event marks the end of write.
 * The write is successfully performed if the event's att_status
field is set to ATT_ERROR_SUCCESS (see bluetooth.h for ATT_ERROR
codes).
 * @param  callback
 * @param  con_handle
 * @param  characteristic
 * @param
configuration
GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION,
GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION
 * @param  service_id    - context provided to callback in events
 * @param  connection_id - contest provided to callback in events
 * @return status
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER                  if no
```

```
HCI connection for con_handle is found
 *
BTSTACK_MEMORY_ALLOC_FAILED                                  if no
GATT client for con_handle could be allocated
 *
GATT_CLIENT_IN_WRONG_STATE                                   if GATT
client is not ready
 *
GATT_CLIENT_CHARACTERISTIC_NOTIFICATION_NOT_SUPPORTED     if
configuring notification, but characteristic has no notification
property set
 *
GATT_CLIENT_CHARACTERISTIC_INDICATION_NOT_SUPPORTED       if
configuring indication, but characteristic has no indication
property set
 *
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE         if
configuration is invalid
 *
ERROR_CODE_SUCCESS                                           if query
is successfully registered
 */
uint8_t
gatt_client_write_client_characteristic_configuration_with_context(
callback, hci_con_handle_t con_handle,

gatt_client_characteristic_t * characteristic, uint16_t
configuration, uint16_t service_id, uint16_t connection_id);

/**
 * @brief Register for changes to the Service Changed and Database
Hash Characteristics of the remote GATT Service
 * *
 * When configured, GATT_EVENT_QUERY_COMPLETE event is emitted
 * If supported, the Database Hash is read as well
 *
 * Requires ENABLE_GATT_CLIENT_CACHING
 *
 * @param callback
 */
```

```
void
gatt_client_add_service_changed_handler(btstack_packet_callback_rec
* callback);

/**
 * @brief Remove callback for service changes
 *
 * Requires ENABLE_GATT_CLIENT_CACHING
 *
 * @param callback
 */
void
gatt_client_remove_service_changed_handler(btstack_packet_callback_
* callback);

/**
 * @brief Register for notifications and indications of a
characteristic enabled by
 * the gatt_client_write_client_characteristic_configuration
function.
 * @param notification struct used to store registration
 * @param callback
 * @param con_handle or GATT_CLIENT_ANY_CONNECTION to receive
updates from all connected devices
 * @param characteristic or NULL to receive updates for all
characteristics
 */
void
gatt_client_listen_for_characteristic_value_updates(gatt_client_not
* notification, btstack_packet_handler_t callback,
hci_con_handle_t con_handle, gatt_client_characteristic_t *
characteristic);

/**
 * @brief Stop listening to characteristic value updates
registered with
 * the gatt_client_listen_for_characteristic_value_updates
function.
 * @param notification struct used in
gatt_client_listen_for_characteristic_value_updates
```

```
 */
void
gatt_client_stop_listening_for_characteristic_value_updates(gatt_cl
* notification);

/**
 * @brief Register for notifications and indications of
characteristic in a service
 * the gatt_client_write_client_characteristic_configuration
function.
 * @param notification struct used to store registration
 * @param callback
 * @param con_handle or GATT_CLIENT_ANY_CONNECTION to receive
updates from all connected devices
 * @param service
 * @param end_handle
 * @param service_id    - context provided to callback in events
 * @param connection_id - contest provided to callback in events
 */
void
gatt_client_listen_for_service_characteristic_value_updates(gatt_cl
* notification,

btstack_packet_handler_t callback,

hci_con_handle_t con_handle,

gatt_client_service_t * service,

uint16_t service_id,

uint16_t connection_id);

/**
 * @brief Stop listening to characteristic value updates for
registered service with
 * the gatt_client_listen_for_characteristic_value_updates
function.
 * @param notification struct used in
gatt_client_listen_for_characteristic_value_updates
```

```
 */
void
gatt_client_stop_listening_for_service_characteristic_value_updates
* notification);


/**
 * @brief Transactional write. It can be called as many times as
it is needed to write the characteristics within the same
transaction.
 * Call the gatt_client_execute_write function to commit the
transaction.
 * @param  callback
 * @param  con_handle
 * @param  attribute_handle
 * @param  offset of value
 * @param  value_length
 * @param  value is not copied, make sure memory is accessible
until write is done, i.e. GATT_EVENT_QUERY_COMPLETE is received
 */
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 value_length, uint8_t * value);

/**
 * @brief Commit transactional write. GATT_EVENT_QUERY_COMPLETE is
received.
 * @param  callback
 * @param  con_handle
 * @return status
 */
uint8_t gatt_client_execute_write(btstack_packet_handler_t
callback, hci_con_handle_t con_handle);

/**
 * @brief Abort transactional write. GATT_EVENT_QUERY_COMPLETE is
received.
 * @param  callback
 * @param  con_handle
 * @return status
```

```
 */
uint8_t gatt_client_cancel_write(btstack_packet_handler_t
callback, hci_con_handle_t con_handle);

/**
 * @brief Request callback when regular gatt query can be sent
 * @note callback might happen during call to this function
 * @param callback_registration to point to callback function and
context information
 * @param con_handle
 * @return ERROR_CODE_SUCCESS if ok, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection
handle unknown,
 *       - ERROR_CODE_MEMORY_CAPACITY_EXCEEDED if GATT client
memory exceeded, or
 *       - ERROR_CODE_COMMAND_DISALLOWED if GATT client already
registered callback
 */
uint8_t
gatt_client_request_to_send_gatt_query(btstack_context_callback_reg
* callback_registration, hci_con_handle_t con_handle);

/**
 * @brief Remove queued callback for regular gatt queries, to be
used on disconnect for example
 * @param callback_registration
 * @param con_handle
 * @return ERROR_CODE_SUCCESS if ok
 */
uint8_t
gatt_client_remove_gatt_query(btstack_context_callback_registratior
* callback_registration, hci_con_handle_t con_handle);

/**
 * @brief Request callback when writing characteristic value
without response is possible
 * @note callback might happen during call to this function
 * @param callback_registration to point to callback function and
context information
 * @param con_handle
```

```
 * @return ERROR_CODE_SUCCESS if ok,
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection handle
unknown, and ERROR_CODE_COMMAND_DISALLOWED if callback already
registered
 */
uint8_t
gatt_client_request_to_write_without_response(btstack_context_callb
* callback_registration, hci_con_handle_t con_handle);


// the following functions are marked as deprecated and will be
removed eventually
/**
 * @brief Requests GATT_EVENT_CAN_WRITE_WITHOUT_RESPONSE that
guarantees
 * a single successful
gatt_client_write_value_of_characteristic_without_response call.
 * @deprecated please use
gatt_client_request_to_write_without_response instead
 * @param  callback
 * @param  con_handle
 * @return status
 */
uint8_t
gatt_client_request_can_write_without_response_event(btstack_packet
callback, hci_con_handle_t con_handle);

#ifdef ENABLE_GATT_CLIENT_CACHING
/**
 * @brief Get GATT Database Hash for remote device
 * @param con_handle
 * @return database hash or NULL if not available
 */
const uint8_t * gatt_client_get_database_hash(hci_con_handle_t
con_handle);

/**
 * @brief Get next cache id for remote device. This allows to
enumerate all GATT Service Client requets
 * @param con_handle
```

```
 * @return
 */
uint16_t gatt_client_get_next_cache_id(hci_con_handle_t
con_handle);
#endif

/**
 * @brief Map ATT Error Code to (extended) Error Codes
 * @param att_error_code
 * @return
 */
uint8_t gatt_client_att_status_to_error_code(uint8_t
att_error_code);
```

GATT Service Client API

gatt_service_client.h

```
/**
 * @brief Initialize GATT Service Client infrastructure
 */
void gatt_service_client_init(void);

/**
 * @brief Register new GATT Service Client with list of
Characteristic UUID16s
 * @param client
 * @param connection_and_notification_handler packet handler that
receives events: GATT_EVENT_NOTIFICATION,
GATT_EVENT_IDENTIFICATION and two HCI_EVENT_GATTSERVICE_META
subevents: GATTSERVICE_SUBEVENT_CLIENT_CONNECTED/DISCONNECTED
 * @param characteristic_uuid16s
 * @param characteristic_uuid16s_num
 */
void
gatt_service_client_register_client_with_uuid16s(gatt_service_clier
*client, btstack_packet_handler_t
connection_and_notification_handler,
                                          const uint16_t
*characteristic_uuid16s, uint16_t characteristic_uuid16s_num);
```

```
/**
 * @brief Register new GATT Service Client with list of
Characteristic UUID128s
 * @param client
 * @param connection_and_notification_handler packet handler that
receives events: GATT_EVENT_NOTIFICATION,
GATT_EVENT_IDENTIFICATION and two HCI_EVENT_GATTSERVICE_META
subevents: GATTSERVICE_SUBEVENT_CLIENT_CONNECTED/DISCONNECTED
 * @param characteristic_uuid128s
 * @param characteristic_uuid128s_num
 */

void
gatt_service_client_register_client_with_uuid128s(gatt_service_clie
*client, btstack_packet_handler_t packet_handler,
                                          const uuid128_t
*characteristic_uuid128s, uint16_t characteristic_uuid128s_num);

/**
 * @brief Get Characteristic UUID16 for given Characteristic index
 *
 * @param client
 * @param characteristic_index
 * @return uuid16 or 0 if index out of range
 */
uint16_t gatt_service_client_characteristic_uuid16_for_index(const
gatt_service_client_t * client, uint8_t characteristic_index);

/**
 * @brief Get Characteristic UUID128 for given Characteristic
index
 *
 * @param client
 * @param characteristic_index
 * @return uuid128 or NULL if index out of range
 */
const uuid128_t *
gatt_service_client_characteristic_uuid128_for_index(const
gatt_service_client_t * client, uint8_t characteristic_index);
```

```
/**
 * @brief Unregister GATT Service Client
 * @param client
 */
void gatt_service_client_unregister_client(gatt_service_client_t *
client);

/**
 * @brief Connect to the first instance of Primary GATT Service
with UUID16
 *
 * @param con_handle
 * @param client
 * @param connection
 * @param service_uuid16
 * @param characteristics
 * @param characteristics_num
 * @return
 */
uint8_t
gatt_service_client_connect_primary_service_with_uuid16(hci_con_har
con_handle, gatt_service_client_t *client,

gatt_service_client_connection_t *connection,
                                                        uint16_t
service_uuid16,

gatt_service_client_characteristic_t *characteristics,
                                                        uint8_t
characteristics_num);

/**
 * @brief Connect to the n-th instance of Primary GATT Service
with UUID16
 *
 * @param con_handle
 * @param client
 * @param connection
 * @param service_uuid128
```

```
 * @param characteristics
 * @param characteristics_num
 * @return
 */
uint8_t
gatt_service_client_connect_primary_service_with_uuid128(hci_con_ha
con_handle, gatt_service_client_t *client,

gatt_service_client_connection_t *connection,

const uuid128_t * service_uuid128,

gatt_service_client_characteristic_t *characteristics,

uint8_t characteristics_num);

/**
 * @brief Connect to the Secondary GATT Service with given handle
range
 *
 * UUID16 and Service Index are stored for GATT Service Client
user only
 *
 * @param con_handle
 * @param client
 * @param connection
 * @param service_uuid16
 * @param service_start_handle
 * @param service_end_handle
 * @param service_index
 * @param characteristics
 * @param characteristics_num
 * @return
 */
uint8_t
gatt_service_client_connect_secondary_service_with_uuid16(hci_con_h
con_handle, gatt_service_client_t *client,

gatt_service_client_connection_t *connection,
                                                        uint16_t
```

```
service_uuid16, uint8_t service_index,
                                                                uint16_t
service_start_handle, uint16_t service_end_handle,

gatt_service_client_characteristic_t *characteristics,
                                                                uint8_t
characteristics_num);

/**
 * @brief Disconnect service client
 * @param client
 * @param connection
 * @return
 */
uint8_t
gatt_service_client_disconnect(gatt_service_client_connection_t
*connection);

/**
 * @brief Check if Characteristic is available and can be queried
 * @param client
 * @param connection
 * @param characteristic_index
 * @return ERROR_CODE_SUCCESS if ready,
ERROR_CODE_COMMAND_DISALLOWED or
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE otherwise
 */
uint8_t gatt_service_client_can_query_characteristic(const
gatt_service_client_connection_t *connection,
                                                    uint8_t
characteristic_index);

/**
 * @brief Get Characteristic Value Handle for given Characteristic
index
 * *
 * @param connection
 * @param characteristic_index
 * @return
 */
```

```
uint16_t
gatt_service_client_characteristic_value_handle_for_index(const
gatt_service_client_connection_t *connection,

uint8_t characteristic_index);

/**
 * @brief Get Characteristic index Handle for given Characteristic
Value Handle
 * *
 * @param connection
 * @param value_handle
 * @return index of characteristic in initial list or
GATT_SERVICE_CLIENT_INVALID_INDEX
 */
uint8_t
gatt_service_client_characteristic_index_for_value_handle(const
gatt_service_client_connection_t *connection,

uint16_t value_handle);

/**
 * @brief Get connection id
 * @param client
 * @param connection
 * @returns connection_id
 */
uint16_t gatt_service_client_get_connection_id(const
gatt_service_client_connection_t * connection);

/**
 * @brief Get connection handle
 * @param client
 * @param connection
 * @returns con_handle
 */
hci_con_handle_t gatt_service_client_get_con_handle(const
gatt_service_client_connection_t * connection);

/**
```

```
 * @brief Get service index provided in connect call
 * @param client
 * @param connection
 * @returns connection_id
 */
uint8_t gatt_service_client_get_service_index(const
gatt_service_client_connection_t * connection);

/**
 * @brief Get remote MTU
 * @param client
 * @param connection
 * @returns MTU or 0 in case of error
 */
uint16_t gatt_service_client_get_mtu(const
gatt_service_client_connection_t *connection);



/**
 * @brief Dump characteristic value handles
 * @param client
 * @param connection
 * @param characteristic_names
 */
void gatt_service_client_dump_characteristic_value_handles(const
gatt_service_client_connection_t *connection,
                                                           const
char **characteristic_names);

/**
 * Return connection for given CID
 * @param client
 * @param connection_cid
 * @return
 */
gatt_service_client_connection_t *
gatt_service_client_get_connection_for_cid(const
gatt_service_client_t *client, uint16_t connection_cid);

/**
```

```
 * Substitute subevent ID with the given one, and send the event
 * @param callback
 * @param packet
 * @param size
 * @param subevent_id
 */
void
gatt_service_client_replace_subevent_id_and_emit(btstack_packet_har
callback, uint8_t * packet, uint16_t size, uint8_t subevent_id);

/**
 * @brief De-Init
 * @param client
 */
void gatt_service_client_deinit(void);
```

Device Database API

le_device_db.h

```
/**
 * @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
 * @return 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
 * @return number of device in db
 */
int le_device_db_count(void);

/**
 * @brief get max number of devices in db for enumeration
 * @return max number of device in db
 */
int le_device_db_max_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
 * @breif secure_connection
 */
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, int secure_connection);

/**
```

```
 * @brief get remote encryption info
 * @brief index
 * @brief ediv
 * @brief rand
 * @brief ltk
 * @brief key size
 * @brief authenticated
 * @brief authorized
 * @breif secure_connection
 */
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, int * secure_connection);

#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
 * @return 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
 * @return 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);
```

Device Database TLV API

le_device_db_tlv.h : @TLV 'BTD' x

```
/**
 * @brief configure le device db for use with btstack tlv instance
 * @param btstack_tlv_impl to use
 * @param btstack_tlv_context
 */

void le_device_db_tlv_configure(const btstack_tlv_t *
btstack_tlv_impl, void * btstack_tlv_context);
```

Security Manager API

sm.h : @TLV 'SMER' @TLV 'SMIR'

```
/**
 * @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
 * @note If not set and btstack_tlv is configured, ER key is
generated and stored in TLV by SM
 * @param er key
 */
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
 * @note If not set and btstack_tlv is configured, IR key is
generated and stored in TLV by SM
```

```
 * @param ir key
 */
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 address_type, bd_addr_t addr, uint8_t * oob_data));

/**
 * @brief Add event packet handler.
 * @param callback_handler
 */
void sm_add_event_handler(btstack_packet_callback_registration_t *
callback_handler);

/**
 * @brief Remove event packet handler.
 * @param callback_handler
 */
void
sm_remove_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 Enable/disable Secure Connections Mode only
 * @param enable secure connections only mode
 */
void sm_set_secure_connections_only_mode(bool enable);

/**
 * @brief Let Peripheral request an encrypted connection right
after connecting
 * @param enable
 * @note Not used normally. Bonding is triggered by access to
protected attributes in ATT Server
 */
void sm_set_request_security(bool enable);

/**
 * @brief Trigger Security Request
 * @deprecated please use sm_request_pairing instead
 */
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 Used by att_server.c and gatt_client.c to request user
authentication
 * @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 Support for signed writes, used by att_server.
 * @return ready
 */
int sm_cmac_ready(void);

/**
 * @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
```

```
 * @param address_type
 * @param address
 * @return 0 if successfully added to lookup queue
 * @note Triggers SM_IDENTITY_RESOLVING_* events
 */
int sm_address_resolution_lookup(uint8_t address_type, bd_addr_t
address);

/**
 * @brief Get Identity Resolving state
 * @param con_handle
 * @return irk_lookup_state_t
 * @note return IRK_LOOKUP_IDLE if connection does not exist
 */
irk_lookup_state_t sm_identity_resolving_state(hci_con_handle_t
con_handle);

/**
 * @brief Identify device in LE Device DB.
 * @param con_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 Get LTK for encrypted connection
 * @param con_handle
 * @param ltk buffer to store long term key
 * @return ERROR_CODE_SUCCESS ok
 *         ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if no
connection for this con handle exists
 *         ERROR_CODE_PIN_OR_KEY_MISSING if connection is not
encrypted
 */
uint8_t sm_get_ltk(hci_con_handle_t con_handle, sm_key_t ltk);

/**
 * @brief Use fixec passkey for Legacy and SC instead of
generating a random number
 * @note Can be used to improve security over Just Works if no
```

```
keyboard or displary are present and
 *       individual random passkey can be printed on the device
during production
 * @param passkey
 */
void sm_use_fixed_passkey_in_display_role(uint32_t passkey);

/**
 * @brief Allow connection re-encryption in Peripheral (Responder)
role for LE Legacy Pairing
 *       without entry for Central device stored in LE Device DB
 * @note BTstack in Peripheral Role (Responder) supports LE Legacy
Pairing without a persistent LE Device DB as
 *       the LTK is reconstructed from a local secret IRK and EDIV
+ Random stored on Central (Initiator) device
 *       On the downside, it's not really possible to delete a
pairing if this is enabled.
 * @param allow encryption using reconstructed LTK without stored
entry (Default: 1)
 */
void sm_allow_ltk_reconstruction_without_le_device_db_entry(int
allow);

/**
 * @brief Generate OOB data for LE Secure Connections
 * @note This generates a 128 bit random number ra and then
calculates Ca = f4(PKa, PKa, ra, 0)
 *       New OOB data should be generated for each pairing. Ra is
used for subsequent OOB pairings
 * @param callback
 * @return status
 */
uint8_t sm_generate_sc_oob_data(void (*callback)(const uint8_t *
confirm_value, const uint8_t * random_value));

/**
 * @brief Registers OOB Data Callback for LE Secure Conections.
The callback should set all arguments and return 1 if OOB data is
availble
 * @note the oob_sc_local_random usually is the random_value
```

```
returend by sm_generate_sc_oob_data
 * @param get_oob_data_callback
 */
void sm_register_sc_oob_data_callback( int
(*get_sc_oob_data_callback)(uint8_t address_type, bd_addr_t addr,
uint8_t * oob_sc_peer_confirm, uint8_t * oob_sc_peer_random));

/**
 * @bbrief Register LTK Callback that allows to provide a custom
LTK on re-encryption. The callback returns true if LTK was
modified
 * @param get_ltk_callback
 */
void sm_register_ltk_callback( bool (*get_ltk_callback)
(hci_con_handle_t con_handle, uint8_t address_type, bd_addr_t
addr, uint8_t * ltk));
```

Audio Interface API

btstack_audio.h : Abstraction layer for 16-bit audio playback and recording within BTstack.

Most embedded implementations, e.g. the one for ESP32, use a single I2S interface which requires that the sample rate is the same for sink and source roles

```
typedef struct {
    btstack_time_us_t timestamp;
} btstack_audio_context_t;

typedef struct {

    /**
     * @brief Setup audio codec for specified samplerate and
number of channels
     * @param Channels (1=mono, 2=stereo)
     * @param Sample rate
     * @param Playback callback with sample buffer, number of
samples, and, if available playback time of first sample
     * @return 0 on success
```

```
     */
    int (*init)(uint8_t channels,
                uint32_t samplerate,
                void (*playback) (int16_t *  buffer,
                                  uint16_t num_samples,
                                  const btstack_audio_context_t *
timeinfo));

    /**
     * @brief Get the current playback sample rate, may differ
from the
     *        specified sample rate
     */
    uint32_t (*get_samplerate)(void);

    /**
     * @brief Set volume
     * @param Volume 0..127
     */
    void (*set_volume)(uint8_t volume);

    /**
     * @brief Start stream
     */
    void (*start_stream)(void);

    /**
     * @brief Stop stream
     */
    void (*stop_stream)(void);

    /**
     * @brief Close audio codec
     */
    void (*close)(void);

} btstack_audio_sink_t;


typedef struct {
```

```
    /**
     * @brief Setup audio codec for specified samplerate and
number of channels
     * @param Channels (1=mono, 2=stereo)
     * @param Sample rate
     * @param Recording callback with sample buffer, number of
samples, and, if available recording time of first sample
     * @return 0 on success
     */
    int (*init)(uint8_t channels,
                uint32_t samplerate,
                void (*recording)(const int16_t * buffer,
                                  uint16_t num_samples,
                                  const btstack_audio_context_t *
timeinfo));

    /**
     * @brief Get the current recording sample rate, may differ
from the
     *        specified sameple rate
     */
    uint32_t (*get_samplerate)(void);

    /**
     * @brief Set Gain
     * @param Gain 0..127
     */
    void (*set_gain)(uint8_t gain);

    /**
     * @brief Start stream
     */
    void (*start_stream)(void);

    /**
     * @brief Stop stream
     */
    void (*stop_stream)(void);
```

```
    /**
     * @brief Close audio codec
     */
    void (*close)(void);

} btstack_audio_source_t;


/**
 * @brief Get BTstack Audio Sink Instance
 * @return btstack_audio_sink implementation
 */
const btstack_audio_sink_t *
btstack_audio_sink_get_instance(void);

/**
 * @brief Get BTstack Audio Source Instance
 * @return btstack_audio_source implementation
 */
const btstack_audio_source_t *
btstack_audio_source_get_instance(void);


/**
 * @brief Get BTstack Audio Sink Instance
 * @param btstack_audio_sink implementation
 */
void btstack_audio_sink_set_instance(const btstack_audio_sink_t *
audio_sink_impl);

/**
 * @brief Get BTstack Audio Source Instance
 * @param btstack_audio_source implementation
 */
void btstack_audio_source_set_instance(const
btstack_audio_source_t * audio_source_impl);


// common implementations
const btstack_audio_sink_t *
```

```
btstack_audio_portaudio_sink_get_instance(void);
const btstack_audio_source_t *
btstack_audio_portaudio_source_get_instance(void);

const btstack_audio_sink_t *
btstack_audio_alsa_sink_get_instance(void);

const btstack_audio_sink_t *
btstack_audio_embedded_sink_get_instance(void);
const btstack_audio_source_t *
btstack_audio_embedded_source_get_instance(void);

const btstack_audio_sink_t *
btstack_audio_esp32_sink_get_instance(void);
const btstack_audio_source_t *
btstack_audio_esp32_source_get_instance(void);

// platform-specific extension
void btstack_audio_portaudio_sink_set_device(const char *
device_name);
void btstack_audio_portaudio_source_set_device(const char *
device_name);
```

base64 Decoder API

btstack_base64_decoder.h

```
#define BTSTACK_BASE64_DECODER_MORE     -1
#define BTSTACK_BASE64_DECODER_COMPLETE -2
#define BTSTACK_BASE64_DECODER_INVALID  -3
#define BTSTACK_BASE64_DECODER_FULL     -4

/**
 * @brief Initialize base64 decoder
 * @param context
 */
void btstack_base64_decoder_init(btstack_base64_decoder_t *
context);

/**
```

```
 * @brief Decode single byte
 * @brief context
 * @return value, or BTSTACK_BASE64_DECODER_MORE,
BTSTACK_BASE64_DECODER_COMPLETE, BTSTACK_BASE64_DECODER_INVALID
 */
int  btstack_base64_decoder_process_byte(btstack_base64_decoder_t
* context, uint8_t c);

/**
 * @brief Decode block
 * @brief input_data base64 encoded message
 * @brief input_size
 * @breif output_buffer
 * @brief output_max_size
 * @return output_size or BTSTACK_BASE64_DECODER_INVALID,
BTSTACK_BASE64_DECODER_FULL
 */
int btstack_base64_decoder_process_block(const uint8_t *
input_data, uint32_t input_size, uint8_t * output_buffer, uint32_t
output_max_size);
```

base64 Encoder API

btstack_base64_encoder.h

Chipset Driver API

btstack_chipset.h : The API implements custom chipset initialization and support of proprietary extensions to set UART baud rate, Bluetooth Address, and similar.

```
typedef struct {
    /**
     * chipset driver name
     */
    const char * name;

    /**
     * init driver
```

```
     * allows to reset init script index
     * @param transport_config
     */
    void (*init)(const void * transport_config);

    /**
     * support custom init sequences after RESET command
     * @param  hci_cmd_buffer to store generated command
     * @return result see btstack_chipset_result_t
     */
    btstack_chipset_result_t (*next_command)(uint8_t *
hci_cmd_buffer);

    /**
     * provide UART Baud Rate change command.
     * @param baudrate
     * @param hci_cmd_buffer to store generated command
     */
    void (*set_baudrate_command)(uint32_t baudrate, uint8_t
*hci_cmd_buffer);

    /** provide Set BD Addr command
     * @param baudrate
     * @param hci_cmd_buffer to store generated command
     */
    void (*set_bd_addr_command)(bd_addr_t addr, uint8_t
*hci_cmd_buffer);

} btstack_chipset_t;
```

Bluetooth Power Control API

btstack_control.h : The Bluetooth Hardware Control API allows HCI to manage Bluetooth chipsets via direct hardware controls.

```
typedef struct {
    void (*init) (const void *config);
    int  (*on)   (void);  // <-- turn BT module on and configure
    int  (*off)  (void);  // <-- turn BT module off
    int  (*sleep)(void);  // <-- put BT module to sleep    - only
```

```
to be called after ON
    int  (*wake) (void);  // <-- wake BT module from sleep - only
to be called after SLEEP
    void (*register_for_power_notifications)(void (*cb)
(POWER_NOTIFICATION_t event));
} btstack_control_t;
```

Debug Messages API

btstack_debug.h : Allow to funnel debug and error messages.

```
/**
 * @brief Log Security Manager key via log_info
 * @param name
 * @param key to log
 */
void log_info_key(const char * name, sm_key_t key);

/**
 * @brief Hexdump via log_info
 * @param data
 * @param size
 */
void log_info_hexdump(const void *data, int size);

/**
 * @brief Hexdump via log_debug
 * @param data
 * @param size
 */
void log_debug_hexdump(const void *data, int size);
```

EM9304 SPI API

btstack_em9304_spi.h : BTstack's Hardware Abstraction Layer for EM9304 connected via SPI with additional RDY Interrupt line.

```
#include <stdint.h>
typedef struct {
```

```
    /**
     * @brief Open SPI
     */
    int (*open)(void);

    /**
     * @brief Close SPI
     */
    int (*close)(void);

    /**
     * @brief Check if full duplex operation via transceive is
supported
     * @return 1 if supported
     */
    int  (*get_fullduplex_support)(void);

    /**
     * @brief Set callback for RDY
     * @param callback or NULL to disable callback
     */
    void (*set_ready_callback)(void (*callback)(void));

    /**
     * @brief Set callback for transfer complete
     * @param callback
     */
    void (*set_transfer_done_callback)(void (*callback)(void));

    /**
     * @brief Set Chip Selet
     * @param enable
     */
    void (*set_chip_select)(int enable);

    /**
     * @brief Poll READY state
     */
    int (*get_ready)(void);
```

```
    /**
     * @brief Transmit and Receive bytes via SPI
     * @param tx_data buffer to transmit
     * @param rx_data buffer to receive into
     * @param len
     */
    void (*transceive)(const uint8_t * tx_data, uint8_t * rx_data,
uint16_t len);

    /**
     * @brief Transmit bytes via SPI
     * @param tx_data buffer to transmit
     * @param len
     */
    void (*transmit)(const uint8_t * tx_data, uint16_t len);

    /**
     * @brief Receive bytes via SPI
     * @param rx_data buffer to receive into
     * @param len
     */
    void (*receive)(uint8_t * rx_data, uint16_t len);

} btstack_em9304_spi_t;

/**
 * @brief Get EM9304 SPI instance
 */
const btstack_em9304_spi_t *
btstack_em9304_spi_embedded_instance(void);
```

Finite State Machine (FSM) API

btstack_fsm.h

```
/*
 * @brief Request the transition from the current state to the
given new state
 * @param me the current state machine
```

```
 * @param target the new state to transit to
 * @result transition status
 */
btstack_fsm_state_t btstack_fsm_transit(btstack_fsm_t * const me,
btstack_fsm_state_handler_t target);

/*
 * @brief Constructs a new state hierarchical machine machine,
with storage for maximum hierarchy depth.
 * @param me the current state machine
 * @param initial the initial state
 */
void btstack_fsm_constructor(btstack_fsm_t * const me,
btstack_fsm_state_handler_t initial);

/*
 * @brief Takes the initial transition of the state machine and
sending it a BTSTACK_HSM_INIT_SIG
 * @param me the current state machine
 * @param e event
 */
void btstack_fsm_init(btstack_fsm_t * const me,
btstack_fsm_event_t const * const e);

/*
 * @brief Dispatches the given event to the state machine, if a
transition is requested, leave the old states and enter the new
on.
 *        Handling entering/exiting all states on the way.
 * @param me the current state machine
 * @param e event
 */
btstack_fsm_state_t btstack_fsm_dispatch(btstack_fsm_t * const me,
btstack_fsm_event_t const * const e);

/*
 * @brief Dispatches the given event to the state machine until it
was handled.
 * @param me the current state machine
 * @param e event
```

```
 */
void btstack_fsm_dispatch_until(btstack_fsm_t * const me,
btstack_fsm_event_t const * const e);
```

Human Interface Device (HID) API

btstack_hid.h

```
/*
 * @brief Get boot descriptor data
 * @result data
 */
const uint8_t * btstack_hid_get_boot_descriptor_data(void);

/*
 * @brief Get boot descriptor length
 * @result length
 */
uint16_t btstack_hid_get_boot_descriptor_len(void);
```

HID Parser API

btstack_hid_parser.h : Single-pass HID Report Parser: HID Report is directly parsed without preprocessing HID Descriptor to minimize memory.

```
// HID Descriptor Iterator

/**
 * @brief Init HID Descriptor iterator
 * @param iterator
 * @param hid_descriptor
 * @param hid_descriptor_len
 */
void
btstack_hid_descriptor_iterator_init(btstack_hid_descriptor_iteratc
* iterator, const uint8_t * hid_descriptor, uint16_t
hid_descriptor_len);
```

```
/**
 * @brief Check if HID Descriptor Items are available
 * @param iterator
 * @return
 */
bool
btstack_hid_descriptor_iterator_has_more(btstack_hid_descriptor_ite
* iterator);

/**
 * @brief Get next HID Descriptor Item
 * @param iterator
 * @return
 */
const hid_descriptor_item_t *
btstack_hid_descriptor_iterator_get_item(btstack_hid_descriptor_ite
* iterator);

/**
 * @brief Returns if HID Descriptor was well formed
 * @param iterator
 * @return
 */
bool
btstack_hid_descriptor_iterator_valid(btstack_hid_descriptor_iterat
* iterator);


// HID Descriptor Usage Iterator

/**
 * @brief Initialize usages iterator for HID Descriptor and report
type
 * @param parser
 * @param hid_descriptor
 * @param hid_descriptor_len
 * @param hid_report_type
 */
void btstack_hid_usage_iterator_init(btstack_hid_usage_iterator_t
* iterator, const uint8_t * hid_descriptor, uint16_t
```

```
hid_descriptor_len, hid_report_type_t hid_report_type);

/**
 * @brief Checks if more usages are available
 * @param parser
 */
bool
btstack_hid_usage_iterator_has_more(btstack_hid_usage_iterator_t *
iterator);

/**
 * @brief Get current usage item
 * @param parser
 * @param item
 */
void
btstack_hid_usage_iterator_get_item(btstack_hid_usage_iterator_t *
iterator, btstack_hid_usage_item_t * item);


// HID Report Iterator

/**
 * @brief Initialize HID Parser.
 * @param parser state
 * @param hid_descriptor
 * @param hid_descriptor_len
 * @param hid_report_type
 * @param hid_report
 * @param hid_report_len
 */
void btstack_hid_parser_init(btstack_hid_parser_t * parser, const
uint8_t * hid_descriptor, uint16_t hid_descriptor_len,
hid_report_type_t hid_report_type, const uint8_t * hid_report,
uint16_t hid_report_len);

/**
 * @brief Checks if more fields are available
 * @param parser
 */
```

```
bool btstack_hid_parser_has_more(btstack_hid_parser_t * parser);

/**
 * @brief Get next field
 * @param parser
 * @param usage_page
 * @param usage
 * @param value provided in HID report
 */
void btstack_hid_parser_get_field(btstack_hid_parser_t * parser,
uint16_t * usage_page, uint16_t * usage, int32_t * value);



/**
 * @brief Parses descriptor and returns report size for given
report ID and report type
 * @param report_id or HID_REPORT_ID_UNDEFINED
 * @param report_type
 * @param hid_descriptor
 * @param hid_descriptor_len
 * @return report size in bytes or 0 on parsing error
 */
int btstack_hid_get_report_size_for_id(uint16_t report_id,
hid_report_type_t report_type, const uint8_t *hid_descriptor,
                                       uint16_t
hid_descriptor_len);

/**
 * @brief Parses descriptor and returns status for given report ID
 * @param report_id
 * @param hid_descriptor
 * @param hid_descriptor_len
 * @return status for report id
 */
hid_report_id_status_t btstack_hid_report_id_valid(uint16_t
report_id, const uint8_t * hid_descriptor, uint16_t
hid_descriptor_len);

/**
 * @brief Parses descriptor and returns true if report ID found
```

```
 * @param hid_descriptor
 * @param hid_descriptor_len
 * @return true if report ID declared in descriptor
 */
bool btstack_hid_report_id_declared(const uint8_t *hid_descriptor,
uint16_t hid_descriptor_len);
```

Hierarchical State Machine (HSM) API

btstack_hsm.h

```
/*
 * @brief Request the transition from the current state to the
given new state
 * @param me the current state machine
 * @param target the new state to transit to
 * @result transition status
 */
btstack_hsm_state_t btstack_hsm_transit(btstack_hsm_t * const me,
btstack_hsm_state_handler_t const target);

/*
 * @brief Specifies the upper state in a state hierarchy
 * @param me the current state machine
 * @param target the next parent state in the hierarchy
 * @result transition status
 */
btstack_hsm_state_t btstack_hsm_super(btstack_hsm_t * const me,
btstack_hsm_state_handler_t const target);

/*
 * @brief Placeholder state to mark the top most state in a state
hierarchy, the root state. Ignores all events.
 * @param me the current state machine
 * @param e event
 * @result ignored status
 */
btstack_hsm_state_t btstack_hsm_top(btstack_hsm_t * const me,
btstack_hsm_event_t const * const e);
```

```
/*
 * @brief Constructs a new state hierarchical machine machine,
with storage for maximum hierarchy depth.
 * @param me the current state machine
 * @param initial the initial state
 * @param array of btstack_hsm_state_handler_t elements with the
same number of elements as the maximum number of nested state
machines.
 * @param The number of nested state machines.
 */
void btstack_hsm_constructor(btstack_hsm_t * const me,
btstack_hsm_state_handler_t initial, btstack_hsm_state_handler_t
path[], int8_t depth);

/*
 * @brief Takes the initial transition of the state machine and
sending it a BTSTACK_HSM_INIT_SIG
 * @param me the current state machine
 * @param e event
 */
void btstack_hsm_init(btstack_hsm_t * const me,
btstack_hsm_event_t const * const e);

/*
 * @brief Dispatches the given event to the state machine, if a
transition is requested, leave the old states and enter the new
on.
 *        Honoring the hierarchy and handling entering/exiting all
states on the way.
 * @param me the current state machine
 * @param e event
 */
btstack_hsm_state_t btstack_hsm_dispatch(btstack_hsm_t * const me,
btstack_hsm_event_t const * const e);
```

LC3 Interface API

btstack_lc3.h : Interface for LC3 implementations

```
typedef enum {
    BTSTACK_LC3_FRAME_DURATION_10000US,
    BTSTACK_LC3_FRAME_DURATION_7500US
} btstack_lc3_frame_duration_t;

typedef struct {

    /**
     * Configure Decoder
     * @param context
     * @param sample_rate
     * @param frame_duration
     * @param octets_per_frame
     * @return status
     */
    uint8_t (*configure)(void * context, uint32_t sample_rate,
btstack_lc3_frame_duration_t frame_duration, uint16_t
octets_per_frame);

    /**
     * Decode LC3 Frame into signed 16-bit samples
     * @param context
     * @param bytes
     * @param BFI Bad Frame Indication flags
     * @param pcm_out buffer for decoded PCM samples
     * @param stride count between two consecutive samples
     * @param BEC_detect Bit Error Detected flag
     * @return status
     */
     uint8_t (*decode_signed_16)(void * context, const uint8_t
*bytes, uint8_t BFI,
                         int16_t* pcm_out, uint16_t stride, uint8_t
* BEC_detect);

    /**
     * Decode LC3 Frame into signed 24-bit samples, sign-extended
to 32-bit
     * @param context
     * @param bytes
     * @param BFI Bad Frame Indication flags
```

```
     * @param pcm_out buffer for decoded PCM samples
     * @param stride count between two consecutive samples
     * @param BEC_detect Bit Error Detected flag
     * @return status
     */
    uint8_t (*decode_signed_24)(void * context, const uint8_t
*bytes, uint8_t BFI,
                                int32_t* pcm_out, uint16_t stride,
uint8_t * BEC_detect);

} btstack_lc3_decoder_t;

typedef struct {
    /**
     * Configure Decoder
     * @param context
     * @param sample_rate
     * @param frame_duration
     * @param octets_per_frame
     * @return status
     */
    uint8_t (*configure)(void * context, uint32_t sample_rate,
btstack_lc3_frame_duration_t frame_duration, uint16_t
octets_per_frame);

    /**
     * Encode LC3 Frame with 16-bit signed PCM samples
     * @param context
     * @param pcm_in buffer for decoded PCM samples
     * @param stride count between two consecutive samples
     * @param bytes
     * @return status
     */
    uint8_t (*encode_signed_16)(void * context, const int16_t*
pcm_in, uint16_t stride, uint8_t *bytes);

    /**
     * Encode LC3 Frame with 24-bit signed PCM samples, sign-
extended to 32 bit
     * @param context
```

```
     * @param pcm_in buffer for decoded PCM samples
     * @param stride count between two consecutive samples
     * @param bytes
     * @return status
     */
    uint8_t (*encode_signed_24)(void * context, const int32_t*
pcm_in, uint16_t stride, uint8_t *bytes);

} btstack_lc3_encoder_t;

/**
 * @brief Map enum to ISO Interval in us
 * @param frame_duration enum
 * @return frame_duratoin in us or 0 for invalid frame_duration
enum
 */
uint16_t
btstack_lc3_frame_duration_in_us(btstack_lc3_frame_duration_t
frame_duration);

/**
 * @bbrief Calculate number of samples per ISO Interval
 * @param sample_rate
 * @param frame_duration
 * @return
 */
uint16_t btstack_lc3_samples_per_frame(uint32_t sample_rate,
btstack_lc3_frame_duration_t frame_duration);
```

LC3 Google Adapter API

btstack_lc3_google.h

```
typedef struct {
    lc3_decoder_mem_48k_t           decoder_mem;
    lc3_decoder_t                   decoder;    // pointer
    uint32_t                        sample_rate;
    btstack_lc3_frame_duration_t    frame_duration;
    uint16_t                        octets_per_frame;
} btstack_lc3_decoder_google_t;
```

```
typedef struct {
    lc3_encoder_mem_48k_t           encoder_mem;
    lc3_encoder_t                   encoder;    // pointer
    uint32_t                        sample_rate;
    btstack_lc3_frame_duration_t    frame_duration;
    uint16_t                        octets_per_frame;
} btstack_lc3_encoder_google_t;

/**
 * Init LC3 Decoder Instance
 * @param context for EHIMA LC3 decoder
 */
const btstack_lc3_decoder_t *
btstack_lc3_decoder_google_init_instance(btstack_lc3_decoder_google_t
* context);

/**
 * Init LC3 Decoder Instance
 * @param context for EHIMA LC3 decoder
 */
const btstack_lc3_encoder_t *
btstack_lc3_encoder_google_init_instance(btstack_lc3_encoder_google_t
* context);
```

Adapter for Fraunhofer LC3plus Coddec API

btstack_lc3plus_fraunhofer.h : only uses suitable subset for lc3 testing

```
typedef struct {
    btstack_lc3_frame_duration_t    frame_duration;
    uint16_t                        octets_per_frame;
    uint16_t                        samples_per_frame;
    uint32_t                        sample_rate;
    // decoder must be 4-byte aligned
    uint8_t                         decoder[LC3PLUS_DEC_MAX_SIZE];
} btstack_lc3plus_fraunhofer_decoder_t;

typedef struct {
```

```
    btstack_lc3_frame_duration_t    frame_duration;
    uint16_t                        octets_per_frame;
    uint32_t                        sample_rate;
    // encoder must be 4-byte aligned
    uint8_t                         encoder[LC3PLUS_ENC_MAX_SIZE];
} btstack_lc3plus_fraunhofer_encoder_t;

/**
 * Init LC3 Decoder Instance
 * @param context for Fraunhofer LC3plus decoder
 */
const btstack_lc3_decoder_t *
btstack_lc3plus_fraunhofer_decoder_init_instance(btstack_lc3plus_fr
* context);

/**
 * Init LC3 Encoder Instance
 * @param context for Fraunhofer LC3plus encoder
 */
const btstack_lc3_encoder_t *
btstack_lc3plus_fraunhofer_encoder_init_instance(btstack_lc3plus_fr
* context);
```

Linked List API

btstack_linked_list.h

```
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;


/**
 * @brief Test if list is empty.
 * @param list
 * @return true if list is empty
 */
bool btstack_linked_list_empty(btstack_linked_list_t * list);

/**
 * @brief Add item to list as first element.
 * @param list
 * @param item
 * @return true if item was added, false if item already in list
 */
bool btstack_linked_list_add(btstack_linked_list_t * list,
btstack_linked_item_t *item);

/**
 * @brief Add item to list as last element.
 * @param list
 * @param item
 * @return true if item was added, false if item already in list
 */
bool btstack_linked_list_add_tail(btstack_linked_list_t * list,
btstack_linked_item_t *item);

/**
 * @brief Pop (get + remove) first element.
 * @param list
 * @return first element or NULL if list is empty
 */
btstack_linked_item_t *
btstack_linked_list_pop(btstack_linked_list_t * list);

/**
 * @brief Remove item from list
 * @param list
 * @param item
```

```
 * @return true if item was removed, false if it is no't in list
 */
bool btstack_linked_list_remove(btstack_linked_list_t * list,
btstack_linked_item_t *item);

/**
 * @brief Get first element.
 * @param list
 * @return first element or NULL if list is empty
 */
btstack_linked_item_t *
btstack_linked_list_get_first_item(btstack_linked_list_t * list);

/**
 * @breif Get previous element
 * @param list
 * @param item
 * @return previous element or NULL if not found or element was
first in liest
 */
btstack_linked_item_t *
btstack_linked_list_get_previous_item(btstack_linked_list_t *
list, btstack_linked_item_t * item);

/**
 * @brief Get last element.
 * @param list
 * @return first element or NULL if list is empty
 */
btstack_linked_item_t *
btstack_linked_list_get_last_item(btstack_linked_list_t * list);

/**
 * @brief Counts number of items in list
 * @return number of items in list
 */
int btstack_linked_list_count(btstack_linked_list_t * list);

/**
 * @brief Split list into matching and not-matching items based on
```

```
a criteria
 * @param input list of items
 * @param matches list of matching items
 * @param other all other items
 * @param predicate is called with linked list item and context,
returns true if item matches
 * @param context
 */
void btstack_linked_list_split(btstack_linked_list_t * input,
                               btstack_linked_list_t * matches,
                               btstack_linked_list_t * other,
                               bool (*predicate)(const
btstack_linked_item_t *item, void * context), void * context);

/**
 * @brief Filter matching items into second list
 * @param input list of items
 * @param matches list of matching items
 * @param predicate is called with linked list item and context,
returns true if item matches
 * @param context
 */
void btstack_linked_list_filter(btstack_linked_list_t * input,
                               btstack_linked_list_t * matches,
                               bool (*predicate)(const
btstack_linked_item_t *item, void * context), void * context);

/**
 * @brief Initialize Linked List Iterator
 * @note robust against removal of current element by
btstack_linked_list_remove
 * @param it iterator context
 * @param list
 */
void
btstack_linked_list_iterator_init(btstack_linked_list_iterator_t *
it, btstack_linked_list_t * list);

/**
 * @brief Has next element
```

```
 * @param it iterator context
 * @return true if next element is available
 */
bool
btstack_linked_list_iterator_has_next(btstack_linked_list_iterator_
* it);

/**
 * @brief Get next list eleemnt
 * @param it iterator context
 * @return list element
 */
btstack_linked_item_t *
btstack_linked_list_iterator_next(btstack_linked_list_iterator_t *
it);

/**
 * @brief Remove current list element from list
 * @param it iterator context
 */
void
btstack_linked_list_iterator_remove(btstack_linked_list_iterator_t
* it);
```

Linked Queue API

btstack_linked_queue.h

```
typedef struct {
    btstack_linked_item_t * head;
    btstack_linked_item_t * tail;
} btstack_linked_queue_t;

/**
 * @brief Tests if queue is empty
 * @return true if emtpy
 */
bool btstack_linked_queue_empty(btstack_linked_queue_t * queue);

/**
```

```
 * @brief Append item to queue
 * @param queue
 * @param item
 */
void btstack_linked_queue_enqueue(btstack_linked_queue_t * queue,
btstack_linked_item_t * item);

/**
 * @brief Pop next item from queue
 * @param queue
 * @return item or NULL if queue empty
 */
btstack_linked_item_t *
btstack_linked_queue_dequeue(btstack_linked_queue_t * queue);

/**
 * @brief Get first item from queue
 * @param queue
 * @return item or NULL if queue empty
 */
btstack_linked_item_t *
btstack_linked_queue_first(btstack_linked_queue_t * queue);
```

Network Interface API

btstack_network.h

```
/**
 * @brief Initialize network interface
 * @param send_packet_callback
 */
void btstack_network_init(void (*send_packet_callback)(const
uint8_t * packet, uint16_t size));

/**
 * @brief Bring up network interfacd
 * @param network_address
 * @return 0 if ok
 */
int  btstack_network_up(bd_addr_t network_address);
```

```
/**
 * @brief Shut down network interfacd
 * @param network_address
 * @return 0 if ok
 */
int  btstack_network_down(void);

/**
 * @brief Receive packet on network interface, e.g., forward
packet to TCP/IP stack
 * @param packet
 * @param size
 */
void btstack_network_process_packet(const uint8_t * packet,
uint16_t size);

/**
 * @brief Notify network interface that packet from
send_packet_callback was sent and the next packet can be
delivered.
 */
void btstack_network_packet_sent(void);

/**
 * @brief Get network name after network was activated
 * @note e.g. tapX on Linux, might not be useful on all platforms
 * @return network name
 */
const char * btstack_network_get_name(void);
```

Lienar Resampling API

btstack_resample.h : Linear resampling for 16-bit audio code samples using 16 bit/16 bit fixed point math.

```
/**
 * @brief Init resample context
 * @param num_channels
 * @return btstack_audio implementation
```

```
 */
void btstack_resample_init(btstack_resample_t * context, int
num_channels);

/**
 * @brief Set resampling factor
 * @param factor as fixed point value, identity is 0x10000
 */
void btstack_resample_set_factor(btstack_resample_t * context,
uint32_t factor);

/**
 * @brief Process block of input samples
 * @note size of output buffer is not checked
 * @param input_buffer
 * @param num_frames
 * @param output_buffer
 * @return number destination frames
 */
uint16_t btstack_resample_block(btstack_resample_t * context,
const int16_t * input_buffer, uint32_t num_frames, int16_t *
output_buffer);
```

Ring Buffer API

btstack_ring_buffer.h

```
/**
 * Init ring buffer
 * @param ring_buffer object
 * @param storage
 * @param storage_size in bytes
 */
void btstack_ring_buffer_init(btstack_ring_buffer_t * ring_buffer,
uint8_t * storage, uint32_t storage_size);

/**
 * Reset ring buffer to initial state (empty)
 * @param ring_buffer object
 */
```

```
void btstack_ring_buffer_reset(btstack_ring_buffer_t *
ring_buffer);

/**
 * Check if ring buffer is empty
 * @param ring_buffer object
 * @return TRUE if empty
 */
int btstack_ring_buffer_empty(btstack_ring_buffer_t *
ring_buffer);

/**
 * Get number of bytes available for read
 * @param ring_buffer object
 * @return number of bytes available for read
 */
uint32_t btstack_ring_buffer_bytes_available(btstack_ring_buffer_t
* ring_buffer);

/**
 * Get free space available for write
 * @param ring_buffer object
 * @return number of bytes available for write
 */
uint32_t btstack_ring_buffer_bytes_free(btstack_ring_buffer_t *
ring_buffer);

/**
 * Write bytes into ring buffer
 * @param ring_buffer object
 * @param data to store
 * @param data_length
 * @return 0 if ok, ERROR_CODE_MEMORY_CAPACITY_EXCEEDED if not
enough space in buffer
 */
int btstack_ring_buffer_write(btstack_ring_buffer_t * ring_buffer,
uint8_t * data, uint32_t data_length);

/**
 * Read from ring buffer
```

```
 * @param ring_buffer object
 * @param buffer to store read data
 * @param length to read
 * @param number_of_bytes_read
 */
void btstack_ring_buffer_read(btstack_ring_buffer_t * ring_buffer,
uint8_t * buffer, uint32_t length, uint32_t *
number_of_bytes_read);
```

Sample rate compensation API

btstack_sample_rate_compensation.h : Prevents buffer over/under-run at the audio receiver by compensating for varying/different playback/receiving sample rates.

Intended to measure the L2CAP packet sample rate and with the provided playback sample rate calculates a compensation ratio which compensates for drift between playback and reception.

Requires the audio interface to provide the current playback sample rate.

```
#define FLOAT_TO_Q15(a)   ((signed)((a)*(UINT16_C(1)<<15)+0.5f))
#define FLOAT_TO_Q8(a)    ((signed)((a)*(UINT16_C(1)<<8)+0.5f))
#define FLOAT_TO_Q7(a)    ((signed)((a)*(UINT16_C(1)<<7)+0.5f))

#define Q16_TO_FLOAT(a)   ((float)(a)/(UINT32_C(1)<<16))
#define Q15_TO_FLOAT(a)   ((float)(a)/(UINT32_C(1)<<15))
#define Q8_TO_FLOAT(a)    ((float)(a)/(UINT32_C(1)<<8))
#define Q7_TO_FLOAT(a)    ((float)(a)/(UINT32_C(1)<<7))

//#define DEBUG_RATIO_CALCULATION

typedef struct {
    uint32_t count;         // 17bit are usable to count samples,
recommended for max 96kHz
    uint32_t last;          // time stamp of last measurement
    uint32_t rate_state;    // unsigned Q17.8
    uint32_t ratio_state;   // unsigned Q16.16
    uint32_t constant_playback_sample_rate; // playback sample
```

```
rate if no real one is available
#ifdef DEBUG_RATIO_CALCULATION
    double sample_rate;
    double ratio;
#endif
} btstack_sample_rate_compensation_t;

/**
 * @brief Initialize sample rate compensation
 * @param self pointer to current instance
 * @param time stamp at which to start sample rate measurement
 */
void
btstack_sample_rate_compensation_init( btstack_sample_rate_compensa
*self, uint32_t timestamp_ms, uint32_t sample_rate, uint32_t
ratioQ15 );

/**
 * @brief reset sample rate compensation
 * @param self pointer to current instance
 * @param time stamp at which to start sample rate measurement
 */
void
btstack_sample_rate_compensation_reset( btstack_sample_rate_compens
*self, uint32_t timestamp_ms );

/**
 * @brief update sample rate compensation with the current
playback sample rate decoded samples
 * @param self pointer to current instance
 * @param time stamp for current samples
 * @param samples for current time stamp
 * @param playback sample rate
 */
uint32_t
btstack_sample_rate_compensation_update( btstack_sample_rate_comper
*self, uint32_t timestamp_ms, uint32_t samples, uint32_t
playback_sample_rate );
```

SCO Transport API

btstack_sco_transport.h : Hardware abstraction for PCM/I2S Inteface used for HSP/HFP profiles.

```
typedef struct {

    /**
     * register packet handler for SCO HCI packets
     */
    void (*register_packet_handler)(void (*handler)(uint8_t
packet_type, uint8_t *packet, uint16_t size));

    /**
     * open transport
     * @param con_handle of SCO connection
     * @param sco_format
     */
    void (*open)(hci_con_handle_t con_handle, sco_format_t
sco_format);

    /**
     * send SCO packet
     */
    void (*send_packet)(const uint8_t *buffer, uint16_t length);

    /**
     * close transport
     * @param con_handle of SCO connection
     */
    void (*close)(hci_con_handle_t con_handle);

} btstack_sco_transport_t;
```

SLIP encoder/decoder API

btstack_slip.h

```
// ENCODER
```

```
/**
 * @brief Initialise SLIP encoder with data
 * @param data
 * @param len
 */
void btstack_slip_encoder_start(const uint8_t * data, uint16_t
len);

/**
 * @brief Check if encoder has data ready
 * @return True if data ready
 */
int  btstack_slip_encoder_has_data(void);

/**
 * @brief Get next byte from encoder
 * @return Next bytes from encoder
 */
uint8_t btstack_slip_encoder_get_byte(void);

// DECODER

/**
 * @brief Initialise SLIP decoder with buffer
 * @param buffer to store received data
 * @param max_size of buffer
 */
void btstack_slip_decoder_init(uint8_t * buffer, uint16_t
max_size);

/**
 * @brief Process received byte
 * @param input
 */

void btstack_slip_decoder_process(uint8_t input);

/**
 * @brief Get size of decoded frame
 * @return size of frame. Size = 0 => frame not complete
```

```
 */

uint16_t btstack_slip_decoder_frame_size(void);
```

Tag-Value-Length Persistent Storage (TLV) API

btstack_tlv.h : Inteface for BTstack's Tag Value Length Persistent Storage implementations used to store pairing/bonding data.

```
typedef struct {

    /**
     * Get Value for Tag
     * @param context
     * @param tag
     * @param buffer
     * @param buffer_size
     * @return size of value
     */
    int (*get_tag)(void * context, uint32_t tag, uint8_t * buffer,
uint32_t buffer_size);

    /**
     * Store Tag
     * @param context
     * @param tag
     * @param data
     * @param data_size
     * @return 0 on success
     */
    int (*store_tag)(void * context, uint32_t tag, const uint8_t *
data, uint32_t data_size);

    /**
     * Delete Tag
     *  @note it is not expected that delete operation fails,
please use at least log_error in case of errors
     * @param context
```

```
     * @param tag
     */
    void (*delete_tag)(void * context,  uint32_t tag);

} btstack_tlv_t;

/**
 * @brief Make TLV implementation available to BTstack components
via Singleton
 * @note Usually called by port after BD_ADDR was retrieved from
Bluetooth Controller
 * @param tlv_impl
 * @param tlv_context
 */
void btstack_tlv_set_instance(const btstack_tlv_t * tlv_impl, void
* tlv_context);

/**
 * @brief Get current TLV implementation. Used for bonding
information, but can be used by application, too.
 * @param tlv_impl
 * @param tlv_context
 */
void btstack_tlv_get_instance(const btstack_tlv_t ** tlv_impl,
void ** tlv_context);
```

Empty TLV Instance API

btstack_tlv_none.h : Empty implementation for BTstack's Tag Value Length Persistent Storage implementations No keys are stored. Can be used as placeholder during porting to new platform.

```
/**
 * Init Tag Length Value Store
 * @param context btstack_tlv_none_t
 * @param hal_flash_bank_impl    of hal_flash_bank interface
 * @Param hal_flash_bank_context of hal_flash_bank_interface
 */
const btstack_tlv_t * btstack_tlv_none_init_instance(void);
```

UART API

btstack_uart.h : Common types for UART transports

```
typedef struct {
    /**
     * init transport
     * @param uart_config
     */
    int (*init)(const btstack_uart_config_t * uart_config);

    /**
     * open transport connection
     */
    int (*open)(void);

    /**
     * close transport connection
     */
    int (*close)(void);

    /**
     * set callback for block received. NULL disables callback
     */
    void (*set_block_received)(void (*block_handler)(void));

    /**
     * set callback for sent. NULL disables callback
     */
    void (*set_block_sent)(void (*block_handler)(void));

    /**
     * set baudrate
     */
    int (*set_baudrate)(uint32_t baudrate);

    /**
     * set parity
     */
    int  (*set_parity)(int parity);
```

```
    /**
     * set flowcontrol
     */
    int  (*set_flowcontrol)(int flowcontrol);

    /**
     * receive block
     */
    void (*receive_block)(uint8_t *buffer, uint16_t len);

    /**
     * send block
     */
    void (*send_block)(const uint8_t *buffer, uint16_t length);


    /** Support for different Sleep Modes in TI's H4 eHCILL and in
H5 - can be set to NULL if not used */

    /**
     * query supported wakeup mechanisms
     * @return supported_sleep_modes mask
     */
     int (*get_supported_sleep_modes)(void);

    /**
     * set UART sleep mode - allows to turn off UART and it's
clocks to save energy
     * Supported sleep modes:
     * - off: UART active, RTS low if receive_block was called and
block not read yet
     * - RTS high, wake on CTS: RTS should be high. On CTS pulse,
UART gets enabled again and RTS goes to low
     * - RTS low, wake on RX: data on RX will trigger UART enable,
bytes might get lost
     */
    void (*set_sleep)(btstack_uart_sleep_mode_t sleep_mode);

    /**
```

```
     * set wakeup handler - needed to notify hci transport of
wakeup requests by Bluetooth controller
     * Called upon CTS pulse or RX data. See sleep modes.
     */
    void (*set_wakeup_handler)(void (*wakeup_handler)(void));


    /** Support for HCI H5 Transport Mode - can be set to NULL for
H4 */

    /**
     * H5/SLIP only: set callback for frame received. NULL
disables callback
     */
    void (*set_frame_received)(void (*frame_handler)(uint16_t
frame_size));

    /**
     * H5/SLIP only: set callback for frame sent. NULL disables
callback
     */
    void (*set_frame_sent)(void (*block_handler)(void));

    /**
     * H5/SLIP only: receive SLIP frame
     */
    void (*receive_frame)(uint8_t *buffer, uint16_t len);

    /**
     * H5/SLIP only: send SLIP frame
     */
    void (*send_frame)(const uint8_t *buffer, uint16_t length);

} btstack_uart_t;
```

UART Block API

btstack_uart_block.h : Compatibility layer for ports that use btstack_uart_block_t

```
typedef btstack_uart_t btstack_uart_block_t;

// existing block-only implementations
const btstack_uart_block_t *
btstack_uart_block_windows_instance(void);
const btstack_uart_block_t *
btstack_uart_block_embedded_instance(void);
const btstack_uart_block_t *
btstack_uart_block_freertos_instance(void);

// mapper for extended implementation
static inline const btstack_uart_block_t *
btstack_uart_block_posix_instance(void){
    return (const btstack_uart_block_t *)
btstack_uart_posix_instance();
}
/* API_STOP */

#if defined __cplusplus
}
#endif

#endif
```

UART SLIP Wrapper API

btstack_uart_slip_wrapper.h : Compatibility layer to use new H5 implementation with btstack_uart.h implementations without SLIP support. Using this compatibility layer caused increased processing as it uses single byte UART reads.

If you’re using H5, please consider implement the H5/SLIP functions in your btstack_uart.h or hal_uart_dma.h implementation.

```
/**
 * @brief Initialize SLIP wrapper for existing
btstack_uart_block_t instance without SLIP support
 * @param uart_block_without_slip
 * @return btstack_uart_t instance with SLIP support for use with
hci_trasnport_h5
```

```
 */
const btstack_uart_t * btstack_uart_slip_wrapper_instance(const
btstack_uart_t * uart_without_slip);
```

General Utility Functions API

btstack_util.h

```
/**
 * @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 Calculate delta between two uint32_t points in time
 * @return time_a - time_b - result > 0 if time_a is newer than
time_b
 */
int32_t btstack_time_delta(uint32_t time_a, uint32_t time_b);

/**
 * @brief Calculate delta between two uint16_t points in time
 * @return time_a - time_b - result > 0 if time_a is newer than
time_b
 */
int16_t btstack_time16_delta(uint16_t time_a, uint16_t time_b);

/**
 * @brief Read 08/16/24/32 bit little endian value from buffer
```

```
 * @param buffer
 * @param position in buffer
 * @return value
 */
uint8_t  little_endian_read_08(const uint8_t * buffer, int
position);
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 08/16/32 bit little endian value into buffer
 * @param buffer
 * @param position in buffer
 * @param value
 */
void little_endian_store_08(uint8_t * buffer, uint16_t position,
uint8_t value);
void little_endian_store_16(uint8_t * buffer, uint16_t position,
uint16_t value);
void little_endian_store_24(uint8_t * buffer, uint16_t position,
uint32_t value);
void little_endian_store_32(uint8_t * buffer, uint16_t position,
uint32_t value);

/**
 * @brief Read 08/16/24/32 bit big endian value from buffer
 * @param buffer
 * @param position in buffer
 * @return value
 */
uint32_t big_endian_read_08(const uint8_t* buffer, int position);
uint32_t big_endian_read_16(const uint8_t * buffer, int position);
uint32_t big_endian_read_24(const uint8_t * buffer, int position);
uint32_t big_endian_read_32(const uint8_t * buffer, int position);
```

```
/**
 * @brief Write 08/16/32 bit big endian value into buffer
 * @param buffer
 * @param position in buffer
 * @param value
 */
void big_endian_store_08(uint8_t * buffer, uint16_t position,
uint8_t value);
void big_endian_store_16(uint8_t * buffer, uint16_t position,
uint16_t value);
void big_endian_store_24(uint8_t * buffer, uint16_t position,
uint32_t value);
void big_endian_store_32(uint8_t * buffer, uint16_t position,
uint32_t value);


/**
 * @brief Swap bytes in 16 bit integer
 */
static inline uint16_t btstack_flip_16(uint16_t value){
    return (uint16_t)((value & 0xffu) << 8) | (value >> 8);
}

/**
 * @brief Check for big endian system
 * @return 1 if on big endian
 */
static inline int btstack_is_big_endian(void){
    uint16_t test_value = 0x0100;
    return (int) *(uint8_t*) &test_value;
}

/**
 * @brief Check for little endian system
 * @return 1 if on little endian
 */
static inline int btstack_is_little_endian(void){
    uint16_t test_value = 0x0001;
    return (int) *(uint8_t*) &test_value;
}
```

```
/**
 * @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 Check if all bytes in buffer are zero
 * @param buffer
 * @param size
 * @return true if all bytes is buffer are zero
 */
bool btstack_is_null(const uint8_t * buffer, uint16_t size);

/**
 * @brief Check if all bytes in a bd_addr_t are zero
 * @param addr
 * @return true if all bytes in addr are zero
 */
bool btstack_is_null_bd_addr( const bd_addr_t addr );

/**
 * @brief ASCII character for 4-bit nibble
```

```
 * @return character
 */
char char_for_nibble(uint8_t 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 0 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
 * @param delimiter
 * @return pointer to Bluetooth address string
 */
char * bd_addr_to_str_with_delimiter(const bd_addr_t addr, char
delimiter);

/**
 * @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 Replace address placeholder '00:00:00:00:00:00' with
Bluetooth address
 * @param buffer
 * @param size
 * @param address
 */
void btstack_replace_bd_addr_placeholder(uint8_t * buffer,
uint16_t size, const bd_addr_t address);

/**
 * @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 true if it can be expressed as UUID32
 */
bool 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);

/**
 * @brief Return number of digits of a uint32 number
 * @param uint32_number
 * @return num_digits
 */
int string_len_for_uint32(uint32_t i);

/**
 * @brief Return number of set bits in a uint32 number
 * @param uint32_number
 * @return num_set_bits
 */
int count_set_bits_uint32(uint32_t x);

/**
 * @brief Check CRC8 using ETSI TS 101 369 V6.3.0.
 * @note Only used by RFCOMM
 * @param data
 * @param len
 * @param check_sum
 */
uint8_t btstack_crc8_check(uint8_t * data, uint16_t len, uint8_t
check_sum);
```

```
/**
 * @brief Calculate CRC8 using ETSI TS 101 369 V6.3.0.
 * @note Only used by RFCOMM
 * @param data
 * @param len
 */
uint8_t btstack_crc8_calc(uint8_t * data, uint16_t len);


/**
 * @brief Calculate the initial CRC32 value using ISO 3309 (HDLC),
polynomial (normal) 0x04c11db7
 * @note Used by OTS Service.
 *
 * @return  The initial crc value.
 */
uint32_t btstack_crc32_init(void);

/**
 * @brief Update the CRC32 value with new data.
 *
 * @param crc      The current crc value.
 * @param data     Pointer to a buffer of \a data_len bytes.
 * @param data_len Number of bytes in the \a data buffer.
 * @return             The updated crc value.
 */
uint32_t btstack_crc32_update(uint32_t crc, const uint8_t * data,
uint32_t data_len);

/**
 * @brief Calculate the final CRC32 value.
 *
 * @param crc  The current crc value.
 * @return     The final crc value.
 */
uint32_t btstack_crc32_finalize(uint32_t crc);

/**
 * @brief Get next cid
 * @param current_cid
```

```
 * @return next cid skiping 0
 */
uint16_t btstack_next_cid_ignoring_zero(uint16_t current_cid);

/**
 * @brief Copy string (up to dst_size-1 characters) from src into
dst buffer with terminating '\0'
 * @note replaces strncpy + dst[dst_size-1] = '\0'
 * @param dst
 * @param dst_size
 * @param src
 * @retun bytes_copied including trailing 0
 */
uint16_t btstack_strcpy(char * dst, uint16_t dst_size, const char
* src);

/**
 * @brief Append src string to string in dst buffer with
terminating '\0'
 * @note max total string length will be dst_size-1 characters
 * @param dst
 * @param dst_size
 * @param src
 */
void btstack_strcat(char * dst, uint16_t dst_size, const char *
src);

/**
 * @brief Calculated the number of characters that would get
printed
 * @note same as calling snprintf without a buffer
 * @param format
 * @param argsq
 * @return number of characters, or negative value on error
 */
int btstack_printf_strlen(const char * format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 1, 2)))
#endif
;
```

```
/**
 * @brief Format string into buffer with '\0' and assert it is
large enough
 * @note same as calling snprintf and assert that the string was
not truncated
 * @param buffer
 * @param size of buffer
 * @param format
 * @param argsq
 * @return number of characters
 */
uint16_t btstack_snprintf_assert_complete(char * buffer, size_t
size, const char * format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 3, 4)))
#endif
;

/**
 * @brief Format string into buffer, truncated if necessary.
Output string is '\0' terminated
 * @note similar to calling snprintf but returns the length of the
output string
 * @param buffer
 * @param size of buffer
 * @param format
 * @param argsq
 * @return number of characters
 */
uint16_t btstack_snprintf_best_effort(char * buffer, size_t size,
const char * format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 3, 4)))
#endif
;

/**
 * Returns the number of leading 0-bits in x, starting at the most
```

```
significant bit position.
 * If x is 0, the result is undefined.
 * @note maps to __builtin_clz for gcc and clang
 * @param value
 * @return number of leading 0-bits
 */
uint8_t btstack_clz(uint32_t value);

/**
 * @brief Copy chunk of data into a virtual buffer backed by a
physical buffer.
 * Used to provide chunk of data of larger buffer that is
constructed on the fly, e.g. serializing data struct
 *
 * For example, copy field2 to buffer, with buffer_offset = 11
 *
 *           field1  field2  field3     field4        field5
filed6
 * struct:
-------|-------|----------|-------------|-------|--------------
 * buffer:             ------------------
 * result:             ----|
 *
 * When also copying field3 and field4 to buffer, with
buffer_offset = 11
 *
 *           field1  field2  field3     field4        field5
filed6
 * struct:
-------|-------|----------|-------------|-------|--------------
 * buffer:             ------------------
 * result:             ----|----------|--
 *
 * @param field_data
 * @param field_len
 * @param field_offset  position of field in complete data block
 * @param buffer_data
 * @param buffer_len
 * @param buffer_offset position of buffer in complete data block
 * @return bytes_copied number of bytes actually stored in buffer
```

```
 */
uint16_t btstack_virtual_memcpy(
    const uint8_t * field_data, uint16_t field_len, uint16_t
field_offset,
    uint8_t * buffer, uint16_t buffer_size, uint16_t
buffer_offset);

/**
 * Convert bytes to hex string
 * @param dst buffer for hex string, needs to be twice as large as
src_size + 1
 * @param src_data
 * @param src_size
 */
void btstack_bytes_to_hex(char * dst, const uint8_t * src_data,
uint16_t src_size);

/**
 * Convert hex string to bytes
 * @param dst
 * @param dst_size
 * @param src
 * @return true if conversion was successful
 */
bool btstack_hex_to_bytes(uint8_t * dst, uint16_t dst_size, const
char * src);
```

General Utility Functions API

btstack_version.h

A2DP Sink API

a2dp_sink.h : Audio/Video Distribution Transport Protocol A2DP Sink is a device that accepts streamed media data.

```
/**
 * @brief Create A2DP Sink service record.
 * @param service
```

```
 * @param service_record_handle
 * @param supported_features 16-bit bitmap, see AVDTP_SINK_SF_*
values in avdtp.h
 * @param service_name or NULL for default value. Provide
"" (empty string) to skip attribute
 * @param service_provider_name or NULL for default value. Provide
"" (empty string) to skip attribute
 */
void a2dp_sink_create_sdp_record(uint8_t * service, uint32_t
service_record_handle, uint16_t supported_features, const char *
service_name, const char * service_provider_name);

/**
 * @brief Initialize up A2DP Sink device.
 */
void a2dp_sink_init(void);

/**
 * @brief Create a stream endpoint of type SINK, and register
media codec by specifying its capabilities and the default
configuration.
 * @param media_type                see avdtp_media_type_t values
in avdtp.h (audio, video or multimedia)
 * @param media_codec_type          see avdtp_media_codec_type_t
values in avdtp.h
 * @param codec_capabilities        media codec capabilities as
defined in A2DP spec, section 4 - Audio Codec Interoperability
Requirements.
 * @param codec_capabilities_len    media codec capabilities
length
 * @param codec_configuration       default media codec
configuration
 * @param codec_configuration_len   media codec configuration
length
 *
 * @return local_stream_endpoint
 */
avdtp_stream_endpoint_t *
a2dp_sink_create_stream_endpoint(avdtp_media_type_t media_type,
avdtp_media_codec_type_t media_codec_type,
```

```
                                                                const
uint8_t *codec_capabilities, uint16_t codec_capabilities_len,

uint8_t * codec_configuration, uint16_t codec_configuration_len);

/**
 *  @brief Unregister stream endpoint and free it's memory
 *  @param stream_endpoint created by
a2dp_sink_create_stream_endpoint
 */
void a2dp_sink_finalize_stream_endpoint(avdtp_stream_endpoint_t *
stream_endpoint);

/**
 * @brief Register callback for the A2DP Sink client. It will
receive following subevents of HCI_EVENT_A2DP_META HCI event
type:
 * - A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION:
indicates from remote chosen SBC media codec configuration
 * - A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_OTHER_CONFIGURATION:
indicates from remote chosen other then SBC media codec
configuration
 * - A2DP_SUBEVENT_STREAM_ESTABLISHED:
received when stream to a remote device is established
 * - A2DP_SUBEVENT_STREAM_STARTED:
received when stream is started
 * - A2DP_SUBEVENT_STREAM_SUSPENDED:
received when stream is paused
 * - A2DP_SUBEVENT_STREAM_STOPED:
received when stream is aborted or stopped
 * - A2DP_SUBEVENT_STREAM_RELEASED:
received when stream is released
 * - A2DP_SUBEVENT_SIGNALING_CONNECTION_RELEASED:
received when signaling channel is disconnected
 *
 * @param callback
 */
void a2dp_sink_register_packet_handler(btstack_packet_handler_t
callback);
```

```
/**
 * @brief Register media handler for the A2DP Sink client.
 * @param callback
 * @param packet
 * @param size
 */
void a2dp_sink_register_media_handler(void (*callback)(uint8_t
local_seid, uint8_t *packet, uint16_t size));

/**
 * @brief Establish stream.
 * @param remote
 * @param out_a2dp_cid      Assigned A2DP channel identifier used
for furhter A2DP commands.
 */
uint8_t a2dp_sink_establish_stream(bd_addr_t remote, uint16_t *
out_a2dp_cid);

#ifdef ENABLE_AVDTP_ACCEPTOR_EXPLICIT_START_STREAM_CONFIRMATION
/**
 * @brief Accept starting the stream on
A2DP_SUBEVENT_START_STREAM_REQUESTED event.
 * @param a2dp_cid      A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
uint8_t a2dp_sink_start_stream_accept(uint16_t a2dp_cid, uint8_t
local_seid);

/**
 * @brief Reject starting the stream on
A2DP_SUBEVENT_START_STREAM_REQUESTED event.
 * @param a2dp_cid      A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
uint8_t a2dp_sink_start_stream_reject(uint16_t a2dp_cid, uint8_t
local_seid);
#endif

/**
 * @brief Release stream and disconnect from remote.
```

```
 * @param a2dp_cid          A2DP channel identifier.
 */
void a2dp_sink_disconnect(uint16_t a2dp_cid);

/**
 * @brief Select and configure SBC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     SBC Configuration
 * @return status
 */
uint8_t a2dp_sink_set_config_sbc(uint16_t a2dp_cid, uint8_t
local_seid, uint8_t remote_seid, const avdtp_configuration_sbc_t *
configuration);

/**
 * @brief Select and configure MPEG AUDIO endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     MPEG AUDIO Configuration
 * @return status
 */
uint8_t a2dp_sink_set_config_mpeg_audio(uint16_t a2dp_cid, uint8_t
local_seid, uint8_t remote_seid, const
avdtp_configuration_mpeg_audio_t * configuration);

/**
 * @brief Select and configure MPEG AAC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     MPEG AAC Configuration
 * @return status
 */
uint8_t a2dp_sink_set_config_mpeg_aac(uint16_t a2dp_cid,  uint8_t
local_seid,  uint8_t remote_seid, const
avdtp_configuration_mpeg_aac_t * configuration);
```

```
/**
 * @brief Select and configure ATRAC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     ATRAC Configuration
 * @return status
 */
uint8_t a2dp_sink_set_config_atrac(uint16_t a2dp_cid,  uint8_t
local_seid, uint8_t remote_seid, const avdtp_configuration_atrac_t
* configuration);

/**
 * @brief Select and configure Non-A2DP endpoint. Bytes 0-3 of
codec information contain Vendor ID, bytes 4-5 contain Vendor
Specific Codec ID (little endian)
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param media_codec_information
 * @param media_codec_information_len
 * @return status
 */
uint8_t a2dp_sink_set_config_other(uint16_t a2dp_cid,  uint8_t
local_seid, uint8_t remote_seid, const uint8_t *
media_codec_information, uint8_t media_codec_information_len);

/**
 * @brief Register media configuration validator. Can reject
insuitable configuration or report stream endpoint as currently
busy
 * @note validator has to return AVDTP error codes like:
AVDTP_ERROR_CODE_SEP_IN_USE or
AVDTP_ERROR_CODE_UNSUPPORTED_CONFIGURATION
 *       the callback receives the media configuration in the same
format as the existing
A2dP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 *       and similar
 * @param callback
 */
```

```
void a2dp_sink_register_media_config_validator(uint8_t (*callback)
(const avdtp_stream_endpoint_t * stream_endpoint, const uint8_t *
event, uint16_t size));

/**
 * @brief De-Init A2DP Sink device.
 */
void a2dp_sink_deinit(void);
```

A2DP Source API

a2dp_source.h : Audio/Video Distribution Transport Protocol A2DP Source is a device that streames media data.

```
/**
 * @brief Create A2DP Source service record.
 * @param service
 * @param service_record_handle
 * @param supported_features 16-bit bitmap, see AVDTP_SOURCE_SF_*
values in avdtp.h
 * @param service_name or NULL for default value. Provide
"" (empty string) to skip attribute
 * @param service_provider_name or NULL for default value. Provide
"" (empty string) to skip attribute
 */
void a2dp_source_create_sdp_record(uint8_t * service, uint32_t
service_record_handle, uint16_t supported_features, const char *
service_name, const char * service_provider_name);

/**
 * @brief Initialize up A2DP Source device.
 */
void a2dp_source_init(void);

/**
 * @brief Create a stream endpoint of type SOURCE, and register
media codec by specifying its capabilities and the default
configuration.
 * @param media_type                See avdtp_media_type_t values
in avdtp.h (audio, video or multimedia).
```

```
 * @param media_codec_type          See avdtp_media_codec_type_t
values in avdtp.h
 * @param codec_capabilities        Media codec capabilities as
defined in A2DP spec, section 4 - Audio Codec Interoperability
Requirements.
 * @param codec_capabilities_len    Media codec capabilities
length.
 * @param codec_configuration       Default media codec
configuration.
 * @param codec_configuration_len   Media codec configuration
length.
 *
 * @return local_stream_endpoint
 */
avdtp_stream_endpoint_t *
a2dp_source_create_stream_endpoint(avdtp_media_type_t media_type,
avdtp_media_codec_type_t media_codec_type,
                                                                const
uint8_t *codec_capabilities, uint16_t codec_capabilities_len,

uint8_t * codec_configuration, uint16_t codec_configuration_len);

/**
 *  @brief Unregister stream endpoint and free it's memory
 *  @param stream_endpoint created by
a2dp_source_create_stream_endpoint
 */
void a2dp_source_finalize_stream_endpoint(avdtp_stream_endpoint_t
* stream_endpoint);

/**
 * @brief Register callback for the A2DP Source client. It will
receive following subevents of HCI_EVENT_A2DP_META HCI event
type:
 * - A2DP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW:
Indicates that the next media packet can be sent.
 *
 * - A2DP_SUBEVENT_SIGNALING_CONNECTION_ESTABLISHED
Received when signaling connection with a remote is established.
 * - A2DP_SUBEVENT_SIGNALING_CONNECTION_RELEASED
```

```
Received when signaling connection with a remote is released
 * - A2DP_SUBEVENT_STREAM_ESTABLISHED
Received when stream to a remote device is established.
 * - A2DP_SUBEVENT_STREAM_STARTED
Received when stream is started.
 * - A2DP_SUBEVENT_STREAM_SUSPENDED
Received when stream is paused.
 * - A2DP_SUBEVENT_STREAM_STOPED
received when stream is aborted or stopped.
 * - A2DP_SUBEVENT_STREAM_RELEASED
Received when stream is released.
 * - A2DP_SUBEVENT_SIGNALING_DELAY_REPORTING_CAPABILITY
Currently the only capability that is passed to client
 * - A2DP_SUBEVENT_SIGNALING_CAPABILITIES_DONE
Signals that all capabilities are reported
 * - A2DP_SUBEVENT_SIGNALING_DELAY_REPORT
Delay report
 * - A2DP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
SBC configuration
 * - A2DP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
Signals that a media packet can be sent
 * - A2DP_SUBEVENT_COMMAND_REJECTED
Command reject
 * @param callback
 */
void a2dp_source_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Open stream.
 * @param remote_addr
 * @param avdtp_cid         Assigned A2DP channel identifier used
for further A2DP commands.
 */
uint8_t a2dp_source_establish_stream(bd_addr_t remote_addr,
uint16_t *avdtp_cid);

    /**
     * @brief Reconfigure stream.
     * @param local_seid          ID assigned to a local stream
```

```
endpoint
     * @param sampling_frequency  New sampling frequency to use.
Cannot be called while stream is active
     */
uint8_t a2dp_source_reconfigure_stream_sampling_frequency(uint16_t
a2dp_cid, uint32_t sampling_frequency);

/**
 * @brief Start stream.
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
uint8_t a2dp_source_start_stream(uint16_t a2dp_cid, uint8_t
local_seid);

/**
 * @brief Pause stream.
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
uint8_t a2dp_source_pause_stream(uint16_t a2dp_cid, uint8_t
local_seid);

/**
 * @brief Release stream and disconnect from remote.
 * @param a2dp_cid          A2DP channel identifier.
 */
uint8_t a2dp_source_disconnect(uint16_t a2dp_cid);

/**
 * @brief Request to send a media packet. Packet can be then sent
on reception of A2DP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
event.
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
void a2dp_source_stream_endpoint_request_can_send_now(uint16_t
a2dp_cid, uint8_t local_seid);

/**
```

```
 * @brief Return maximal media payload size, does not include
media header.
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @return max_media_payload_size_without_media_header
 */
int a2dp_max_media_payload_size(uint16_t a2dp_cid, uint8_t
local_seid);

/**
 * @brief Send media payload.
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param marker
 * @param timestamp         in sample rate units
 * @param payload
 * @param payload_size
 * @param marker
 * @return status
 */
uint8_t a2dp_source_stream_send_media_payload_rtp(uint16_t
a2dp_cid, uint8_t local_seid, uint8_t marker, uint32_t timestamp,
                                          uint8_t *payload,
uint16_t payload_size);

/**
 * @brief Send media packet
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param packet
 * @param size
 * @return status
 */
uint8_t a2dp_source_stream_send_media_packet(uint16_t a2dp_cid,
uint8_t local_seid, const uint8_t * packet, uint16_t size);

/**
 * @brief Select and configure SBC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
```

```
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     SBC Configuration
 * @return status
 */
uint8_t a2dp_source_set_config_sbc(uint16_t a2dp_cid, uint8_t
local_seid, uint8_t remote_seid, const avdtp_configuration_sbc_t *
configuration);

/**
 * @brief Select and configure MPEG AUDIO endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     MPEG AUDIO Configuration
 * @return status
 */
uint8_t a2dp_source_set_config_mpeg_audio(uint16_t a2dp_cid,
uint8_t local_seid, uint8_t remote_seid, const
avdtp_configuration_mpeg_audio_t * configuration);

/**
 * @brief Select and configure MPEG AAC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     MPEG AAC Configuration
 * @return status
 */
uint8_t a2dp_source_set_config_mpeg_aac(uint16_t a2dp_cid,
uint8_t local_seid,  uint8_t remote_seid, const
avdtp_configuration_mpeg_aac_t * configuration);

/**
 * @brief Select and configure ATRAC endpoint
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param configuration     ATRAC Configuration
 * @return status
 */
```

```
uint8_t a2dp_source_set_config_atrac(uint16_t a2dp_cid,  uint8_t
local_seid, uint8_t remote_seid, const avdtp_configuration_atrac_t
* configuration);

/**
 * @brief Select and configure Non-A2DP endpoint. Bytes 0-3 of
codec information contain Vendor ID, bytes 4-5 contain Vendor
Specific Codec ID (little endian)
 * @param a2dp_cid          A2DP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param remote_seid       ID of a remote stream endpoint.
 * @param media_codec_information
 * @param media_codec_information_len
 * @return status
 */
uint8_t a2dp_source_set_config_other(uint16_t a2dp_cid,  uint8_t
local_seid, uint8_t remote_seid, const uint8_t *
media_codec_information, uint8_t media_codec_information_len);

/**
 * @brief Register media configuration validator. Can reject
insuitable configuration or report stream endpoint as currently
busy
 * @note validator has to return AVDTP error codes like:
AVDTP_ERROR_CODE_SEP_IN_USE or
AVDTP_ERROR_CODE_UNSUPPORTED_CONFIGURATION
 *       the callback receives the media configuration in the same
format as the existing
A2dP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 *       and similar
 * @param callback
 */
void a2dp_source_register_media_config_validator(uint8_t
(*callback)(const avdtp_stream_endpoint_t * stream_endpoint, const
uint8_t * event, uint16_t size));

/**
 * @brief De-Init A2DP Source device.
 */
void a2dp_source_deinit(void);
```

AVDTP Sink API

avdtp_sink.h : Audio/Video Distribution Transport Protocol (AVDTP) Sink is a device that accepts streamed media data.

```
/**
 * @brief Set up AVDTP Sink device.
 */
void avdtp_sink_init(void);

// returns avdtp_stream_endpoint_t *
avdtp_stream_endpoint_t *
avdtp_sink_create_stream_endpoint(avdtp_sep_type_t sep_type,
avdtp_media_type_t media_type);

/**
 *  @brief Unregister stream endpoint and free it's memory
 *  @param stream_endpoint created by
avdtp_sink_create_stream_endpoint
 */
void avdtp_sink_finalize_stream_endpoint(avdtp_stream_endpoint_t *
stream_endpoint);

void avdtp_sink_register_media_transport_category(uint8_t seid);
void avdtp_sink_register_reporting_category(uint8_t seid);
void avdtp_sink_register_delay_reporting_category(uint8_t seid);
void avdtp_sink_register_recovery_category(uint8_t seid, uint8_t
maximum_recovery_window_size, uint8_t
maximum_number_media_packets);
void avdtp_sink_register_header_compression_category(uint8_t seid,
uint8_t back_ch, uint8_t media, uint8_t recovery);
void avdtp_sink_register_multiplexing_category(uint8_t seid,
uint8_t fragmentation);

void avdtp_sink_register_media_codec_category(uint8_t seid,
avdtp_media_type_t media_type, avdtp_media_codec_type_t
media_codec_type, const uint8_t *media_codec_info, uint16_t
media_codec_info_len);
void avdtp_sink_register_content_protection_category(uint8_t seid,
uint16_t cp_type, const uint8_t * cp_type_value, uint8_t
```

```
cp_type_value_len);

/**
 * @brief Register callback for the AVDTP Sink client.
 * @param callback
 */
void avdtp_sink_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Connect to device with a bluetooth address. (and perform
configuration?)
 * @param bd_addr
 * @param avdtp_cid Assigned avdtp cid
 */
uint8_t avdtp_sink_connect(bd_addr_t bd_addr, uint16_t *
avdtp_cid);

void avdtp_sink_register_media_handler(void (*callback)(uint8_t
local_seid, uint8_t *packet, uint16_t size));
/**
 * @brief Disconnect from device with connection handle.
 * @param avdtp_cid
 */
uint8_t avdtp_sink_disconnect(uint16_t avdtp_cid);

/**
 * @brief Discover stream endpoints
 * @param avdtp_cid
 */
uint8_t avdtp_sink_discover_stream_endpoints(uint16_t avdtp_cid);

/**
 * @brief Get capabilities
 * @param avdtp_cid
 */
uint8_t avdtp_sink_get_capabilities(uint16_t avdtp_cid, uint8_t
acp_seid);

/**
```

```
 * @brief Get all capabilities
 * @param avdtp_cid
 */
uint8_t avdtp_sink_get_all_capabilities(uint16_t avdtp_cid,
uint8_t acp_seid);

/**
 * @brief Set configuration
 * @param avdtp_cid
 */
uint8_t avdtp_sink_set_configuration(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid, uint16_t configured_services_bitmap,
avdtp_capabilities_t configuration);

/**
 * @brief Reconfigure stream
 * @param avdtp_cid
 * @param seid
 */
uint8_t avdtp_sink_reconfigure(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid, uint16_t configured_services_bitmap,
avdtp_capabilities_t configuration);

/**
 * @brief Get configuration
 * @param avdtp_cid
 */
uint8_t avdtp_sink_get_configuration(uint16_t avdtp_cid, uint8_t
acp_seid);


/**
 * @brief Open stream
 * @param avdtp_cid
 * @param seid
 */
uint8_t avdtp_sink_open_stream(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid);
```

```
/**
 * @brief Start stream
 * @param local_seid
 */
uint8_t avdtp_sink_start_stream(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Abort stream
 * @param local_seid
 */
uint8_t avdtp_sink_abort_stream(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Start stream
 * @param local_seid
 */
uint8_t avdtp_sink_stop_stream(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Suspend stream
 * @param local_seid
 */
uint8_t avdtp_sink_suspend(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Report delay
 * @param local_seid
 * @param delay_100us
 */
uint8_t avdtp_sink_delay_report(uint16_t avdtp_cid, uint8_t
local_seid, uint16_t delay_100us);

/**
 * @brief Register media configuration validator. Can reject
insuitable configuration or report stream endpoint as currently
busy
```

```
 * @note validator has to return AVDTP error codes like:
AVDTP_ERROR_CODE_SEP_IN_USE or
AVDTP_ERROR_CODE_UNSUPPORTED_CONFIGURATION
 *       the callback receives the media configuration in the same
format as the existing
AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 *       and similar
 * @param callback
 */
void avdtp_sink_register_media_config_validator(uint8_t
(*callback)(const avdtp_stream_endpoint_t * stream_endpoint, const
uint8_t * event, uint16_t size));

/**
 * @brief De-Init AVDTP Sink.
 */
void avdtp_sink_deinit(void);

// AVDTP_SI_DELAYREPORT
```

AVDTP Source API

avdtp_source.h : Audio/Video Distribution Transport Protocol (AVDTP) Source is a device that streames media data.

```
/**
 * @brief Register media transport category with local stream
endpoint identified by seid
 * @param seid
 */
void avdtp_source_register_media_transport_category(uint8_t seid);

/**
 * @brief Register reporting category with local stream endpoint
identified by seid
 * @param seid
 */
void avdtp_source_register_reporting_category(uint8_t seid);

/**
```

```
 * @brief Register delay reporting category with local stream
endpoint identified by seid
 * @param seid
 */
void avdtp_source_register_delay_reporting_category(uint8_t seid);

/**
 * @brief Register recovery category with local stream endpoint
identified by seid
 * @param seid
 * @param maximum_recovery_window_size
 * @param maximum_number_media_packets
 */
void avdtp_source_register_recovery_category(uint8_t seid, uint8_t
maximum_recovery_window_size, uint8_t
maximum_number_media_packets);

/**
 * @brief Register content protection category with local stream
endpoint identified by seid
 * @param seid
 * @param cp_type
 * @param cp_type_value
 * @param cp_type_value_len
 */
void avdtp_source_register_content_protection_category(uint8_t
seid, uint16_t cp_type, const uint8_t * cp_type_value, uint8_t
cp_type_value_len);

/**
 * @brief Register header compression category with local stream
endpoint identified by seid
 * @param seid
 * @param back_ch
 * @param media
 * @param recovery
 */
void avdtp_source_register_header_compression_category(uint8_t
seid, uint8_t back_ch, uint8_t media, uint8_t recovery);
```

```
/**
 * @brief Register media codec category with local stream endpoint
identified by seid
 * @param seid
 * @param media_type
 * @param media_codec_type
 * @param media_codec_info
 * @param media_codec_info_len
 */
void avdtp_source_register_media_codec_category(uint8_t seid,
avdtp_media_type_t media_type, avdtp_media_codec_type_t
media_codec_type, const uint8_t *media_codec_info, uint16_t
media_codec_info_len);

/**
 * @brief Register multiplexing category with local stream
endpoint identified by seid
 * @param seid
 * @param fragmentation
 */
void avdtp_source_register_multiplexing_category(uint8_t seid,
uint8_t fragmentation);

/**
 * @brief Initialize up AVDTP Source device.
 */
void avdtp_source_init(void);

/**
 * @brief Register callback for the AVDTP Source client. See
btstack_defines.h for AVDTP_SUBEVENT_* events
 *
 * @param callback
 */
void avdtp_source_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Connect to device with a bluetooth address. (and perform
configuration?)
```

```
 * @param bd_addr
 * @param avdtp_cid Assigned avdtp cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
BTSTACK_MEMORY_ALLOC_FAILED, SDP_QUERY_BUSY
 */
uint8_t avdtp_source_connect(bd_addr_t bd_addr, uint16_t *
avdtp_cid);

/**
 * @brief Disconnect from device with connection handle.
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER
 */
uint8_t avdtp_source_disconnect(uint16_t avdtp_cid);

/**
 * @brief Discover stream endpoints
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_discover_stream_endpoints(uint16_t
avdtp_cid);

/**
 * @brief Get capabilities
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_get_capabilities(uint16_t avdtp_cid, uint8_t
acp_seid);

/**
 * @brief Get all capabilities
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
```

```
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_get_all_capabilities(uint16_t avdtp_cid,
uint8_t acp_seid);

/**
 * @brief Set configuration
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_set_configuration(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid, uint16_t configured_services_bitmap,
avdtp_capabilities_t configuration);

/**
 * @brief Reconfigure stream
 * @param avdtp_cid
 * @param seid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_reconfigure(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid, uint16_t configured_services_bitmap,
avdtp_capabilities_t configuration);

/**
 * @brief Get configuration
 * @param avdtp_cid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_get_configuration(uint16_t avdtp_cid, uint8_t
acp_seid);
```

```
/**
 * @brief Open stream
 * @param avdtp_cid
 * @param seid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_open_stream(uint16_t avdtp_cid, uint8_t
int_seid, uint8_t acp_seid);

/**
 * @brief Start stream
 * @param local_seid
* @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_start_stream(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Abort stream
 * @param local_seid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_abort_stream(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Start stream
 * @param local_seid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_stop_stream(uint16_t avdtp_cid, uint8_t
local_seid);
```

```
/**
 * @brief Suspend stream
 * @param local_seid
 * @return status ERROR_CODE_SUCCESS if succesful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t avdtp_source_suspend(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Create stream endpoint
 * @param sep_type          AVDTP_SOURCE or AVDTP_SINK, see
avdtp.h
 * @param media_type        AVDTP_AUDIO, AVDTP_VIDEO or
AVDTP_MULTIMEDIA, see avdtp.h
 * @return pointer to newly created stream endpoint, or NULL if
allocation failed
 */
avdtp_stream_endpoint_t *
avdtp_source_create_stream_endpoint(avdtp_sep_type_t sep_type,
avdtp_media_type_t media_type);

/**
 *  @brief Unregister stream endpoint and free it's memory
 *  @param stream_endpoint created by
avdtp_sink_create_stream_endpoint
 */
void avdtp_source_finalize_stream_endpoint(avdtp_stream_endpoint_t
* stream_endpoint);

/**
 * @brief Send media packet
 * @param avdtp_cid         AVDTP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param packet
 * @param size
 * @return status
 */
```

```
uint8_t avdtp_source_stream_send_media_packet(uint16_t avdtp_cid,
uint8_t local_seid, const uint8_t * packet, uint16_t size);

/**
 * @brief Send media payload including RTP header
 * @param avdtp_cid         AVDTP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 * @param marker
 * @param timestamp         in sample rate units
 * @param payload
 * @param size
 * @return status
 */
uint8_t avdtp_source_stream_send_media_payload_rtp(uint16_t
avdtp_cid, uint8_t local_seid, uint8_t marker, uint32_t timestamp,
                                            const uint8_t *payload,
uint16_t size);

/**
 * @brief Request to send a media packet. Packet can be then sent
on reception of AVDTP_SUBEVENT_STREAMING_CAN_SEND_MEDIA_PACKET_NOW
event.
 * @param avdtp_cid         AVDTP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
void avdtp_source_stream_endpoint_request_can_send_now(uint16_t
avddp_cid, uint8_t local_seid);

/**
 * @brief Return maximal media payload size, does not include
media header.
 * @param avdtp_cid         AVDTP channel identifier.
 * @param local_seid        ID of a local stream endpoint.
 */
int avdtp_max_media_payload_size(uint16_t avdtp_cid, uint8_t
local_seid);

/**
 * @brief Register media configuration validator. Can reject
insuitable configuration or report stream endpoint as currently
```

```
busy
 * @note validator has to return AVDTP error codes like:
AVDTP_ERROR_CODE_SEP_IN_USE or
AVDTP_ERROR_CODE_UNSUPPORTED_CONFIGURATION
 *       the callback receives the media configuration in the same
format as the existing
AVDTP_SUBEVENT_SIGNALING_MEDIA_CODEC_SBC_CONFIGURATION
 *       and similar
 * @param callback
 */
void avdtp_source_register_media_config_validator(uint8_t
(*callback)(const avdtp_stream_endpoint_t * stream_endpoint, const
uint8_t * event, uint16_t size));

/**
 * @brief De-Init AVDTP Source.
 */
void avdtp_source_deinit(void);
```

AVRCP Browsing API

avrcp_browsing.h

```
/**
 * @brief Set up AVRCP Browsing service
 */
void avrcp_browsing_init(void);

/**
 * @brief Register callback for the AVRCP Browsing Controller
client.
 * @param callback
 */
void
avrcp_browsing_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief   Connect to AVRCP Browsing service on a remote device,
emits AVRCP_SUBEVENT_BROWSING_CONNECTION_ESTABLISHED with status
```

```
 * @param   remote_addr
 * @param   ertm_buffer
 * @param   ertm_buffer_size
 * @param   ertm_config
 * @param   avrcp_browsing_cid  outgoing parameter, valid if
status == ERROR_CODE_SUCCESS
 * @return status
 */
uint8_t avrcp_browsing_connect(bd_addr_t remote_addr, uint8_t *
ertm_buffer, uint32_t ertm_buffer_size, l2cap_ertm_config_t *
ertm_config, uint16_t * avrcp_browsing_cid);

/**
 * @brief Configure incoming connection for Browsing Service.
 * @param avrcp_browsing_cid
 * @param ertm_buffer
 * @param ertm_buffer_size
 * @param ertm_config
 * @return status
 */
uint8_t avrcp_browsing_configure_incoming_connection(uint16_t
avrcp_browsing_cid, uint8_t * ertm_buffer, uint32_t
ertm_buffer_size, l2cap_ertm_config_t * ertm_config);

/**
 * @brief Decline incoming connection Browsing Service.
 * @param avrcp_browsing_cid
 * @return status
 */
uint8_t avrcp_browsing_decline_incoming_connection(uint16_t
avrcp_browsing_cid);

/**
 * @brief   Disconnect from AVRCP Browsing service
 * @param   avrcp_browsing_cid
 * @return status
 */
uint8_t avrcp_browsing_disconnect(uint16_t avrcp_browsing_cid);

/**
```

```
 * @brief De-Init AVRCP Browsing
 */
void avrcp_browsing_deinit(void);
```

AVRCP Browsing Controller API

avrcp_browsing_controller.h

```
/**
 * @brief Set up AVRCP Browsing Controller device.
 */
void avrcp_browsing_controller_init(void);

/**
 * @brief Register callback for the AVRCP Browsing Controller
client to receive:
 * - event AVRCP_SUBEVENT_BROWSING_DONE marking the end of
operation. If the browsing status field of this field is equal to
AVRCP_BROWSING_ERROR_CODE_SUCCESS, operation was successful.
 * - data packet marked by AVRCP_BROWSING_DATA_PACKET subpacket
type containg the response load.
 *
 * @param callback
 */
void
avrcp_browsing_controller_register_packet_handler(btstack_packet_ha
callback);

/**
 * @brief Retrieve a list of media players.
 * @param avrcp_browsing_cid
 * @param start_item
 * @param end_item
 * @param attr_bitmap Use AVRCP_MEDIA_ATTR_ALL for all, and
AVRCP_MEDIA_ATTR_NONE for none. Otherwise, see
avrcp_media_attribute_id_t for the bitmap position of attrs.
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
```

```
with previous query.
 */
uint8_t avrcp_browsing_controller_get_media_players(uint16_t
avrcp_browsing_cid, uint32_t start_item, uint32_t end_item,
uint32_t attr_bitmap);

/**
 * @brief Retrieve a list of folders and media items of the
browsed player.
 * @param avrcp_browsing_cid
 * @param start_item
 * @param end_item
 * @param attr_bitmap Use AVRCP_MEDIA_ATTR_ALL for all, and
AVRCP_MEDIA_ATTR_NONE for none. Otherwise, see
avrcp_media_attribute_id_t for the bitmap position of attrs.
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_controller_browse_file_system(uint16_t
avrcp_browsing_cid, uint32_t start_item, uint32_t end_item,
uint32_t attr_bitmap);

/**
 * @brief Retrieve a list of media items of the browsed player.
 * @param avrcp_browsing_cid
 * @param start_item
 * @param end_item
 * @param attr_bitmap Use AVRCP_MEDIA_ATTR_ALL for all, and
AVRCP_MEDIA_ATTR_NONE for none. Otherwise, see
avrcp_media_attribute_id_t for the bitmap position of attrs.
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_controller_browse_media(uint16_t
```

```
avrcp_browsing_cid, uint32_t start_item, uint32_t end_item,
uint32_t attr_bitmap);

/**
 * @brief Retrieve a list of folders and media items of the
addressed player.
 * @param avrcp_browsing_cid
 * @param start_item
 * @param end_item
 * @param attr_bitmap Use AVRCP_MEDIA_ATTR_ALL for all, and
AVRCP_MEDIA_ATTR_NONE for none. Otherwise, see
avrcp_media_attribute_id_t for the bitmap position of attrs.
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_controller_browse_now_playing_list(uint16_t
avrcp_browsing_cid, uint32_t start_item, uint32_t end_item,
uint32_t attr_bitmap);

/**
 * @brief Set browsed player. Calling this command is required
prior to browsing the player's file system. Some players may
support browsing only when set as the Addressed Player.
 * @param avrcp_browsing_cid
 * @param browsed_player_id
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_controller_set_browsed_player(uint16_t
avrcp_browsing_cid, uint16_t browsed_player_id);

/**
 * @brief Get total num attributes
 * @param avrcp_browsing_cid
```

```
 * @param scope
 */
uint8_t
avrcp_browsing_controller_get_total_nr_items_for_scope(uint16_t
avrcp_browsing_cid, avrcp_browsing_scope_t scope);

/**
 * @brief Navigate one level up or down in thhe virtual
filesystem. Requires that s browsed player is set.
 * @param avrcp_browsing_cid
 * @param direction     0-folder up, 1-folder down
 * @param folder_uid    8 bytes long
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_controller_change_path(uint16_t
avrcp_browsing_cid, uint8_t direction, uint8_t * folder_uid);
uint8_t avrcp_browsing_controller_go_up_one_level(uint16_t
avrcp_browsing_cid);
uint8_t avrcp_browsing_controller_go_down_one_level(uint16_t
avrcp_browsing_cid, uint8_t * folder_uid);


/**
 * @brief Retrives metadata information (title, artist,
album, ...) about a media element with given uid.
 * @param avrcp_browsing_cid
 * @param uid            media element uid
 * @param uid_counter    Used to detect change to the media
database on target device. A TG device that supports the UID
Counter shall update the value of the counter on each change to
the media database.
 * @param attr_bitmap    0x00000000 - retrieve all, chek
avrcp_media_attribute_id_t in avrcp.h for detailed bit position
description.
 * @param scope          check avrcp_browsing_scope_t in avrcp.h
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
```

```
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_controller_get_item_attributes_for_scope(uint16_t
avrcp_browsing_cid, uint8_t * uid, uint16_t uid_counter, uint32_t
attr_bitmap, avrcp_browsing_scope_t scope);

/**
 * @brief Searches are performed from the current folder in the
Browsed Players virtual filesystem. The search applies to the
current folder and all folders below that.
 * @param avrcp_browsing_cid
 * @param search_str_len
 * @param search_str
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query,
 *       - ERROR_CODE_PARAMETER_OUT_OF_MANDATORY_RANGE if search
string is NULL.
 */
uint8_t avrcp_browsing_controller_search(uint16_t
avrcp_browsing_cid, uint16_t search_str_len, char * search_str);

/**
 * @brief De-Init AVRCP Browsing Controller
 */
void avrcp_browsing_controller_deinit(void);
```

AVRCP Browsing Target API

avrcp_browsing_target.h

```
/**
 * @brief Set up AVRCP Browsing Target client.
 */
```

```
void avrcp_browsing_target_init(void);

/**
 * @brief Register callback for the AVRCP Browsing Target client
to receive events:
 * - AVRCP_SUBEVENT_BROWSING_SEARCH
 * - AVRCP_SUBEVENT_BROWSING_GET_FOLDER_ITEMS
 * - AVRCP_PDU_ID_GET_TOTAL_NUMBER_OF_ITEMS
 * - AVRCP_SUBEVENT_BROWSING_SET_BROWSED_PLAYER
 * - AVRCP_SUBEVENT_BROWSING_CHANGE_PATH
 * - AVRCP_SUBEVENT_BROWSING_GET_ITEM_ATTRIBUTES
 *
 * @param callback
 */
void
avrcp_browsing_target_register_packet_handler(btstack_packet_handle
callback);

/**
 * @brief Accept set browsed player
 * @param browsing_cid
 * @param uid_counter
 * @param browsed_player_id
 * @param response
 * @param response_size
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_target_send_accept_set_browsed_player(uint16_t
browsing_cid, uint16_t uid_counter, uint16_t browsed_player_id,
uint8_t * response, uint16_t response_len);

/**
 * @brief Reject set browsed player
 * @param browsing_cid
 * @param status
```

```
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_target_send_reject_set_browsed_player(uint16_t
browsing_cid, avrcp_status_code_t status);

/**
 * @brief Send answer to get folder items query on event
AVRCP_SUBEVENT_BROWSING_GET_FOLDER_ITEMS. The first byte of this
event defines the scope of the query, see avrcp_browsing_scope_t.
 * @param browsing_cid
 * @param uid_counter
 * @param attr_list
 * @param attr_list_size
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_target_send_get_folder_items_response(uint16_t
browsing_cid, uint16_t uid_counter, uint8_t * attr_list, uint16_t
attr_list_size, uint16_t num_items);

/**
 * @brief Send answer to get total number of items query on event
AVRCP_SUBEVENT_BROWSING_GET_TOTAL_NUM_ITEMS. The first byte of
this event defines the scope of the query, see
avrcp_browsing_scope_t.
 * @param browsing_cid
 * @param uid_counter
 * @param total_num_items
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
```

```
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_target_send_get_total_num_items_response(uint16_t
browsing_cid, uint16_t uid_counter, uint32_t total_num_items);

/**
 * @brief Send answer to change path command on event
AVRCP_SUBEVENT_BROWSING_CHANGE_PATH.
 * @param browsing_cid
 * @param status
 * @param num_items
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_target_send_change_path_response(uint16_t
browsing_cid, avrcp_status_code_t status, uint32_t num_items);

/**
 * @brief Send answer to get item attribute response on event
AVRCP_SUBEVENT_BROWSING_GET_ITEM_ATTRIBUTES.
 * @param browsing_cid
 * @param status
 * @param attr_list
 * @param attr_list_size
 * @param num_items
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t
avrcp_browsing_target_send_get_item_attributes_response(uint16_t
browsing_cid, avrcp_status_code_t status, uint8_t * attr_list,
uint16_t attr_list_size, uint8_t num_items);
```

```
/**
 * @brief Send answer to search query on event
AVRCP_SUBEVENT_BROWSING_SEARCH.
 * @param browsing_cid
 * @param uid_counter
 * @param total_num_items
 * @return ERROR_CODE_SUCCESS if successful, otherwise:
 *       - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if unknown
connection cid,
 *       - ERROR_CODE_COMMAND_DISALLOWED if client is not done
with previous query.
 */
uint8_t avrcp_browsing_target_send_search_response(uint16_t
browsing_cid, avrcp_status_code_t status, uint16_t uid_counter,
uint32_t num_items);

/**
 * @brief De-Init AVRCP Browsing Target
 */
void avrcp_browsing_target_deinit(void);
```

AVRCP Controller API

avrcp_controller.h

```
typedef enum {
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_CATEGORY_PLAYER_OR_RECORDER
= 0,

AVRCP_CONTROLLER_SUPPORTED_FEATURE_CATEGORY_MONITOR_OR_AMPLIFIER,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_CATEGORY_TUNER,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_CATEGORY_MENU,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_RESERVED_4,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_RESERVED_5,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_BROWSING,

AVRCP_CONTROLLER_SUPPORTED_FEATURE_COVER_ART_GET_IMAGE_PROPERTIES,
    AVRCP_CONTROLLER_SUPPORTED_FEATURE_COVER_ART_GET_IMAGE,
```

```
AVRCP_CONTROLLER_SUPPORTED_FEATURE_COVER_ART_GET_LINKED_THUMBNAIL
} avrcp_controller_supported_feature_t;

/**
 * @brief AVRCP Controller service record.
 * @param service
 * @param service_record_handle
 * @param supported_features 16-bit bitmap, see
AVRCP_FEATURE_MASK_* in avrcp.h
 * @param service_name or NULL for default value. Provide
"" (empty string) to skip attribute
 * @param service_provider_name or NULL for default value. Provide
"" (empty string) to skip attribute
 */
void avrcp_controller_create_sdp_record(uint8_t * service,
uint32_t service_record_handle, uint16_t supported_features, const
char * service_name, const char * service_provider_name);

/**
 * @brief Set up AVRCP Controller service.
 */
void avrcp_controller_init(void);

/**
 * @brief Register callback for the AVRCP Controller client.
 * @param callback
 */
void
avrcp_controller_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Set max num fragments in whuch message can be
transmited.
 * @param avrcp_cid
 * @param max_num_fragments
 * @return status
 */
uint8_t avrcp_controller_set_max_num_fragments(uint16_t avrcp_cid,
uint8_t max_num_fragments);
```

```
/**
 * @brief Unit info.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_unit_info(uint16_t avrcp_cid);

/**
 * @brief Subunit info.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_subunit_info(uint16_t avrcp_cid);

/**
 * @brief Get capabilities.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_get_supported_company_ids(uint16_t
avrcp_cid);

/**
 * @brief Get supported Events.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_get_supported_events(uint16_t avrcp_cid);


/**
 * @brief Start continuous cmd (play, pause, volume up, ...).
Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and
status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_start_press_and_hold_cmd(uint16_t
```

```
avrcp_cid, avrcp_operation_id_t operation_id);

/**
 * @brief Stops continuous cmd (play, pause, volume up, ...).
Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and
status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_release_press_and_hold_cmd(uint16_t
avrcp_cid);

/**
 * Send PASS THROUGH command. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @param operation_id
 * @return
 */
uint8_t avrcp_controller_send_pass_through_cmd(uint16_t avrcp_cid,
avrcp_operation_id_t operation_id);

/**
 * @brief Play. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_play(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_play(uint16_t avrcp_cid);

/**
 * @brief Stop. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_stop(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_stop(uint16_t avrcp_cid);
```

```
/**
 * @brief Pause. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_pause(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_pause(uint16_t avrcp_cid);

/**
 * @brief Single step - fast forward. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_fast_forward(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_fast_forward(uint16_t
avrcp_cid);


/**
 * @brief Single step rewind. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_rewind(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_rewind(uint16_t
avrcp_cid);

/**
 * @brief Forward. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_forward(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_forward(uint16_t
avrcp_cid);
```

```
/**
 * @brief Backward. Event AVRCP_SUBEVENT_OPERATION_COMPLETE
returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_backward(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_backward(uint16_t
avrcp_cid);

/**
 * @brief Turn up volume. Event AVRCP_SUBEVENT_OPERATION_COMPLETE
returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_volume_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_volume_up(uint16_t
avrcp_cid);
/**
 * @brief Turn down volume. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_volume_down(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_volume_down(uint16_t
avrcp_cid);

/**
 * @brief Mute sound. Event AVRCP_SUBEVENT_OPERATION_COMPLETE
returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_mute(uint16_t avrcp_cid);
uint8_t avrcp_controller_press_and_hold_mute(uint16_t avrcp_cid);

/**
 * @brief Record. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
```

```
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_record(uint16_t avrcp_cid);

/**
 * @brief Eject. Event AVRCP_SUBEVENT_OPERATION_COMPLETE returns
operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_eject(uint16_t avrcp_cid);

// Basic Group Navigation
/**
 * @brief Move to the first song in the next group.
 * @param avrcp_cid
 */
uint8_t avrcp_controller_next_group(uint16_t avrcp_cid);

/**
 * @brief Move to the first song in the previous group.
 * @param avrcp_cid
 */
uint8_t avrcp_controller_previous_group(uint16_t avrcp_cid);

// Category 3
uint8_t avrcp_controller_0(uint16_t avrcp_cid);
uint8_t avrcp_controller_1(uint16_t avrcp_cid);
uint8_t avrcp_controller_2(uint16_t avrcp_cid);
uint8_t avrcp_controller_3(uint16_t avrcp_cid);
uint8_t avrcp_controller_4(uint16_t avrcp_cid);
uint8_t avrcp_controller_5(uint16_t avrcp_cid);
uint8_t avrcp_controller_6(uint16_t avrcp_cid);
uint8_t avrcp_controller_7(uint16_t avrcp_cid);
uint8_t avrcp_controller_8(uint16_t avrcp_cid);
uint8_t avrcp_controller_9(uint16_t avrcp_cid);
uint8_t avrcp_controller_dot(uint16_t avrcp_cid);
uint8_t avrcp_controller_enter(uint16_t avrcp_cid);
```

```
uint8_t avrcp_controller_clear(uint16_t avrcp_cid);
uint8_t avrcp_controller_channel_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_channel_down(uint16_t avrcp_cid);
uint8_t avrcp_controller_previous_channel(uint16_t avrcp_cid);
uint8_t avrcp_controller_sound_select(uint16_t avrcp_cid);
uint8_t avrcp_controller_input_select(uint16_t avrcp_cid);
uint8_t avrcp_controller_display_information(uint16_t avrcp_cid);
uint8_t avrcp_controller_help(uint16_t avrcp_cid);
uint8_t avrcp_controller_power(uint16_t avrcp_cid);
uint8_t avrcp_controller_angle(uint16_t avrcp_cid);
uint8_t avrcp_controller_subpicture(uint16_t avrcp_cid);
uint8_t avrcp_controller_F1(uint16_t avrcp_cid);
uint8_t avrcp_controller_F2(uint16_t avrcp_cid);
uint8_t avrcp_controller_F3(uint16_t avrcp_cid);
uint8_t avrcp_controller_F4(uint16_t avrcp_cid);
uint8_t avrcp_controller_F5(uint16_t avrcp_cid);

// Category 4
uint8_t avrcp_controller_select(uint16_t avrcp_cid);
uint8_t avrcp_controller_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_down(uint16_t avrcp_cid);
uint8_t avrcp_controller_left(uint16_t avrcp_cid);
uint8_t avrcp_controller_right(uint16_t avrcp_cid);
uint8_t avrcp_controller_right_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_right_down(uint16_t avrcp_cid);
uint8_t avrcp_controller_left_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_left_down(uint16_t avrcp_cid);
uint8_t avrcp_controller_root_menu(uint16_t avrcp_cid);
uint8_t avrcp_controller_setup_menu(uint16_t avrcp_cid);
uint8_t avrcp_controller_contents_menu(uint16_t avrcp_cid);
uint8_t avrcp_controller_favorite_menu(uint16_t avrcp_cid);
uint8_t avrcp_controller_exit(uint16_t avrcp_cid);
uint8_t avrcp_controller_page_up(uint16_t avrcp_cid);
uint8_t avrcp_controller_page_down(uint16_t avrcp_cid);
/**
 * @brief Get play status. Returns event of type
AVRCP_SUBEVENT_PLAY_STATUS (length, position, play_status).
 * If TG does not support SongLength And SongPosition on TG, then
TG shall return 0xFFFFFFFF.
 * @param avrcp_cid
```

```
 * @return status
 */
uint8_t avrcp_controller_get_play_status(uint16_t avrcp_cid);

/**
 * @brief Enable notification. Response via
AVRCP_SUBEVENT_NOTIFICATION_STATE.
 * @param avrcp_cid
 * @param event_id
 * @return status
 */
uint8_t avrcp_controller_enable_notification(uint16_t avrcp_cid,
avrcp_notification_event_id_t event_id);

/**
 * @brief Disable notification. Response via
AVRCP_SUBEVENT_NOTIFICATION_STATE.
 * @param avrcp_cid
 * @param event_id
 * @return status
 */
uint8_t avrcp_controller_disable_notification(uint16_t avrcp_cid,
avrcp_notification_event_id_t event_id);

/**
 * @brief Get info on now playing media using subset of attribute
IDs
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_get_element_attributes(uint16_t
avrcp_cid, uint8_t num_attributes, avrcp_media_attribute_id_t *
attributes);

/**
 * @brief Get info on now playing media using all IDs.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_get_now_playing_info(uint16_t avrcp_cid);
```

```
/**
 * @brief Get info on now playing media using specific media
attribute ID.
 * @param media_attribute_id
 * @param avrcp_cid
 * @return status
 */
uint8_t
avrcp_controller_get_now_playing_info_for_media_attribute_id(uint16
avrcp_cid, avrcp_media_attribute_id_t media_attribute_id);

/**
 * @brief Set absolute volume 0-127 (corresponds to 0-100%).
Response via AVRCP_SUBEVENT_SET_ABSOLUTE_VOLUME_RESPONSE
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_set_absolute_volume(uint16_t avrcp_cid,
uint8_t volume);



/**
 * @brief Skip to next playing media. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_skip(uint16_t avrcp_cid);

uint8_t
avrcp_controller_query_player_application_setting_attributes(uint16
avrcp_cid);
uint8_t
avrcp_controller_query_player_application_setting_values(uint16_t
avrcp_cid, avrcp_player_application_setting_attribute_id_t
attribute_id);

uint8_t
avrcp_controller_query_player_application_setting_attribute_text(ui
```

```
avrcp_cid, uint8_t attr_ids_num,
avrcp_player_application_setting_attribute_id_t * attr_ids);
uint8_t
avrcp_controller_query_player_application_setting_value_text(uint16
avrcp_cid, avrcp_player_application_setting_attribute_id_t
attr_id, uint8_t attr_id_values_num, uint8_t * attr_id_values);

uint8_t avrcp_controller_inform_displayable_characterset(uint16_t
avrcp_cid, uint8_t character_set_num, uint16_t * character_set);
uint8_t avrcp_controller_inform_battery_status(uint16_t avrcp_cid,
avrcp_battery_status_t battery_status);

/**
 * @brief Query repeat and shuffle mode. Response via
AVRCP_SUBEVENT_SHUFFLE_AND_REPEAT_MODE.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_query_shuffle_and_repeat_modes(uint16_t
avrcp_cid);

/**
 * @brief Set shuffle mode. Event
AVRCP_SUBEVENT_OPERATION_COMPLETE returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_set_shuffle_mode(uint16_t avrcp_cid,
avrcp_shuffle_mode_t mode);

/**
 * @brief Set repeat mode. Event AVRCP_SUBEVENT_OPERATION_COMPLETE
returns operation id and status.
 * @param avrcp_cid
 * @return status
 */
uint8_t avrcp_controller_set_repeat_mode(uint16_t avrcp_cid,
avrcp_repeat_mode_t mode);

/**
```

```
 * @brief The PlayItem command starts playing an item indicated by
the UID. It is routed to the Addressed Player.
 * @param avrcp_cid
 * @param uid
 * @param uid_counter
 * @param scope
 **/
uint8_t avrcp_controller_play_item_for_scope(uint16_t avrcp_cid,
uint8_t * uid, uint16_t uid_counter, avrcp_browsing_scope_t
scope);

/**
 * @brief Adds an item indicated by the UID to the Now Playing
queue.
 * @param avrcp_cid
 * @param uid
 * @param uid_counter
 * @param scope
 **/
uint8_t
avrcp_controller_add_item_from_scope_to_now_playing_list(uint16_t
avrcp_cid, uint8_t * uid, uint16_t uid_counter,
avrcp_browsing_scope_t scope);

/**
 * @brief Set addressed player.
 * @param avrcp_cid
 * @param addressed_player_id
 */
uint8_t avrcp_controller_set_addressed_player(uint16_t avrcp_cid,
uint16_t addressed_player_id);

/**
 * @brief Send custom command
 * @param avrcp_cid
 * @param command_type
 * @param subunit_type
 * @param subunit_id
 * @param pdu_id
 * @param company_id
```

```
 * @param data
 * @param data_len
 */
uint8_t avrcp_controller_send_custom_command(uint16_t avrcp_cid,
    avrcp_command_type_t command_type,
    avrcp_subunit_type_t subunit_type, avrcp_subunit_id_t
subunit_id,
    avrcp_pdu_id_t pdu_id, uint32_t company_id,
    const uint8_t * data, uint16_t data_len);

/**
 * @brief De-Init AVRCP Controller
 */
void avrcp_controller_deinit(void);
```

AVRCP Cover Art Client API

avrcp_cover_art_client.h

```
/**
 * @brief Set up AVRCP Cover Art client
 */
void avrcp_cover_art_client_init(void);

/**
 * @brief   Connect to AVRCP Cover Art service on a remote device,
emits AVRCP_SUBEVENT_COVER_ART_CONNECTION_ESTABLISHED with status
 * @param   packet_handler
 * @param   remote_addr
 * @param   ertm_buffer
 * @param   ertm_buffer_size
 * @param   ertm_config
 * @param   avrcp_cover_art_cid  outgoing parameter, valid if
status == ERROR_CODE_SUCCESS
 * @return status
 */
uint8_t avrcp_cover_art_client_connect(avrcp_cover_art_client_t
*cover_art_client, btstack_packet_handler_t packet_handler,
                                bd_addr_t remote_addr, uint8_t
*ertm_buffer, uint32_t ertm_buffer_size,
```

```
                                const l2cap_ertm_config_t
*ertm_config, uint16_t *avrcp_cover_art_cid);

/**
 * @brief Request cover art thumbnail for cover with a given image
handle retrieved via
 *        - avrcp_controller_get_now_playing_info or
 *        - avrcp_controller_get_element_attributes(...
AVRCP_MEDIA_ATTR_DEFAULT_COVER_ART ...)
 * @param avrcp_cover_art_cid
 * @param image_handle
 * @return status
 */
uint8_t avrcp_cover_art_client_get_linked_thumbnail(uint16_t
avrcp_cover_art_cid, const char * image_handle);

/**
 * @brief Request cover art image for given image handle retrieved
via
 *        - avrcp_controller_get_now_playing_info or
 *        - avrcp_controller_get_element_attributes(...
AVRCP_MEDIA_ATTR_DEFAULT_COVER_ART ...)
 *        and given image descriptor
 * @param avrcp_cover_art_cid
 * @param image_handle
 * @param image_descriptor
 * @return status
 */
uint8_t avrcp_cover_art_client_get_image(uint16_t
avrcp_cover_art_cid, const char * image_handle, const char *
image_descriptor);

/**
 * @brief Request image properties for given image handle
retrieved via
 *        - avrcp_controller_get_now_playing_info or
 *        - avrcp_controller_get_element_attributes(...
AVRCP_MEDIA_ATTR_DEFAULT_COVER_ART ...)
 * @param avrcp_cover_art_cid
 * @param image_handle
```

```
 * @return status
 */
uint8_t avrcp_cover_art_client_get_image_properties(uint16_t
avrcp_cover_art_cid, const char * image_handle);

/**
 * @brief   Disconnect from AVRCP Cover Art service
 * @param   avrcp_cover_art_cid
 * @return status
 */
uint8_t avrcp_cover_art_client_disconnect(uint16_t
avrcp_cover_art_cid);

/**
 * @brief De-Init AVRCP Cover Art Client
 */
void avrcp_cover_art_client_deinit(void);
```

AVRCP Media Item Iterator API

avrcp_media_item_iterator.h

```
typedef struct avrcp_media_item_context {
     const uint8_t * data;
     uint16_t   offset;
     uint16_t   length;
} avrcp_media_item_context_t;

// Media item data iterator
void avrcp_media_item_iterator_init(avrcp_media_item_context_t
*context, uint16_t avrcp_media_item_len, const uint8_t *
avrcp_media_item_data);
int  avrcp_media_item_iterator_has_more(const
avrcp_media_item_context_t * context);
void avrcp_media_item_iterator_next(avrcp_media_item_context_t *
context);

// Access functions
uint32_t          avrcp_media_item_iterator_get_attr_id(const
avrcp_media_item_context_t * context);
```

```
uint16_t         avrcp_media_item_iterator_get_attr_charset(const
avrcp_media_item_context_t * context);
uint16_t
avrcp_media_item_iterator_get_attr_value_len(const
avrcp_media_item_context_t * context);
const uint8_t *  avrcp_media_item_iterator_get_attr_value(const
avrcp_media_item_context_t * context);
```

AVRCP Target API

avrcp_target.h

```
typedef enum {
    AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_PLAYER_OR_RECORDER =
0,
    AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_MONITOR_OR_AMPLIFIER,
    AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_TUNER,
    AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_MENU,
    AVRCP_TARGET_SUPPORTED_FEATURE_PLAYER_APPLICATION_SETTINGS, //
AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_PLAYER_OR_RECORDER must be
1 for this feature to be set
    AVRCP_TARGET_SUPPORTED_FEATURE_GROUP_NAVIGATION,            //
AVRCP_TARGET_SUPPORTED_FEATURE_CATEGORY_PLAYER_OR_RECORDER must be
1 for this feature to be set
    AVRCP_TARGET_SUPPORTED_FEATURE_BROWSING,

AVRCP_TARGET_SUPPORTED_FEATURE_MULTIPLE_MEDIA_PLAYER_APPLICATIONS,
    AVRCP_TARGET_SUPPORTED_FEATURE_COVER_ART,
} avrcp_target_supported_feature_t;

/**
 * @brief AVRCP Target service record.
 * @param service
 * @param service_record_handle
 * @param supported_features 16-bit bitmap, see
AVRCP_FEATURE_MASK_* in avrcp.h
 * @param service_name or NULL for default value. Provide
"" (empty string) to skip attribute
 * @param service_provider_name or NULL for default value. Provide
"" (empty string) to skip attribute
```

```
 */
void    avrcp_target_create_sdp_record(uint8_t * service, uint32_t
service_record_handle, uint16_t supported_features, const char *
service_name, const char * service_provider_name);

/**
 * @brief Set up AVRCP Target service.
 */
void    avrcp_target_init(void);

/**
 * @brief Register callback for the AVRCP Target client.
 * @param callback
 */
void
avrcp_target_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Select Player that is controlled by Controller
 * @param callback
 * @note Callback should return if selected player is valid
 */
void avrcp_target_register_set_addressed_player_handler(bool
(*callback)(uint16_t player_id));

/**
 * @brief Register a list of Company IDs supported by target.
 * @param avrcp_cid
 * @param num_companies
 * @param companies
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_support_companies(uint16_t avrcp_cid, uint8_t
num_companies, const uint32_t *companies);

/**
 * @brief Register event ID supported by target.
 * @param avrcp_cid
```

```
 * @param event_id
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found,
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE for unsupported
event id, otherwise ERROR_CODE_SUCCESS,
 */
uint8_t avrcp_target_support_event(uint16_t avrcp_cid,
avrcp_notification_event_id_t event_id);

/**
 * @brief Send a play status.
 * @note  The avrcp_target_packet_handler will receive
AVRCP_SUBEVENT_PLAY_STATUS_QUERY event. Use this function to
respond.
 * @param avrcp_cid
 * @param song_length_ms
 * @param song_position_ms
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_play_status(uint16_t avrcp_cid, uint32_t
song_length_ms, uint32_t song_position_ms, avrcp_playback_status_t
status);

/**
 * @brief Set Now Playing Info that is send to Controller if
notifications are enabled
 * @param avrcp_cid
 * @param current_track
 * @param total_tracks
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, ERROR_CODE_COMMAND_DISALLOWED if no track
is provided, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_set_now_playing_info(uint16_t avrcp_cid,
const avrcp_track_t * current_track, uint16_t total_tracks);

/**
 * @brief Set Playing status and send to Controller
 * @param avrcp_cid
```

```
 * @param playback_status
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_set_playback_status(uint16_t avrcp_cid,
avrcp_playback_status_t playback_status);

/**
 * @brief Set Unit Info
 * @param avrcp_cid
 * @param unit_type
 * @param company_id
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_set_unit_info(uint16_t avrcp_cid,
avrcp_subunit_type_t unit_type, uint32_t company_id);

/**
 * @brief Set Subunit Info
 * @param avrcp_cid
 * @param subunit_type
 * @param subunit_info_data
 * @param subunit_info_data_size
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_set_subunit_info(uint16_t avrcp_cid,
avrcp_subunit_type_t subunit_type, const uint8_t *
subunit_info_data, uint16_t subunit_info_data_size);

/**
 * @brief Send Playing Content Changed Notification if enabled
 * @param avrcp_cid
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_playing_content_changed(uint16_t avrcp_cid);

/**
```

```
 * @brief Send Addressed Player Changed Notification if enabled
 * @param avrcp_cid
 * @param player_id
 * @param uid_counter
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_addressed_player_changed(uint16_t avrcp_cid,
uint16_t player_id, uint16_t uid_counter);

/**
 * @brief Set Battery Status Changed and send notification if
enabled
 * @param avrcp_cid
 * @param battery_status
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_battery_status_changed(uint16_t avrcp_cid,
avrcp_battery_status_t battery_status);

/**
 * @brief Overwrite the absolute volume requested by controller
with the actual absolute volume.
 * This function can only be called on
AVRCP_SUBEVENT_NOTIFICATION_VOLUME_CHANGED event, which indicates
a set absolute volume request by controller.
 * If the absolute volume requested by controller does not match
the granularity of volume control the TG provides, you can use
this function to adjust the actual value.
 *
 * @param avrcp_cid
 * @param absolute_volume
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_adjust_absolute_volume(uint16_t avrcp_cid,
uint8_t absolute_volume);
```

```
/**
 * @brief Set Absolute Volume and send notification if enabled
 * @param avrcp_cid
 * @param absolute_volume
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_volume_changed(uint16_t avrcp_cid, uint8_t
absolute_volume);

/**
 * @brief Set Track and send notification if enabled
 * @param avrcp_cid
 * @param trackID
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_track_changed(uint16_t avrcp_cid, uint8_t *
trackID);

/**
 * @brief Send Operation Rejected message
 * @param avrcp_cid
 * @param opid
 * @param operands_length
 * @param operand
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_operation_rejected(uint16_t avrcp_cid,
avrcp_operation_id_t opid, uint8_t operands_length, uint8_t
operand);

/**
 * @brief Send Operation Accepted message
 * @param avrcp_cid
 * @param opid
 * @param operands_length
 * @param operand
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
```

```
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_operation_accepted(uint16_t avrcp_cid,
avrcp_operation_id_t opid, uint8_t operands_length, uint8_t
operand);

/**
 * @brief Send Operation Not Implemented message
 * @param avrcp_cid
 * @param opid
 * @param operands_length
 * @param operand
 * @return status ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection is not found, otherwise ERROR_CODE_SUCCESS
 */
uint8_t avrcp_target_operation_not_implemented(uint16_t avrcp_cid,
avrcp_operation_id_t opid, uint8_t operands_length, uint8_t
operand);

uint8_t avrcp_target_uids_changed(uint16_t avrcp_cid, uint16_t
uid_counter);
uint8_t avrcp_target_send_response_for_play_item_cmd(uint16_t
avrcp_cid, avrcp_status_code_t status);
uint8_t
avrcp_target_send_response_for_add_to_now_playing_cmd(uint16_t
avrcp_cid, avrcp_status_code_t status);

/**
 * @brief De-Init AVRCP Browsing Target
 */
void avrcp_target_deinit(void);
```

BNEP API

bnep.h

```
/**
 * @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.
 * @deprecated use gap_set_security_level instead
 */
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);

/**
 * @brief De-Init BNEP
 */
void bnep_deinit(void);
```

Link Key DB API

btstack_link_key_db.h : Interface to provide link key storage.

```
typedef struct {

    // management
```

```
    /**
     * @brief Open the Link Key DB
     */
    void (*open)(void);

    /**
     * @brief Sets BD Addr of local Bluetooth Controller.
     * @note Only needed if Bluetooth Controller can be swapped,
e.g. USB Dongles on desktop systems
     */
    void (*set_local_bd_addr)(bd_addr_t bd_addr);

    /**
     * @brief Close the Link Key DB
     */
    void (*close)(void);

    // get/set/delete link key

    /**
     * @brief Get Link Key for given address
     * @param addr to lookup
     * @param link_key (out)
     * @param type (out)
     * @return 1 on success
     */
    int  (*get_link_key)(bd_addr_t bd_addr, link_key_t link_key,
link_key_type_t * type);

    /**
     * @brief Update/Store Link key
     * @brief addr
     * @brief link_key
     * @brief type of link key
     */
    void (*put_link_key)(bd_addr_t bd_addr, link_key_t link_key,
link_key_type_t type);

    /**
```

```
     * @brief Delete Link Keys
     * @brief addr
     */
    void (*delete_link_key)(bd_addr_t bd_addr);

    // iterator: it's allowed to delete

    /**
     * @brief Setup iterator
     * @param it
     * @return 1 on success
     */
    int (*iterator_init)(btstack_link_key_iterator_t * it);

    /**
     * @brief Get next Link Key
     * @param it
     * @brief addr
     * @brief link_key
     * @brief type of link key
     * @return 1, if valid link key found
     */
    int  (*iterator_get_next)(btstack_link_key_iterator_t * it,
bd_addr_t bd_addr, link_key_t link_key, link_key_type_t * type);

    /**
     * @brief Frees resources allocated by iterator_init
     * @note Must be called after iteration to free resources
     * @param it
     */
    void (*iterator_done)(btstack_link_key_iterator_t * it);

} btstack_link_key_db_t;
```

In-Memory Link Key Storage API

btstack_link_key_db_memory.h

```
/*
 * @brief
```

```
 */
const btstack_link_key_db_t *
btstack_link_key_db_memory_instance(void);

typedef struct {
    btstack_linked_item_t item;
    bd_addr_t bd_addr;
    link_key_t link_key;
    link_key_type_t link_key_type;
} btstack_link_key_db_memory_entry_t;
```

Static Link Key Storage API

btstack_link_key_db_static.h : Static Link Key Storage implementation to use during development/porting: - Link keys have to be manually added to this file to make them usable + Link keys are preserved on reflash in constrast to the program flash based link key store

```
/*
 * @brief
 */
const btstack_link_key_db_t *
btstack_link_key_db_static_instance(void);
```

Link Key TLV Storage API

btstack_link_key_db_tlv.h : Interface to provide link key storage via BTstack's TLV storage.

@TLV 'BLT' x

```
/**
 * Init Link Key DB using TLV
 * @param btstack_tlv_impl of btstack_tlv interface
 * @Param btstack_tlv_context of btstack_tlv_interface
 */
const btstack_link_key_db_t *
btstack_link_key_db_tlv_get_instance(const btstack_tlv_t *
btstack_tlv_impl, void * btstack_tlv_context);
```

Device ID Server API

device_id_server.h : Create Device ID SDP Records.

```
/**
 * @brief Create SDP record for Device ID service
 * @param service buffer - needs to large enough
 * @param service_record_handle
 * @param vendor_id_source usually
DEVICE_ID_VENDOR_ID_SOURCE_BLUETOOTH or
DEVICE_ID_VENDOR_ID_SOURCE_USB
 * @param vendor_id
 * @param product_it
 * @param version
 */
void device_id_create_sdp_record(uint8_t *service, uint32_t
service_record_handle, uint16_t vendor_id_source, uint16_t
vendor_id, uint16_t product_id, uint16_t version);
```

GATT SDP API

gatt_sdp.h

```
/**
 * @brief Creates SDP record forG ATT 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 gatt_start_handle
 * @param gatt_end_handle
 */
void gatt_create_sdp_record(uint8_t *service, uint32_t
service_record_handle, uint16_t gatt_start_handle, uint16_t
gatt_end_handle);
```

GOEP Client API

goep_client.h : Communicate with remote OBEX server - General Object Exchange

```
typedef struct {
    btstack_linked_item_t item;

    uint16_t         cid;

    goep_client_state_t     state;
    bd_addr_t        bd_addr;
    uint16_t         uuid;
    hci_con_handle_t con_handle;
    uint8_t          incoming;

    btstack_packet_handler_t client_handler;
    btstack_context_callback_registration_t sdp_query_request;

    uint8_t          rfcomm_port;
    uint16_t         l2cap_psm;
    uint16_t         bearer_cid;
    uint16_t         bearer_mtu;

    uint16_t         record_index;

    // cached higher layer information PBAP + MAP
    uint32_t         profile_supported_features;
    uint8_t          map_mas_instance_id;
    uint8_t          map_supported_message_types;
    uint16_t         map_version;

    // needed to select one of multiple MAS Instances
    struct {
        uint32_t supported_features;
        uint8_t  instance_id;
        uint8_t  supported_message_types;
        uint8_t  rfcomm_port;
        uint16_t version;
#ifdef ENABLE_GOEP_L2CAP
```

```
        uint16_t l2cap_psm;
#endif
    } mas_info;

    uint8_t          obex_opcode;
    uint32_t         obex_connection_id;
    int              obex_connection_id_set;

#ifdef ENABLE_GOEP_L2CAP
    l2cap_ertm_config_t ertm_config;
    uint16_t            ertm_buffer_size;
    uint8_t           * ertm_buffer;
#endif
} goep_client_t;


// remote does not expose PBAP features in SDP record
#define PBAP_FEATURES_NOT_PRESENT ((uint32_t) -1)
#define MAP_FEATURES_NOT_PRESENT ((uint32_t) -1)
#define PROFILE_FEATURES_NOT_PRESENT ((uint32_t) -1)

/**
 * Setup GOEP Client
 */
void goep_client_init(void);

/**
 * @brief Connect to a GEOP server with specified UUID on a remote
device.
 * @param goep_client
 * @param l2cap_ertm_config
 * @param l2cap_ertm_buffer_size
 * @param l2cap_ertm_buffer
 * @param handler
 * @param addr
 * @param uuid
 * @param instance_id e.g. used to connect to different MAP Access
Servers, default: 0
 * @param out_cid
 * @return
```

```
 */
uint8_t goep_client_connect(goep_client_t *goep_client,
l2cap_ertm_config_t *l2cap_ertm_config, uint8_t
*l2cap_ertm_buffer,
                    uint16_t l2cap_ertm_buffer_size,
btstack_packet_handler_t handler, bd_addr_t addr, uint16_t uuid,
                    uint8_t instance_id, uint16_t *out_cid);

/**
 * @brief Connect to a GEOP server over L2CAP with specified PSM
on a remote device.
 * @note In contrast to goep_client_connect which searches for a
OBEX service with a given UUID, this
 *       function only supports GOEP v2.0 (L2CAP) to a specified
L2CAP PSM
 * @param goep_client
 * @param l2cap_ertm_config
 * @param l2cap_ertm_buffer_size
 * @param l2cap_ertm_buffer
 * @param handler
 * @param addr
 * @param l2cap_psm
 * @param out_cid
 * @return
 */
uint8_t goep_client_connect_l2cap(goep_client_t *goep_client,
l2cap_ertm_config_t *l2cap_ertm_config, uint8_t
*l2cap_ertm_buffer,
                    uint16_t l2cap_ertm_buffer_size,
btstack_packet_handler_t handler, bd_addr_t addr, uint16_t
l2cap_psm,
                    uint16_t *out_cid);

/**
 * @brief Disconnects GOEP connection with given identifier.
 * @param goep_cid
 * @return status
 */
uint8_t goep_client_disconnect(uint16_t goep_cid);
```

```
/**
 * @brief Request emission of GOEP_SUBEVENT_CAN_SEND_NOW as soon
as possible
 * @note GOEP_SUBEVENT_CAN_SEND_NOW might be emitted during call
to this function
 *       so packet handler should be ready to handle it
 * @param goep_cid
 */
void goep_client_request_can_send_now(uint16_t goep_cid);

/**
 * @brief Get Opcode from last created request, needed for parsing
of OBEX response packet
 * @param goep_cid
 * @return opcode
 */
uint8_t goep_client_get_request_opcode(uint16_t goep_cid);

/**
 * @brief Get PBAP Supported Features found in SDP record during
connect
 */
uint32_t goep_client_get_pbap_supported_features(uint16_t
goep_cid);

/**
 * @brief Get MAP Supported Features found in SDP record during
connect
 */
uint32_t goep_client_get_map_supported_features(uint16_t
goep_cid);

/**
 * @brief Get MAP MAS Instance ID found in SDP record during
connect
 */
uint8_t goep_client_get_map_mas_instance_id(uint16_t goep_cid);

/**
 * @brief Get MAP MAS Supported Message Types found in SDP record
```

```
during connect
 */
uint8_t goep_client_get_map_supported_message_types(uint16_t
goep_cid);

/**
 * @brief Check if GOEP 2.0 or higher features can be used
 * @return true if GOEP Version 2.0 or higher
 */
bool goep_client_version_20_or_higher(uint16_t goep_cid);

/**
 * @brief Set Connection ID used for newly created requests
 * @param goep_cid
 */
void goep_client_set_connection_id(uint16_t goep_cid, uint32_t
connection_id);

/**
 * @brief Start Connect request
 * @param goep_cid
 * @param obex_version_number
 * @param flags
 * @param maximum_obex_packet_length
 */
void goep_client_request_create_connect(uint16_t goep_cid, uint8_t
obex_version_number, uint8_t flags, uint16_t
maximum_obex_packet_length);

/**
 * @brief Start Disconnect request
 * @param goep_cid
 */
void goep_client_request_create_disconnect(uint16_t goep_cid);

/**
 * @brief Create Get request
 * @param goep_cid
 */
void goep_client_request_create_get(uint16_t goep_cid);
```

```
/**
 * @brief Create Abort request
 * @param goep_cid
 */
void goep_client_request_create_abort(uint16_t goep_cid);

/**
 * @brief Start Set Path request
 * @param goep_cid
 */
void goep_client_request_create_set_path(uint16_t goep_cid,
uint8_t flags);

/**
 * @brief Create Put request
 * @param goep_cid
 */
void goep_client_request_create_put(uint16_t goep_cid);

/**
 * @brief Get max size of body data that can be added to current
response with goep_client_body_add_static
 * @param goep_cid
 * @param data
 * @param length
 * @return size in bytes or 0
 */
uint16_t goep_client_request_get_max_body_size(uint16_t goep_cid);

/**
 * @brief Add SRM Enable
 * @param goep_cid
 */
void goep_client_header_add_srm_enable(uint16_t goep_cid);

/**
 * @brief Add SRMP Waiting
 * @param goep_cid
 */
```

```
void goep_client_header_add_srmp_waiting(uint16_t goep_cid);

/**
 * @brief Add header with single byte value (8 bit)
 * @param goep_cid
 * @param header_type
 * @param value
 */
void goep_client_header_add_byte(uint16_t goep_cid, uint8_t
header_type, uint8_t value);

/**
 * @brief Add header with word value (32 bit)
 * @param goep_cid
 * @param header_type
 * @param value
 */
void goep_client_header_add_word(uint16_t goep_cid, uint8_t
header_type, uint32_t value);

/**
 * @brief Add header with variable size
 * @param goep_cid
 * @param header_type
 * @param header_data
 * @param header_data_length
 */
void goep_client_header_add_variable(uint16_t goep_cid, uint8_t
header_type, const uint8_t * header_data, uint16_t
header_data_length);

/**
 * @brief Add name header to current request
 * @param goep_cid
 * @param name
 */
void goep_client_header_add_name(uint16_t goep_cid, const char *
name);

/**
```

```
 * @brief Add name header to current request
 * @param goep_cid
 * @param name
 * @param name_len
 */
void goep_client_header_add_name_prefix(uint16_t goep_cid, const
char * name, uint16_t name_len);

/**
 * @brief Add string encoded as unicode to current request
 * @param goep_cid
 * @param name
 * @param name_len
 */
void goep_client_header_add_unicode_prefix(uint16_t goep_cid,
uint8_t header_id, const char * name, uint16_t name_len);

/**
 * @brief Add target header to current request
 * @param goep_cid
 * @param target
 * @param length of target
 */
void goep_client_header_add_target(uint16_t goep_cid, const
uint8_t * target, uint16_t length);

/**
 * @brief Add type header to current request
 * @param goep_cid
 * @param type
 */
void goep_client_header_add_type(uint16_t goep_cid, const char *
type);

/**
 * @brief Add count header to current request
 * @param goep_cid
 * @param count
 */
void goep_client_header_add_count(uint16_t goep_cid, uint32_t
```

```
count);

/**
 * @brief Add length header to current request
 * @param goep_cid
 * @param length
 */
void goep_client_header_add_length(uint16_t goep_cid, uint32_t
length);

/**
 * @brief Add application parameters header to current request
 * @param goep_cid
 * @param data
 * @param lenght of application parameters
 */
void goep_client_header_add_application_parameters(uint16_t
goep_cid, const uint8_t * data, uint16_t length);

/**
 * @brief Add application parameters header to current request
 * @param goep_cid
 * @param data
 * @param lenght of challenge response
 */
void goep_client_header_add_challenge_response(uint16_t goep_cid,
const uint8_t * data, uint16_t length);

/**
 * @brief Add body
 * @param goep_cid
 * @param data
 * @param lenght
 */
void goep_client_body_add_static(uint16_t goep_cid, const uint8_t
* data, uint32_t length);

/**
 * @brief Query remaining buffer size
 * @param goep_cid
```

```
 * @return size
 */
uint16_t goep_client_body_get_outgoing_buffer_len(uint16_t
goep_cid);

/**
 * @brief Add body
 * @param goep_cid
 * @param data
 * @param length
 * @param ret_length
 */
void goep_client_body_fillup_static(uint16_t goep_cid, const
uint8_t * data, uint32_t length, uint32_t * ret_length);

/**
 * @brief Execute prepared request
 * @param goep_cid
 * @param daa
 */
int goep_client_execute(uint16_t goep_cid);

/**
 * @brief Execute prepared request with final bit
 * @param goep_cid
 * @param final
 */
int goep_client_execute_with_final_bit(uint16_t goep_cid, bool
final);

/**
 * @brief De-Init GOEP Client
 */
void goep_client_deinit(void);
```

HFP Audio Gateway (AG) API

hfp_ag.h

```
typedef struct {
    uint8_t type;
    const char * number;
    hfp_phone_service_t service;
} hfp_phone_number_t;

/**
 * @brief Create HFP Audio Gateway (AG) SDP service record.
 * @param service
 * @param rfcomm_channel_nr
 * @param name or NULL for default value. Provide "" (empty
string) to skip attribute
 * @param ability_to_reject_call
 * @param supported_features 32-bit bitmap, see HFP_AGSF_* values
in hfp.h
 * @param codecs_nr
 * @param codecs
 */
void hfp_ag_create_sdp_record_with_codecs(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,
                                          uint8_t codecs_nr, const
uint8_t * codecs);

/**
 * @brief Set up HFP Audio Gateway (AG) device without additional
supported features.
 * @param rfcomm_channel_nr
 */
void hfp_ag_init(uint8_t rfcomm_channel_nr);

/**
 * @brief Set codecs.
 * @param codecs_nr
 * @param codecs
 */
void hfp_ag_init_codecs(uint8_t codecs_nr, const 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, const
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, const
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 Provide Apple Device information upon AT+XAPL
 * @param device
 * @param features
 */
void hfp_ag_init_apple_identification(const char * device, uint8_t
features);
```

```
/**
 * @brief Register callback for the HFP Audio Gateway (AG)
client.
 * @param callback
 */
void hfp_ag_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Register custom AT command.
 * @param hfp_custom_at_command (with 'AT+' prefix)
 */
void hfp_ag_register_custom_at_command(hfp_custom_at_command_t *
custom_at_command);

/**
 * @brief Enable/Disable 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 of HF
```

```
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *                  - ERROR_CODE_COMMAND_DISALLOWED if connection
already exists or connection in wrong state, or
 *                  - BTSTACK_MEMORY_ALLOC_FAILED
 *
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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
HSP_SUBEVENT_AUDIO_CONNECTION_COMPLETE.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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
```

```
HSP_SUBEVENT_AUDIO_DISCONNECTION_COMPLETE.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid gain
range
 */
uint8_t hfp_ag_set_microphone_gain(hci_con_handle_t acl_handle,
int gain);

/**
 * @brief Set speaker gain.
 *
 * @param acl_handle
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid gain
range
 */
uint8_t hfp_ag_set_speaker_gain(hci_con_handle_t acl_handle, int
gain);

/**
 * @brief Set battery level.
 *
 * @param battery_level Valid range: [0,5]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid battery
level range
 */
```

```
uint8_t hfp_ag_set_battery_level(int battery_level);

/**
 * @brief Clear last dialed number.
 */
void hfp_ag_clear_last_dialed_number(void);

/**
 * @brief Set last dialed number.
 */
void hfp_ag_set_last_dialed_number(const char * number);

/**
 * @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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if call waiting
notification is not enabled
 */
uint8_t hfp_ag_notify_incoming_call_waiting(hci_con_handle_t
acl_handle);

// Voice Recognition

/**
 * @brief Activate voice recognition and emit
HFP_SUBEVENT_VOICE_RECOGNITION_ACTIVATED event with status
ERROR_CODE_SUCCESS
 * if successful. Prerequisite is established SLC.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
```

```
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_VOICE_RECOGNITION_FUNCTION is not supported by HF
and AG, or already activated
 */
uint8_t hfp_ag_activate_voice_recognition(hci_con_handle_t
acl_handle);

/**
 * @brief Deactivate voice recognition and emit
HFP_SUBEVENT_VOICE_RECOGNITION_DEACTIVATED event with status
ERROR_CODE_SUCCESS
 * if successful. Prerequisite is established SLC.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_VOICE_RECOGNITION_FUNCTION is not supported by HF
and AG, or already deactivated
 */
uint8_t hfp_ag_deactivate_voice_recognition(hci_con_handle_t
acl_handle);

/**
 * @brief Notify HF that sound will be played and
HFP_SUBEVENT_ENHANCED_VOICE_RECOGNITION_AG_IS_STARTING_SOUND event
with status ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
 *
 * @param acl_handle
 * @param activate
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_ENHANCED_VOICE_RECOGNITION_STATUS is not supported
by HF and AG
 */
```

```
uint8_t
hfp_ag_enhanced_voice_recognition_report_sending_audio(hci_con_hanc
acl_handle);

/**
 * @brief Notify HF that AG is ready for input and emit
HFP_SUBEVENT_ENHANCED_VOICE_RECOGNITION_AG_READY_TO_ACCEPT_AUDIO_IN
event with status ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
 *
 * @param acl_handle
 * @param activate
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_ENHANCED_VOICE_RECOGNITION_STATUS is not supported
by HF and AG
 */
uint8_t
hfp_ag_enhanced_voice_recognition_report_ready_for_audio(hci_con_ha
acl_handle);

/**
 * @brief Notify that AG is processing input and emit
HFP_SUBEVENT_ENHANCED_VOICE_RECOGNITION_AG_IS_PROCESSING_AUDIO_INPU
event with status ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
 *
 * @param acl_handle
 * @param activate
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_ENHANCED_VOICE_RECOGNITION_STATUS is not supported
by HF and AG
 */
uint8_t
hfp_ag_enhanced_voice_recognition_report_processing_input(hci_con_h
```

```
acl_handle);

/**
 * @brief Send enhanced audio recognition message and
HFP_SUBEVENT_ENHANCED_VOICE_RECOGNITION_AG_MESSAGE_SENT event with
status ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
 *
 * @param acl_handle
 * @param activate
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist,
                -
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if the message
size exceeds the HFP_MAX_VR_TEXT_SIZE, or the command does not fit
into a single packet frame,
 *              - ERROR_CODE_COMMAND_DISALLOWED if HF and AG do
not support features: HFP_(HF/
AG)SF_ENHANCED_VOICE_RECOGNITION_STATUS and HFP_(HF/
AG)SF_VOICE_RECOGNITION_TEXT
 */
uint8_t
hfp_ag_enhanced_voice_recognition_send_message(hci_con_handle_t
acl_handle, hfp_voice_recognition_state_t state,
hfp_voice_recognition_message_t msg);

/**
 * @brief Send a phone number back to the HF.
 *
 * @param acl_handle
 * @param phone_number
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 Outgoing call initiated
 */
void hfp_ag_outgoing_call_initiated(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 dropped event to the AG.
 */
void hfp_ag_call_dropped(void);

/**
 * @brief Set network registration status.
 * @param status 0 - not registered, 1 - registered
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid
registration status
 */
uint8_t hfp_ag_set_registration_status(int registration_status);

/**
 * @brief Set network signal strength.
 *
 * @param signal_strength [0-5]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid signal
strength range
 */
uint8_t hfp_ag_set_signal_strength(int signal_strength);

/**
 * @brief Set roaming status.
```

```
 *
 * @param roaming_status 0 - no roaming, 1 - roaming active
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid roaming
status
 */
uint8_t hfp_ag_set_roaming_status(int roaming_status);

/**
 * @brief Set subscriber number information, e.g. the phone number
 * @param numbers
 * @param numbers_count
 */
void hfp_ag_set_subscriber_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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @param error
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if extended audio
gateway error report is disabled
 */
uint8_t
hfp_ag_report_extended_audio_gateway_error_result_code(hci_con_hanc
acl_handle, hfp_cme_error_t error);

/**
 * @brief Send unsolicited result code (most likely a response to
a vendor-specific command not part of standard HFP).
 * @note  Emits HFP_SUBEVENT_CUSTOM_AT_MESSAGE_SENT when result
code was sent
 *
 * @param unsolicited_result_code to send
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
```

```
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if extended audio
gateway error report is disabled
 */
uint8_t hfp_ag_send_unsolicited_result_code(hci_con_handle_t
acl_handle, const char * unsolicited_result_code);

/**
 * @brief Send result code for AT command received via
HFP_SUBEVENT_CUSTOM_AT_COMMAND
 * @note  Emits HFP_SUBEVENT_COMPLETE when result code was sent
 *
 * @param ok for OK, or ERROR
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if extended audio
gateway error report is disabled
 */
uint8_t hfp_ag_send_command_result_code(hci_con_handle_t
acl_handle, bool ok);

/**
 * @brief De-Init HFP AG
 */
void hfp_ag_deinit(void);

/**
 * @brief Create HFP Audio Gateway (AG) SDP service record.
 * @deprecated Use hfp_ag_create_sdp_record_with_codecs instead
 * @param service
 * @param rfcomm_channel_nr
 * @param name
 * @param ability_to_reject_call
 * @param supported_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);
```

HFP Audio Encoder API

hfp_codec.h : @brief Create SCO packet with H2 Synchronization Header and encoded audio samples

```
typedef struct hfp_codec hfp_codec_t;

#ifdef ENABLE_HFP_WIDE_BAND_SPEECH
/**
 * @brief Initialize HFP Audio Codec for mSBC
 * @param hfp_codec
 * @param msbc_encoder_context for msbc encoder
 * @return status
 */
void hfp_codec_init_msbc_with_codec(hfp_codec_t * hfp_codec, const
btstack_sbc_encoder_t * sbc_encoder, void * sbc_encoder_context);

/**
 * @brief Initialize HFP Audio Codec for mSBC
 * @deprecated Please use hfp_codec_init_msbc_with_codec
 * @param hfp_codec
 * @param msbc_encoder_context for msbc encoder
 * @return status
 */
void hfp_codec_init_msbc(hfp_codec_t * hfp_codec,
btstack_sbc_encoder_state_t * msbc_encoder_context);
#endif

#ifdef ENABLE_HFP_SUPER_WIDE_BAND_SPEECH
/**
 * @brief Initialize HFP Audio Codec for LC3-SWB
 * @param hfp_codec
 * @param lc3_encoder
 * @param lc3_encoder_context for lc3_encoder
 * @return status
 */
void hfp_codec_init_lc3_swb(hfp_codec_t * hfp_codec, const
```

```
btstack_lc3_encoder_t * lc3_encoder, void * lc3_encoder_context);
#endif

/**
 * @brief Get number of audio samples per HFP SCO frame
 * @param hfp_codec
 * @return num audio samples per 7.5ms SCO packet
 */
uint16_t hfp_codec_num_audio_samples_per_frame(const hfp_codec_t *
hfp_codec);

/**
 * @brief Checks if next frame can be encoded
 * @param hfp_codec
 */
bool hfp_codec_can_encode_audio_frame_now(const hfp_codec_t *
hfp_codec);

/**
 * Get number of bytes ready for sending
 * @param hfp_codec
 * @return num bytes ready for current packet
 */
uint16_t hfp_codec_num_bytes_available(const hfp_codec_t *
hfp_codec);

/**
 * Encode audio samples for HFP SCO packet
 * @param hfp_codec
 * @param pcm_samples - complete audio frame of
hfp_msbc_num_audio_samples_per_frame int16 samples
 */
void hfp_codec_encode_audio_frame(hfp_codec_t * hfp_codec, int16_t
* pcm_samples);

/**
 * Read from stream into SCO packet buffer
 * @param hfp_codec
 * @param buffer to store stream
 * @param size num bytes to read from stream
```

```
 */
void hfp_codec_read_from_stream(hfp_codec_t * hfp_codec, uint8_t *
buffer, uint16_t size);

/**
 * @param hfp_codec
 */
void hfp_codec_deinit(hfp_codec_t * hfp_codec);
```

HFP GSM Model API

hfp_gsm_model.h

```
typedef struct {
    bool used_slot;
    hfp_enhanced_call_status_t enhanced_status;
    hfp_enhanced_call_dir_t direction;
    hfp_enhanced_call_mode_t mode;
    hfp_enhanced_call_mpty_t mpty;
    // TODO: sort on drop call, so that index corresponds to table
index
    int index;
    uint8_t clip_type;
    char    clip_number[25];
} hfp_gsm_call_t;

hfp_callheld_status_t hfp_gsm_callheld_status(void);
hfp_call_status_t hfp_gsm_call_status(void);
hfp_callsetup_status_t hfp_gsm_callsetup_status(void);

int hfp_gsm_get_number_of_calls(void);
char * hfp_gsm_last_dialed_number(void);
void hfp_gsm_clear_last_dialed_number(void);
void hfp_gsm_set_last_dialed_number(const char* number);

hfp_gsm_call_t * hfp_gsm_call(int index);

int hfp_gsm_call_possible(void);

uint8_t hfp_gsm_clip_type(void);
```

```
char *  hfp_gsm_clip_number(void);

void hfp_gsm_init(void);
void hfp_gsm_deinit(void);

void hfp_gsm_handler(hfp_ag_call_event_t event, uint8_t index,
uint8_t type, const char * number);
```

HFP Hands-Free (HF) API

hfp_hf.h

```
/**
 * @brief Create HFP Hands-Free (HF) SDP service record.
 * @param service
 * @param rfcomm_channel_nr
 * @param name or NULL for default value. Provide "" (empty
string) to skip attribute
 * @param supported_features 32-bit bitmap, see HFP_HFSF_* values
in hfp.h
 * @param codecs_nr  number of codecs in codecs argument
 * @param codecs
 */
void hfp_hf_create_sdp_record_with_codecs(uint8_t * service,
uint32_t service_record_handle, int rfcomm_channel_nr,
                                          const char * name,
uint16_t supported_features, uint8_t codecs_nr, const uint8_t *
codecs);

/**
 * @brief Set up HFP Hands-Free (HF) device without additional
supported features.
 * @param rfcomm_channel_nr
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *                  - L2CAP_SERVICE_ALREADY_REGISTERED,
 *                  - RFCOMM_SERVICE_ALREADY_REGISTERED or
 *                  - BTSTACK_MEMORY_ALLOC_FAILED if allocation of
any of RFCOMM or L2CAP services failed
 */
uint8_t hfp_hf_init(uint8_t rfcomm_channel_nr);
```

```
/**
 * @brief Set codecs.
 * @param codecs_nr  number of codecs in codecs argument
 * @param codecs
 */
void hfp_hf_init_codecs(uint8_t codecs_nr, const 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, const 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 Set microphone gain used during SLC for Volume
Synchronization.
 *
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_INVALID_HCI_COMMAND_PARAMETERS if
invalid gain range
```

```
 */
uint8_t hfp_hf_set_default_microphone_gain(uint8_t gain);

/**
 * @brief Set speaker gain used during SLC for Volume
Synchronization.
 *
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_INVALID_HCI_COMMAND_PARAMETERS if
invalid gain range
 */
uint8_t hfp_hf_set_default_speaker_gain(uint8_t gain);

/**
 * @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
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_COMMAND_DISALLOWED if connection
already exists, or
 *              - BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
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 acl_handle
 * @param indicators_status_bitmap 32-bit bitmap, 0 - indicator is
disabled, 1 - indicator is enabled
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
hfp_hf_set_status_update_for_individual_ag_indicators(hci_con_handl
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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if connection in
wrong state
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
hfp_hf_enable_report_extended_audio_gateway_error_result_code(hci_c
acl_handle);

/**
 * @brief Disable Extended Audio Gateway Error result codes in the
AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
 uint8_t
hfp_hf_disable_report_extended_audio_gateway_error_result_code(hci_
acl_handle);

/**
 * @brief Establish audio connection.
 * The status of audio connection establishment is reported via
HFP_SUBEVENT_AUDIO_CONNECTION_ESTABLISHED.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
```

```
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if connection in
wrong state
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_release_audio_connection(hci_con_handle_t
acl_handle);

/**
 * @brief Answer incoming call.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if answering
incoming call with wrong callsetup status
 */
uint8_t hfp_hf_answer_incoming_call(hci_con_handle_t acl_handle);

/**
 * @brief Reject incoming call.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
```

```
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_swap_calls(hci_con_handle_t acl_handle);

/**
 * @brief Add a held call to the conversation.
 *
```

```
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_terminate_call(hci_con_handle_t acl_handle);

/**
 * @brief Terminate all held calls.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_terminate_held_calls(hci_con_handle_t acl_handle);

/**
```

```
 * @brief Initiate outgoing voice call by providing the
destination phone number to the AG.
 *
 * @param acl_handle
 * @param number
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @param memory_id
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_activate_call_waiting_notification(hci_con_handle_t
acl_handle);

/**
 * @brief Disable the “Call Waiting notification” function in the
AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
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 acl_handle
 */
uint8_t hfp_hf_activate_calling_line_notification(hci_con_handle_t
acl_handle);

/**
```

```
 * @brief Disable the “Calling Line Identification notification”
function in the AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t
hfp_hf_deactivate_calling_line_notification(hci_con_handle_t
acl_handle);


/**
 * @brief Deactivate echo canceling (EC) and noise reduction (NR)
in the AG and emit
 * HFP_SUBEVENT_ECHO_CANCELING_NOISE_REDUCTION_DEACTIVATE with
status ERROR_CODE_SUCCESS if AG supports EC and AG.
 * If the AG supports its own embedded echo canceling and/or noise
reduction function,
 * it shall have EC and NR activated until this function is
called.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if HFP_(HF/
AG)SF_EC_NR_FUNCTION feature is not supported by AG and HF
 */
uint8_t
hfp_hf_deactivate_echo_canceling_and_noise_reduction(hci_con_handle
acl_handle);

/**
 * @brief Activate voice recognition and emit
HFP_SUBEVENT_VOICE_RECOGNITION_ACTIVATED event with status
ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
Prerequisite is established SLC.
```

```
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_VOICE_RECOGNITION_FUNCTION is not supported by HF
and AG, or already activated
 */
uint8_t hfp_hf_activate_voice_recognition(hci_con_handle_t
acl_handle);

/**
 * @brief Dectivate voice recognition and emit
HFP_SUBEVENT_VOICE_RECOGNITION_DEACTIVATED event with status
ERROR_CODE_SUCCESS
 * if successful, otherwise ERROR_CODE_COMMAND_DISALLOWED.
Prerequisite is established SLC.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_VOICE_RECOGNITION_FUNCTION is not supported by HF
and AG, or already activated
 */
uint8_t hfp_hf_deactivate_voice_recognition(hci_con_handle_t
acl_handle);


/**
 * @brief Indicate that the HF is ready to accept audio.
Prerequisite is established voice recognition session.
 * The HF may call this function during an ongoing AVR (Audio
Voice Recognition) session to terminate audio output from
 * the AG (if there is any) and prepare the AG for new audio
input.
 *
 * @param acl_handle
```

```
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_COMMAND_DISALLOWED if feature
HFP_(HF/AG)SF_ENHANCED_VOICE_RECOGNITION_STATUS is not supported
by HF and AG, or wrong VRA status
 */
uint8_t
hfp_hf_enhanced_voice_recognition_report_ready_for_audio(hci_con_ha
acl_handle);


/**
 * @brief Set microphone gain.
 *
 * @param acl_handle
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid gain
range
 */
uint8_t hfp_hf_set_microphone_gain(hci_con_handle_t acl_handle,
int gain);

/**
 * @brief Set speaker gain.
 *
 * @param acl_handle
 * @param gain Valid range: [0,15]
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if invalid gain
range
 */
uint8_t hfp_hf_set_speaker_gain(hci_con_handle_t acl_handle, int
gain);

/**
 * @brief Instruct the AG to transmit a DTMF code.
```

```
 *
 * @param acl_handle
 * @param dtmf_code
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_query_current_call_status(hci_con_handle_t
acl_handle);

/**
 * @brief Release a call with index in the AG.
 *
 * @param acl_handle
```

```
 * @param index
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @param index
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_rrh_query_status(hci_con_handle_t acl_handle);

/**
 * @brief Put an incoming call on hold in the AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
```

```
exist
 */
uint8_t hfp_hf_rrh_hold_call(hci_con_handle_t acl_handle);

/**
 * @brief Accept held incoming call in the AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_rrh_accept_held_call(hci_con_handle_t acl_handle);

/**
 * @brief Reject held incoming call in the AG.
 *
 * @param acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t 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 acl_handle
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if connection does not
exist
 */
uint8_t hfp_hf_query_subscriber_number(hci_con_handle_t
acl_handle);

/**
 * @brief Set HF indicator.
 *
 * @param acl_handle
```

```
 * @param assigned_number
 * @param value
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if indicator is
either not found or not enabled
 */
uint8_t hfp_hf_set_hf_indicator(int assigned_number, int value);

/**
 * @brief Tests if in-band ringtone is active on AG (requires SLC)
 *
 * @param acl_handler
 * @return 0 if unknown acl_handle or in-band ring-tone disabled,
otherwise 1
 */
int hfp_hf_in_band_ringtone_active(hci_con_handle_t acl_handle);

/**
 * @brief Provide Apple Accessory information after SLC
 * @param vendor_id
 * @param product_id
 * @param version
 * @param features bitmask: bit 0 = reserved, bit 1 = battery
reporting, bit 2 = docked or powered, bit 3 = Siri
 */
void hfp_hf_apple_set_identification(uint16_t vendor_id, uint16_t
product_id, const char * version, uint8_t features);

/**
 * @brief Set Apple Accessory Battery Level
 * @param battery_level range: 0..9
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_INVALID_HCI_COMMAND_PARAMETERS battery level out of
range
 */
uint8_t hfp_hf_apple_set_battery_level(uint8_t battery_level);

/**
```

```
 * @brief Set Apple Accessory Docked State
 * @param docked 0 = undocked, 1 = docked
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
ERROR_CODE_INVALID_HCI_COMMAND_PARAMETERS docked state invalid
 */
uint8_t hfp_hf_apple_set_docked_state(uint8_t docked);

/**
 * @brief Send AT command (most likely a vendor-specific command
not part of standard HFP).
 * @note  Result (OK/ERROR) is reported via
HFP_SUBEVENT_CUSTOM_AT_MESSAGE_SENT
 *        To receive potential unsolicited result code, add
ENABLE_HFP_AT_MESSAGES to get all message via
HFP_SUBEVENT_AT_MESSAGE_RECEIVED
 *
 * @param acl_handle
 * @param at_command to send
 * @return status ERROR_CODE_SUCCESS if successful, otherwise:
 *              - ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER if
connection does not exist, or
 *              - ERROR_CODE_COMMAND_DISALLOWED if extended audio
gateway error report is disabled
 */
uint8_t hfp_hf_send_at_command(hci_con_handle_t acl_handle, const
char * at_command);

/**
 * @brief Register custom AT command.
 * @param hfp_custom_at_command (with '+' prefix)
 */
void hfp_hf_register_custom_at_command(hfp_custom_at_command_t *
custom_at_command);

/**
 * @brief De-Init HFP HF
 */
void hfp_hf_deinit(void);

/**
```

```
 * @brief Create HFP Hands-Free (HF) SDP service record.
 * @deprecated Use hfp_hf_create_sdp_record_with_codecs instead
 * @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);

#ifdef ENABLE_TESTING_SUPPORT
bool test_hfp_hf_vra_state_machine(hfp_connection_t *
hfp_connection, hfp_hf_vra_event_type_t event);
#endif
```

HFP mSBC Encoder API

hfp_msbc.h

```
/**
 *
 */
void hfp_msbc_init(void);

/**
 *
 */
int  hfp_msbc_num_audio_samples_per_frame(void);

/**
 *
 */
int  hfp_msbc_can_encode_audio_frame_now(void);

/**
 * @param pcm_samples - complete audio frame of
hfp_msbc_num_audio_samples_per_frame int16 samples
```

```
 */
void hfp_msbc_encode_audio_frame(int16_t * pcm_samples);

/**
 *
 */
int  hfp_msbc_num_bytes_in_stream(void);

/**
 *
 * @param buffer to store stream
 * @param size num bytes to read from stream
 */
void hfp_msbc_read_from_stream(uint8_t * buffer, int size);

/**
 * @brief De-Init HFP mSBC Codec
 */
void hfp_msbc_deinit(void);
```

HID Device API

hid_device.h

```
typedef struct {
    uint16_t        hid_device_subclass;
    uint8_t         hid_country_code;
    uint8_t         hid_virtual_cable;
    uint8_t         hid_remote_wake;
    uint8_t         hid_reconnect_initiate;
    bool            hid_normally_connectable;
    bool            hid_boot_device;
    uint16_t        hid_ssr_host_max_latency;
    uint16_t        hid_ssr_host_min_timeout;
    uint16_t        hid_supervision_timeout;
    const uint8_t * hid_descriptor;
    uint16_t        hid_descriptor_size;
    // ServiceName, or NULL
    const char *    device_name;
} hid_sdp_record_t;
```

```
/**
 * @brief Create HID Device SDP service record.
 * @param service Empty buffer in which a new service record will
be stored.
 * @param have_remote_audio_control
 * @param service
 * @param service_record_handle
 * @param params
 */
void hid_create_sdp_record(uint8_t * service, uint32_t
service_record_handle, const hid_sdp_record_t * params);

/**
 * @brief Set up HID Device
 * @param boot_protocol_mode_supported
 * @param hid_descriptor_len
 * @param hid_descriptor
 */
void hid_device_init(bool boot_protocol_mode_supported, uint16_t
hid_descriptor_len, const uint8_t * hid_descriptor);

/**
 * @brief Register callback for the HID Device client.
 * @param callback
 */
void hid_device_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Register get report callback for the HID Device client.
 * @param callback
 */
void hid_device_register_report_request_callback(int (*callback)
(uint16_t hid_cid, hid_report_type_t report_type, uint16_t
report_id, int * out_report_size, uint8_t * out_report));

/**
 * @brief Register set report callback for the HID Device client.
 * @param callback
```

```
 */
void hid_device_register_set_report_callback(void (*callback)
(uint16_t hid_cid, hid_report_type_t report_type, int report_size,
uint8_t * report));

/**
 * @brief Register callback to receive report data for the HID
Device client.
 * @param callback
 */
void hid_device_register_report_data_callback(void (*callback)
(uint16_t cid, hid_report_type_t report_type, uint16_t report_id,
int report_size, uint8_t * report));

/*
 * @brief Create HID connection to HID Host
 * @param addr
 * @param hid_cid to use for other commands
 * @result status
 */
uint8_t hid_device_connect(bd_addr_t addr, uint16_t * hid_cid);

/*
 * @brief Disconnect from HID Host
 * @param hid_cid
 */
void hid_device_disconnect(uint16_t hid_cid);

/**
 * @brief Request can send now event to send HID Report
 * Generates an HID_SUBEVENT_CAN_SEND_NOW subevent
 * @param hid_cid
 */
void hid_device_request_can_send_now_event(uint16_t hid_cid);

/**
 * @brief Send HID message on interrupt channel
 * @param hid_cid
 */
void hid_device_send_interrupt_message(uint16_t hid_cid, const
```

```
uint8_t * message, uint16_t message_len);

/**
 * @brief Send HID message on control channel
 * @param hid_cid
 */
void hid_device_send_control_message(uint16_t hid_cid, const
uint8_t * message, uint16_t message_len);

/**
 * @brief Retutn 1 if boot protocol mode active
 * @param hid_cid
 */
int hid_device_in_boot_protocol_mode(uint16_t hid_cid);

/**
 * @brief De-Init HID Device
 */
void hid_device_deinit(void);
```

HID Host API

hid_host.h

```
/**
 * @brief Set up HID Host and provide storage for HID Descriptors
 * If storage for HID Descriptors is provided, the HID Host will
fetch and store the HID Descriptors
 * for each connection and inform the application via
HID_SUBEVENT_DESCRIPTOR_AVAILABLE event.
 * The app can then access the HID descriptor for a particular
connection with
 * hid_descriptor_storage_get_descriptor_data and
hid_descriptor_storage_get_descriptor_len.
 * @note if hid_descriptor_storage is NULL, the HID Host will not
store the HID Descriptor
 * @param hid_descriptor_storage
 * @param hid_descriptor_storage_len
 */
void hid_host_init(uint8_t * hid_descriptor_storage, uint16_t
```

```
hid_descriptor_storage_len);

/**
 * @brief Register callback for the HID Host.
 * @param callback
 */
void hid_host_register_packet_handler(btstack_packet_handler_t
callback);

/**
 * @brief Create HID connection to HID Device and emit
HID_SUBEVENT_CONNECTION_OPENED event with status code,
 * followed by HID_SUBEVENT_DESCRIPTOR_AVAILABLE that informs if
the HID Descriptor was found. In the case of incoming
 * connection, i.e. HID Device initiating the connection, the
HID_SUBEVENT_DESCRIPTOR_AVAILABLE is delayed, and the reports
 * may already come via HID_SUBEVENT_REPORT event. It is up to the
application code if
 * these reports should be buffered or ignored until the
descriptor is available.
 * @note  HID_PROTOCOL_MODE_REPORT_WITH_FALLBACK_TO_BOOT will try
ti set up REPORT mode, but fallback to BOOT mode if necessary.
 * @note  HID_SUBEVENT_DESCRIPTOR_AVAILABLE possible status values
are:
 * - ERROR_CODE_SUCCESS if descriptor available,
 * - ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE if not,
and
 * - ERROR_CODE_MEMORY_CAPACITY_EXCEEDED if descriptor is larger
then the available space
 * @param remote_addr
 * @param protocol_mode see hid_protocol_mode_t in btstack_hid.h
 * @param hid_cid to use for other commands
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_COMMAND_DISALLOWED, BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t hid_host_connect(bd_addr_t remote_addr,
hid_protocol_mode_t protocol_mode, uint16_t * hid_cid);


/**
```

```
 * @brief Accept incoming HID connection, this should be called
upon receiving HID_SUBEVENT_INCOMING_CONNECTION event.
 * @param hid_cid
 * @param protocol_mode see hid_protocol_mode_t in btstack_hid.h
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_accept_connection(uint16_t hid_cid,
hid_protocol_mode_t protocol_mode);

/**
 * @brief Decline incoming HID connection, this should be called
upon receiving HID_SUBEVENT_INCOMING_CONNECTION event.
 * @param hid_cid
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_decline_connection(uint16_t hid_cid);

/**
 * @brief Disconnect from HID Device and emit
HID_SUBEVENT_CONNECTION_CLOSED event.
 * @param hid_cid
 */
void hid_host_disconnect(uint16_t hid_cid);

// Control messages:

/**
 * @brief Send SUSPEND control signal to connected HID Device. A
Bluetooth HID Device which receives a SUSPEND control signal
 * may optionally disconnect from the Bluetooth HID Host.
 * @param hid_cid
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_suspend(uint16_t hid_cid);
```

```
/**
 * @brief Order connected HID Device to exit suspend mode.
 * The Bluetooth HID Device shall send a report to the Bluetooth
HID Host.
 * @param hid_cid
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_exit_suspend(uint16_t hid_cid);

/**
 * @brief Unplug connected HID Device.
 * @note This is the only command that can be also received from
HID Device. It will be indicated by receiving
 * HID_SUBEVENT_VIRTUAL_CABLE_UNPLUG event, as well as
disconnecting HID Host from device.
 * @param hid_cid
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_virtual_cable_unplug(uint16_t hid_cid);

/**
 * @brief Set Protocol Mode on the Bluetooth HID Device and emit
HID_SUBEVENT_SET_PROTOCOL_RESPONSE event with handshake_status,
see hid_handshake_param_type_t
 * @param hid_cid
 * @param protocol_mode see hid_protocol_mode_t in btstack_hid.h
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_set_protocol_mode(uint16_t hid_cid,
hid_protocol_mode_t protocol_mode);

/**
 * @brief Retrieve the Protocol Mode of the Bluetooth HID Device
```

```
and emit HID_SUBEVENT_GET_PROTOCOL_RESPONSE with handshake_status,
see hid_handshake_param_type_t
 * @param hid_cid
 * @param protocol_mode see hid_protocol_mode_t in btstack_hid.h
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_get_protocol(uint16_t hid_cid);

/**
 * @brief Send report to a Bluetooth HID Device and emit
HID_SUBEVENT_SET_REPORT_RESPONSE with handshake_status, see
hid_handshake_param_type_t. The Bluetooth HID Host shall send
complete reports.
 * @param hid_cid
 * @param report_type see hid_report_type_t in btstack_hid.h
 * @param report_id or HID_REPORT_ID_UNDEFINED
 * @param report
 * @param report_len
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_set_report(uint16_t hid_cid,
hid_report_type_t report_type, uint16_t report_id, const uint8_t *
report, uint8_t report_len);

/**
 * @brief Request a HID report from the Bluetooth HID Device and
emit HID_SUBEVENT_GET_REPORT_RESPONSE event with with
handshake_status, see hid_handshake_param_type_t.
 * Polling Bluetooth HID Devices using the GET_REPORT transfer is
costly in terms of time and overhead,
 * and should be avoided whenever possible. The GET_REPORT
transfer is typically only used by applications
 * to determine the initial state of a Bluetooth HID Device. If
the state of a report changes frequently,
 * then the report should be reported over the more efficient
Interrupt channel, see hid_host_send_report.
```

```
 * @param hid_cid
 * @param report_type see hid_report_type_t in btstack_hid.h
 * @param report_id or HID_REPORT_ID_UNDEFINED
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_get_report(uint16_t hid_cid,
hid_report_type_t report_type, uint16_t report_id);

/**
 * @brief Send HID output report on interrupt channel.
 * @param hid_cid
 * @param report_id or HID_REPORT_ID_UNDEFINED
 * @param report
 * @param report_len
 * @result status ERROR_CODE_SUCCESS on success, otherwise
ERROR_CODE_UNKNOWN_CONNECTION_IDENTIFIER,
ERROR_CODE_COMMAND_DISALLOWED
 */
uint8_t hid_host_send_report(uint16_t hid_cid, uint16_t report_id,
const uint8_t * report, uint8_t report_len);

/**
 * @brief Get descriptor data
 * @param hid_cid
 * @result data
 */
const uint8_t *
hid_descriptor_storage_get_descriptor_data(uint16_t hid_cid);

/**
 * @brief Get descriptor length
 * @param hid_cid
 * @result length
 */
uint16_t hid_descriptor_storage_get_descriptor_len(uint16_t
hid_cid);

/**
```

```
 * @brief De-Init HID Device
 */
void hid_host_deinit(void);
```

HSP Audio Gateway API

hsp_ag.h

```
/**
 * @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);

/**
 * @brief Set packet types used for outgoing SCO connection
requests
 * @param common single packet_types: SCO_PACKET_TYPES_*
 */
void hsp_ag_set_sco_packet_types(uint16_t packet_types);

/**
 * @brief De-Init HSP AG
 */
void hsp_ag_deinit(void);
```

HSP Headset API

hsp_hs.h

```
/**
 * @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);

/**
 * @brief Set packet types used for incoming SCO connection
requests
 * @param common single packet_types: SCO_PACKET_TYPES_*
 */
void hsp_hs_set_sco_packet_types(uint16_t packet_types);

/**
 * @brief De-Init HSP AG
```

```
 */
void hsp_hs_deinit(void);
```

Personal Area Network (PAN) API

pan.h

```
/**
 * @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_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,
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);
```

PBAP Client API

pbap.h

```
// PBAP Supported Features

#define PBAP_SUPPORTED_FEATURES_DOWNLOAD
(1<<0)
#define PBAP_SUPPORTED_FEATURES_BROWSING
(1<<1)
#define PBAP_SUPPORTED_FEATURES_DATABASE_IDENTIFIER
(1<<2)
#define PBAP_SUPPORTED_FEATURES_FOLDER_VERSION_COUNTERS
(1<<3)
#define PBAP_SUPPORTED_FEATURES_VCARD_SELECTING
(1<<4)
#define PBAP_SUPPORTED_FEATURES_ENHANCED_MISSED_CALLS
(1<<5)
#define PBAP_SUPPORTED_FEATURES_X_BT_UCI_VCARD_PROPERTY
(1<<6)
#define PBAP_SUPPORTED_FEATURES_X_BT_UID_VCARD_PROPERTY
(1<<7)
#define PBAP_SUPPORTED_FEATURES_CONTACT_REFERENCING
(1<<8)
#define PBAP_SUPPORTED_FEATURES_DEFAULT_CONTACT_IMAGE_FORMAT
(1<<9)

// PBAP Property Mask - also used for vCardSelector
#define PBAP_PROPERTY_MASK_VERSION              (1<< 0) // vCard
Version
#define PBAP_PROPERTY_MASK_FN                   (1<< 1) //
Formatted Name
#define PBAP_PROPERTY_MASK_N                    (1<< 2) //
Structured Presentation of Name
#define PBAP_PROPERTY_MASK_PHOTO                (1<< 3) //
Associated Image or Photo
#define PBAP_PROPERTY_MASK_BDAY                 (1<< 4) //
Birthday
#define PBAP_PROPERTY_MASK_ADR                  (1<< 5) //
Delivery Address
```

```
#define PBAP_PROPERTY_MASK_LABEL                (1<< 6) //
Delivery
#define PBAP_PROPERTY_MASK_TEL                  (1<< 7) //
Telephone Number
#define PBAP_PROPERTY_MASK_EMAIL                (1<< 8) //
Electronic Mail Address
#define PBAP_PROPERTY_MASK_MAILER               (1<< 9) //
Electronic Mail
#define PBAP_PROPERTY_MASK_TZ                   (1<<10) // Time
Zone
#define PBAP_PROPERTY_MASK_GEO                  (1<<11) //
Geographic Position
#define PBAP_PROPERTY_MASK_TITLE                (1<<12) // Job
#define PBAP_PROPERTY_MASK_ROLE                 (1<<13) // Role
within the Organization
#define PBAP_PROPERTY_MASK_LOGO                 (1<<14) //
Organization Logo
#define PBAP_PROPERTY_MASK_AGENT                (1<<15) // vCard
of Person Representing
#define PBAP_PROPERTY_MASK_ORG                  (1<<16) // Name of
Organization
#define PBAP_PROPERTY_MASK_NOTE                 (1<<17) //
Comments
#define PBAP_PROPERTY_MASK_REV                  (1<<18) //
Revision
#define PBAP_PROPERTY_MASK_SOUND                (1<<19) //
Pronunciation of Name
#define PBAP_PROPERTY_MASK_URL                  (1<<20) // Uniform
Resource Locator
#define PBAP_PROPERTY_MASK_UID                  (1<<21) // Unique
ID
#define PBAP_PROPERTY_MASK_KEY                  (1<<22) // Public
Encryption Key
#define PBAP_PROPERTY_MASK_NICKNAME             (1<<23) //
Nickname
#define PBAP_PROPERTY_MASK_CATEGORIES           (1<<24) //
Categories
#define PBAP_PROPERTY_MASK_PROID                (1<<25) // Product
ID
#define PBAP_PROPERTY_MASK_CLASS                (1<<26) // Class
```

```
information
#define PBAP_PROPERTY_MASK_SORT_STRING          (1<<27) // String
used for sorting operations
#define PBAP_PROPERTY_MASK_X_IRMC_CALL_DATETIME (1<<28) // Time
stamp
#define PBAP_PROPERTY_MASK_X_BT_SPEEDDIALKEY    (1<<29) // Speed-
dial shortcut
#define PBAP_PROPERTY_MASK_X_BT_UCI             (1<<30) // Uniform
Caller Identifier
#define PBAP_PROPERTY_MASK_X_BT_UID             (1<<31) //
Bluetooth Contact Unique Identifier

// PBAP vCardSelectorOperator
#define PBAP_VCARD_SELECTOR_OPERATOR_OR          0
#define PBAP_VCARD_SELECTOR_OPERATOR_AND         1

// PBAP Format
typedef enum {
    PBAP_FORMAT_VCARD_21 = 0,
    PBAP_FORMAT_VCRAD_30
} pbap_format_vcard_t;

// PBAP Object Types
typedef enum {
    PBAP_OBJECT_TYPE_INVALID = 0,
    PBAP_OBJECT_TYPE_PHONEBOOOK,
    PBAP_OBJECT_TYPE_VCARD_LISTING,
    PBAP_OBJECT_TYPE_VCARD,
} pbap_object_type_t;

// PBAP Phonebooks
typedef enum {
    PBAP_PHONEBOOK_INVALID = 0,
    PBAP_PHONEBOOK_TELECOM_CCH,
    PBAP_PHONEBOOK_TELECOM_FAV,
    PBAP_PHONEBOOK_TELECOM_ICH,
    PBAP_PHONEBOOK_TELECOM_MCH,
    PBAP_PHONEBOOK_TELECOM_OCH,
    PBAP_PHONEBOOK_TELECOM_PB,
    PBAP_PHONEBOOK_TELECOM_SPD,
```

```
    PBAP_PHONEBOOK_SIM_TELECOM_CCH,
    PBAP_PHONEBOOK_SIM_TELECOM_ICH,
    PBAP_PHONEBOOK_SIM_TELECOM_MCH,
    PBAP_PHONEBOOK_SIM_TELECOM_OCH,
    PBAP_PHONEBOOK_SIM_TELECOM_PB
} pbap_phonebook_t;

// lengths
#define PBAP_DATABASE_IDENTIFIER_LEN 16
#define PBAP_FOLDER_VERSION_LEN 16
```

PBAP Client API

pbap_client.h

```
/**
 * Setup PhoneBook Access Client
 */
void pbap_client_init(void);

/**
 * @brief Create PBAP connection to a Phone Book Server (PSE)
server on a remote device.
 * If the server requires authentication, a
PBAP_SUBEVENT_AUTHENTICATION_REQUEST is emitted, which
 * can be answered with pbap_authentication_password(..).
 * The status of PBAP connection establishment is reported via
PBAP_SUBEVENT_CONNECTION_OPENED event,
 * i.e. on success status field is set to ERROR_CODE_SUCCESS.
 *
 * This function allows for multiple parallel connections.
 *
 * @param client storage for connection state. Must stay valid
until connection closes
 * @param l2cap_ertm_config
 * @param l2cap_ertm_buffer_size
 * @param l2cap_ertm_buffer
 * @param handler
 * @param addr
 * @param out_cid to use for further commands
```

```
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_MEMORY_ALLOC_FAILED if PBAP or GOEP connection already
exists.
 */

uint8_t pbap_client_connect(pbap_client_t * client,
l2cap_ertm_config_t *l2cap_ertm_config, uint8_t
*l2cap_ertm_buffer,
                            uint16_t l2cap_ertm_buffer_size,
btstack_packet_handler_t handler, bd_addr_t addr, uint16_t *
out_cid);

/**
 * @brief Create PBAP connection to a Phone Book Server (PSE)
server on a remote device.
 * If the server requires authentication, a
PBAP_SUBEVENT_AUTHENTICATION_REQUEST is emitted, which
 * can be answered with pbap_authentication_password(..).
 * The status of PBAP connection establishment is reported via
PBAP_SUBEVENT_CONNECTION_OPENED event,
 * i.e. on success status field is set to ERROR_CODE_SUCCESS.
 *
 * This function uses a single pbap_client_t and l2cap ertm buffer
instance and can only be used for a single connection.
 * Fur multiple parallel connections, use pbap_client_connect.
 *
 * @param handler
 * @param addr
 * @param out_cid to use for further commands
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_MEMORY_ALLOC_FAILED if PBAP or GOEP connection already
exists.
 */
uint8_t pbap_connect(btstack_packet_handler_t handler, bd_addr_t
addr, uint16_t * out_cid);

/**
 * Create SDP Record for Phonebook Access Client
 * @param service
 * @param service_record_handle
```

```
 * @param service_name
 */
void pbap_client_create_sdp_record(uint8_t *service, uint32_t
service_record_handle, const char *service_name);

/**
 * @brief Provide password for OBEX Authentication after receiving
PBAP_SUBEVENT_AUTHENTICATION_REQUEST.
 * The status of PBAP connection establishment is reported via
PBAP_SUBEVENT_CONNECTION_OPENED event,
 * i.e. on success status field is set to ERROR_CODE_SUCCESS.
 *
 * @param pbap_cid
 * @param password (null terminated string) - not copied, needs to
stay valid until connection completed
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_authentication_password(uint16_t pbap_cid, const char
* password);

/**
 * @brief Disconnects PBAP connection with given identifier.
 * Event PBAP_SUBEVENT_CONNECTION_CLOSED indicates that PBAP
connection is closed.
 *
 * @param pbap_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_disconnect(uint16_t pbap_cid);

/**
 * @brief Set current folder on PSE. The status is reported via
PBAP_SUBEVENT_OPERATION_COMPLETED event.
 *
 * @param pbap_cid
 * @param path - note: path is not copied
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
```

```
 */
uint8_t pbap_set_phonebook(uint16_t pbap_cid, const char * path);

/**
 * @brief Set vCard Selector for get/pull phonebook. No event is
emitted.
 *
 * @param pbap_cid
 * @param vcard_selector - combination of PBAP_PROPERTY_MASK_*
properties
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_set_vcard_selector(uint16_t pbap_cid, uint32_t
vcard_selector);

/**
 * @brief Set vCard Selector for get/pull phonebook. No event is
emitted.
 *
 * @param pbap_cid
 * @param vcard_selector_operator -
PBAP_VCARD_SELECTOR_OPERATOR_OR (default) or
PBAP_VCARD_SELECTOR_OPERATOR_AND
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_set_vcard_selector_operator(uint16_t pbap_cid, int
vcard_selector_operator);

/**
 * @brief Set Property Selector. No event is emitted.
 * @param pbap_cid
 * @param property_selector - combination of PBAP_PROPERTY_MASK_*
properties
 * @return
 */
uint8_t pbap_set_property_selector(uint16_t pbap_cid, uint32_t
property_selector);
```

```
/**
 * @brief Set number of items returned by pull phonebook.
 * @param pbap_cid
 * @param max_list_count
 * @return
 */
uint8_t pbap_set_max_list_count(uint16_t pbap_cid, uint16_t
max_list_count);

/**
 * @bbrief Set start offset for pull phonebook
 * @param pbap_cid
 * @param list_start_offset
 * @return
 */
uint8_t pbap_set_list_start_offset(uint16_t pbap_cid, uint16_t
list_start_offset);

/**
 * @bbrief Set order for pbap_pull_vcard_listing
 * @param pbap_cid
 * @param order
 * @return
 */
uint8_t pbap_set_order(uint16_t pbap_cid, uint8_t order);

/**
 * @bbrief Set search property for pbap_pull_vcard_listing
 * @param pbap_cid
 * @param search_property
 * @return
 */
uint8_t pbap_set_search_property(uint16_t pbap_cid, uint8_t
search_property);

/**
 * @bbrief Set search property for pbap_pull_vcard_listing
 * @param pbap_cid
 * @param search_value
 * @return
```

```
 */
uint8_t pbap_set_search_value(uint16_t pbap_cid, const char *
search_value);

/**
 * @brief Get size of phone book from PSE. The result is reported
via PBAP_SUBEVENT_PHONEBOOK_SIZE event.
 * @note This function uses the PullPhoneBook command
 *
 * @param pbap_cid
 * @param path - note: path is not copied, common path 'telecom/
pb.vcf'
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_get_phonebook_size(uint16_t pbap_cid, const char *
path);

/**
 * @brief Pull phone book from PSE. The result is reported via
registered packet handler (see pbap_connect function),
 * with packet type set to PBAP_DATA_PACKET. Event
PBAP_SUBEVENT_OPERATION_COMPLETED marks the end of the phone
book.
 *
 * @param pbap_cid
 * @param path - note: path is not copied, common path 'telecom/
pb.vcf'
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_pull_phonebook(uint16_t pbap_cid, const char * path);

/**
 * @brief Get size of phone book from PSE. The result is reported
via PBAP_SUBEVENT_PHONEBOOK_SIZE event.
 *
 * @note This function uses the PullvCardListing command
 * @param pbap_cid
 * @param path - note: path is not copied, common path 'telecom/
```

```
pb.vcf'
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_get_vcard_listing_size(uint16_t pbap_cid, const char
* path);

/**
 * @brief Pull vCard listing. vCard data is emitted via
PBAP_SUBEVENT_CARD_RESULT event.
 * Event PBAP_SUBEVENT_OPERATION_COMPLETED marks the end of vCard
listing.
 *
 * @param pbap_cid
 * @param path
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_pull_vcard_listing(uint16_t pbap_cid, const char *
path);

/**
 * @brief Pull vCard entry. The result is reported via callback
(see pbap_connect function),
 * with packet type set to PBAP_DATA_PACKET.
 * Event PBAP_SUBEVENT_OPERATION_COMPLETED marks the end of the
vCard entry.
 *
 * @param pbap_cid
 * @param path
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_pull_vcard_entry(uint16_t pbap_cid, const char *
path);

/**
 * @brief Lookup contact(s) by phone number. vCard data is emitted
via PBAP_SUBEVENT_CARD_RESULT event.
 * Event PBAP_SUBEVENT_OPERATION_COMPLETED marks the end of the
```

```
lookup.
 *
 * @param pbap_cid
 * @param phone_number
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_lookup_by_number(uint16_t pbap_cid, const char *
phone_number);

/**
 * @brief Abort current operation. No event is emitted.
 *
 * @param pbap_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_abort(uint16_t pbap_cid);


/**
 * @brief Set flow control mode - default is off. No event is
emitted.
 * @note When enabled, pbap_next_packet needs to be called after a
packet was processed to receive the next one
 *
 * @param pbap_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_set_flow_control_mode(uint16_t pbap_cid, int enable);

/**
 * @brief Trigger next packet from PSE when Flow Control Mode is
enabled.
 * @param pbap_cid
 * @return status ERROR_CODE_SUCCESS on success, otherwise
BTSTACK_BUSY if in a wrong state.
 */
uint8_t pbap_next_packet(uint16_t pbap_cid);
```

```
/**
 * @brief De-Init PBAP Client
 */
void pbap_client_deinit(void);
```

RFCOMM API

rfcomm.h

```
/**
 * @brief Set up RFCOMM.
 */
void rfcomm_init(void);

/**
 * @brief Set security level required for incoming connections,
need to be called before registering services.
 * @deprecated use gap_set_security_level instead
 */
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.
 * @return status
 */
uint8_t 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
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
L2CAP_SERVICE_ALREADY_REGISTERED or BTSTACK_MEMORY_ALLOC_FAILED
 */
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
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
L2CAP_SERVICE_ALREADY_REGISTERED or BTSTACK_MEMORY_ALLOC_FAILED
 */
uint8_t
rfcomm_register_service_with_initial_credits(btstack_packet_handler
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.
 * @return status
 */
uint8_t rfcomm_accept_connection(uint16_t rfcomm_cid);

/**
 * @brief Deny incoming RFCOMM connection.
 * @return status
 */
uint8_t rfcomm_decline_connection(uint16_t rfcomm_cid);

/**
 * @brief Grant more incoming credits to the remote side for the
given RFCOMM channel identifier.
 * @return status
 */
uint8_t rfcomm_grant_credits(uint16_t rfcomm_cid, uint8_t
credits);

/**
 * @brief Checks if RFCOMM can send packet.
 * @param rfcomm_cid
 * @result true if can send now
 */
```

```
bool 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
 * @prarm status
 */
uint8_t 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
 * @return status
 */
uint8_t 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
 * @return status
 */
uint8_t 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
 * @return status
 */
uint8_t 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
 * @return status
 */
uint8_t 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, uint8_t flow_control);

/**
 * @brief Query remote port
 * @param rfcomm_cid
 * @return status
 */
uint8_t rfcomm_query_port_configuration(uint16_t rfcomm_cid);

/**
 * @brief Query max frame size
 * @param rfcomm_cid
 * @return max frame size
 */
uint16_t rfcomm_get_max_frame_size(uint16_t rfcomm_cid);

/**
 * @brief Reserve packet buffer to allow to create RFCOMM packet
in place
 * @note Must only be called after a 'can send now' check or event
 * @note Asserts if packet buffer is already reserved
 *
 * 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)
```

```
 * }
 */
void rfcomm_reserve_packet_buffer(void);

/**
 * @brief Get outgoing buffer
 * @return pointer to outgoing rfcomm buffer
 */
uint8_t * rfcomm_get_outgoing_buffer(void);

/**
 * @brief Send packet prepared in outgoing buffer
 * @note This releases the outgoing rfcomm buffer
 * @param rfcomm_cid
 * @param len
 * @return status
 */
uint8_t rfcomm_send_prepared(uint16_t rfcomm_cid, uint16_t len);

/**
 * @brief Release outgoing buffer in case rfcomm_send_prepared was
not called
*/
void rfcomm_release_packet_buffer(void);

/**
 * @brief Enable L2CAP ERTM mode for RFCOMM. request callback is
used to provide ERTM buffer. released callback returns buffer
 *
 * @note on request_callback, the app must set the ertm_config,
buffer, size fields to enable ERTM for the current connection
 * @note if buffer is not set, BASIC mode will be used instead
 *
 * @note released_callback provides ertm_id from earlier request
to match request and release
 *
 * @param request_callback
 * @param released_callback
 */
void rfcomm_enable_l2cap_ertm(void
```

```
request_callback(rfcomm_ertm_request_t * request), void
released_callback(uint16_t ertm_id));


// Event getters for RFCOMM_EVENT_PORT_CONFIGURATION

/**
 * @brief Get field rfcomm_cid from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return rfcomm_cid
 */
static inline uint16_t
rfcomm_event_port_configuration_get_rfcomm_cid(const uint8_t *
event){
    return little_endian_read_16(event, 2);
}

/**
 * @brief Get field local from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return remote - false for local port, true for remote port
 */
static inline bool
rfcomm_event_port_configuration_get_remote(const uint8_t * event){
    return event[4] != 0;
}

/**
 * @brief Get field baud_rate from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return baud_rate
 */

static inline rpn_baud_t
rfcomm_event_port_configuration_get_baud_rate(const uint8_t *
event){
    return (rpn_baud_t) event[5];
```

```
}

/**
 * @brief Get field data_bits from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return data_bits
 */

static inline rpn_data_bits_t
rfcomm_event_port_configuration_get_data_bits(const uint8_t *
event){
    return (rpn_data_bits_t) (event[6] & 3);
}
/**
 * @brief Get field stop_bits from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return stop_bits
 */
static inline rpn_stop_bits_t
rfcomm_event_port_configuration_get_stop_bits(const uint8_t *
event){
    return (rpn_stop_bits_t) ((event[6] >> 2) & 1);
}

/**
 * @brief Get field parity from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return parity
 */
static inline rpn_parity_t
rfcomm_event_port_configuration_get_parity(const uint8_t * event){
    return (rpn_parity_t) ((event[6] >> 3) & 7);
}

/**
 * @brief Get field flow_control from event
RFCOMM_EVENT_PORT_CONFIGURATION
```

```
 * @param event packet
 * @return flow_control
 */

static inline uint8_t
rfcomm_event_port_configuration_get_flow_control(const uint8_t *
event){
    return event[7] & 0x3f;
}

/**
 * @brief Get field xon from event RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return xon
 */
static inline uint8_t
rfcomm_event_port_configuration_get_xon(const uint8_t * event){
    return event[8];
}

/**
 * @brief Get field xoff from event
RFCOMM_EVENT_PORT_CONFIGURATION
 * @param event packet
 * @return xoff
 */
static inline uint8_t
rfcomm_event_port_configuration_get_xoff(const uint8_t * event){
    return event[9];
}

/**
 * @brief De-Init RFCOMM
 */
void rfcomm_deinit(void);
```

SDP Client API

sdp_client.h

```
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 SDP Client Init
 */
void sdp_client_init(void);

/**
 * @brief Checks if the SDP Client is ready
 * @deprecated Please use sdp_client_register_query_callback
instead
 * @return true when no query is active
 */
bool sdp_client_ready(void);

/**
 * @brief Requests a callback, when the SDP Client is ready and
can be used
 * @note The callback might happens before
sdp_client_register_query_callback has returned
 * @param callback_registration
 */
uint8_t
sdp_client_register_query_callback(btstack_context_callback_registr
* callback_registration);

/**
 * @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

/**
 * @brief De-Init SDP Client
 */
void sdp_client_deinit(void);
```

SDP Client RFCOMM API

sdp_client_rfcomm.h

```
/**
 * @brief Searches SDP records on a remote device for RFCOMM
services with a given 16-bit UUID anywhere.
 * @note calls sdp_service_search_pattern_for_uuid16 that uses
global buffer
 * @param callback handler
 * @param remote BD_ADDR
 * @param uuid16
 */
```

```
 uint8_t
sdp_client_query_rfcomm_channel_and_name_for_uuid(btstack_packet_ha
callback, bd_addr_t remote, uint16_t uuid16);

/**
 * @brief Searches SDP records on a remote device for RFCOMM
services with a given 16-bit UUID in its ServiceClassIDList
 * @note calls sdp_service_search_pattern_for_uuid16 that uses
global buffer
 * @param callback handler
 * @param remote BD_ADDR
 * @param uuid16
 */
uint8_t
sdp_client_query_rfcomm_channel_and_name_for_service_class_uuid(bts
callback, bd_addr_t remote, uint16_t uuid16);

/**
 * @brief Searches SDP records on a remote device for RFCOMM
services with a given 128-bit UUID anywhere
 * @note calls sdp_service_search_pattern_for_uuid128 that uses
global buffer
 * @param callback handler
 * @param remote BD_ADDR
 * @param uuid128
 */
uint8_t
sdp_client_query_rfcomm_channel_and_name_for_uuid128(btstack_packet
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
callback, bd_addr_t remote, const uint8_t *
des_serviceSearchPattern);
```

SDP Server API

sdp_server.h

```
/**
 * @brief Set up SDP Server.
 */
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);

/**
 * @brief gets service record handle from record
 * @resutl service record handle or 0
 */
uint32_t sdp_get_service_record_handle(const uint8_t * record);

/**
 * @brief Finds an unused valid service record handle
 * @result handle
 */
uint32_t sdp_create_service_record_handle(void);

/**
 * @brief gets record for handle
 * @result record
 */
```

```
uint8_t * sdp_get_record_for_handle(uint32_t handle);

/**
 * @brief De-Init SDP Server
 */
void sdp_deinit(void);
```

SDP Utils API

sdp_util.h

```
// OFFSETS FOR LOCALIZED ATTRIBUTES -
BLUETOOTH_ATTRIBUTE_LANGUAGE_BASE_ATTRIBUTE_ID_LIST
#define SDP_Offset_ServiceName        0x0000
#define SDP_Offset_ServiceDescription 0x0001
#define SDP_Offset_ProviderName       0x0002

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);
uint32_t  de_get_len(const uint8_t * header);


// @note returned "string" is not NULL terminated
const uint8_t * de_get_string(const uint8_t * element);

de_size_t de_get_size_type(const uint8_t * header);
de_type_t de_get_element_type(const uint8_t * header);
uint32_t  de_get_header_size(const uint8_t * header);
// returns true if uint16 was read
bool      de_element_get_uint16(const uint8_t * element, uint16_t
* value);
uint32_t  de_get_data_size(const uint8_t * header);
uint32_t  de_get_uuid32(const uint8_t * element);
// returns true if uint16 was read
bool      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 *sequence);
void      de_pop_sequence(uint8_t * parent, uint8_t * child);
void      de_add_number(uint8_t *sequence, de_type_t type,
de_size_t size, uint32_t value);
void      de_add_data(uint8_t *sequence, de_type_t type, uint16_t
size, uint8_t *data);
void      de_add_uuid128(uint8_t * sequence, uint8_t * uuid);

// returns data element  len if date element is smaller than size
uint32_t de_get_len_safe(const uint8_t * header, uint32_t size);

// MARK: DES iterator
typedef struct {
    uint8_t * element;
    uint16_t pos;
    uint16_t length;
```

```
} des_iterator_t;

bool      des_iterator_init(des_iterator_t * it, uint8_t *
element);
bool      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);
bool      sdp_set_attribute_value_for_attribute_id(uint8_t *
record, uint16_t attributeID, uint32_t value);
bool      sdp_record_matches_service_search_pattern(uint8_t
*record, uint8_t *serviceSearchPattern);
uint16_t  sdp_get_filtered_size(uint8_t *record, uint8_t
*attributeIDList);
bool      sdp_filter_attributes_in_attributeIDList(uint8_t
*record, uint8_t *attributeIDList, uint16_t startOffset, uint16_t
maxBytes, uint16_t *usedBytes, uint8_t *buffer);
bool      sdp_attribute_list_contains_id(uint8_t *attributeIDList,
uint16_t attributeID);
bool      sdp_attribute_list_valid(uint8_t *attributeIDList);
bool      sdp_valid_service_search_pattern(uint8_t
*service_search_pattern);
/*
 * @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);
```

SPP Server API

spp_server.h : Create SPP SDP Records.

```
/**
 * @brief Create SDP record for SPP service with official SPP
Service Class
 * @param service buffer - needs to large enough
 * @param service_record_handle
 * @param rfcomm_channel
 * @param name or NULL for default value. Provide "" (empty
string) to skip attribute
 */
void spp_create_sdp_record(uint8_t *service, uint32_t
service_record_handle, int rfcomm_channel, const char *name);

/**
 * @brief Create SDP record for SPP service with custom service
UUID (e.g. for use with Android)
 * @param service buffer - needs to large enough
 * @param service_record_handle
 * @param service_uuid128 buffer
 * @param rfcomm_channel
 * @param name
 */
void spp_create_custom_sdp_record(uint8_t *service, uint32_t
service_record_handle, const uint8_t * service_uuid128, int
rfcomm_channel, const char *name);
```

Genral Access Profile (GAP) API

gap.h

```
// Classic + LE

/**
```

```
 * @brief Read RSSI
 * @param con_handle
 * @events: GAP_EVENT_RSSI_MEASUREMENT
 */
int gap_read_rssi(hci_con_handle_t con_handle);


/**
 * @brief Gets local address.
 */
void gap_local_bd_addr(bd_addr_t address_buffer);

/**
 * @brief Disconnect connection with handle
 * @param handle
 * @return status
 */
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);

/**
 * @brief Get HCI connection role
 * @param con_handle
 * @result hci_role_t HCI_ROLE_MASTER / HCI_ROLE_SLAVE /
HCI_ROLE_INVALID (if connection does not exist)
 */
hci_role_t gap_get_role(hci_con_handle_t connection_handle);

// Classic

/**
 * @brief Request role switch
 * @note this only requests the role switch. A
```

```
HCI_EVENT_ROLE_CHANGE is emitted and its status field will
indicate if the switch was succesful
 * @param addr
 * @param hci_role_t HCI_ROLE_MASTER / HCI_ROLE_SLAVE
 * @result status
 */
uint8_t gap_request_role(const bd_addr_t addr, hci_role_t role);

/**
 * @brief Sets local name.
 * @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 stays valid
 */
void gap_set_local_name(const char * local_name);

/**
 * @brief Set Extended Inquiry Response data
 * @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 HCI_EXTENDED_INQUIRY_RESPONSE_DATA_LEN
(240) bytes, is not copied make sure memory stays valid
 */
void gap_set_extended_inquiry_response(const uint8_t * data);

/**
 * @brief Set class of device
 */
void gap_set_class_of_device(uint32_t class_of_device);

/**
 * @brief Set default link policy settings for all classic ACL
links
 * @param default_link_policy_settings - see LM_LINK_POLICY_* in
bluetooth.h
 * @note common value: LM_LINK_POLICY_ENABLE_ROLE_SWITCH |
LM_LINK_POLICY_ENABLE_SNIFF_MODE to enable role switch and sniff
```

```
mode
 */
void gap_set_default_link_policy_settings(uint16_t
default_link_policy_settings);

/**
 * @brief Set Allow Role Switch param for outgoing classic ACL
links
 * @param allow_role_switch - true: allow remote device to request
role switch, false: stay master
 */
void gap_set_allow_role_switch(bool allow_role_switch);

/**
 * @brief Set  link supervision timeout for outgoing classic ACL
links
 * @param default_link_supervision_timeout * 0.625 ms, default
0x7d00 = 20 seconds, 0 = no link supervision timeout
 */
void gap_set_link_supervision_timeout(uint16_t
link_supervision_timeout);

/**
 * @brief Enable link watchdog. If no ACL packet is sent within
timeout_ms, the link will get disconnected
 * note: current implementation uses the automatic flush timeout
controller feature with a max timeout of 1.28s
 * @param timeout_ms
 */
void gap_enable_link_watchdog(uint16_t timeout_ms);

/**
 * @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);

/**
 * @brief Set security mode for all outgoing and incoming
connections. Default: GAP_SECURITY_MODE_4
 * @param security_mode is GAP_SECURITY_MODE_2 or
GAP_SECURITY_MODE_4
 * @return status ERROR_CODE_SUCCESS or
ERROR_CODE_UNSUPPORTED_FEATURE_OR_PARAMETER_VALUE
 */
uint8_t gap_set_security_mode(gap_security_mode_t security_mode);

/**
 * @brief Get security mode
 * @return security_mode
 */
gap_security_mode_t gap_get_security_mode(void);

/**
 * @brief Set security level for all outgoing and incoming
connections. Default: LEVEL_2
 * @param security_level
 * @note has to be called before services or profiles are
initialized
 */
void gap_set_security_level(gap_security_level_t security_level);

/**
 * @brief Get security level
 * @return security_level
 */
gap_security_level_t gap_get_security_level(void);

/**
 * @brief Set Secure Connections Only Mode for BR/EDR (Classic)
Default: false
 * @param enable
 */
void gap_set_secure_connections_only_mode(bool enable);
```

```
/**
 * @breif Get Secure Connections Only Mode
 * @param enabled
 */
bool gap_get_secure_connections_only_mode(void);

/**
 * @brief Set minimal security level for registered services
 * @param security_level
 * @note Called by L2CAP based on registered services
 */
void gap_set_minimal_service_security_level(gap_security_level_t
security_level);

/**
 * @brief Register filter for rejecting classic connections.
Callback will return 1 accept connection, 0 on reject.
 */
void gap_register_classic_connection_filter(int (*accept_callback)
(bd_addr_t addr, hci_link_type_t link_type));

/* Configure Secure Simple Pairing */

/**
 * @brief Enable will enable SSP during init. Default: true
 */
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 Enable/disable Secure Connections. Default: true if
supported by Controller
 */
void gap_secure_connections_enable(bool enable);

/**
 * @brief Query if Secure Connections can be used for Classic
connections.
 * @note Requires gap_secure_connections_enable == true and
Controller to support it
 */
bool gap_secure_connections_active(void);

/**
 * @brief If set, BTstack will confirm a numeric comparison and
enter '000000' if requested.
 */
void gap_ssp_set_auto_accept(int auto_accept);

/**
 * @brief Set required encryption key size for GAP Levels 1-3 on
classic connections.
 * @note If you need to reduce the required encryption key size,
please consider enabling
 *
ENABLE_MUTUAL_AUTHENTICATION_FOR_LEGACY_SECURE_CONNECTIONS to
prevent BIAS attacks.
 *       However, the re-authentication for Legacy Secure
Connections can cause a link loss
 *       in some Controller combinations.
 * @param encryption_key_size in bytes. Valid 7..16, default: 16
 */
void gap_set_required_encryption_key_size(uint8_t
encryption_key_size);

/**
 * @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);

/**
 * @brief map link keys to secure connection yes/no
 */
bool gap_secure_connection_for_link_key_type(link_key_type_t
link_key_type);

/**
 * @brief map link keys to authenticated
 */
bool gap_authenticated_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);

bool
gap_mitm_protection_required_for_security_level(gap_security_level_
level);

/**
 * @brief Set Page Scan Type
 * @param page_scan_interval * 0.625 ms, range: 0x0012..0x1000,
default: 0x0800
 * @param page_scan_windows  * 0.625 ms, range:
0x0011..page_scan_interval, default: 0x0012
```

```
 */
void gap_set_page_scan_activity(uint16_t page_scan_interval,
uint16_t page_scan_window);

/**
 * @brief Set Page Scan Type
 * @param page_scan_mode
 */
void gap_set_page_scan_type(page_scan_type_t page_scan_type);

/**
 * @brief Set Page Timeout
 * @param page_timeout * 0.625 ms, range: 0x0001..0xffff, default:
0x6000 (ca 15 seconds)
 */
void gap_set_page_timeout(uint16_t page_timeout);

// LE

/**
 * @brief Set parameters for LE Scan
 * @param scan_type 0 = passive, 1 = active
 * @param scan_interval range 0x0004..0x4000, unit 0.625 ms
 * @param scan_window range 0x0004..0x4000, unit 0.625 ms
 * @param scanning_filter_policy 0 = all devices, 1 = all from
whitelist
 * @return status
 */
uint8_t gap_set_scan_params(uint8_t scan_type, uint16_t
scan_interval, uint16_t scan_window,
                            uint8_t scanning_filter_policy);

/**
 * @brief Set parameters for LE Scan
 * @deprecated Use gap_set_scan_params instead
 */
void gap_set_scan_parameters(uint8_t scan_type, uint16_t
scan_interval, uint16_t scan_window);

/**
```

```
 * @brief Set duplicate filter for LE Scan
 * @param enabled if enabled, only one advertisements per BD_ADDR
is reported, default: false
 */
void gap_set_scan_duplicate_filter(bool enabled);

/**
 * @brief Set PHYs for LE Scan
 * @param phy bitmask: 1 = LE 1M PHY, 4 = LE Coded PHY
 */
void gap_set_scan_phys(uint8_t phys);

/**
 * @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 static
 */
void gap_random_address_set(const 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 Parameters
 * @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 update interval for resolvable private addresses
generated by the Controller
 * @param update_s timeout for updates in seconds
 * @return status
 */
uint8_t
gap_extended_advertising_set_resolvable_private_address_update(uint
update_s);

/**
 * @brief Provide storage for new advertising set and setup on
Controller
 * @note Using RPA as own_address_type requires
ENABLE_LE_ENHANCED_CONNECTION_COMPLETE_EVENT is required for
pairing
 * @param storage to use by stack, needs to stay valid until adv
set is removed with gap_extended_advertising_remove
 * @param advertising_parameters
 * @param out_advertising_handle to use with other adv config
commands
 * @return status
 * @events: GAP_SUBEVENT_ADVERTISING_SET_INSTALLED
 */
```

```
uint8_t gap_extended_advertising_setup(le_advertising_set_t *
storage, const le_extended_advertising_parameters_t *
advertising_parameters, uint8_t * out_advertising_handle);

/**
 * @param Set advertising params for advertising set
 * @note Using RPA as own_address_type requires
ENABLE_LE_ENHANCED_CONNECTION_COMPLETE_EVENT is required for
pairing
 * @param advertising_handle
 * @param advertising_parameters
 * @return status
 */
uint8_t gap_extended_advertising_set_params(uint8_t
advertising_handle, const le_extended_advertising_parameters_t *
advertising_parameters);

/**
 * @param Get advertising params for advertising set, e.g. to
update params
 * @param advertising_handle
 * @param advertising_parameters
 * @return status
 */
uint8_t gap_extended_advertising_get_params(uint8_t
advertising_handle, le_extended_advertising_parameters_t *
advertising_parameters);

/**
 * @param Set periodic advertising params for advertising set
 * @param advertising_handle
 * @param advertising_parameters
 * @return status
 */
uint8_t gap_periodic_advertising_set_params(uint8_t
advertising_handle, const le_periodic_advertising_parameters_t *
advertising_parameters);

/**
 * @param Get params for periodic advertising set, e.g. to update
```

```
params
 * @param advertising_handle
 * @param advertising_parameters
 * @return status
 */
uint8_t gap_periodic_advertising_get_params(uint8_t
advertising_handle, le_periodic_advertising_parameters_t *
advertising_parameters);

/**
 * @param Set random addrress for advertising set
 * @param advertising_handle
 * @param random_address
 * @return status
 */
uint8_t gap_extended_advertising_set_random_address(uint8_t
advertising_handle, bd_addr_t random_address);

/**
 * @brief Set Advertising Data for a advertisement set
 * @param advertising_handle
 * @param advertising_data_length
 * @param advertising_data
 * @return status
 */
uint8_t gap_extended_advertising_set_adv_data(uint8_t
advertising_handle, uint16_t advertising_data_length, const
uint8_t * advertising_data);

/**
 * @brief Set Scan Response Data for a advertisement set
 * @param advertising_handle
 * @param scan_response_data_length
 * @param scan_response_data
 * @return status
 */
uint8_t gap_extended_advertising_set_scan_response_data(uint8_t
advertising_handle, uint16_t scan_response_data_length, const
uint8_t * scan_response_data);
```

```
/**
 * @brief Set data for periodic advertisement set
 * @param advertising_handle
 * @param periodic_data_length
 * @param periodic_data
 * @return status
 */
uint8_t gap_periodic_advertising_set_data(uint8_t
advertising_handle, uint16_t periodic_data_length, const uint8_t *
periodic_data);

/**
 * @brief Start advertising advertising set
 * @param advertising_handle
 * @param timeout in 10ms, or 0 == no timeout
 * @param num_extended_advertising_events Controller shall send,
or 0 == no max number
 * @return status
 */
uint8_t gap_extended_advertising_start(uint8_t advertising_handle,
uint16_t timeout, uint8_t num_extended_advertising_events);

/**
 * @brief Stop advertising
 * @param advertising_handle
 * @return status
 */
uint8_t gap_extended_advertising_stop(uint8_t advertising_handle);

/**
 * @brief Start periodic advertising for given advertising set
 * @param advertising_handle
 * @param include_adi
 * @return status
 */
uint8_t gap_periodic_advertising_start(uint8_t advertising_handle,
bool include_adi);

/**
 * @brief Stop periodic advertising for given advertising set
```

```
 * @param advertising_handle
 * @return status
 */
uint8_t gap_periodic_advertising_stop(uint8_t advertising_handle);

/**
 * @brief Set Default Periodic Advertising Sync Transfer
Parameters
 * @note The parameters are used for all subsequent connections
over the LE transport.
 *       If mode != 0, an
HCI_LE_Periodic_Advertising_Sync_Transfer_Received event will be
emitted by the Controller
 * @param mode 0 = ignore (default), 1 = periodic advertising
events disabled
 *             2 = periodic advertising events enabled with
duplicate filtering
 *             3 = periodic advertising events enabled with
duplicate filtering
 * @param skip The number of periodic advertising packets that can
be skipped after a successful receive
 * @param sync_timeout Range: 0x000A to 0x4000, Time = N*10 ms,
Time Range: 100 ms to 163.84 s
 * @param cte_type  bit 0 = Do not sync to packets with an AoA
Constant Tone Extension
 *                  bit 1 = Do not sync to packets with an AoD
Constant Tone Extension with 1 us slots
 *                  bit 2 = Do not sync to packets with an AoD
Constant Tone Extension with 2 us slots
 *                  bit 3 = Do not sync to packets without a
Constant Tone Extension
 * @return status
 */
uint8_t
gap_periodic_advertising_sync_transfer_set_default_parameters(uint8
mode, uint16_t skip, uint16_t sync_timeout, uint8_t cte_type);

/**
 * @brief Send Periodic Advertising Sync Transfer to connected
device
```

```
 * @param con_handle of connected device
 * @param service_data 16-bit data to transfer to remote host
 * @param sync_handle of synchronized periodic advertising train
to transfer
 * @return status
 */
uint8_t
gap_periodic_advertising_sync_transfer_send(hci_con_handle_t
con_handle, uint16_t service_data, hci_con_handle_t sync_handle);

/**
 * @brief Send Periodic Advertising Set Info Transfer to connected
device
 * @param con_handle of connected device
 * @param service_data 16-bit data to transfer to remote host
 * @param advertising_handle of local periodic advertising train
to transfer
 * @return status
 */
uint8_t
gap_periodic_advertising_set_info_transfer_send(hci_con_handle_t
con_handle, uint16_t service_data, uint8_t advertising_handle);

/**
 * @brief Remove advertising set from Controller
 * @param advertising_handle
 * @return status
 * @events GAP_SUBEVENT_ADVERTISING_SET_REMOVED
 */
uint8_t gap_extended_advertising_remove(uint8_t
advertising_handle);

/**
 * @brief Create Broadcast Isochronous Group (BIG)
 * @param storage to use by stack, needs to stay valid until adv
set is removed with gap_big_terminate
 * @param big_params
 * @return status
 * @events GAP_SUBEVENT_BIG_CREATED unless interrupted by call to
gap_big_terminate
```

```
 */
uint8_t gap_big_create(le_audio_big_t * storage,
le_audio_big_params_t * big_params);

/**
 * @brief Terminate Broadcast Isochronous Group (BIG)
 * @param big_handle
 * @return status
 * @events GAP_SUBEVENT_BIG_TERMINATED
 */
uint8_t gap_big_terminate(uint8_t big_handle);

/**
 * @brief Synchronize to Broadcast Isochronous Group (BIG)
 * @param storage to use by stack, needs to stay valid until adv
set is removed with gap_big_terminate
 * @param big_sync_params
 * @return status
 * @events GAP_SUBEVENT_BIG_SYNC_CREATED unless interrupted by
call to gap_big_sync_terminate
 */
uint8_t gap_big_sync_create(le_audio_big_sync_t * storage,
le_audio_big_sync_params_t * big_sync_params);

/**
 * @brief Stop synchronizing to Broadcast Isochronous Group (BIG).
Triggers GAP_SUBEVENT_BIG_SYNC_STOPPED
 * @note Also used to stop synchronizing before BIG Sync was
established
 * @param big_handle
 * @return status
 * @events GAP_SUBEVENT_BIG_SYNC_STOPPED
 */
uint8_t gap_big_sync_terminate(uint8_t big_handle);

/**
 * @brief Create Connected Isochronous Group (CIG)
 * @param storage to use by stack, needs to stay valid until CIG
removed with gap_cig_remove
 * @param cig_params
```

```
 * @return status
 * @events GAP_SUBEVENT_CIG_CREATED unless interrupted by call to
gap_cig_remove
 */
uint8_t gap_cig_create(le_audio_cig_t * storage,
le_audio_cig_params_t * cig_params);

/**
 * @brief Remove Connected Isochronous Group (CIG)
 * @param cig_id
 * @return status
 * @events GAP_SUBEVENT_CIG_TERMINATED
 */
uint8_t gap_cig_remove(uint8_t cig_id);

/**
 * @brief Create Connected Isochronous Streams (CIS)
 * @note number of CIS from cig_params in gap_cig_create is used
 * @param cig_id
 * @param cis_con_handles array of CIS Connection Handles
 * @param acl_con_handles array of ACL Connection Handles
 * @return status
 * @events GAP_SUBEVENT_CIS_CREATED unless interrupted by call to
gap_cig_remove
 */
uint8_t gap_cis_create(uint8_t cig_id, hci_con_handle_t
cis_con_handles [], hci_con_handle_t acl_con_handles []);

/**
 * @brief Accept Connected Isochronous Stream (CIS)
 * @param cis_con_handle
 * @return status
 * @events GAP_SUBEVENT_CIS_CREATED
 */
uint8_t gap_cis_accept(hci_con_handle_t cis_con_handle);

#ifdef ENABLE_LE_AUDIO_CODEC_OFFLOAD
/**
 * @brief Set codec configuration send to Controller for LC3
Offloading
```

```
 * @param cis_handle
 * @param data_direction
 * @param offload_config
 */
uint8_t gap_cis_set_codec_configuration(hci_con_handle_t
cis_handle, uint8_t data_direction,
                                        const
le_audio_offload_config_t* offload_config);
#endif

/**
 * @brief Reject Connected Isochronous Stream (CIS)
 * @param cis_con_handle
 * @return status
 * @events GAP_SUBEVENT_CIS_CREATED
 */
uint8_t gap_cis_reject(hci_con_handle_t cis_con_handle);

/**
 * @brief Set connection parameters for outgoing connections and
connection parameter updates
 * @param conn_scan_interval (unit: 0.625 msec), default: 60 ms
 * @param conn_scan_window (unit: 0.625 msec), default: 30 ms
 * @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_scan_interval,
uint16_t conn_scan_window,
    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 Set initiating PHYs for outgoing connections
 * @param phy bitmask: 1 = LE 1M PHY, 2 = LE 2M PHY, 4 = LE Coded
```

```
PHY
 */
void gap_set_connection_phys(uint8_t phys);

/**
 * @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)
 * @return 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)
 * @return 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 Test if connection parameters are inside in existing
rage
 * @param conn_interval_min (unit: 1.25ms)
 * @param conn_interval_max (unit: 1.25ms)
 * @param conn_latency
 * @param supervision_timeout (unit: 10ms)
 * @return 1 if included
 */
int
gap_connection_parameter_range_included(le_connection_parameter_rar
* existing_range, uint16_t le_conn_interval_min, uint16_t
le_conn_interval_max, uint16_t le_conn_latency, uint16_t
le_supervision_timeout);

/**
 * @brief Request an update of the connection subrating for a
given LE connection
 * @param handle
 * @param subrate_min
 * @param subrate_max
 * @param max_latency (in units of subrated connection intervals)
 * @param continuation_number (Minimum number of underlying
connection events to remain active after a packet containing a
Link Layer PDU with a non-zero Length field is sent or received)
 * @param supervision_timeout (unit: 10ms) range: 10..3200 (100 ms
to 32 s)
 * @return status
 */
uint8_t gap_request_connection_subrating(hci_con_handle_t
```

```
con_handle, uint16_t subrate_min, uint16_t subrate_max,
                                        uint16_t max_latency,
uint16_t continuation_number, uint16_t supervision_timeout);

/**
 * @brief Set max number of connections in LE Peripheral role (if
Bluetooth Controller supports it)
 * @note: default: 1
 * @param max_peripheral_connections
 */
void gap_set_max_number_peripheral_connections(int
max_peripheral_connections);

/**
 * @brief Add Device to Whitelist
 * @param address_typ
 * @param address
 * @return status
 */
uint8_t gap_whitelist_add(bd_addr_type_t address_type, const
bd_addr_t address);

/**
 * @brief Remove Device from Whitelist
 * @param address_typ
 * @param address
 * @return status
 */
uint8_t gap_whitelist_remove(bd_addr_type_t address_type, const
bd_addr_t address);

/**
 * @brief Clear Whitelist
 * @return status
 */
uint8_t gap_whitelist_clear(void);

/**
 * @brief Connect to remote LE device
 * @return status
```

```
 */
uint8_t gap_connect(const bd_addr_t addr, bd_addr_type_t
addr_type);

/**
 * @brief Connect with Whitelist
 * @note Explicit whitelist management and this connect with
whitelist replace deprecated gap_auto_connection_* functions
 * @return status
 */
uint8_t gap_connect_with_whitelist(void);

/**
 * @brief Cancel connection process initiated by gap_connect
 * @return status
 */
uint8_t gap_connect_cancel(void);

/**
 * @brief Auto Connection Establishment - Start Connecting to
device
 * @deprecated Please setup Whitelist with gap_whitelist_* and
start connecting with gap_connect_with_whitelist
 * @param address_type
 * @param address
 * @return status
 */
uint8_t gap_auto_connection_start(bd_addr_type_t address_type,
const bd_addr_t address);

/**
 * @brief Auto Connection Establishment - Stop Connecting to
device
 * @deprecated Please setup Whitelist with gap_whitelist_* and
start connecting with gap_connect_with_whitelist
 * @param address_type
 * @param address
 * @return status
 */
uint8_t gap_auto_connection_stop(bd_addr_type_t address_type,
```

```
const bd_addr_t address);

/**
 * @brief Auto Connection Establishment - Stop everything
 * @deprecated Please setup Whitelist with gap_whitelist_* and
start connecting with gap_connect_with_whitelist
 * @note  Convenience function to stop all active auto connection
attempts
 * @return status
 */
uint8_t gap_auto_connection_stop_all(void);

/**
 * @brief Set LE PHY
 * @param con_handle
 * @param all_phys 0 = set rx/tx, 1 = set only rx, 2 = set only tx
 * @param tx_phys 1 = 1M, 2 = 2M, 4 = Coded
 * @param rx_phys 1 = 1M, 2 = 2M, 4 = Coded
 * @param phy_options 0 = no preferred coding for Coded, 1 = S=2
coding (500 kbit), 2 = S=8 coding (125 kbit)
 * @return status
 */
uint8_t gap_le_set_phy(hci_con_handle_t con_handle, uint8_t
all_phys, uint8_t tx_phys, uint8_t rx_phys, uint16_t phy_options);

/**
 * @brief Get connection interval
 * @param con_handle
 * @return connection interval, otherwise 0 if error
 */
uint16_t gap_le_connection_interval(hci_con_handle_t con_handle);

/**
 *
 * @brief Get encryption key size.
 * @param con_handle
 * @return 0 if not encrypted, 7-16 otherwise
 */
uint8_t gap_encryption_key_size(hci_con_handle_t con_handle);
```

```
/**
 * @brief Get authentication property.
 * @param con_handle
 * @return true if bonded with OOB/Passkey (AND MITM protection)
 */
bool gap_authenticated(hci_con_handle_t con_handle);

/**
 * @brief Get secure connection property
 * @param con_handle
 * @return true if bonded usiung LE Secure Connections
 */
bool gap_secure_connection(hci_con_handle_t con_handle);

/**
 * @brief Queries authorization state.
 * @param con_handle
 * @return authorization_state for the current session
 */
authorization_state_t gap_authorization_state(hci_con_handle_t
con_handle);

/**
 * @brief Get bonded property (BR/EDR/LE)
 * @note LE: has to be called after identity resolving is complete
 * @param con_handle
 * @return true if bonded
 */
bool gap_bonded(hci_con_handle_t con_handle);

// Classic
#ifdef ENABLE_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 Deletes link key for remote device with baseband
address.
 * @param addr
 * @note On most desktop ports, the Link Key DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 */
void gap_drop_link_key_for_bd_addr(bd_addr_t addr);

/**
 * @brief Delete all stored link keys
 * @note On most desktop ports, the Link Key DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 */
void gap_delete_all_link_keys(void);

/**
 * @brief Store link key for remote device with baseband address
 * @param addr
 * @param link_key
 * @param link_key_type
 * @note On most desktop ports, the Link Key DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
```

```
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 */
void gap_store_link_key_for_bd_addr(bd_addr_t addr, link_key_t
link_key, link_key_type_t type);

/**
 * @brief Get link for remote device with basband address
 * @param addr
 * @param link_key (out) is stored here
 * @param link_key_type (out) is stored here
 * @note On most desktop ports, the Link Key DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 */
bool gap_get_link_key_for_bd_addr(bd_addr_t addr, link_key_t
link_key, link_key_type_t * type);

/**
 * @brief Setup Link Key iterator
 * @param it
 * @return 1 on success
 * @note On most desktop ports, the Link Key DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 */
int gap_link_key_iterator_init(btstack_link_key_iterator_t * it);

/**
 * @brief Get next Link Key
 * @param it
 * @brief addr
 * @brief link_key
```

```
 * @brief type of link key
 * @return 1, if valid link key found
 * @see note on gap_link_key_iterator_init
 */
int gap_link_key_iterator_get_next(btstack_link_key_iterator_t *
it, bd_addr_t bd_addr, link_key_t link_key, link_key_type_t *
type);

/**
 * @brief Frees resources allocated by iterator_init
 * @note Must be called after iteration to free resources
 * @param it
 * @see note on gap_link_key_iterator_init
 */
void gap_link_key_iterator_done(btstack_link_key_iterator_t * it);

/**
 * @brief Start GAP Classic Inquiry
 * @param duration in 1.28s units
 * @return status
 * @events: GAP_EVENT_INQUIRY_RESULT, GAP_EVENT_INQUIRY_COMPLETE
 */
int gap_inquiry_start(uint8_t duration_in_1280ms_units);

/**
 * @brief Start GAP Classic Periodic Inquiry
 * @param duration in 1.28s units
 * @param max_period_length between consecutive inquiries in 1.28s
units
 * @param min_period_length between consecutive inquiries in 1.28s
units
 * @return status
 * @events: GAP_EVENT_INQUIRY_RESULT, GAP_EVENT_INQUIRY_COMPLETE
 */
uint8_t gap_inquiry_periodic_start(uint8_t duration, uint16_t
max_period_length, uint16_t min_period_length);

/**
 * @brief Stop GAP Classic Inquiry (regular or periodic)
 * @return 0 if ok
```

```
 * @events GAP_EVENT_INQUIRY_COMPLETE
 */
int gap_inquiry_stop(void);

/**
 * @brief Set LAP for GAP Classic Inquiry
 * @param lap GAP_IAC_GENERAL_INQUIRY (default),
GAP_IAC_LIMITED_INQUIRY
 */
void gap_inquiry_set_lap(uint32_t lap);

/**
 * @brief Set Inquiry Scan Activity
 * @param inquiry_scan_interval range: 0x0012 to 0x1000; only even
values are valid, Time = N * 0.625 ms
 * @param inquiry_scan_window range: 0x0011 to 0x1000; Time = N *
0.625 ms
 */
void gap_inquiry_set_scan_activity(uint16_t inquiry_scan_interval,
uint16_t inquiry_scan_window);

/**
 * @brief Set Inquiry Transmit Power Level
 * @param tx_power range: -70 to 20 dBm
 */
void gap_inquiry_set_transmit_power_level(int8_t tx_power);

/**
 * @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(const bd_addr_t addr, uint8_t
page_scan_repetition_mode, uint16_t clock_offset);

/**
 * @brief Legacy Pairing Pin Code Response
```

```
 * @note data is not copied, pointer has to stay valid
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_pin_code_response(const bd_addr_t addr, const char * pin);

/**
 * @brief Legacy Pairing Pin Code Response for binary data / non-
strings
 * @note data is not copied, pointer has to stay valid
 * @param addr
 * @param pin_data
 * @param pin_len
 * @return 0 if ok
 */
int gap_pin_code_response_binary(const bd_addr_t addr, const
uint8_t * pin_data, uint8_t pin_len);

/**
 * @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(const 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(const bd_addr_t addr);

/**
 * @brief Accept SSP Numeric Comparison
 * @param addr
 * @param passkey
 * @return 0 if ok
 */
int gap_ssp_confirmation_response(const bd_addr_t addr);

/**
 * @brief Abort SSP Numeric Comparison/Pairing
 * @param addr
 * @param pin
 * @return 0 if ok
 */
int gap_ssp_confirmation_negative(const bd_addr_t addr);

/**
 * @brief Generate new OOB data
 * @note OOB data will be provided in GAP_EVENT_LOCAL_OOB_DATA and
be used in future pairing procedures
 */
void gap_ssp_generate_oob_data(void);

/**
 * @brief Report Remote OOB Data
 * @note Pairing Hash and Randomizer are expected in big-endian
byte format
 * @param bd_addr
 * @param c_192 Simple Pairing Hash C derived from P-192 public
key
 * @param r_192 Simple Pairing Randomizer derived from P-192
public key
 * @param c_256 Simple Pairing Hash C derived from P-256 public
key
 * @param r_256 Simple Pairing Randomizer derived from P-256
public key
```

```
 * @return status
 */
uint8_t gap_ssp_remote_oob_data(const bd_addr_t addr, const
uint8_t * c_192, const uint8_t * r_192, const uint8_t * c_256,
const uint8_t * r_256);

/**
 * Send SSP IO Capabilities Reply
 * @note IO Capabilities (Negative) Reply is sent automatically
unless ENABLE_EXPLICIT_IO_CAPABILITIES_REPLY
 * @param addr
 * @return status
 */
uint8_t gap_ssp_io_capabilities_response(const bd_addr_t addr);

/**
 * Send SSP IO Capabilities Negative Reply
 * @note IO Capabilities (Negative) Reply is sent automatically
unless ENABLE_EXPLICIT_IO_CAPABILITIES_REPLY
 * @param addr
 * @return status
 */
uint8_t gap_ssp_io_capabilities_negative(const bd_addr_t addr);

/**
 * Send Link Key Reponse
 * @note Link Key (Negative) Reply is sent automatically unless
ENABLE_EXPLICIT_LINK_KEY_RESPONSE
 * @param addr
 * @param link_key
 * @param type or INVALID_LINK_KEY if link key not available
 * @return status
 */
 uint8_t gap_send_link_key_response(const bd_addr_t addr,
link_key_t link_key, link_key_type_t type);

/**
 * @brief Enter Sniff mode
 * @param con_handle
 * @param sniff_min_interval range: 0x0002 to 0xFFFE; only even
```

```
values are valid, Time = N * 0.625 ms
 * @param sniff_max_interval range: 0x0002 to 0xFFFE; only even
values are valid, Time = N * 0.625 ms
 * @param sniff_attempt Number of Baseband receive slots for sniff
attempt.
 * @param sniff_timeout Number of Baseband receive slots for sniff
timeout.
 * @return status
 */
uint8_t gap_sniff_mode_enter(hci_con_handle_t con_handle, uint16_t
sniff_min_interval, uint16_t sniff_max_interval, uint16_t
sniff_attempt, uint16_t sniff_timeout);

/**
 * @brief Exit Sniff mode
 * @param con_handle
 * @return status
 */
uint8_t gap_sniff_mode_exit(hci_con_handle_t con_handle);

/**
 * @brief Configure Sniff Subrating
 * @param con_handle
 * @param max_latency range: 0x0002 to 0xFFFE; Time = N * 0.625 ms
 * @param min_remote_timeout range:  0x0002 to 0xFFFE; Time = N *
0.625 ms
 * @param min_local_timeout range:  0x0002 to 0xFFFE; Time = N *
0.625 ms
 * @return status
 */
uint8_t gap_sniff_subrating_configure(hci_con_handle_t con_handle,
uint16_t max_latency, uint16_t min_remote_timeout, uint16_t
min_local_timeout);

/**
 * @Brief Set QoS
 * @param con_handle
 * @param service_type
 * @param token_rate
 * @param peak_bandwidth
```

```
 * @param latency
 * @param delay_variation
 * @return status
 */
uint8_t gap_qos_set(hci_con_handle_t con_handle,
hci_service_type_t service_type, uint32_t token_rate, uint32_t
peak_bandwidth, uint32_t latency, uint32_t delay_variation);

#endif

// LE

/**
 * @brief Get own addr type and address used for LE for next scan/
advertisement/connect operation
 */
void gap_le_get_own_address(uint8_t * addr_type, bd_addr_t addr);

/**
 * @brief Get own addr type and address used for LE advertisements
(Peripheral)
 */
void gap_le_get_own_advertisements_address(uint8_t * addr_type,
bd_addr_t addr);

/**
 * @brief Get own addr type and address used for LE Extended
Advertisiing (Peripheral)
 */
void gap_le_get_own_advertising_set_address(uint8_t * addr_type,
bd_addr_t addr, uint8_t advertising_handle);

/**
 * @brief Get own addr type and address used for LE connections
(Central)
 */
void gap_le_get_own_connection_address(uint8_t * addr_type,
bd_addr_t addr);

/**
```

```
 * @brief Get state of connection re-encryption for bonded devices
when in central role
 * @note used by gatt_client and att_server to wait for re-
encryption
 * @param con_handle
 * @return 1 if security setup is active
 */
bool gap_reconnect_security_setup_active(hci_con_handle_t
con_handle);

/**
 * @brief Delete bonding information for remote device
 * @note On most desktop ports, the LE Device DB uses a TLV and
there is one TLV storage per
 *       Controller resp. its Bluetooth Address. As the Bluetooth
Address is retrieved during
 *       power up, this function only works, when the stack is in
working state for these ports.
 * @param address_type
 * @param address
 */
void gap_delete_bonding(bd_addr_type_t address_type, bd_addr_t
address);

/**
 * LE Privacy 1.2 - requires support by Controller and
ENABLE_LE_RESOLVING_LIST to be defined
 */

/**
 * Set Privacy Mode for use in Resolving List. Default:
LE_PRIVACY_MODE_DEVICE
 * @note Only applies for new devices added to resolving list,
please call before startup
 * @param privacy_mode
 */
void gap_set_peer_privacy_mode(le_privacy_mode_t privacy_mode );

/**
 * @brief Load LE Device DB entries into Controller Resolving List
```

```
to allow filtering on
 *        bonded devies with resolvable private addresses
 * @return EROOR_CODE_SUCCESS if supported by Controller
 */
uint8_t gap_load_resolving_list_from_le_device_db(void);

/**
 * @brief Register callback that gets executed during random
address update
 * @note gap_privacy_client_ready needs to be called after
callback is received
 * @param client
 * @return status
 */
void gap_privacy_client_register(gap_privacy_client_t * client);

/**
 * @brief Acknowledge upcoming random address change
 * @param client
 * @return status
 */
void gap_privacy_client_ready(gap_privacy_client_t * client);

/**
 * @brief Unregister callback from random address updates
 * @param client
 * @return status
 */
void gap_privacy_client_unregister(gap_privacy_client_t * client);

/**
 * @brief Get local persistent IRK
 */
const uint8_t * gap_get_persistent_irk(void);
```

Host Controler Interface (HCI) API

hci.h

```
// 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 Enable custom init for chipset driver to send HCI
commands before HCI Reset
 */
void hci_enable_custom_pre_init(void);

/**
 * @brief Configure Bluetooth hardware control. Has to be called
before power on.
 * @[aram hardware_control implementation
 */
void hci_set_control(const btstack_control_t *hardware_control);

#ifdef HAVE_SCO_TRANSPORT
/**
 * @brief Set SCO Transport implementation for SCO over PCM mode
 * @param sco_transport that sends SCO over I2S or PCM interface
 */
void hci_set_sco_transport(const btstack_sco_transport_t
*sco_transport);
#endif

#ifdef ENABLE_CLASSIC
```

```
/**
 * @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);
#endif

/**
 * @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 number of ISO packets to buffer for BIS/CIS
 * @param num_packets (default = 1)
 */
void hci_set_num_iso_packets_to_queue(uint8_t num_packets);

/**
 * @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 inquriy_mode);

/**
 * @brief Requests the change of BTstack power mode.
 * @param power_mode
 * @return 0 if success, otherwise error
 */
int  hci_power_control(HCI_POWER_MODE power_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 Remove event packet handler.
 */
void
hci_remove_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);

/**
 * @brief Registers a packet handler for ISO data. Used for LE
Audio profiles
 */
void hci_register_iso_packet_handler(btstack_packet_handler_t
handler);

// Sending HCI Commands

/**
 * @brief Check if CMD packet can be sent to controller
 * @return true if command can be sent
 */
bool 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.
 * @return status
 */
uint8_t hci_send_cmd(const hci_cmd_t * cmd, ...);


// Sending SCO Packets

/** @brief Get SCO payload length for existing SCO connection and
current SCO Voice setting
 *  @note  Using SCO packets of the exact length is required for
USB transfer in general and some H4 controllers as well
 *  @param sco_con_handle
 *  @return Length of SCO payload in bytes (not audio frames)
incl. 3 byte header
 */
uint16_t hci_get_sco_packet_length_for_connection(hci_con_handle_t
```

```
sco_con_handle);

/** @brief Get SCO packet length for one of the existing SCO
connections and current SCO Voice setting
 *  @deprecated Please use
hci_get_sco_packet_length_for_connection instead
 *  @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
 */
uint16_t 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.
 * @param con_handle
 */
void
hci_request_sco_can_send_now_event_for_con_handle(hci_con_handle_t
con_handle);

/**
 * @brief Request emission of HCI_EVENT_SCO_CAN_SEND_NOW as soon
as possible
 * @deprecated Please use
hci_request_sco_can_send_now_event_for_con_handle instead
 * @note HCI_EVENT_SCO_CAN_SEND_NOW might be emitted during call
to this function
 *       so packet handler should be ready to handle it.
con_handle will
 *       be set to HCI_CON_HANDLE_INVALID in the event
 */
void hci_request_sco_can_send_now_event(void);

/**
 * @brief Check HCI packet buffer and if SCO packet can be sent to
```

```
controller
 * @param con_handle
 * @return true if sco packet can be sent
 */
bool hci_can_send_sco_packet_now_for_con_handle(hci_con_handle_t
con_handle);

/**
 * @brief Check HCI packet buffer and if SCO packet can be sent to
controller
 * @deprecated Please use
hci_can_send_sco_packet_now_for_con_handle instead
 * @return true if sco packet can be sent
 */
bool hci_can_send_sco_packet_now(void);

/**
 * @brief Check if SCO packet can be sent to controller
 * @return true if sco packet can be sent
 */
bool hci_can_send_prepared_sco_packet_now(void);

/**
 * @brief Send SCO packet prepared in HCI packet buffer
 */
uint8_t hci_send_sco_packet_buffer(int size);

/**
 * @brief Request emission of HCI_EVENT_BIS_CAN_SEND_NOW for all
BIS as soon as possible
 * @param big_handle
 * @note HCI_EVENT_ISO_CAN_SEND_NOW might be emitted during call
to this function
 *       so packet handler should be ready to handle it
 */
uint8_t hci_request_bis_can_send_now_events(uint8_t big_handle);

/**
 * @brief Request emission of HCI_EVENT_CIS_CAN_SEND_NOW for all
outgoing CIS of referenced CIG
```

```
 *
 * @note As CIG_IDs are only unique for Central, a CIS Connection
Handle is used to identify the CIG.
 *       The group_complete field in HCI_EVENT_CIS_CAN_SEND_NOW
can be used to request again.
 *
 * @note HCI_EVENT_CIS_CAN_SEND_NOW might be emitted during call
to this function
 *       so packet handler should be ready to handle it
 *
 * @param cis_con_handle
 */
uint8_t hci_request_cis_can_send_now_events(hci_con_handle_t
cis_con_handle);

/**
 * @brief Send ISO packet prepared in HCI packet buffer
 */
uint8_t hci_send_iso_packet_buffer(uint16_t size);

/**
 * Reserves outgoing packet buffer.
 * @note Must only be called after a 'can send now' check or event
 * @note Asserts if packet buffer is already reserved
 */
void 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_prepared
 */
void hci_release_packet_buffer(void);

/**
* @brief Sets the master/slave policy
```

```
* @param policy (0: attempt to become master, 1: let connecting
device decide)
*/
void hci_set_master_slave_policy(uint8_t policy);

/**
 * @brief Check if Controller supports BR/EDR (Bluetooth Classic)
 * @return true if supported
 * @note only valid in working state
 */
bool hci_classic_supported(void);

/**
 * @brief Check if Controller supports LE (Bluetooth Low Energy)
 * @return true if supported
 * @note only valid in working state
 */
bool hci_le_supported(void);

/**
 * @brief Check if LE Extended Advertising is supported
 * @return true if supported
 */
bool hci_le_extended_advertising_supported(void);

/** @brief Check if address type corresponds to LE connection
 *  @bparam address_type
 *  @erturn true if LE connection
 */
bool hci_is_le_connection_type(bd_addr_type_t address_type);

/** @brief Check if address type corresponds to Identity Address
 *  @bparam address_type
 *  @erturn true if LE connection
 */
bool hci_is_le_identity_address_type(bd_addr_type_t address_type);
```

HCI Logging API

hci_dump.h : Dump HCI trace as BlueZ's hcidump format, Apple's PacketLogger, or stdout.

```
typedef enum {
    HCI_DUMP_INVALID = 0,
    HCI_DUMP_BLUEZ,
    HCI_DUMP_PACKETLOGGER,
    HCI_DUMP_BTSNOOP,
} hci_dump_format_t;

typedef struct {
    // reset output, called if max packets is reached, to limit
file size
    void (*reset)(void);
    // log packet
    void (*log_packet)(uint8_t packet_type, uint8_t in, uint8_t
*packet, uint16_t len);
    // log message
    void (*log_message)(int log_level, const char * format,
va_list argptr);
#ifdef __AVR__ \
    // log message - AVR
    void (*log_message_P)(int log_level, PGM_P * format, va_list
argptr);
#endif
} hci_dump_t;

/**
 * @brief Init HCI Dump
 * @param hci_dump_impl - platform-specific implementation
 */
void hci_dump_init(const hci_dump_t * hci_dump_impl);

/**
 * @brief Enable packet logging
 * @param enabled default: true
 */
void hci_dump_enable_packet_log(bool enabled);
```

```
/**
 * @brief
 */
void hci_dump_enable_log_level(int log_level, int enable);

/**
 * @brief Set max number of packets - output file might be
truncated
 * @param packets - number of packets to log, -1 for unlimited
 */
void hci_dump_set_max_packets(int packets);

/**
 * @brief Dump Packet
 * @param packet_type
 * @param in is 1 for incoming, 0 for outoing
 * @param packet
 * @param len
 */
void hci_dump_packet(uint8_t packet_type, uint8_t in, uint8_t
*packet, uint16_t len);

/**
 * @brief Dump Message
 * @param log_level
 * @param format
 */
void hci_dump_log(int log_level, const char * format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 2, 3)))
#endif
;

#ifdef __AVR__
/*
 * @brief Dump Message using format string stored in PGM memory
(allows to save RAM)
 * @param log_level
 * @param format
```

```
 */
void hci_dump_log_P(int log_level, PGM_P format, ...)
#ifdef __GNUC__
__attribute__ ((format (__printf__, 2, 3)))
#endif
;
#endif

/**
 * @brief Dump internal BTstack event
 * @note only logged if ENABLE_LOG_BTSTACK_EVENTS is defined
 * @param packet
 * @param len
 */
void hci_dump_btstack_event(const uint8_t *packet, uint16_t len);

/**
 * @brief Setup header for PacketLogger format
 * @param buffer
 * @param tv_sec
 * @param tv_us
 * @param packet_type
 * @param in
 * @param len
 */
void hci_dump_setup_header_packetlogger(uint8_t * buffer, uint32_t
tv_sec, uint32_t tv_us, uint8_t packet_type, uint8_t in, uint16_t
len);

/**
 * @brief Setup header for BLUEZ (hcidump) format
 * @param buffer
 * @param tv_sec
 * @param tv_us
 * @param packet_type
 * @param in
 * @param len
 */
void hci_dump_setup_header_bluez(uint8_t * buffer, uint32_t
tv_sec, uint32_t tv_us, uint8_t packet_type, uint8_t in, uint16_t
```

```
len);

/**
 * @brief Setup header for BT Snoop format
 * @param buffer
 * @param ts_usec_high upper 32-bit of 64-bit microsecond
timestamp
 * @param ts_usec_low  lower 2-bit of 64-bit microsecond timestamp
 * @param cumulative_drops since last packet was recorded
 * @param packet_type
 * @param in
 * @param len
 */
void hci_dump_setup_header_btsnoop(uint8_t * buffer, uint32_t
ts_usec_high, uint32_t ts_usec_low, uint32_t cumulative_drops,
uint8_t packet_type, uint8_t in, uint16_t len);
```

HCI Transport API

hci_transport.h : The API allows BTstack to use different transport interfaces.

```
/* 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_LINUX,
} 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;
    int        parity;                // see btstack_uart.h
BTSTACK_UART_PARITY
} hci_transport_config_uart_t;

typedef struct {
    hci_transport_config_type_t type; // ==
HCI_TRANSPORT_CONFIG_LINUX
    int device_id;                    // e.g. 0 for HCI0
} hci_transport_config_linux_t;
```

HCI Transport EM9304 API API

hci_transport_em9304_spi.h : The EM9304 uses an extended SPI interface and this HCI Transport is based on the the btstack_em9304.h interface.

```
/*
 * @brief Setup H4 over SPI instance for EM9304 with
em9304_spi_driver
```

```
 * @param em9304_spi_driver to use
 */
const hci_transport_t * hci_transport_em9304_spi_instance(const
btstack_em9304_spi_t * em9304_spi_driver);
```

HCI Transport H4 API

hci_transport_h4.h

```
/*
 * @brief Setup H4 instance with btstack_uart implementation
 * @param btstack_uart_block_driver to use
 */
const hci_transport_t * hci_transport_h4_instance_for_uart(const
btstack_uart_t * uart_driver);

/*
 * @brief Setup H4 instance with btstack_uart_block implementation
 * @param btstack_uart_block_driver to use
 * @deprecated use hci_transport_h4_instance_for_uart instead
 */
const hci_transport_t * hci_transport_h4_instance(const
btstack_uart_block_t * uart_driver);
```

HCI Transport H5 API

hci_transport_h5.h

```
/*
 * @brief Setup H5 instance with btstack_uart implementation that
supports SLIP frames
 * @param uart_driver to use
 */
const hci_transport_t * hci_transport_h5_instance(const
btstack_uart_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
 * @deprecated Parity can be enabled in UART driver configuration
 */
void hci_transport_h5_enable_bcsp_mode(void);
```

HCI Transport USB API

hci_transport_usb.h

```
/*
 * @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);

/**
 * @brief Specify USB Bluetooth device via bus and port numbers
from hub to device
 */
void hci_transport_usb_set_bus_and_path(uint8_t bus, int len,
uint8_t* port_numbers);

/**
 * @brief Add device to list of known Bluetooth USB Controller
 * @param vendor_id
 * @param product_id
 */
void hci_transport_usb_add_device(uint16_t vendor_id, uint16_t
product_id);
```

L2CAP API

l2cap.h : Logical Link Control and Adaption Protocol

```
//
// PSM numbers from https://www.bluetooth.com/specifications/
assigned-numbers/logical-link-control
//
#define PSM_SDP           BLUETOOTH_PROTOCOL_SDP
#define PSM_RFCOMM        BLUETOOTH_PROTOCOL_RFCOMM
#define PSM_BNEP          BLUETOOTH_PROTOCOL_BNEP
// @TODO: scrape PSMs Bluetooth SIG site and put in
bluetooth_psm.h or bluetooth_l2cap.h
#define PSM_HID_CONTROL   0x11
#define PSM_HID_INTERRUPT 0x13
#define PSM_ATT           0x1f
#define PSM_IPSP          0x23

/**
 * @brief Set up L2CAP and register L2CAP with HCI layer.
 */
void l2cap_init(void);

/**
 * @brief Add event packet handler for LE Connection Parameter
Update events
 */
void
l2cap_add_event_handler(btstack_packet_callback_registration_t *
callback_handler);

/**
 * @brief Remove event packet handler.
 */
void
l2cap_remove_event_handler(btstack_packet_callback_registration_t
* callback_handler);

/**
 * @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 Set the max MTU for LE connections, if not set
l2cap_max_mtu() will be used.
*/
void l2cap_set_max_le_mtu(uint16_t max_mtu);

/**
 * @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 Disconnects L2CAP channel with given identifier.
 * @param local_cid
 * @return status ERROR_CODE_SUCCESS if successful or
L2CAP_LOCAL_CID_DOES_NOT_EXIST
 */
uint8_t l2cap_disconnect(uint16_t local_cid);

/**
```

```
 * @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.
 * @note For channel in credit-based flow control mode, data needs
to stay valid until .. event
 * @param local_cid
 * @param data to send
 * @param len of data
 * @return status
 */
uint8_t l2cap_send(uint16_t local_cid, const uint8_t *data,
uint16_t len);

/**
 * @brief Registers L2CAP service with given PSM and MTU, and
assigns a packet handler.
 * @param packet_handler
 * @param psm
 * @param mtu
 * @param security_level
 * @return status ERROR_CODE_SUCCESS if successful, otherwise
L2CAP_SERVICE_ALREADY_REGISTERED or BTSTACK_MEMORY_ALLOC_FAILED
 */
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 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
 * @return true if packet can be sent
 */
bool 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
 * @return status
 */
uint8_t l2cap_request_can_send_now_event(uint16_t local_cid);

/**
 * @brief Reserve outgoing buffer
 * @note Only for L2CAP Basic Mode Channels
 * @note Must only be called after a 'can send now' check or event
 * @note Asserts if packet buffer is already reserved
 */
void l2cap_reserve_packet_buffer(void);

/**
 * @brief Get outgoing buffer and prepare data.
 * @note Only for L2CAP Basic Mode Channels
 */
uint8_t *l2cap_get_outgoing_buffer(void);

/**
```

```
 * @brief Send L2CAP packet prepared in outgoing buffer to channel
 * @note Only for L2CAP Basic Mode Channels
 */
uint8_t l2cap_send_prepared(uint16_t local_cid, uint16_t len);

/**
 * @brief Release outgoing buffer (only needed if
l2cap_send_prepared is not called)
 * @note Only for L2CAP Basic Mode Channels
 */
void l2cap_release_packet_buffer(void);

//
// Connection-Oriented Channels in Enhanced Retransmission Mode -
ERTM
//

/**
 * @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_ertm_create_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 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_ertm_accept_connection(uint16_t local_cid,
l2cap_ertm_config_t * ertm_contig, uint8_t * buffer, uint32_t
size);

/**
 * @brief Deny incoming incoming L2CAP connection for Enhanced
Retransmission Mode
 * @param local_cid
 * @return status
 */
uint8_t l2cap_ertm_decline_connection(uint16_t local_cid);

/**
 * @brief ERTM Set channel as busy.
 * @note Can be cleared by l2cap_ertm_set_ready
 * @param local_cid
 * @return status
 */
uint8_t l2cap_ertm_set_busy(uint16_t local_cid);

/**
 * @brief ERTM Set channel as ready
 * @note Used after l2cap_ertm_set_busy
 * @param local_cid
 * @return status
 */
uint8_t l2cap_ertm_set_ready(uint16_t local_cid);


//
```

```
// L2CAP Connection-Oriented Channels in LE Credit-Based Flow-
Control Mode - CBM
//

/**
 * @brief Register L2CAP service in LE Credit-Based Flow-Control
Mode
 * @note MTU and initial credits are specified in
l2cap_cbm_accept_connection(..) call
 * @param packet_handler
 * @param psm
 * @param security_level
 */
uint8_t l2cap_cbm_register_service(btstack_packet_handler_t
packet_handler, uint16_t psm, gap_security_level_t
security_level);

/**
 * @brief Unregister L2CAP service in LE Credit-Based Flow-Control
Mode
 * @param psm
 */

uint8_t l2cap_cbm_unregister_service(uint16_t psm);

/*
 * @brief Accept incoming connection LE Credit-Based Flow-Control
Mode
 * @param local_cid             L2CAP 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_cbm_accept_connection(uint16_t local_cid, uint8_t *
receive_sdu_buffer, uint16_t mtu, uint16_t initial_credits);
```

```
/**
 * @brief Deecline connection in LE Credit-Based Flow-Control Mode
 * @param local_cid             L2CAP Channel Identifier
 * @param result                result, see
L2CAP_CBM_CONNECTION_RESULT_SUCCESS in bluetooth.h
 */

uint8_t l2cap_cbm_decline_connection(uint16_t local_cid, uint16_t
result);

/**
 * @brief Create outgoing channel in LE Credit-Based Flow-Control
Mode
 * @param packet_handler        Packet handler for this connection
 * @param con_handle            HCI Connection Handle, LE
transport
 * @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_cbm_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 credits for channel in LE Credit-Based Flow-
Control Mode
 * @param local_cid             L2CAP Channel Identifier
 * @param credits               Number additional credits for peer
 */
uint8_t l2cap_cbm_provide_credits(uint16_t local_cid, uint16_t
```

```
credits);

/**
 * @brief Returns the number of credits provided by peer
 * @param local_cid
 * @return number of credits
 */
uint16_t l2cap_cbm_available_credits(uint16_t local_cid);

//
// L2CAP Connection-Oriented Channels in Enhanced Credit-Based
Flow-Control Mode - ECBM
//

/**
 * @brief Register L2CAP service in Enhanced Credit-Based Flow-
Control Mode
 * @note MTU and initial credits are specified in
l2cap_enhanced_accept_connection(..) call
 * @param packet_handler
 * @param psm
 * @param min_remote_mtu
 * @param security_level
 * @oaram authorization_required
 * @return status
 */
uint8_t l2cap_ecbm_register_service(btstack_packet_handler_t
packet_handler, uint16_t psm, uint16_t min_remote_mtu,
                                    gap_security_level_t
security_level, bool authorization_required);

/**
 * @brief Unregister L2CAP service in Enhanced Credit-Based Flow-
Control Mode
 * @param psm
 * @return status
 */

uint8_t l2cap_ecbm_unregister_service(uint16_t psm);
```

```
/**
 * @brief Set Minimal MPS for channel in Enhanced Credit-Based
Flow-Control Mode
 * @param mps_min
 */
void l2cap_ecbm_mps_set_min(uint16_t mps_min);

/**
 * @brief Set Minimal MPS for channel in Enhanced Credit-Based
Flow-Control Mode
 * @param mps_max
 */
void l2cap_ecbm_mps_set_max(uint16_t mps_max);

/**
 * @brief Create outgoing channel in Enhanced Credit-Based Flow-
Control Mode
 * @note receive_buffer points to an array of receive buffers with
num_channels elements
 * @note out_local_cid points to an array where CID is stored with
num_channel elements
 * @param packet_handler        Packet handler for this connection
 * @param con_handle            HCI Connection Handle
 * @param security_level        Minimum required security level
 * @param psm                   Service PSM to connect to
 * @param num_channels          number of channels to create
 * @param initial_credits       Number of initial credits provided
to peer per channel or L2CAP_LE_AUTOMATIC_CREDITS to enable
automatic credits
 * @param receive_buffer_size   buffer size equals MTU
 * @param receive_buffers       Array of buffers used for
reassembly of L2CAP Information Frames into service data unit
(SDU) with given MTU
 * @param out_local_cids        Array of L2CAP Channel Identifiers
is stored here on success
 * @return status
 */
uint8_t l2cap_ecbm_create_channels(btstack_packet_handler_t
packet_handler, hci_con_handle_t con_handle,
                                   gap_security_level_t
```

```
security_level,
                                        uint16_t psm, uint8_t
num_channels, uint16_t initial_credits, uint16_t
receive_buffer_size,
                                        uint8_t ** receive_buffers,
uint16_t * out_local_cids);

/**
 * @brief  Accept incoming connection Enhanced Credit-Based Flow-
Control Mode
 * @param local_cid            from
L2CAP_EVENT_INCOMING_DATA_CONNECTION
 * @param num_channels
 * @param initial_credits      Number of initial credits provided
to peer per channel or L2CAP_LE_AUTOMATIC_CREDITS to enable
automatic credits
 * @param receive_buffer_size
 * @param receive_buffers      Array of buffers used for
reassembly of L2CAP Information Frames into service data unit
(SDU) with given MTU
 * @param out_local_cids       Array of L2CAP Channel Identifiers
is stored here on success
 * @return status
 */
uint8_t l2cap_ecbm_accept_channels(uint16_t local_cid, uint8_t
num_channels, uint16_t initial_credits,
                                           uint16_t
receive_buffer_size, uint8_t ** receive_buffers, uint16_t *
out_local_cids);
/**
 * @brief Decline connection in Enhanced Credit-Based Flow-Control
Mode
 * @param local_cid           from
L2CAP_EVENT_INCOMING_DATA_CONNECTION
 * @param result              See
L2CAP_ECBM_CONNECTION_RESULT_ALL_SUCCESS in bluetooth.h
 * @return status
 */
uint8_t l2cap_ecbm_decline_channels(uint16_t local_cid, uint16_t
result);
```

```
/**
 * @brief Provide credits for channel in Enhanced Credit-Based
Flow-Control Mode
 * @param local_cid             L2CAP Channel Identifier
 * @param credits               Number additional credits for peer
 * @return status
 */
uint8_t l2cap_ecbm_provide_credits(uint16_t local_cid, uint16_t
credits);

/**
 * @brief Request emission of L2CAP_EVENT_ECBM_CAN_SEND_NOW as
soon as possible
 * @note L2CAP_EVENT_ECBM_CAN_SEND_NOW might be emitted during
call to this function
 *       so packet handler should be ready to handle it
 * @param local_cid             L2CAP Channel Identifier
 * @return status
 */
uint8_t l2cap_ecbm_request_can_send_now_event(uint16_t local_cid);

/**
 * @brief Reconfigure MPS/MTU of local channels
 * @param num_cids
 * @param local_cids            array of local_cids to reconfigure
 * @param receive_buffer_size   buffer size equals MTU
 * @param receive_buffers       Array of buffers used for
reassembly of L2CAP Information Frames into service data unit
(SDU) with given MTU
 * @return status
 */
uint8_t l2cap_ecbm_reconfigure_channels(uint8_t num_cids, uint16_t
* local_cids, int16_t receive_buffer_size, uint8_t **
receive_buffers);

/**
 * @brief Trigger pending connection responses after pairing
completed
 * @note Must be called after receiving an SM_PAIRING_COMPLETE
```

```
event, will be removed eventually
 * @param con_handle
 */
void
l2cap_ecbm_trigger_pending_connection_responses(hci_con_handle_t
con_handle);

/**
 * @brief Returns the number of outgoing credits provided by peer
 * @param local_cid
 * @return number of credits
 */
uint16_t l2cap_ecbm_available_credits(uint16_t local_cid);

/**
 * @brief De-Init L2CAP
 */
void l2cap_deinit(void);
```

Audio Stream Control Service Client API

broadcast_audio_scan_service_client.h

```
typedef enum {
    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_IDLE = 0,
    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W2_QUERY_SERVICE,
    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W4_SERVICE_RESULT,

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W2_QUERY_CHARACTERISTICS,

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W4_CHARACTERISTIC_RESULT,


BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W2_QUERY_CHARACTERISTIC_D

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W4_CHARACTERISTIC_DESCRIF


BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W2_REGISTER_NOTIFICATION,
```

```
BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_W4_NOTIFICATION_REGISTERE
    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_STATE_READY,

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_READ_CHARACTERISTIC_CONFIGUR

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_CHARACTERISTIC_CONFIGURATION

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_START_SC

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_START_SC

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_STOP_SCA

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_STOP_SCA

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_ADD_SOUR

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_ADD_SOUR

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_MODIFY_S

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_MODIFY_S

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_SET_BROA

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_SET_BROA

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_WRITE_CONTROL_POINT_REMOVE_S

BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_WRITE_CONTROL_POINT_REMOVE_S

    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W2_READE_RECEIVE_STATE,
    BROADCAST_AUDIO_SCAN_SERVICE_CLIENT_W4_READE_RECEIVE_STATE,
```

```
} broadcast_audio_scan_service_client_state_t;

typedef struct {
    // used for add source command
    bass_source_data_t data;

    // received via notification
    bool    in_use;
    uint8_t source_id;
    le_audio_big_encryption_t big_encryption;
    uint8_t  bad_code[16];

    // characteristic
    uint16_t receive_state_value_handle;
    uint16_t receive_state_ccc_handle;
    uint16_t receive_state_properties;
    uint16_t receive_state_end_handle;
} bass_client_source_t;

typedef struct {
    btstack_linked_item_t item;

    hci_con_handle_t  con_handle;
    uint16_t          cid;
    uint16_t          mtu;
    broadcast_audio_scan_service_client_state_t  state;

    // service
    uint16_t start_handle;
    uint16_t end_handle;
    uint16_t control_point_value_handle;

    // used for memory capacity checking
    uint8_t  service_instances_num;
    uint8_t  receive_states_instances_num;
    // used for notification registration
    uint8_t  receive_states_index;

    uint8_t max_receive_states_num;
```

```
    bass_client_source_t * receive_states;

    // used for write segmentation
    uint8_t  buffer[BASS_CLIENT_MAX_ATT_BUFFER_SIZE];
    uint16_t buffer_offset;
    uint16_t data_size;

    gatt_client_notification_t notification_listener;

    // used for adding and modifying source
    const bass_source_data_t * control_point_operation_data;
    uint8_t control_point_operation_source_id;
    // used for setting the broadcast code
    const uint8_t * broadcast_code;
} bass_client_connection_t;

/**
 * @brief Init Broadcast Audio Scan Service Client. Register
packet handler to receive events:
 * - LEAUDIO_SUBEVENT_BASS_CLIENT_CONNECTED
 * - LEAUDIO_SUBEVENT_BASS_CLIENT_DISCONNECTED
 * - LEAUDIO_SUBEVENT_BASS_CLIENT_SCAN_OPERATION_COMPLETE
 * - LEAUDIO_SUBEVENT_BASS_CLIENT_SOURCE_OPERATION_COMPLETE
 * - LEAUDIO_SUBEVENT_BASS_CLIENT_NOTIFICATION_COMPLETE
 * @param packet_handler for events
 */
void
broadcast_audio_scan_service_client_init(btstack_packet_handler_t
packet_handler);

/**
 * @brief Connect to BASS Service on remote device
 * @note LEAUDIO_SUBEVENT_BASS_CLIENT_CONNECTED will be emitted
 * @param connection struct provided by user, needs to stay valid
until disconnect event is received
 * @param sources buffer to store information on Broadcast Sources
on the service
 * @param num_sources
 * @param con_handle to connect to
 * @param bass_cid connection id for this connection for other
```

```
functions
 * @return status
 */
uint8_t
broadcast_audio_scan_service_client_connect(bass_client_connection_
* connection, bass_client_source_t * sources, uint8_t num_sources,
hci_con_handle_t con_handle, uint16_t * bass_cid);

/**
 * @brief Notify BASS Service that scanning has started
 * @param bass_cid
 * @return status
 */
uint8_t
broadcast_audio_scan_service_client_scanning_started(uint16_t
bass_cid);

/**
 * @brief Notify BASS Service that scanning has stopped
 * @note emits
LEAUDIO_SUBEVENT_BASS_CLIENT_SOURCE_OPERATION_COMPLETE
 * @param bass_cid
 * @return status
 */
uint8_t
broadcast_audio_scan_service_client_scanning_stopped(uint16_t
bass_cid);

/**
 * @brief Add Broadcast Source on service
 * @note LEAUDIO_SUBEVENT_BASS_NOTIFICATION_COMPLETE will contain
source_id for other functions
 * @param bass_cid
 * @param add_source_data data to add, needs to stay valid until
LEAUDIO_SUBEVENT_BASS_CLIENT_SOURCE_OPERATION_COMPLETE
 * @return status
 */
uint8_t broadcast_audio_scan_service_client_add_source(uint16_t
bass_cid, const bass_source_data_t * add_source_data);
```

```
/**
 * @brief Modify information about Broadcast Source on service
 * @param bass_cid
 * @param source_id
 * @param modify_source_data data to modify, needs to stay valid
until LEAUDIO_SUBEVENT_BASS_CLIENT_SOURCE_OPERATION_COMPLETE
 * @return status
 */
uint8_t broadcast_audio_scan_service_client_modify_source(uint16_t
bass_cid, uint8_t source_id, const bass_source_data_t *
modify_source_data);

/**
 * @brief Set Broadcast Code for a Broadcast Source to allow
remote do decrypt audio stream
 * @param bass_cid
 * @param source_id
 * @param broadcast_code
 * @return status
 */
uint8_t
broadcast_audio_scan_service_client_set_broadcast_code(uint16_t
bass_cid, uint8_t source_id, const uint8_t * broadcast_code);

/**
 * @brief Remove information about Broadcast Source
 * @param bass_cid
 * @param source_id
 * @return status
 */
uint8_t broadcast_audio_scan_service_client_remove_source(uint16_t
bass_cid, uint8_t source_id);

/**
 * @param Provide read-only access to Broadcast Receive State of
given Broadcast Source on service
 * @param bass_cid
 * @param source_id
 * @return pointer to source data or NULL, if source_id not found
 */
```

```
const bass_source_data_t *
broadcast_audio_scan_service_client_get_source_data(uint16_t
bass_cid, uint8_t source_id);

/**
 * @param Get BIG Encryption and Bad Code from Broadcast Receive
State of given Broadcast Source on service
 * @param bass_cid
 * @param source_id
 * @param out_big_encryption
 * @param out_bad_code 16-byte buffer
 * @return status
 */
uint8_t
broadcast_audio_scan_service_client_get_encryption_state(uint16_t
bass_cid, uint8_t source_id,

le_audio_big_encryption_t * out_big_encryption, uint8_t *
out_bad_code);

/**
 * @brief Deinit Broadcast Audio Scan Service Client
 */
void broadcast_audio_scan_service_client_deinit(uint16_t
bass_cid);
```

Broadcast Audio Scan Service Server (BASS) API

broadcast_audio_scan_service_server.h : @text The Broadcast Audio Scan Service is used by servers to expose their status with respect to synchronization to broadcast Audio Streams and associated data, including Broadcast_Codes used to decrypt encrypted broadcast Audio Streams. Clients can use the attributes exposed by servers to observe and/or request changes in server behavior.

To use with your application, add `#import <broadcast_audio_scan_service.gatt>` to your .gatt file.

```
// memory for list of these structs is allocated by the
application
typedef struct {
    // assigned by client via control point
    bass_source_data_t data;

    uint8_t  update_counter;
    uint8_t  source_id;
    bool     in_use;

    le_audio_big_encryption_t big_encryption;
    uint8_t  bad_code[16];

    uint16_t bass_receive_state_handle;
    uint16_t bass_receive_state_client_configuration_handle;
    uint16_t bass_receive_state_client_configuration;
} bass_server_source_t;

typedef struct {
    hci_con_handle_t con_handle;
    uint16_t sources_to_notify;

    // used for caching long write
    uint8_t  long_write_buffer[512];
    uint16_t long_write_value_size;
    uint16_t long_write_attribute_handle;
} bass_server_connection_t;

/**
 * @brief Init Broadcast Audio Scan Service Server with ATT DB
 * @param sources_num
 * @param sources
 * @param clients_num
 * @param clients
 */
void broadcast_audio_scan_service_server_init(uint8_t const
sources_num, bass_server_source_t * sources, uint8_t const
clients_num, bass_server_connection_t * clients);

/**
```

```
 * @brief Register packet handler to receive events:
 * - LEAUDIO_SUBEVENT_BASS_SERVER_SCAN_STOPPED
 * - LEAUDIO_SUBEVENT_BASS_SERVER_SCAN_STARTED
 * - LEAUDIO_SUBEVENT_BASS_SERVER_BROADCAST_CODE
 * - LEAUDIO_SUBEVENT_BASS_SERVER_SOURCE_ADDED
 * - LEAUDIO_SUBEVENT_BASS_SERVER_SOURCE_MODIFIED
 * - LEAUDIO_SUBEVENT_BASS_SERVER_SOURCE_DELETED
 * @param packet_handler
 */
void
broadcast_audio_scan_service_server_register_packet_handler(btstack
packet_handler);

/**
 * @brief Set PA state of source.
 * @param source_index
 * @param sync_state
 */
void broadcast_audio_scan_service_server_set_pa_sync_state(uint8_t
source_index, le_audio_pa_sync_state_t sync_state);

/**
 * @brief Add source.
 * @param source_data
 * @param source_index
 */
void broadcast_audio_scan_service_server_add_source(const
bass_source_data_t *source_data, uint8_t * source_index);

/**
 * @brief Deinit Broadcast Audio Scan Service Server
 */
void broadcast_audio_scan_service_server_deinit(void);
```

Volume Offset Control Service Server API

le_audio.h

Broadcast Audio Source Endpoint AD Builder API

le_audio_base_builder.h

Broadcast Audio Source Endpoint AD Parser API

le_audio_base_parser.h

LE Audio Util API

le_audio_util.h

Mesh Provisioning Service Server API

mesh_provisioning_service_server.h

```
/**
 * @brief Init Mesh Provisioning Service Server with ATT DB
 */
void mesh_provisioning_service_server_init(void);

/**
 * @brief Send a Proxy PDU message containing Provisioning PDU
from a Provisioning Server to a Provisioning Client.
 * @param con_handle
 * @param proxy_pdu
 * @param proxy_pdu_size max lenght MESH_PROV_MAX_PROXY_PDU
 */
void mesh_provisioning_service_server_send_proxy_pdu(uint16_t
con_handle, const uint8_t * proxy_pdu, uint16_t proxy_pdu_size);

/**
 * @brief Register callback for the PB-GATT.
 * @param callback
 */
```

```
void
mesh_provisioning_service_server_register_packet_handler(btstack_pa
callback);

/**
 * @brief Request can send now event to send PDU
 * Generates an MESH_SUBEVENT_CAN_SEND_NOW subevent
 * @param con_handle
 */
void
mesh_provisioning_service_server_request_can_send_now(hci_con_handl
con_handle);
```

Mesh Proxy Service Server API

mesh_proxy_service_server.h

```
/**
 * @brief Init Mesh Proxy Service Server with ATT DB
 */
void mesh_proxy_service_server_init(void);

/**
 * @brief Send a Proxy PDU message containing proxy PDU from a
proxy Server to a proxy Client.
 * @param con_handle
 * @param proxy_pdu
 * @param proxy_pdu_size max lenght MESH_PROV_MAX_PROXY_PDU
 */
void mesh_proxy_service_server_send_proxy_pdu(uint16_t con_handle,
const uint8_t * proxy_pdu, uint16_t proxy_pdu_size);

/**
 * @brief Register callback for the PB-GATT.
 * @param callback
 */
void
mesh_proxy_service_server_register_packet_handler(btstack_packet_ha
callback);
```

```
/**
 * @brief Request can send now event to send PDU
 * Generates an MESH_SUBEVENT_CAN_SEND_NOW subevent
 * @param con_handle
 */
void
mesh_proxy_service_server_request_can_send_now(hci_con_handle_t
con_handle);
```

#Events and Errors

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 Please note that the guarantee that a packet can be sent is only valid when the event is received. After returning from the packet handler, BTstack might need to send itself.

Event	Event Code
L2CAP_EVENT_CHANNEL_OPENED	0x70

Event	Event Code
L2CAP_EVENT_CHANNEL_CLOSED	0x71
L2CAP_EVENT_INCOMING_CONNECTION	0x72
L2CAP_EVENT_CAN_SEND_NOW	0x78

Table: 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)

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 Please note that the guarantee that a packet can be sent is only valid when the event is received. After returning from the packet handler, BTstack might need to send itself.

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: 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)

Errors

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

Error	Error Code
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
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: Errors.

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.

Changes

Repository structure

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.

Plural folder names

Folder with plural names have been renamed to the singular form, examples: *doc*, *chipset*, *platform*, *tool*.

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*.

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*.

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.

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.

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.

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.

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*.

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.

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.

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.

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.

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.

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*.

bt_control

Has been split into *src/btstack_chipset.h* and *src/bstack_control.h*

HCI / GAP

HCI functions that are commonly placed in GAP have been moved from *src/hci.h* to *src/gap.h*

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.

SPP Server

The function to create an SPP SDP record has been moved into *spp_server.h*

SDP Client

- SDP Query structs have been replaced by HCI events. You can use btstack_event.h to access the fields.

SDP Server

- Has been renamed to *src/classic/sdp_server.h*.
- The distinction if the API is for embedded or not has been removed.

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.

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

ANCS Client

Renamed to *src/ble/ancs_client*

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_EVENT_CAN_SEND_NOW 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.

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*

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).

Requirements

- bash
- sed
- Coccinelle. On Debian-based distributions, it's available via apt. On OS X, it can be installed via Homebrew.

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.

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.