April 27, 2015

BTstack Manual

Including Quickstart Guide

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

Contents

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. The Revision History is shown in the Appendix N on page 100.

1. Quick Start

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

On Windows, you need to manually install and configure GNU Make, Python, and optionally git :

- GNU Make[1] for Windows: Add its bin folder to the Windows Path in Environment Variables. The bin folder is where make.exe resides, and it's usually located in `C:\ProgramFiles\GnuWin32\bin`.
- Python[2] for Windows: Add Python installation folder to the Windows Path in Environment Variables.

Adding paths to the Windows Path variable:

- Go to: Control Panel→System→Advanced tab→Environment Variables.
- The top part contains a list of User variables.
- Click on the Path variable and then click edit.
- Go to the end of the line, then append the path to the list., for example, `C:\ProgramFiles\GnuWin32\bin` for GNU Make.
- Ensure that there is a semicolon before and after `C:\ProgramFiles\GnuWin32\bin`.

1.2. **Getting BTstack from GitHub.** Use git to clone the latest version:

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

Alternatively, you can download it as a ZIP archive from BTstack's page[3]on GitHub.

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

[1]http://gnuwin32.sourceforge.net/packages/make.htm
[2]http://www.python.org/getit/
[3]https://github.com/bluekitchen/btstack/archive/master.zip

TABLE 1. Overview of platform specific toolchains, programmers, and used chipsets.

Platform	Chipset	Toolchain	Programmer
ez430-rf2560, msp-exp430f5438, msp430f5229lp	CC256x	mspgcc[a]	MSP430Flasher[b], MSPDebug[c]
stm32-f103rb-nucleo	CC256x	arm-gcc[d]	OpenOCD[e]
pic32-harmony	CSR8811	MPLAB XC[f]	PICkit 3[g]
libusb on Linux/OS X	any	any	N/A

[a]http://sourceforge.net/projects/mspgcc/files/Windows/mingw32/
[b]http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer
[c]http://mspdebug.sourceforge.net/
[d]https://launchpad.net/gcc-arm-embedded
[e]http://openocd.org
[f]http://www.microchip.com/pagehandler/en_us/devtools/mplabxc/
[g]http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=pg164130

1.4. **Run the Example.** As a first test, we recommend the SPP Counter example (see Section 6.5). During the startup, for TI chipsets, the init script is transferred, and the Bluetooth stack brought up. After that, the development board is discoverable as "BTstack SPP Counter" and provides a single virtual serial port. When you connect to it, you'll receive a counter value as text every second.

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

1.5.1. *libusb.*

The quickest way to try BTstack is on a Linux or OS X system with an additional USB Bluetooth module. The Makefile in `platforms/libusb` requires pkg-config[4] and libusb[5]-1.0 or higher to be installed.

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

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

```
sudo nvram bluetoothHostControllerSwitchBehavior=never
```

It's also possible to run the examples on Win32 systems. For this:

- Install MSYS[6] and MINGW32[7] using the MINGW installer

[4]http://www.freedesktop.org/wiki/Software/pkg-config/
[5]www.libusb.org
[6]www.mingw.org/wiki/msys
[7]www.mingw.org

- Compile and install libusb-1.0.19 to /usr/local/ in msys command shell
- Setup a USB Bluetooth dongle for use with libusb-1.0:
  - Start Zadig[8]
  - Select Options → "List all devices"
  - Select USB Bluetooth dongle in the big pull down list
  - Select WinUSB (libusb) in the right pull pull down list
  - Select "Replace Driver"

Now, you can run the examples from the *msys* shell the same way as on Linux/OS X.

1.5.2. *Texas Instruments MSP430-based boards.*

Compiler Setup. The MSP430 port of BTstack is developed using the Long Term Support (LTS) version of mspgcc. General information about it and installation instructions are provided on the MSPGCC Wiki[9]. On Windows, you need to download and extract mspgcc[10] to `C:\mspgcc`. Add `C:\mspgcc\bin` folder to the Windows Path in Environment variable as explained in Section 1.1.

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

- MSP430Flasher[11] (windows-only):
  - Use the following command, where you need to replace the `BINARY_FILE_NAME.hex` with the name of your application:

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

- MSPDebug[12]: An example session with the MSP-FET430UIF connected on OS X is given in following listing:

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

1.5.3. *Texas Instruments CC256x-based chipsets.*

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

[8] http://zadig.akeo.ie
[9] http://sourceforge.net/apps/mediawiki/mspgcc/index.php?title=MSPGCC_Wiki
[10] http://sourceforge.net/projects/mspgcc/files/Windows/mingw32/
[11] http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer
[12] http://mspdebug.sourceforge.net/

- Download the BTS file[13] for your CC256x-based module.
- Copy the included .bts file into `btstack/chipset-cc256x`
- In `chipset-cc256x`, run the Python script: *./convert_bts_init_scripts.py*

The common code for all CC256x chipsets is provided by *bt_control_cc256x.c*. During the setup, *bt_control_cc256x_instance* function is used to get a *bt_control_t* instance and passed to *hci_init* function.

Note: Depending on the CC256x-based module you're using, you'll need to update the reference `bluetooth_init_cc256...` in the Makefile to match the downloaded file.

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

1.5.4. *MSP-EXP430F5438 + CC256x Platform.*

Hardware Setup. We assume that a PAN1315, PAN1317, or PAN1323 module is plugged into RF1 and RF2 of the MSP-EXP430F5438 board and the "RF3 Adapter board" is used or at least simulated. See User Guide[14].

1.5.5. *STM32F103RB Nucleo + CC256x Platform.*

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

1.5.6. *PIC32 Bluetooth Audio Development Kit.*

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

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

2. 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.

[13] http://processors.wiki.ti.com/index.php/CC256x_Downloads
[14] http://processors.wiki.ti.com/index.php/PAN1315EMK_User_Guide#RF3_Connector

- *No blocking anywhere* - If Bluetooth processing is required, its result will be delivered as an event via registered packet handlers.
- *No artificially limited buffers/pools* - Incoming and outgoing data packets are not queued.
- *Statically bounded memory (optionally)* - The number of maximum connections/channels/services can be configured.

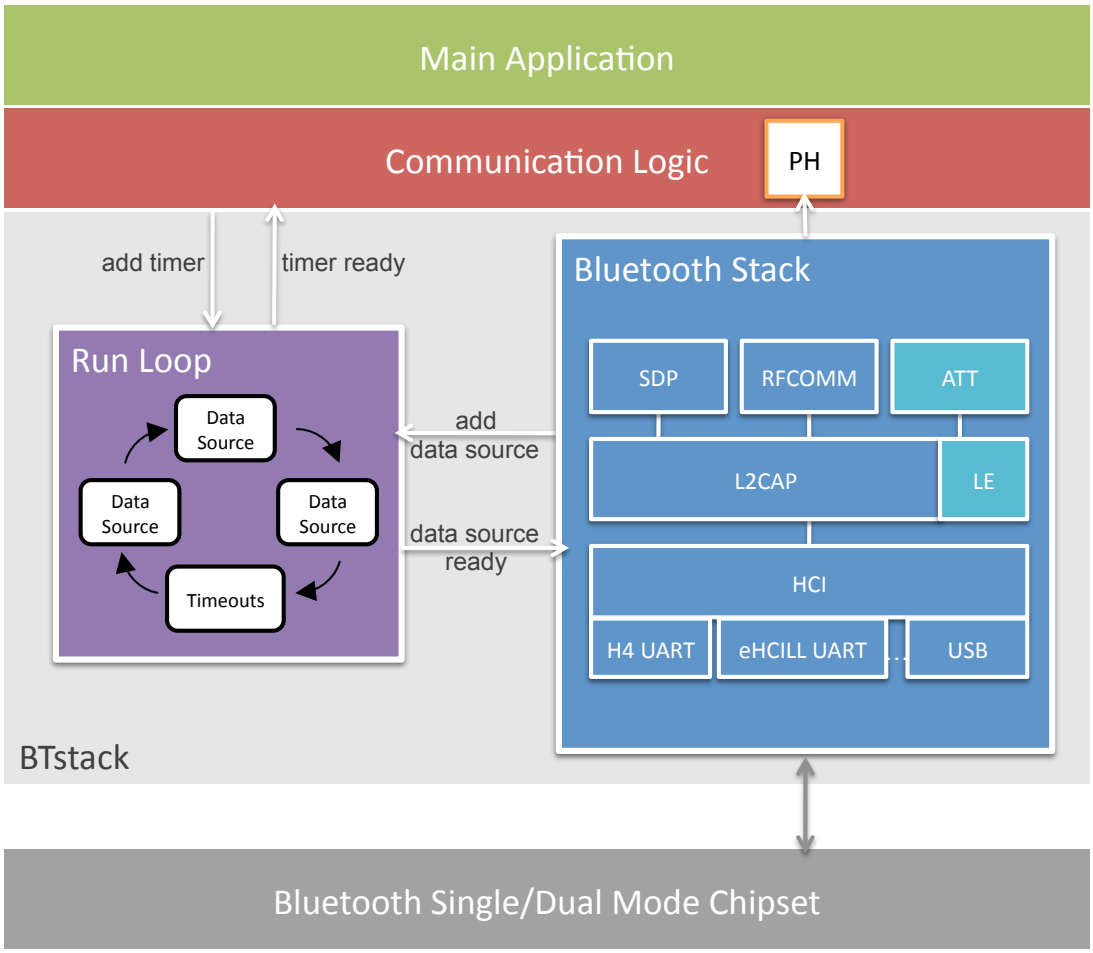

FIGURE 1. BTstack-based single-threaded application. The Main Application contains the application logic, e.g., reading a sensor value and providing it via the Communication Logic as a SPP Server. The Communication Logic is often modeled as a finite state machine with events and data coming from either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

Figure 1 shows the general architecture of a BTstack-based application that includes the BTstack run loop.

2.1. **Single threaded design.** BTstack does not use or require multi-threading. It uses a single run loop to handle data sources and timers. Data sources represent

communication interfaces like an UART or an USB driver. Timers are used by BTstack to implement various Bluetooth-related timeouts. For example, to disconnect a Bluetooth baseband channel without an active L2CAP channel after 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, see Section 8.2.

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

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

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

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

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

3. How to use BTstack

BTstack implements a set of basic Bluetooth protocols. To make use of these to connect to other devices or to provide own services, BTstack has to be properly configured during application startup.

In the following, we provide an overview of the memory management, the run loop, and services that are 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 over packet handlers, of which an overview is provided here. Finally, we describe the RFCOMM credit-based flow-control, which may be necessary for

resource-constraint devices. Complete examples for the MSP430 platforms will be presented in Chapter 6.

3.1. **Memory configuration.** 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 config file. To initialize the static pools, you need to call *btstack_memory_init* function. An example of memory configuration for a single SPP service with a minimal L2CAP MTU is shown in Listing 2.
- dynamically using the *malloc/free* functions, if HAVE_MALLOC is defined in config file.

If both HAVE_MALLOC and maximal size of a pool are defined in the config file, the statical allocation will take precedence. In case that both are omitted, an error will be raised.

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

```
btstack_memory_init();
```

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

Data sources and timers are represented by the *data_source_t* and *timer_source_t* structs respectively. Each of these structs contain 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.

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 *embedded_trigger* function. The call to *embedded_trigger* sets an internal flag that is checked in the critical section just before entering sleep mode.

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 1. Note that BTstack expects to get called periodically to keep its time, see Section 7.1 for more on the tick hardware abstraction.

```
#define TIMER_PERIOD_MS 1000
timer_source_t periodic_timer;

void register_timer(timer_source_t *timer, uint16_t period){
    run_loop_set_timer(timer, period);
    run_loop_add_timer(timer);
}

void timer_handler(timer_source_t *ts){
    // do something,
    ... e.g., increase counter,

    // then re-register timer
    register_timer(ts, TIMER_PERIOD_MS);
}

void timer_setup(){
    // set one-shot timer
    run_loop_set_timer_handler(&periodic_timer, &timer_handler);
    register_timer(&periodic_timer, TIMER_PERIOD_MS);
}
```

Listing 1. Periodic counter

The Run loop API is provided in Appendix A. To enable the use of timers, make sure that you defined HAVE_TICK in the config file.

In your code, you'll have to configure the run loop before you start it as shown in Listing 21. The application can register data sources as well as timers, e.g., periodical sampling of sensors, or communication over the UART.

The run loop is set up by calling *run_loop_init* function for embedded systems:

```
run_loop_init(RUN_LOOP_EMBEDDED);
```

3.3. **BTstack initialization.** To initialize BTstack you need to initialize the memory and the run loop as explained in Sections 3.1 and 3.2 respectively, then setup HCI and all needed higher level protocols.

The HCI initialization has to adapt BTstack to the used platform and requires four arguments. These are:

- *Bluetooth hardware control*: 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 *bt_control_t* encapsulates common functionality that is not covered by the Bluetooth specification. As an example, the *bt_control_cc256x_in-stance* function returns a pointer to a control struct suitable for the CC256x chipset.

```
bt_control_t * control = bt_control_cc256x_instance();
```

- *HCI Transport implementation*: On embedded systems, a Bluetooth module can be connected via USB or an UART port. BTstack implements two UART based protocols: HCI UART Transport Layer (H4) and H4 with eHCILL support, a lightweight low-power variant by Texas Instruments. These are accessed by linking the appropriate file (`src/hci_transport_h4_dma.c` resp. `src/hci_transport_h4_ehcill_dma.c`) and then getting a pointer to HCI Transport implementation. For more information on adapting HCI Transport to different environments, see Section 7.3.

```
hci_transport_t * transport = hci_transport_h4_dma_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. First, the HCI command to change the baud rate is sent, then it is necessary to wait for the confirmation event from the Bluetooth module. Only now, can the UART baud rate changed. As an example, the CC256x has to be initialized at 115200 and can then be used at higher speeds.

```
hci_uart_config_t* config = hci_uart_config_cc256x_instance();
```

- *Persistent storage* - specifies where to persist data like link keys or remote device names. 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 Section 7.4.

```
remote_device_db_t * remote_db = &remote_device_db_memory;
```

After these are ready, HCI is initialized like this:

```
#define HCI_ACL_PAYLOAD_SIZE 52
#define MAX_SPP_CONNECTIONS 1
#define MAX_NO_HCI_CONNECTIONS MAX_SPP_CONNECTIONS
#define MAX_NO_L2CAP_SERVICES 2
#define MAX_NO_L2CAP_CHANNELS (1+MAX_SPP_CONNECTIONS)
#define MAX_NO_RFCOMM_MULTIPLEXERS MAX_SPP_CONNECTIONS
#define MAX_NO_RFCOMM_SERVICES 1
#define MAX_NO_RFCOMM_CHANNELS MAX_SPP_CONNECTIONS
#define MAX_NO_DB_MEM_DEVICE_NAMES 0
#define MAX_NO_DB_MEM_LINK_KEYS 3
#define MAX_NO_DB_MEM_SERVICES 1
```

LISTING 2. Memory configuration for an SPP service with a minimal L2CAP MTU.

```
hci_init(transport, config, control, remote_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.

3.4. **Services.** One important construct of BTstack is *service*. A service represents a server side component that handles incoming connections. So far, BTstack provides L2CAP 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.

3.5. **Where to get data - packet handlers.** 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 M 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 packet handler - handles HCI and general BTstack events if L2CAP is not used (rare case).

Table 2. Functions for registering packet handlers

Packet Handler	Registering Function
HCI packet handler	*hci_register_packet_handler*
L2CAP packet handler	*l2cap_register_packet_handler*
L2CAP service packet handler	*l2cap_register_service_internal*
L2CAP channel packet handler	*l2cap_create_channel_internal*
RFCOMM packet handler	*rfcomm_register_packet_handler*

- L2CAP packet handler - handles HCI and general BTstack events.
- 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 packet handler - handles RFCOMM incoming/outgoing events and data.

These handlers are registered with the functions listed in Table 2.

HCI and general BTstack events are delivered to the packet handler specified by *l2cap_register_packet_handler* function, or *hci_register_packet_handler*, if L2CAP is not used. 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_internal* is used. Currently, RFCOMM provides only a single packet handler specified by *rfcomm_register_packet_handler* for all RFCOMM connections, but this will be fixed in the next API overhaul.

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_internal*; 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_internal*. In this call, you'll also specify a packet handler to accept and receive keyboard data.

4. 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.

In the following, we first explain how the various Bluetooth protocols are used in BTstack. In the next chapter, we go over the profiles.

4.1. **HCI - Host Controller Interface.**

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

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

However, as many features of the GAP Classic can be achieved by sending a single HCI command, not all GAP convenience functions are listed in `src/gap.h`. If there's no special GAP function, please consider sending the HCI command directly, as explained in the following.

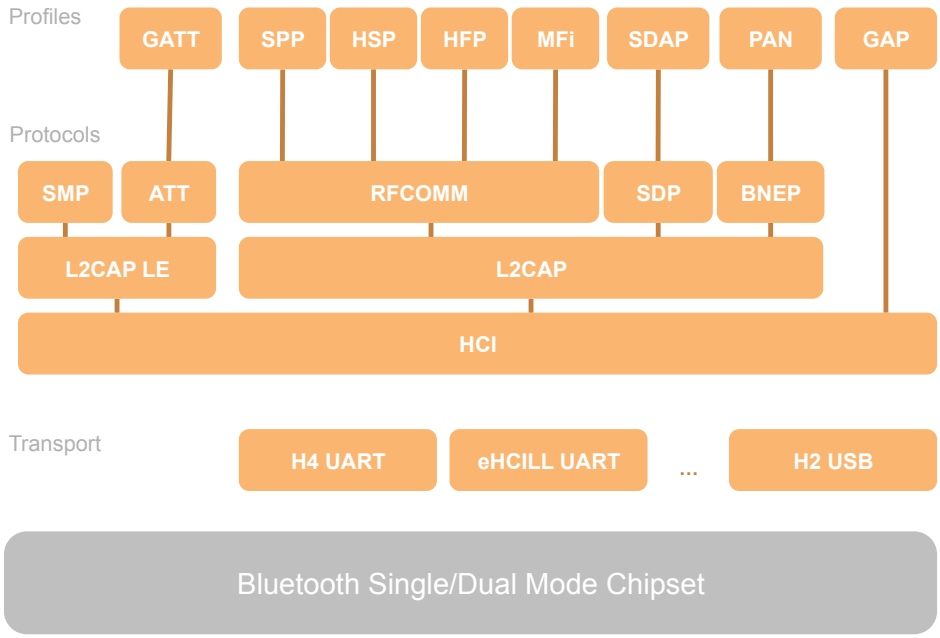

FIGURE 2. BTstack Protocol Architecture

4.1.1. *Defining custom HCI command templates.* Each HCI command is assigned a 2-byte OpCode used to uniquely identify different types of commands. The OpCode parameter is divided into two fields, called the OpCode Group Field (OGF) and OpCode Command Field (OCF), see Bluetooth Specification[15] - Core Version 4.0, Volume 2, Part E, Chapter 5.4. Listing 3 shows the OGFs provided by BTstack in `src/hci.h` file. 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. BTstack provides the *hci_cmd_t* struct as a compact format to define HCI command packets, see Listing 4, and `include/btstack/hci_cmds.h` file in the source code. The OpCode of a command can be calculated using the OPCODE macro.

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

LISTING 3. Supported OpCode Group Fields.

```
// Calculate combined ogf/ocf value.
```

[15]https://www.bluetooth.org/Technical/Specifications/adopted.htm

TABLE 3. Supported Format Specifiers of HCI Command Parameter

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)

```
#define OPCODE(ogf, ocf) (ocf | ogf << 10)

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

extern const hci_cmd_t hci_write_local_name;
...
```

LISTING 4. hci_cmds.h defines HCI command template.

Listing 5 illustrates the *hci_write_local_name* HCI command template from BTstack library. 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 3 lists the format specifiers supported by BTstack. Check `src/hci_cmds.c` for other predefined HCI commands and info on their parameters.

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

LISTING 5. Example of HCI command template.

4.1.2. *Sending HCI command based on a template.*

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

LISTING 6. Send hci_write_local_name command that takes a string as a parameter.

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

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 use HCI automatically.

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

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

```
btstack_packet_handler_t l2cap_packet_handler;

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

void create_outgoing_l2cap_channel(bd_addr_t address, uint16_t psm,
    uint16_t mtu){
     l2cap_create_channel_internal(NULL, l2cap_packet_handler,
         remote_bd_addr, psm, mtu);
}

void l2cap_packet_handler(uint8_t packet_type, uint16_t channel,
    uint8_t *packet, uint16_t size){
    if (packet_type == HCI_EVENT_PACKET &&
          packet[0] == L2CAP_EVENT_CHANNEL_OPENED){
        if (packet[2]) {
            printf("Connection failed\n\r");
            return;
        }
        printf("Connected\n\r");
    }
    if (packet_type == L2CAP_DATA_PACKET){
        // handle L2CAP data packet
```

```
        return;
    }
}
```

LISTING 7. L2CAP handler for outgoing L2CAP channel.

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_internal* function. The *l2cap_-create_channel_internal* 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 7.

4.2.2. *Provide an L2CAP service.* To provide an L2CAP service, the application on a local Bluetooth device must init the L2CAP layer and register the service with *l2cap_register_service_internal*. 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_internal* and *l2cap_deny_connection_internal* functions respectively. If a connection is accepted and the incoming L2CAP channel gets successfully opened, the L2CAP service can send L2CAP data packets to the connected device with *l2cap_send_internal*.

```
void btstack_setup(){
    ...
    l2cap_init();
    l2cap_register_service_internal(NULL, packet_handler, 0x11,100);
}

void packet_handler (uint8_t packet_type, uint16_t channel, uint8_t
   *packet, uint16_t size){
      ...
      if (packet_type == L2CAP_DATA_PACKET){
        // handle L2CAP data packet
        return;
    }
    switch(event){
        ...
        case L2CAP_EVENT_INCOMING_CONNECTION:
            bt_flip_addr(event_addr, &packet[2]);
            handle     = READ_BT_16(packet, 8);
            psm        = READ_BT_16(packet, 10);
            local_cid  = READ_BT_16(packet, 12);
            printf("L2CAP incoming connection requested.");
            l2cap_accept_connection_internal(local_cid);
            break;
        case L2CAP_EVENT_CHANNEL_OPENED:
            bt_flip_addr(event_addr, &packet[3]);
            psm = READ_BT_16(packet, 11);
            local_cid = READ_BT_16(packet, 13);
```

```
            handle = READ_BT_16(packet, 9);
            if (packet[2] == 0) {
                printf("Channel successfully opened.");
            } else {
                printf("L2CAP connection failed. status code.");
            }
            break;
        case L2CAP_EVENT_CREDITS:
        case DAEMON_EVENT_HCI_PACKET_SENT:
            tryToSend();
            break;
        case L2CAP_EVENT_CHANNEL_CLOSED:
            break;
    }
}
```

LISTING 8. Providing an L2CAP service.

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. In such case, the application can try sending again upon reception of DAEMON_EVENT_HCI_PACKET_SENT or L2CAP_EVENT_CREDITS event. The first event signals that the internal BTstack outgoing buffer became free again, the second one signals the same for ACL buffers in the Bluetooth chipset. Listing 8 provides L2CAP service example code.

4.2.3. *L2CAP LE - L2CAP Low Energy Protocol.* In addition to the full L2CAP implementation in the *src* folder, BTstack provides an optimized v*l2cap_le* implementation in the *ble* folder. This L2CAP LE variant can be used for single-mode devices and provides the base for the ATT and SMP protocols.

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

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

credits are provided by the application such that it can manage its receive buffers explicitly.

4.3.2. *Access an RFCOMM service on a remote device.* To communicate with an RFCOMM service on a remote device, the application on a local Bluetooth device initiates the RFCOMM layer using the *rfcomm_init* function, and then creates an outgoing RFCOMM channel to a given server channel on a remote device using the *rfcomm_create_channel_internal* function. The *rfcomm_create_channel_internal* 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_internal* - see Section 4.3.5.

The packet handler that is given as an input parameter of the RFCOMM create channel function will be assigned to the new outgoing RFCOMM channel. This handler receives the RFCOMM_EVENT_OPEN_CHAN-NEL_COMPLETE and RFCOMM_EVENT_CHANNEL_CLOSED events, and RFCOMM data packets, as shown in Listing 12.

4.3.3. *Provide an RFCOMM service.* To provide an RFCOMM service, the application on a local Bluetooth device must first init the L2CAP and RFCOMM layers and then register the service with *rfcomm_register_service_internal*. 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_internal* and *rfcomm_deny_connection_internal* 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_internal* and receive data packets by the packet handler provided by the *rfcomm_register_service_internal* call.

Sending of RFCOMM 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. In such case, the application can try sending again upon reception of DAEMON_EVENT_HCI_PACKET_SENT or RFCOMM_EVENT_CREDITS event. The first event signals that the internal BTstack outgoing buffer became free again, the second one signals that the remote side allowed to send another packet. Listing 13 provides the RFCOMM service example code.

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

Listing 9. RFCOMM service with automatic credit management.

4.3.4. *Living with a single output buffer.* 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.

```
void prepareData(void){
    ...
}

void tryToSend(void){
    if (!dataLen) return;
    if (!rfcomm_channel_id) return;

    int err = rfcomm_send_internal(rfcomm_channel_id,  dataBuffer,
       dataLen);
    switch (err){
        case 0:
            // packet is sent prepare next one
            prepareData();
            break;
        case RFCOMM_NO_OUTGOING_CREDITS:
        case BTSTACK_ACL_BUFFERS_FULL:
            break;
        default:
           printf("rfcomm_send_internal() -> err %d\n\r", err);
        break;
    }
}
```

LISTING 10. Preparing and sending data.

```
void packet_handler (uint8_t packet_type, uint16_t channel, uint8_t
   *packet, uint16_t size){
    ...
    switch(event){
        case RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE:
            if (status) {
                printf("RFCOMM channel open failed.");
            } else {
                rfcomm_channel_id = READ_BT_16(packet, 12);
                rfcomm_mtu = READ_BT_16(packet, 14);
                printf("RFCOMM channel opened, mtu = %u.",
                   rfcomm_mtu);
            }
            break;
        case RFCOMM_EVENT_CREDITS:
        case DAEMON_EVENT_HCI_PACKET_SENT:
            tryToSend();
            break;
        case RFCOMM_EVENT_CHANNEL_CLOSED:
            rfcomm_channel_id = 0;
            break;
       ...
       }
}
```

LISTING 11. Managing the speed of RFCOMM packet generation.

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.

BTstack returns BTSTACK_ACL_BUFFERS_FULL, if the outgoing buffer is full and RFCOMM_NO_OUTGOING_CREDITS, if no outgoing credits are available. In Listing 10, we show how to resend data packets when credits or outgoing buffers become available.

RFCOMM's mandatory credit-based flow-control imposes an additional constraint on sending a data packet - at least one new RFCOMM credit must be available. BTstack signals the availability of a credit by sending an RFCOMM credit (RFCOMM_EVENT_CREDITS) event.

These two events represent two orthogonal mechanisms that deal with flow control. Taking these mechanisms in account, the application should try to send data packets when one of these two events is received, see Listing 11 for a RFCOMM example.

If the management of credits is manual, credits are provided by the application such that it can manage its receive buffers explicitly, see Listing 14.

Manual credit management is recommended when received RFCOMM data cannot be processed immediately. In the SPP flow control example in Section 6.6, 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 15. 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.

4.3.5. *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, see Listing 9, 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

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

Listing 14. RFCOMM service with manual credit management.

```
void init_rfcomm(){
    ...
    rfcomm_init();
    rfcomm_register_packet_handler(packet_handler);
}

void create_rfcomm_channel(uint8_t packet_type, uint8_t *packet,
    uint16_t size){
    rfcomm_create_channel_internal(connection, addr, rfcomm_channel)
        ;
}

void rfcomm_packet_handler(uint8_t packet_type, uint16_t channel,
    uint8_t *packet, uint16_t size){
    if (packet_type == HCI_EVENT_PACKET && packet[0] ==
        RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE){
        if (packet[2]) {
            printf("Connection failed\n\r");
            return;
        }
        printf("Connected\n\r");
    }

    if (packet_type == RFCOMM_DATA_PACKET){
        // handle RFCOMM data packets
        return;
    }
}
```

LISTING 12. RFCOMM handler for outgoing RFCOMM channel.

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

LISTING 15. Providing new credits

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

4.4.1. *Create and announce SDP records.* BTstack contains a complete SDP server and allows to register SDP records. An SDP record is a list of SDP Attribute {*ID, Value*} pairs that are stored in a Data Element Sequence (DES).

```
void btstack_setup(){
    ...
    rfcomm_init();
    rfcomm_register_service_internal(NULL, rfcomm_channel_nr, mtu);
}

void packet_handler(uint8_t packet_type, uint8_t *packet, uint16_t
    size){
    if (packet_type == RFCOMM_DATA_PACKET){
        // handle RFCOMM data packets
        return;
    }
    ...
    switch (event) {
        ...
        case RFCOMM_EVENT_INCOMING_CONNECTION:
            //data: event(8), len(8), address(48), channel(8),
                rfcomm_cid(16)
            bt_flip_addr(event_addr, &packet[2]);
            rfcomm_channel_nr = packet[8];
            rfcomm_channel_id = READ_BT_16(packet, 9);
            rfcomm_accept_connection_internal(rfcomm_channel_id);
            break;
        case RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE:
           // data: event(8), len(8), status (8), address (48),
               handle(16), server channel(8), rfcomm_cid(16), max
               frame size(16)
            if (packet[2]) {
                printf("RFCOMM channel open failed.");
                break;
            }
           // data: event(8), len(8), status (8), address (48),
               handle (16), server channel(8), rfcomm_cid(16), max
               frame size(16)
           rfcomm_channel_id = READ_BT_16(packet, 12);
           mtu = READ_BT_16(packet, 14);
           printf("RFCOMM channel open succeeded.");
           break;
        case RFCOMM_EVENT_CREDITS:
        case DAEMON_EVENT_HCI_PACKET_SENT:
            tryToSend();
            break;

        case RFCOMM_EVENT_CHANNEL_CLOSED:
            printf("Channel closed.");
            rfcomm_channel_id = 0;
        break;
    }
}
```

Listing 13. Providing RFCOMM service.

The Attribute ID is a 16-bit number, the value can be of other simple types like integers or strings or can itselff contain other DES.

To create an SDP record for an SPP service, you can call *sdp_create_spp_service* from `src/sdp_util.c` with a pointer to a buffer to store the record, the RFCOMM server channel number, and a record name.

For other types of records, you can use the other functions in `src/sdp_util.c`, using the data element *de_** functions. Listing 17 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*.

4.4.2. *Query remote SDP service.* BTstack provides an SDP client to query SDP services of a remote device. The SDP Client API is shown in Appendix F. 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 16.

```
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 (packet[2] == HCI_STATE_WORKING){
                sdp_query_rfcomm_channel_and_name_for_uuid(remote, 0
                    x0003);
            }
            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_query_rfcomm_service_event_t * ve;

    switch (event->type){
        case SDP_QUERY_RFCOMM_SERVICE:
            ve = (sdp_query_rfcomm_service_event_t*) event;
            printf("Service name: '%s', RFCOMM port %u\n", ve->
                service_name, ve->channel_nr);
            break;
        case SDP_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_query_rfcomm_register_callback(handle_query_rfcomm_event,
        NULL);

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

Listing 16. Searching RFCOMM services on a remote device.

```
uint8_t des_buffer[200];
uint8_t* attribute;
de_create_sequence(service);

// 0x0000 "Service Record Handle"
```

```
de_add_number(des_buffer, DE_UINT, DE_SIZE_16,
    SDP_ServiceRecordHandle);
de_add_number(des_buffer, DE_UINT, DE_SIZE_32, 0x10001);

// 0x0001 "Service Class ID List"
de_add_number(des_buffer,  DE_UINT, DE_SIZE_16,
    SDP_ServiceClassIDList);
attribute = de_push_sequence(des_buffer);
{
    de_add_number(attribute,  DE_UUID, DE_SIZE_16, 0x1101 );
}
de_pop_sequence(des_buffer, attribute);
```

LISTING 17. Creating record with the data element (*de_**) functions.

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

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

4.5.2. *Access a BNEP service on a remote device.* To connect to a remote BNEP service, you need to know its UUID. The set of available UUIDs can be queried by a SDP query for the PAN profile. Please see Section 5.3 for details. With the remote UUID, you can create a connection using the *bnep_connect* function. You'll receive a *BNEP_EVENT_OPEN_CHANNEL_COMPLETE* 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*.

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

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

4.6. **ATT - Attribute Protocol.** The ATT protocol is used by an ATT client to read and write attribute values stored on an ATT server. In addition, the ATT server can notify the client about attribute value changes. An attribute has a handle, a type, and a set of properties, see Section 5.5.2.

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. 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 5.5.1 for more.

4.7. **SMP - Security Manager Protocol.** The SMP protocol allows to setup authenticated and encrypted LE connection. After initialization and configuration, SMP handles security related functions on it's 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.

4.7.1. *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.

4.7.2. *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, 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*
- *sm_set_io_capabilities*

4.7.3. *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_IDENTITY_RESOLVING_STARTED* to mark the start of a lookup,

and later:

- *SM_IDENTITY_RESOLVING_SUCCEEDED* on lookup success, or
- *SM_IDENTITY_RESOLVING_FAILED* on lookup failure.

4.7.4. *Bonding process.* In Bluetooth LE, there are three main methods of establishing an encrypted connection. From the most to the least secure, these are: Out-of-Band (OOB) Data , Passkey, and Just Works.

With OOB data, there needs to be a pre-shared secret 16 byte key. In most cases, this is not an option, especially since popular OS like iOS don't provide a way to specify it. It 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*.

Depending on the authentication requirements, available OOB data, and the enabled STK generation methods, BTstack will request feedback from the app in the form of an event:

- *SM_PASSKEY_INPUT_NUMBER*: request user to input a passkey
- *SM_PASSKEY_DISPLAY_NUMBER*: show a passkey to the user
- *SM_JUST_WORKS_REQUEST*: request a user to accept a Just Works pairing

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

After the bonding process, *SM_PASSKEY_DISPLAY_CANCEL* is emitted to update the user interface.

5. Profiles

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

5.1. **GAP - Generic Access Profile: Classic.** The GAP profile defines how devices find each other and establish a secure connection for other profiles. As mentioned before, the GAP functionality is split between `src/gap.h` and `src/hci.h`. Please check both.

5.1.1. *Become discoverable.* A remote unconnected Bluetooth device must be set as "discoverable" in order to be seen by a device performing the inquiry scan. To become discoverable, an application can call *hci_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 18 for an example.

```
int main(void){
    ...
    // make discoverable
    hci_discoverable_control(1);
    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
               hci_discoverable_control(0);
           ...
      }
 }
```

LISTING 18. Setting device as discoverable. OFF by default.

5.1.2. *Discover remote devices.* To scan for remote devices, the *hci_inquiry* command is used. Found remote devices are reported as a part of HCI_EVENT_INQUIRY_RESULT, HCI_EVENT-_INQUIRY_RESULT_WITH_RSSI, or HCI_EVENT_EXTENDED_INQUIRY_RESPONSE events. Each response contains at least the Bluetooth address, the class of device, the page scan repetition mode, and the clock offset of found device. The latter events add information about the received signal strength or provide the Extended Inquiry Result (EIR). A code snippet is shown in Listing 19.

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 in Section 6.2.

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

On Bluetooth devices that conform to the Bluetooth v2.0 or older specification, a PIN code (up to 16 bytes ASCII) has to be entered on both sides. This isn't optimal for embedded systems that do not have full I/O capabilities. To support pairing with older devices using a PIN, see Listing 20.

```
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");
              bt_flip_addr(bd_addr, &packet[2]);

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

```
void print_inquiry_results(uint8_t *packet){
    int event = packet[0];
    int numResponses = packet[2];
    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 = READ_BT_24(packet, 3 + numResponses
                *(6+1+1+1) + i*3);
            clockOffset =   READ_BT_16(packet, 3 + numResponses
                *(6+1+1+1+3) + i*2) & 0x7fff;
            rssi  = 0;
        } else {
            classOfDevice = READ_BT_24(packet, 3 + numResponses
                *(6+1+1)     + i*3);
            clockOffset =   READ_BT_16(packet, 3 + numResponses
                *(6+1+1+3)   + i*2) & 0x7fff;
            rssi  = packet [3 + numResponses*(6+1+1+3+2) + i*1];
        }
        printf("Device found: %s with COD: 0x%06x, pageScan %u,
            clock offset 0x%04x, rssi 0x%02x\n", bd_addr_to_str(addr
            ), classOfDevice, pageScanRepetitionMode, clockOffset,
            rssi);
    }
}

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

LISTING 19. Discovering remote Bluetooth devices.

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

LISTING 20. Answering authentication request with PIN 0000.

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 Section 7.4. 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.

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

5.2. **SPP - Serial Port Profile.** The SPP profile defines how to set up virtual serial ports and connect two Bluetooth enabled devices.

5.2.1. *Accessing an SPP Server on a remote device.* To access a remote SPP server, you first need to query the remote device for its SPP services. Section 4.4.2 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 in Section 4.3.2

5.2.2. *Providing an SPP Server.* To provide an SPP Server, you need to provide an RFCOMM service with a specific RFCOMM channel number as explained in Section 4.3.3. Then, you need to create an SDP record for it and publish it with the SDP server by calling *sdp_register_service_internal*. BTstack provides the *sdp_create_spp_service* function in `src/sdp_utils.c` 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 in Section 6.5.

5.3. **PAN - Personal Area Networking Profile.** The PAN profile uses BNEP to provide on-demand networking capabilities between Bluetooth devices. The PAN profile defines the following roles:

- PAN User (PANU)

- Network Access Point (NAP)
- Group Ad-hoc Network (GN)

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

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

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

Currently, BTstack supports only PANU.

5.3.1. *Accessing a remote PANU service.* To access a remote PANU service, you first need perform an SDP query to get the L2CAP PSM for the requested PANU UUID. With these two pieces of information, you can connect BNEP to the remote PANU service with the *bnep_connect* function. The PANU Demo example in Section 6.7 shows how this is accomplished.

5.3.2. *Providing a PANU service.* To provide a PANU service, you need to provide a BNEP service with the service UUID, e.g. the PANU UUID, and a a maximal ethernet frame size, as explained in Section 4.5.3. Then, you need to create an SDP record for it and publish it with the SDP server by calling *sdp_register_service_internal*. BTstack provides the *pan_create_panu_service* function in *src/pan.c* that requires an empty buffer of approximately 200 bytes, a description, and a security description.

5.4. **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.

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

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

After a connection is established, the Security Manager will try to resolve the peer Bluetooth address as explained in Section 4.7.

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

- *hci_le_set_advertising_data*
- *hci_le_set_advertising_parameters*
- *hci_le_set_advertise_enable*

As these are direct HCI commands, please refer to Section 4.1.2 for details and have a look at the SPP and LE Counter example in Section 6.10.

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 provided with the *hci_le_set_scan_response_data*.

The scan parameters can be set with *le_central_set_scan_parameters*. The scan can be started/stopped with *le_central_start_scan*/*le_central_stop_scan*.

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

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

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

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

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

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

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

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

GATT queries cannot be interleaved. Therefore, you can check if you can perform a GATT query on a particular connection using *gatt_client_is_ready*.

As a result to a GATT query, zero to many *le_event*s are returned before a *GATT_QUERY_COMPLETE* event completes the query.

For more details on the available GATT queries, please consult Appendix H.

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

To save on both code space and memory, BTstack does not provide a GATT Server implementation. Instead, a textual description of the GATT profile is directly converted into a compact internal ATT Attribute database by a GATT profile compiler. The ATT protocol server - implemented by `ble/att_server.c` and `ble/att.c` - 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:

```
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}
...
```

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

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

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

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

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

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

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.

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

  // setup BTstack memory pools
  btstack_memory_init();

  // select embedded run loop
  run_loop_init(RUN_LOOP_EMBEDDED);

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

  // init HCI
  hci_transport_t     * transport = hci_transport_h4_dma_instance();
  remote_device_db_t * remote_db = (remote_device_db_t *) &
     remote_device_db_memory;
  hci_init(transport, NULL, NULL, remote_db);

  // setup example
  btstack_main(argc, argv);

  // go
  run_loop_execute();
}
```

Listing 21. Exemplary platform init in main.c

6. Examples

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

Listing 21 shows a minimal platform setup for an embedded system with a Bluetooth chipset connected via UART.

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

The call to *hci_dump_open* configures BTstack to output all Bluetooth packets and it's own debug and error message via printf. The Python script *tools/create_packet_log.py* can be used to convert the console output into a Bluetooth PacketLogger format that can be opened by the OS X PacketLogger tool as well

as by Wireshark for further inspection. When asking for help, please always include a log created with HCI dump.

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

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

The examples are grouped like this:

- Hello World example:
  - *led_counter*: Hello World: blinking LED without Bluetooth, in Section 6.1.
- GAP example:
  - *gap_inquiry*: GAP Inquiry Example, in Section 6.2.
- SDP Queries examples:
  - *sdp_general_query*: Dump remote SDP Records, in Section 6.3.
  - *sdp_bnep_query*: Dump remote BNEP PAN protocol UUID and L2CAP PSM, in Section 6.4.
- SPP Server examples:
  - *spp_counter*: SPP Server - Heartbeat Counter over RFCOMM, in Section 6.5.
  - *spp_flowcontrol*: SPP Server - Flow Control, in Section 6.6.
- BNEP/PAN example:
  - *panu_demo*: PANU Demo, in Section 6.7.
- Low Energy examples:
  - *gatt_browser*: GATT Client - Discovering primary services and their characteristics, in Section 6.8.
  - *le_counter*: LE Peripheral - Heartbeat Counter over GATT, in Section 6.9.
- Dual Mode example:
  - *spp_and_le_counter*: Dual mode example, in Section 6.10.

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

6.1.1. *Periodic Timer Setup.* As timers in BTstack are single shot, the periodic counter is implemented by re-registering the timer source in the heartbeat handler callback function. Listing 22 shows heartbeat handler adapted to periodically toggle an LED and print number of toggles.

```
static void heartbeat_handler(timer_source_t *ts){
  // increment counter
  char lineBuffer[30];
  sprintf(lineBuffer, "BTstack counter %04u\n\r", ++counter);
  puts(lineBuffer);
```

```
  // toggle LED
  hal_led_toggle();

  // re-register timer
  run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(&heartbeat);
}
```

LISTING 22. Periodic counter.

6.1.2. *Main Application Setup.* Listing 23 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[]){

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

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

LISTING 23. Setup heartbeat timer.

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

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

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

In ACTIVE, the following events are processed:

- Inquiry result event: the list of discovered devices is processed and the Class of Device (CoD), page scan mode, clock offset, and RSSI are stored in a table.
- 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 cached event: prints cached remote names provided by BTstack, if persistent storage is provided.
- 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 5.1.2.

6.2.2. *Main Application Setup.* Listing 24 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[]) {
  hci_register_packet_handler(packet_handler);

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

LISTING 24. Setup packet handler for GAP inquiry.

6.3. **sdp_general_query: Dump remote SDP Records.** The example shows how the SDP Client is used to get a list of service records on a remote device.

6.3.1. *SDP Client Setup.* SDP is based on L2CAP. To receive SDP query events you must register a callback, i.e. query handler, with the SPD parser, as shown in Listing 25. Via this handler, the SDP client will receive the following events:

- SDP_QUERY_ATTRIBUTE_VALUE containing the results of the query in chunks,
- SDP_QUERY_COMPLETE indicating the end of the query and the status

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

static void sdp_client_init(){
  // init L2CAP
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

  sdp_parser_init();
  sdp_parser_register_callback(handle_sdp_client_query_result);
}
```

LISTING 25. SDP client setup.

6.3.2. *SDP Client Query.* To trigger an SDP query to get the a list of service records on a remote device, you need to call sdp_general_query_for_uuid() with the remote address and the UUID of the public browse group, as shown in Listing 27. In this example we used fixed address of the remote device shown in Listing 26. 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 bd_addr_t remote = {0x04,0x0C,0xCE,0xE4,0x85,0xD3};
```

LISTING 26. Address of remote device in big-endian order.

```
static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
  // printf("packet_handler type %u, packet[0] %x\n", packet_type,
      packet[0]);

  if (packet_type != HCI_EVENT_PACKET) return;
  uint8_t event = packet[0];

  switch (event) {
    case BTSTACK_EVENT_STATE:
      if (packet[2] == HCI_STATE_WORKING){
        sdp_general_query_for_uuid(remote, SDP_PublicBrowseGroup);
      }
      break;
    default:
      break;
  }
}
```

LISTING 27. Querying a list of service records on a remote device..

6.3.3. *Handling SDP Client Query Results.* The SDP Client returns the results of the query in chunks. Each result packet contains the record ID, the Attribute ID, and a chunk of the Attribute value. In this example, we append new chunks for the same Attribute ID in a large buffer, see Listing 28.

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_query_rfcomm.c* which retrieves the RFCOMM channel number and the service name.

```
static void handle_sdp_client_query_result(sdp_query_event_t * event
    ){
  sdp_query_attribute_value_event_t * ve;
  sdp_query_complete_event_t * ce;
```

```
  switch (event->type){
    case SDP_QUERY_ATTRIBUTE_VALUE:
      ve = (sdp_query_attribute_value_event_t*) event;

      // handle new record
      if (ve->record_id != record_id){
        record_id = ve->record_id;
        printf("\n---\nRecord nr. %u\n", record_id);
      }

      assertBuffer(ve->attribute_length);

      attribute_value[ve->data_offset] = ve->data;
      if ((uint16_t)(ve->data_offset+1) == ve->attribute_length){
         printf("Attribute 0x%04x: ", ve->attribute_id);
         de_dump_data_element(attribute_value);
      }
      break;
    case SDP_QUERY_COMPLETE:
      ce = (sdp_query_complete_event_t*) event;
      printf("General query done with status %d.\n\n", ce->status);
      exit(0);
      break;
  }
}
```

LISTING 28. Handling query result chunks..

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

6.4.1. *SDP Client Setup.* As with the previous example, you must register a callback, i.e. query handler, with the SPD parser, as shown in Listing 29. Via this handler, the SDP client will receive events:

- SDP_QUERY_ATTRIBUTE_VALUE containing the results of the query in chunks,
- SDP_QUERY_COMPLETE reporting the status and the end of the query.

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

static void sdp_client_init(){
  // init L2CAP
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);
```

```
  sdp_parser_init();
  sdp_parser_register_callback(handle_sdp_client_query_result);
}
```

Listing 29. SDP client setup.

6.4.2. *SDP Client Query.*

```
static 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:
      // BTstack activated, get started
      if (packet[2] == HCI_STATE_WORKING){
        printf("Start SDP BNEP query.\n");
        sdp_general_query_for_uuid(remote, SDP_BNEPProtocol);
      }
      break;
    default:
      break;
  }
}
```

Listing 30. Querying the a list of service records on a remote device..

6.4.3. *Handling SDP Client Query Result.* The SDP Client returns the result of the query in chunks. Each result packet contains the record ID, the Attribute ID, and a chunk of the Attribute value, see Listing 31. 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(sdp_query_event_t * event
    ){
...

        switch(ve->attribute_id){
          // 0x0001 "Service Class ID List"
          case SDP_ServiceClassIDList:
            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 PANU_UUID:
                  case NAP_UUID:
                  case GN_UUID:
                    printf(" ** Attribute 0x%04x: BNEP PAN protocol
                        UUID: %04x\n", ve->attribute_id, uuid);
                    break;
                  default:
                    break;
                }
              }
              break;
...
            case SDP_ProtocolDescriptorList:{
                printf(" ** Attribute 0x%04x: ", ve->attribute_id);

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

                  if (de_get_element_type(element) != DE_UUID)
                    continue;
                  uint32_t uuid = de_get_uuid32(element);
                  switch (uuid){
                    case SDP_L2CAPProtocol:
                      if (!des_iterator_has_more(&prot_it)) continue;
                      des_iterator_next(&prot_it);
                      de_element_get_uint16(des_iterator_get_element(&
                          prot_it), &l2cap_psm);
                      break;
                    case SDP_BNEPProtocol:
                      if (!des_iterator_has_more(&prot_it)) continue;
                      des_iterator_next(&prot_it);
                      de_element_get_uint16(des_iterator_get_element(&
                          prot_it), &bnep_version);
                      break;
                    default:
                      break;
                  }
                }
                printf("l2cap_psm 0x%04x, bnep_version 0x%04x\n",
                    l2cap_psm, bnep_version);
              }
              break;
...
```

```
}
```

LISTING 31. Extracting BNEP Protcol UUID and L2CAP PSM.

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

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

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

6.5.1. *SPP Service Setup.* To provide an SPP service, the L2CAP, RFCOMM, and SDP protocol layers are required. After setting up an RFCOMM service with channel nubmer RFCOMM_SERVER_CHANNEL, an SDP record is created and registered with the SDP server. Example code for SPP service setup is provided in Listing 32. The SDP record created by function sdp_create_spp_service consists of a basic SPP definition that uses the provided RFCOMM channel ID and service name. For more details, please have a look at it in `src/sdp_util.c`. The SDP record is created on the fly in RAM and is deterministic. To preserve valuable RAM, the result could be stored as constant data inside the ROM.

```
void spp_service_setup(){
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

  rfcomm_init();
  rfcomm_register_packet_handler(packet_handler);
  rfcomm_register_service_internal(NULL, RFCOMM_SERVER_CHANNEL, 0
      xffff);  // 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));
...
  service_record_item_t * service_record_item = (
      service_record_item_t *) spp_service_buffer;
  sdp_create_spp_service( (uint8_t*) &service_record_item->
      service_record, RFCOMM_SERVER_CHANNEL, "SPP Counter");
  printf("SDP service buffer size: %u\n", (uint16_t) (sizeof(
      service_record_item_t) + de_get_len((uint8_t*) &
      service_record_item->service_record)));
  sdp_register_service_internal(NULL, service_record_item);
...
}
```

LISTING 32. SPP service setup.

6.5.2. *Periodic Timer Setup.* The heartbeat handler increases the real counter every second, and sends a text string with the counter value, as shown in Listing 33.

```
static timer_source_t heartbeat;

static void  heartbeat_handler(struct timer *ts){
  static int counter = 0;

  if (rfcomm_channel_id){
    char lineBuffer[30];
    sprintf(lineBuffer, "BTstack counter %04u\n", ++counter);
    printf("%s", lineBuffer);
    if (rfcomm_can_send_packet_now(rfcomm_channel_id)) {
      int err = rfcomm_send_internal(rfcomm_channel_id, (uint8_t*)
          lineBuffer, strlen(lineBuffer));
      if (err) {
        log_error("rfcomm_send_internal -> error 0X%02x", err);
      }
    }
  }
  run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(ts);
}

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

Listing 33. Periodic Counter.

6.5.3. *Bluetooth Logic.* The Bluetooth logic is implemented within the packet handler, see Listing 34. 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_OPEN_CHANNEL_COMPLETE, and
- 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_OPEN_CHANNEL_COMPLETE 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.

```
static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
...
        case HCI_EVENT_PIN_CODE_REQUEST:
          // pre-ssp: inform about pin code request
          printf("Pin code request - using '0000'\n");
          bt_flip_addr(event_addr, &packet[2]);
          hci_send_cmd(&hci_pin_code_request_reply, &event_addr, 4,
              "0000");
          break;

        case HCI_EVENT_USER_CONFIRMATION_REQUEST:
          // ssp: inform about user confirmation request
          printf("SSP User Confirmation Request with numeric value
              '%06u'\n", READ_BT_32(packet, 8));
          printf("SSP User Confirmation Auto accept\n");
          break;

        case RFCOMM_EVENT_INCOMING_CONNECTION:
          // data: event (8), len(8), address(48), channel (8),
              rfcomm_cid (16)
          bt_flip_addr(event_addr, &packet[2]);
          rfcomm_channel_nr = packet[8];
          rfcomm_channel_id = READ_BT_16(packet, 9);
          printf("RFCOMM channel %u requested for %s\n",
              rfcomm_channel_nr, bd_addr_to_str(event_addr));
          rfcomm_accept_connection_internal(rfcomm_channel_id);
          break;

        case RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE:
          // data: event(8), len(8), status (8), address (48),
              server channel(8), rfcomm_cid(16), max frame size(16)
          if (packet[2]) {
            printf("RFCOMM channel open failed, status %u\n", packet
                [2]);
```

```
            } else {
              rfcomm_channel_id = READ_BT_16(packet, 12);
              mtu = READ_BT_16(packet, 14);
              printf("RFCOMM channel open succeeded. New RFCOMM
                  Channel ID %u, max frame size %u\n",
                  rfcomm_channel_id, mtu);
            }
            break;
...
}
```

LISTING 34. SPP Server - Heartbeat Counter over RFCOMM.

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

6.6.1. *SPP Service Setup.* Listing 35 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(){
  // init L2CAP
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

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

  // init SDP, create record for SPP and register with SDP
  sdp_init();
  memset(spp_service_buffer, 0, sizeof(spp_service_buffer));
  service_record_item_t * service_record_item = (
      service_record_item_t *) spp_service_buffer;
  sdp_create_spp_service( (uint8_t*) &service_record_item->
      service_record, 1, "SPP Counter");
  printf("SDP service buffer size: %u\n\r", (uint16_t) (sizeof(
      service_record_item_t) + de_get_len((uint8_t*) &
      service_record_item->service_record)));
  sdp_register_service_internal(NULL, service_record_item);
}
```

LISTING 35. Providing one initial credit during RFCOMM service initialization.

6.6.2. *Periodic Timer Setup.* Explicit credit management is recommended when received RFCOMM data cannot be processed immediately. In this example,

delayed processing of received data is simulated with the help of a periodic timer as follows. When the packet handler receives a data packet, it does not provide a new credit, it sets a flag instead, see Listing 37. 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 36.

```
static void  heartbeat_handler(struct timer *ts){
  if (rfcomm_send_credit){
    rfcomm_grant_credits(rfcomm_channel_id, 1);
    rfcomm_send_credit = 0;
  }
  run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(ts);
}
```

Listing 36. Heartbeat handler with manual credit management.

```
// Bluetooth logic
static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
...
    case RFCOMM_DATA_PACKET:
      for (i=0;i<size;i++){
        putchar(packet[i]);
      };
      putchar('\n');
      rfcomm_send_credit = 1;
      break;
...
}
```

Listing 37. Packet handler with manual credit management.

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

6.7.1. *Main application configuration.* In the application configuration, L2CAP and BNEP are initialized and a BNEP service, for server mode, is registered, before the Bluetooth stack gets started, as shown in Listing 38.

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

```
static void handle_sdp_client_query_result(sdp_query_event_t *event)
    ;

static void panu_setup(){
  // Initialize L2CAP
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

  // Initialise BNEP
  bnep_init();
  bnep_register_packet_handler(packet_handler);
  // Minimum L2CAP MTU for bnep is 1691 bytes
  bnep_register_service(NULL, SDP_PANU, 1691);

  // Initialise SDP
  sdp_parser_init();
  sdp_parser_register_callback(handle_sdp_client_query_result);
}
```

LISTING 38. Panu setup.

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

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

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

Listing 39 shows how a packet is received from the TAP network interface and forwarded over the BNEP connection.

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

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

  network_buffer_len = len;
  if (bnep_can_send_packet_now(bnep_cid)) {
    bnep_send(bnep_cid, network_buffer, network_buffer_len);
```

```
    network_buffer_len = 0;
  } else {
    // park the current network packet
    run_loop_remove_data_source(&tap_dev_ds);
  }
  return 0;
}
```

LISTING 39. Process incoming network packets.

6.7.3. *SDP parser callback.* The SDP parsers retrieves the BNEP PAN UUID as explained in Section 6.4.

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

```
static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size)
{
...
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      event = packet[0];
      switch (event) {
        case BTSTACK_EVENT_STATE:
          if (packet[2] == HCI_STATE_WORKING) {
            printf("Start SDP BNEP query.\n");
            sdp_general_query_for_uuid(remote, SDP_BNEPProtocol);
          }
          break;

...

        case BNEP_EVENT_INCOMING_CONNECTION:
          // data: event(8), len(8), bnep source uuid (16), bnep
              destination uuid (16), remote_address (48)
          uuid_source = READ_BT_16(packet, 2);
          uuid_dest   = READ_BT_16(packet, 4);
          mtu         = READ_BT_16(packet, 6);
          bnep_cid    = channel;
          memcpy(&event_addr, &packet[8], sizeof(bd_addr_t));
          printf("BNEP connection from %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);
          hci_local_bd_addr(local_addr);
          tap_fd = tap_alloc(tap_dev_name, local_addr);
          if (tap_fd < 0) {
            printf("Creating BNEP tap device failed: %s\n", strerror
                (errno));
          } else {
            printf("BNEP device \"%s\" allocated.\n", tap_dev_name);
            tap_dev_ds.fd = tap_fd;
            tap_dev_ds.process = process_tap_dev_data;
```

```
            run_loop_add_data_source(&tap_dev_ds);
          }
          break;

...

        case BNEP_EVENT_CHANNEL_TIMEOUT:
          printf("BNEP channel timeout! Channel will be closed\n");
          break;

        case BNEP_EVENT_CHANNEL_CLOSED:
          printf("BNEP channel closed\n");
          run_loop_remove_data_source(&tap_dev_ds);
          if (tap_fd > 0) {
            close(tap_fd);
            tap_fd = -1;
          }
          break;

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

          break;

        default:
          break;
      }
      break;

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

    default:
      break;
  }
}
```

LISTING 40. Packet Handler.

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

In server mode, BNEP_EVENT_INCOMING_CONNECTION is received after a client has connected. 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.

In client mode, BNEP_EVENT_OPEN_CHANNEL_COMPLETE is received after a client has connected or when the connection fails. The status field returns the error code. It is otherwise identical to BNEP_EVENT_INCOMING_CONNECTION before.

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

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

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

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

6.8.1. *GATT client setup.* In the setup phase, a GATT client must register the HCI and GATT client packet handlers, as shown in Listing 41. 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 4.7.

```
static uint16_t gc_id;

// Handles connect, disconnect, and advertising report events,
// starts the GATT client, and sends the first query.
```

```
static void handle_hci_event(void * connection, 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.
void handle_gatt_client_event(le_event_t * event);

static void gatt_client_setup(){
  // Initialize L2CAP and register HCI event handler
  l2cap_init();
  l2cap_register_packet_handler(&handle_hci_event);

  // Initialize GATT client
  gatt_client_init();
  // Register handler for GATT client events
  gc_id = gatt_client_register_packet_handler(&
      handle_gatt_client_event);

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

LISTING 41. Setting up GATT client.

6.8.2. *HCI packet handler.* The HCI packet handler has to start the scanning, to find the first advertising device, to stop scanning, to connect to and later to disconnect from it, to start the GATT client upon the connection is completed, and to send the first query - in this case the gatt_client_discover_primary_services() is called, see Listing 42.

```
static void handle_hci_event(void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
  if (packet_type != HCI_EVENT_PACKET) return;
  advertising_report_t report;

  uint8_t event = packet[0];
  switch (event) {
    case BTSTACK_EVENT_STATE:
      // BTstack activated, get started
      if (packet[2] != HCI_STATE_WORKING) break;
      if (cmdline_addr_found){
        printf("Trying to connect to %s\n", bd_addr_to_str(
            cmdline_addr));
        le_central_connect(cmdline_addr, 0);
        break;
      }
      printf("BTstack activated, start scanning!\n");
      le_central_set_scan_parameters(0,0x0030, 0x0030);
      le_central_start_scan();
      break;
    case GAP_LE_ADVERTISING_REPORT:
```

```
        fill_advertising_report_from_packet(&report, packet);
        // stop scanning, and connect to the device
        le_central_stop_scan();
        le_central_connect(report.address,report.address_type);
        break;
      case HCI_EVENT_LE_META:
        // wait for connection complete
        if (packet[2] != HCI_SUBEVENT_LE_CONNECTION_COMPLETE) break;
        gc_handle = READ_BT_16(packet, 4);
        // query primary services
        gatt_client_discover_primary_services(gc_id, gc_handle);
        break;
      case HCI_EVENT_DISCONNECTION_COMPLETE:
        printf("\nGATT browser - DISCONNECTED\n");
        exit(0);
        break;
      default:
        break;
    }
}
```

LISTING 42. Connecting and disconnecting from the GATT client.

6.8.3. *GATT Client event handler.* Query results and further queries are handled by the GATT client packet handler, as shown in Listing 43. 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 int search_services = 1;

void handle_gatt_client_event(le_event_t * event){
  le_service_t service;
  le_characteristic_t characteristic;
  switch(event->type){
    case GATT_SERVICE_QUERY_RESULT:
      service = ((le_service_event_t *) event)->service;
      dump_service(&service);
      services[service_count++] = service;
      break;
    case GATT_CHARACTERISTIC_QUERY_RESULT:
      characteristic = ((le_characteristic_event_t *) event)->
          characteristic;
      dump_characteristic(&characteristic);
      break;
    case GATT_QUERY_COMPLETE:
      if (search_services){
        // GATT_QUERY_COMPLETE of search services
        service_index = 0;
        printf("\nGATT browser - CHARACTERISTIC for SERVICE ");
```

```
            printUUID128(service.uuid128); printf("\n");
            search_services = 0;
            gatt_client_discover_characteristics_for_service(gc_id,
                gc_handle, &services[service_index]);
        } else {
            // GATT_QUERY_COMPLETE of search characteristics
            if (service_index < service_count) {
              service = services[service_index++];
              printf("\nGATT browser - CHARACTERISTIC for SERVICE ");
              printUUID128(service.uuid128);
              printf(", [0x%04x-0x%04x]\n", service.start_group_handle,
                  service.end_group_handle);

              gatt_client_discover_characteristics_for_service(gc_id,
                  gc_handle, &service);
              break;
            }
            service_index = 0;
            gap_disconnect(gc_handle);
        }
        break;
      default:
        break;
    }
}
```

LISTING 43. Handling of the GATT client queries.

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

6.9.1. *Main Application Setup.* Listing 44 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*. Finally, it configures the heartbeat handler and boots the Bluetooth stack.

```
static int  le_notification_enabled;
static timer_source_t heartbeat;

static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size);
static uint16_t att_read_callback(uint16_t con_handle, uint16_t
    att_handle, uint16_t offset, uint8_t * buffer, uint16_t
    buffer_size);
static int att_write_callback(uint16_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 timer *ts);
```

```
static void le_counter_setup(){
  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

  // setup le device db
  le_device_db_init();

  // setup SM: Display only
  sm_init();

  // setup ATT server
  att_server_init(profile_data, att_read_callback,
      att_write_callback);
  att_dump_attributes();
  // set one-shot timer
  heartbeat.process = &heartbeat_handler;
  run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(&heartbeat);
}
```

Listing 44. Init L2CAP SM ATT Server and start heartbeat timer.

6.9.2. *Managing LE Advertisements.* To be discoverable, LE Advertisements are enabled using direct HCI Commands. As there's no guarantee that a HCI command can always be sent, the gap_run function is used to send each command as requested by the *todos* variable. First, the advertisement data is set with hci_le_set_advertising_data. Then the advertisement parameters including the advertisement interval is set with hci_le_set_advertising_parameters. Finally, advertisements are enabled with hci_le_set_advertise_enable. See Listing 45.

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.

```
const uint8_t adv_data[] = {
  // Flags general discoverable
  0x02, 0x01, 0x06,
  // Name
  0x0b, 0x09, 'L', 'E', ' ', 'C', 'o', 'u', 'n', 't', 'e', 'r',
};

uint8_t adv_data_len = sizeof(adv_data);
enum {
  SET_ADVERTISEMENT_PARAMS = 1 << 0,
  SET_ADVERTISEMENT_DATA   = 1 << 1,
  ENABLE_ADVERTISEMENTS    = 1 << 2,
};
static uint16_t todos = 0;

static void gap_run(){

  if (!hci_can_send_command_packet_now()) return;
```

```
  if (todos & SET_ADVERTISEMENT_DATA){
    printf("GAP_RUN: set advertisement data\n");
    todos &= ~SET_ADVERTISEMENT_DATA;
    hci_send_cmd(&hci_le_set_advertising_data, adv_data_len,
        adv_data);
    return;
  }

  if (todos & SET_ADVERTISEMENT_PARAMS){
    todos &= ~SET_ADVERTISEMENT_PARAMS;
    uint8_t adv_type = 0;    // default
    bd_addr_t null_addr;
    memset(null_addr, 0, 6);
    uint16_t adv_int_min = 0x0030;
    uint16_t adv_int_max = 0x0030;
    hci_send_cmd(&hci_le_set_advertising_parameters, adv_int_min,
        adv_int_max, adv_type, 0, 0, &null_addr, 0x07, 0x00);
    return;
  }

  if (todos & ENABLE_ADVERTISEMENTS){
    printf("GAP_RUN: enable advertisements\n");
    todos &= ~ENABLE_ADVERTISEMENTS;
    hci_send_cmd(&hci_le_set_advertise_enable, 1);
    return;
  }
}
```

Listing 45. Handle GAP Tasks.

6.9.3. *Packet Handler.* The packet handler is only used to trigger advertisements after BTstack is ready and after disconnects, see Listing 46. Advertisements are not automatically re-enabled after a connection was closed, even though they have been active before.

```
static void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
  switch (packet_type) {
    case HCI_EVENT_PACKET:
      switch (packet[0]) {

        case BTSTACK_EVENT_STATE:
          // BTstack activated, get started
          if (packet[2] == HCI_STATE_WORKING) {
            todos = SET_ADVERTISEMENT_PARAMS |
                SET_ADVERTISEMENT_DATA | ENABLE_ADVERTISEMENTS;
            gap_run();
          }
          break;

        case HCI_EVENT_DISCONNECTION_COMPLETE:
```

```
            todos = ENABLE_ADVERTISEMENTS;
            le_notification_enabled = 0;
            gap_run();
            break;

          default:
            break;
        }
        break;

      default:
        break;
    }
    gap_run();
}
```

LISTING 46. Packet Handler.

6.9.4. *Heartbeat Handler.* The heartbeat handler updates the value of the single Characteristic provided in this example, and sends a notification for this characteristic if enabled, see Listing 47.

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

static void  heartbeat_handler(struct timer *ts){
  counter++;
  counter_string_len = sprintf(counter_string, "BTstack counter %04u
     \n", counter);
  puts(counter_string);

  if (le_notification_enabled) {
    att_server_notify(
       ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
       , (uint8_t*) counter_string, counter_string_len);
  }
  run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(ts);
}
```

LISTING 47. Hearbeat Handler.

6.9.5. *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 48.

```
// 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(uint16_t con_handle, uint16_t
    att_handle, uint16_t offset, uint8_t * buffer, uint16_t
    buffer_size){
  if (att_handle ==
      ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
      ){
    if (buffer){
      memcpy(buffer, &counter_string[offset], counter_string_len -
          offset);
    }
    return counter_string_len - offset;
  }
  return 0;
}
```

Listing 48. ATT Read.

6.9.6. *ATT Write.* The only valid ATT write in this example is to the Client Characteristic Configuration, which configures notification and indication. If the ATT handle matches the client configuration handle, the new configuration value is stored and used in the heartbeat handler to decide if a new value should be sent. See Listing 49.

```
static int att_write_callback(uint16_t con_handle, uint16_t
    att_handle, uint16_t transaction_mode, uint16_t offset, uint8_t
    *buffer, uint16_t buffer_size){
  if (att_handle !=
      ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_CLIENT_CONFIGURATION_
      ) return 0;
  le_notification_enabled = READ_BT_16(buffer, 0) ==
      GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION;
  return 0;
}
```

Listing 49. ATT Write.

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

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

6.10.1. *Advertisements.* The Flags attribute in the Advertisement Data indicates if a device is in dual-mode or not. Flag 0x02 indicates LE General Discoverable, Dual-Mode device. See Listing 50.

```
const uint8_t adv_data[] = {
  // Flags: General Discoverable
  0x02, 0x01, 0x02,
  // Name
  0x0b, 0x09, 'L', 'E', ' ', 'C', 'o', 'u', 'n', 't', 'e', 'r',
};
```

LISTING 50. Advertisement data: Flag 0x02 indicates a dual mode device.

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

6.10.3. *Heartbeat Handler.* Similar to the packet handler, the heartbeat handler is the combination of the individual ones. After updating the counter, it sends an RFCOMM packet if an RFCOMM connection is active, and an LE notification if the remote side has requested notifications.

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

  counter++;
  counter_string_len = sprintf(counter_string, "BTstack counter %04u
      \n", counter);
  // printf("%s", counter_string);

  if (rfcomm_channel_id){
    if (rfcomm_can_send_packet_now(rfcomm_channel_id)){
      int err = rfcomm_send_internal(rfcomm_channel_id, (uint8_t*)
          counter_string, counter_string_len);
      if (err) {
        log_error("rfcomm_send_internal -> error 0X%02x", err);
      }
    }
  }

  if (le_notification_enabled) {
    att_server_notify(
       ATT_CHARACTERISTIC_0000FF11_0000_1000_8000_00805F9B34FB_01_VALUE_HANDLE
       , (uint8_t*) counter_string, counter_string_len);
  }
  run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(ts);
}
```

LISTING 51. Combined Heartbeat handler.

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

```
int btstack_main(void);
int btstack_main(void)
{
  hci_discoverable_control(1);

  l2cap_init();
  l2cap_register_packet_handler(packet_handler);

  rfcomm_init();
  rfcomm_register_packet_handler(packet_handler);
  rfcomm_register_service_internal(NULL, RFCOMM_SERVER_CHANNEL, 0
      xffff);

  // init SDP, create record for SPP and register with SDP
  sdp_init();
  memset(spp_service_buffer, 0, sizeof(spp_service_buffer));
...
  service_record_item_t * service_record_item = (
      service_record_item_t *) spp_service_buffer;
  sdp_create_spp_service( (uint8_t*) &service_record_item->
      service_record, RFCOMM_SERVER_CHANNEL, "SPP Counter");
  printf("SDP service buffer size: %u\n", (uint16_t) (sizeof(
      service_record_item_t) + de_get_len((uint8_t*) &
      service_record_item->service_record)));
  sdp_register_service_internal(NULL, service_record_item);
...

  hci_ssp_set_io_capability(SSP_IO_CAPABILITY_DISPLAY_YES_NO);

  // setup le device db
  le_device_db_init();

  // setup SM: Display only
  sm_init();

  // setup ATT server
  att_server_init(profile_data, att_read_callback,
      att_write_callback);
  att_dump_attributes();
  // set one-shot timer
  heartbeat.process = &heartbeat_handler;
  run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
  run_loop_add_timer(&heartbeat);

  // turn on!
  hci_power_control(HCI_POWER_ON);

  return 0;
}
```

Listing 52. Init L2CAP RFCOMM SDO SM ATT Server and start heartbeat timer.

7. Porting to Other Platforms

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

7.1. **Time Abstraction Layer.** BTstack requires a way to learn about passing time. *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.

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

For this, you need to define *HAVE_TICK* in *btstack-config.h*:

```
#define HAVE_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.

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

For this, you need to define *HAVE_TIME_MS* in *btstack-config.h*:

```
#define HAVE_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);
```

7.2. **Bluetooth Hardware Control API.** The Bluetooth hardware control API can provide the HCI layer with a custom initialization script, a vendor-specific baud rate change command, and system power notifications. It is also used to control the power mode of the Bluetooth module, i.e., turning it on/off and putting to sleep. In addition, it provides an error handler *hw_error* that is

called when a Hardware Error is reported by the Bluetooth module. The callback allows for persistent logging or signaling of this failure. add recipe

Overall, the struct *bt_control_t* encapsulates common functionality that is not covered by the Bluetooth specification. As an example, the *bt_control_cc256x_instance* function returns a pointer to a control struct suitable for the CC256x chipset.

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

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

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

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

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

The main HCI H4 implementations for embedded system is *hci_h4_transport_dma* function. This function calls the following sequence: *hal_uart_dma_init*, *hal_uart_dma_set_block_received* and *hal_uart_dma_set_block_sent* functions. After 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.

7.3.2. *H4 with eHCILL support.* With the standard H4 protocol interface, it is not possible for either the host nor the baseband controller to enter a sleep mode. Besides the official H5 protocol, various chip vendors came up with proprietary solutions to this. The eHCILL support by Texas Instruments allows both the

```
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);

    // remote name
    int (*get_name)(bd_addr_t bd_addr, device_name_t *device_name);
    void(*put_name)(bd_addr_t bd_addr, device_name_t *device_name);
    void(*delete_name)(bd_addr_t bd_addr);
} remote_device_db_t;
```

Listing 53. Persistent Storage Interface.

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);
```

7.4. **Persistent Storage API.** 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 API provides an interface to implement concrete drivers for a particular system. As an example and for testing purposes, BTstack provides the memory-only implementation *remote_device_db_memory*. An implementation has to conform to the interface in Listing 53.

8. 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 Application", "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.

8.1. **Adapting BTstack for Single-Threaded Environments.** In a single-threaded environment, all application components run on the same (single) thread and use direct function calls as shown in Figure 3.

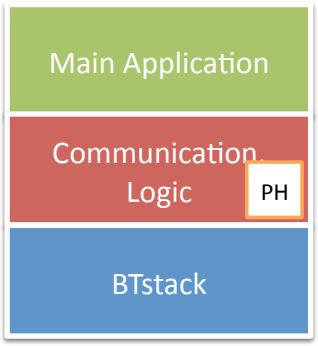

FIGURE 3. BTstack in single-threaded environment.

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:

- *run_loop_cocoa.c* is an implementation for the CoreFoundation Framework used in OS X and iOS. All run loop functions are implemented in terms of CoreFoundation calls, data sources and timers are modeled as CFSockets and CFRunLoopTimer respectively.
- *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.

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

- The Communication Logic is implemented on a dedicated BTstack thread, and the Main Application communicates with the BTstack thread via application-specific messages over an Interprocess Communication (IPC) as depicted in Figure 4. This option results in less code and quick adaption.
- BTstack must be extended to run standalone, i.e, as a Daemon, on a dedicated thread and the Main Application controls this daemon via BTstack

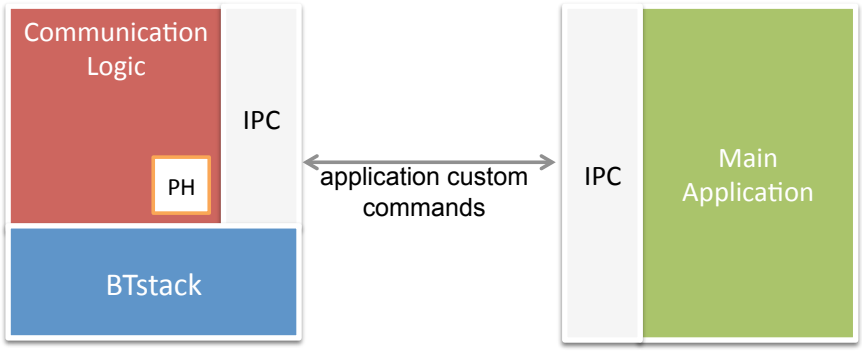

FIGURE 4. BTstack in multi-threaded environment - monolithic solution.

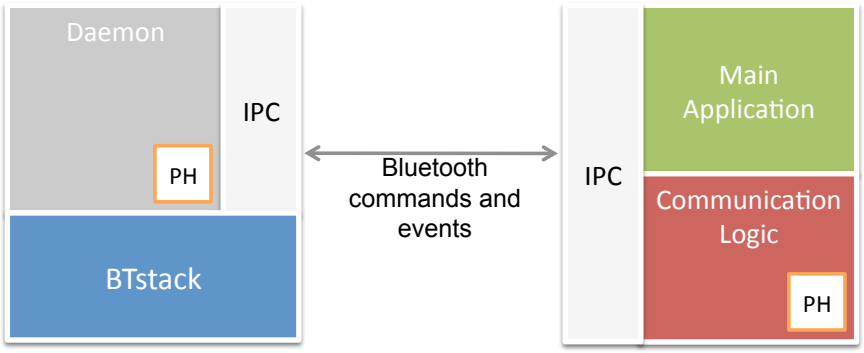

FIGURE 5. BTstack in multi-threaded environment - solution with daemon.

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 5. This option requires more code but provides more flexibility.

Appendix A. Run Loop API

```
/**
 * @brief Set timer based on current time in milliseconds.
 */
void run_loop_set_timer(timer_source_t *a, uint32_t timeout_in_ms);

/**
 * @brief Set callback that will be executed when timer expires.
 */
void run_loop_set_timer_handler(timer_source_t *ts, void (*process)(
    timer_source_t *_ts));

/**
 * @brief Add/Remove timer source.
 */
void run_loop_add_timer(timer_source_t *timer);
int  run_loop_remove_timer(timer_source_t *timer);

/**
 * @brief Init must be called before any other run_loop call. Use
     RUN_LOOP_EMBEDDED for embedded devices.
 */
void run_loop_init(RUN_LOOP_TYPE type);

/**
 * @brief Set data source callback.
 */
void run_loop_set_data_source_handler(data_source_t *ds, int (*
    process)(data_source_t *_ds));

/**
 * @brief Add/Remove data source.
 */
void run_loop_add_data_source(data_source_t *dataSource);
int  run_loop_remove_data_source(data_source_t *dataSource);

/**
 * @brief Execute configured run loop. This function does not return
     .
 */
void run_loop_execute(void);

// hack to fix HCI timer handling
#ifdef HAVE_TICK
/**
 * @brief Sets how many milliseconds has one tick.
 */
uint32_t embedded_ticks_for_ms(uint32_t time_in_ms);
/**
 * @brief Queries the current time in ticks.
 */
```

```
uint32_t embedded_get_ticks(void);
/**
 * @brief Queries the current time in ms
 */
uint32_t embedded_get_time_ms(void);
/**
 * @brief Allows to update BTstack system ticks based on another
     already existing clock.
 */
void embedded_set_ticks(uint32_t ticks);
#endif
#ifdef EMBEDDED
/**
 * @brief Sets an internal flag that is checked in the critical
     section just before entering sleep mode. Has to be called by
     the interrupt handler of a data source to signal the run loop
     that a new data is available.
 */
void embedded_trigger(void);
/**
 * @brief Execute run_loop once. It can be used to integrate BTstack
     's timer and data source processing into a foreign run loop (it
      is not recommended).
 */
void embedded_execute_once(void);
#endif
```

Appendix B. HCI API

```
le_connection_parameter_range_t
    gap_le_get_connection_parameter_range();
void gap_le_set_connection_parameter_range(
    le_connection_parameter_range_t range);

/* LE Client Start */

le_command_status_t le_central_start_scan(void);
le_command_status_t le_central_stop_scan(void);
le_command_status_t le_central_connect(bd_addr_t addr,
    bd_addr_type_t addr_type);
le_command_status_t le_central_connect_cancel(void);
le_command_status_t gap_disconnect(hci_con_handle_t handle);
void le_central_set_scan_parameters(uint8_t scan_type, uint16_t
    scan_interval, uint16_t scan_window);

/* LE Client End */

void hci_connectable_control(uint8_t enable);
void hci_close(void);

/**
 * @note New functions replacing: hci_can_send_packet_now[
    _using_packet_buffer]
 */
int hci_can_send_command_packet_now(void);

/**
 * @brief Gets local address.
 */
void hci_local_bd_addr(bd_addr_t address_buffer);

/**
 * @brief Set up HCI. Needs to be called before any other function.
 */
void hci_init(hci_transport_t *transport, void *config, bt_control_t
    *control, remote_device_db_t const* remote_device_db);

/**
 * @brief Set class of device that will be set during Bluetooth init
    .
 */
void hci_set_class_of_device(uint32_t class_of_device);

/**
 * @brief Set Public BD ADDR - passed on to Bluetooth chipset if
    supported in bt_control_h
 */
void hci_set_bd_addr(bd_addr_t addr);
```

```
/**
 * @brief Registers a packet handler. Used if L2CAP is not used (
    rarely).
 */
void hci_register_packet_handler(void (*handler)(uint8_t packet_type
   , uint8_t *packet, uint16_t size));

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

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

/**
 * @brief Creates and sends HCI command packets based on a template
    and a list of parameters. Will return error if outgoing data
    buffer is occupied.
 */
int hci_send_cmd(const hci_cmd_t *cmd, ...);

/**
 * @brief Deletes link key for remote device with baseband address.
 */
void hci_drop_link_key_for_bd_addr(bd_addr_t addr);

/* Configure Secure Simple Pairing */

/**
 * @brief Enable will enable SSP during init.
 */
void hci_ssp_set_enable(int enable);

/**
 * @brief If set, BTstack will respond to io capability request
    using authentication requirement.
 */
void hci_ssp_set_io_capability(int ssp_io_capability);
void hci_ssp_set_authentication_requirement(int
   authentication_requirement);

/**
 * @brief If set, BTstack will confirm a numeric comparison and
    enter '000000' if requested.
 */
void hci_ssp_set_auto_accept(int auto_accept);

/**
 * @brief Get addr type and address used in advertisement packets.
 */
```

```
void hci_le_advertisement_address(uint8_t * addr_type, bd_addr_t
    addr);
```

Appendix C. L2CAP API

```
/**
 * @brief Set up L2CAP and register L2CAP with HCI layer.
 */
void l2cap_init(void);

/**
 * @brief Registers a packet handler that handles HCI and general
     BTstack events.
 */
void l2cap_register_packet_handler(void (*handler)(void * connection
    , uint8_t packet_type, uint16_t channel, uint8_t *packet,
    uint16_t size));

/**
 * @brief Creates L2CAP channel to the PSM of a remote device with
     baseband address. A new baseband connection will be initiated
     if necessary.
 */
void l2cap_create_channel_internal(void * connection,
    btstack_packet_handler_t packet_handler, bd_addr_t address,
    uint16_t psm, uint16_t mtu);

/**
 * @brief Disconnects L2CAP channel with given identifier.
 */
void l2cap_disconnect_internal(uint16_t local_cid, uint8_t reason);

/**
 * @brief Queries the maximal transfer unit (MTU) for L2CAP channel
     with given identifier.
 */
uint16_t l2cap_get_remote_mtu_for_local_cid(uint16_t local_cid);

/**
 * @brief Sends L2CAP data packet to the channel with given
     identifier.
 */
int l2cap_send_internal(uint16_t local_cid, uint8_t *data, uint16_t
    len);

/**
 * @brief Registers L2CAP service with given PSM and MTU, and
     assigns a packet handler. On embedded systems, use NULL for
     connection parameter.
 */
void l2cap_register_service_internal(void *connection,
    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.  On embedded
     systems, use NULL for connection parameter.
 */
void l2cap_unregister_service_internal(void *connection, uint16_t
    psm);

/**
 * @brief Accepts/Deny incoming L2CAP connection.
 */
void l2cap_accept_connection_internal(uint16_t local_cid);
void l2cap_decline_connection_internal(uint16_t local_cid, uint8_t
    reason);

/**
 * @brief Request LE connection parameter update
 */
int l2cap_le_request_connection_parameter_update(uint16_t handle,
    uint16_t interval_min, uint16_t interval_max, uint16_t
    slave_latency, uint16_t timeout_multiplier);

/**
 * @brief Non-blocking UART write
 */
int  l2cap_can_send_packet_now(uint16_t local_cid);
int  l2cap_reserve_packet_buffer(void);
void l2cap_release_packet_buffer(void);

/**
 * @brief Get outgoing buffer and prepare data.
 */
uint8_t *l2cap_get_outgoing_buffer(void);

int l2cap_send_prepared(uint16_t local_cid, uint16_t len);

int l2cap_send_prepared_connectionless(uint16_t handle, uint16_t cid
    , uint16_t len);

/**
 * @brief Bluetooth 4.0 - allows to register handler for Attribute
     Protocol and Security Manager Protocol.
 */
void l2cap_register_fixed_channel(btstack_packet_handler_t
    packet_handler, uint16_t channel_id);

uint16_t l2cap_max_mtu(void);
uint16_t l2cap_max_le_mtu(void);

int  l2cap_send_connectionless(uint16_t handle, uint16_t cid,
    uint8_t *data, uint16_t len);
```

Appendix D. RFCOMM API

```
/**
 * @brief Set up RFCOMM.
 */
void rfcomm_init(void);

/**
 * @brief Set security level required for incoming connections, need
     to be called before registering services.
 */
void rfcomm_set_required_security_level(gap_security_level_t
    security_level);

/**
 * @brief Register packet handler.
 */
void rfcomm_register_packet_handler(void (*handler)(void *
    connection, uint8_t packet_type, uint16_t channel, uint8_t *
    packet, uint16_t size));

/**
 * @brief Creates RFCOMM connection (channel) to a given server
     channel on a remote device with baseband address. A new
     baseband connection will be initiated if necessary. This
     channel will automatically provide enough credits to the remote
      side
 */
void rfcomm_create_channel_internal(void * connection, bd_addr_t
    addr, uint8_t channel);

/**
 * @brief Creates RFCOMM connection (channel) to a given server
     channel on a remote device with baseband address. new baseband
     connection will be initiated if necessary. This channel will
     use explicit credit management. During channel establishment,
     an initial  amount of credits is provided.
 */
void rfcomm_create_channel_with_initial_credits_internal(void *
    connection, bd_addr_t addr, uint8_t server_channel, uint8_t
    initial_credits);

/**
 * @brief Disconnects RFCOMM channel with given identifier.
 */
void rfcomm_disconnect_internal(uint16_t rfcomm_cid);

/**
 * @brief Registers RFCOMM service for a server channel and a
     maximum frame size, and assigns a packet handler. On embedded
     systems, use NULL for connection parameter. This channel
     provides automatically enough credits to the remote side.
```

```
 */
void rfcomm_register_service_internal(void * connection, 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. On embedded
     systems, use NULL for connection parameter. This channel will
     use explicit credit management. During channel establishment,
     an initial amount of credits is provided.
 */
void rfcomm_register_service_with_initial_credits_internal(void *
    connection, uint8_t channel, uint16_t max_frame_size, uint8_t
    initial_credits);

/**
 * @brief Unregister RFCOMM service.
 */
void rfcomm_unregister_service_internal(uint8_t service_channel);

/**
 * @brief Accepts/Deny incoming RFCOMM connection.
 */
void rfcomm_accept_connection_internal(uint16_t rfcomm_cid);
void rfcomm_decline_connection_internal(uint16_t rfcomm_cid);

/**
 * @brief Grant more incoming credits to the remote side for the
     given RFCOMM channel identifier.
 */
void rfcomm_grant_credits(uint16_t rfcomm_cid, uint8_t credits);

/**
 * @brief Checks if RFCOMM can send packet. Returns yes if packet
     can be sent.
 */
int rfcomm_can_send_packet_now(uint16_t rfcomm_cid);

/**
 * @brief Sends RFCOMM data packet to the RFCOMM channel with given
     identifier.
 */
int  rfcomm_send_internal(uint16_t rfcomm_cid, uint8_t *data,
    uint16_t len);

/**
 * @brief Sends Local Line Status, see LINE_STATUS_..
 */
int rfcomm_send_local_line_status(uint16_t rfcomm_cid, uint8_t
    line_status);

/**
 * @brief Send local modem status. see MODEM_STAUS_..
 */
```

```
int rfcomm_send_modem_status(uint16_t rfcomm_cid, uint8_t
    modem_status);

/**
 * @brief Configure remote port
 */
int rfcomm_send_port_configuration(uint16_t rfcomm_cid, rpn_baud_t
    baud_rate, rpn_data_bits_t data_bits, rpn_stop_bits_t stop_bits,
     rpn_parity_t parity, rpn_flow_control_t flow_control);

/**
 * @brief Query remote port
 */
int rfcomm_query_port_configuration(uint16_t rfcomm_cid);

/**
 * @brief Allow to create RFCOMM packet in outgoing buffer.
 */
int        rfcomm_reserve_packet_buffer(void);
void       rfcomm_release_packet_buffer(void);
uint8_t * rfcomm_get_outgoing_buffer(void);
uint16_t  rfcomm_get_max_frame_size(uint16_t rfcomm_cid);
int        rfcomm_send_prepared(uint16_t rfcomm_cid, uint16_t len);
```

Appendix E. SDP API

```
/**
 * @brief Set up SDP.
 */
void sdp_init(void);

void sdp_register_packet_handler(void (*handler)(void * connection,
    uint8_t packet_type, uint16_t channel, uint8_t *packet, uint16_t
     size));

#ifdef EMBEDDED
/**
 * @brief Register service record internally - this version doesn't
     copy the record therefore it must be forever accessible.
     Preconditions:
    - AttributeIDs are in ascending order;
    - ServiceRecordHandle is first attribute and valid.
 * @return ServiceRecordHandle or 0 if registration failed.
 */
uint32_t sdp_register_service_internal(void *connection,
    service_record_item_t * record_item);
#endif

#ifndef EMBEDDED
/**
 * @brief Register service record internally - this version creates
     a copy of the record precondition: AttributeIDs are in
     ascending order => ServiceRecordHandle is first attribute if
     present.
 * @return ServiceRecordHandle or 0 if registration failed
 */
uint32_t sdp_register_service_internal(void *connection, uint8_t *
    service_record);
#endif

/**
 * @brief Unregister service record internally.
 */
void sdp_unregister_service_internal(void *connection, uint32_t
    service_record_handle);
```

Appendix F. SDP Client API

```
/**
 * @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 a registered callback.
 */
void sdp_client_query(bd_addr_t remote, uint8_t *
    des_serviceSearchPattern, uint8_t * des_attributeIDList);

#ifdef HAVE_SDP_EXTRA_QUERIES
void sdp_client_service_attribute_search(bd_addr_t remote, uint32_t
    search_serviceRecordHandle, uint8_t * des_attributeIDList);
void sdp_client_service_search(bd_addr_t remote, uint8_t *
    des_serviceSearchPattern);
#endif
```

Appendix G. SDP RFCOMM Query API

```
/**
 * @brief SDP Query RFCOMM event to deliver channel number and
     service name byte by byte.
 */
typedef struct sdp_query_rfcomm_service_event {
    uint8_t type;
    uint8_t channel_nr;
    uint8_t * service_name;
} sdp_query_rfcomm_service_event_t;

/**
 * @brief Registers a callback to receive RFCOMM service and query
     complete event.
 */
void sdp_query_rfcomm_register_callback(void(*sdp_app_callback)(
    sdp_query_event_t * event, void * context), void * context);

void sdp_query_rfcomm_deregister_callback();

/**
 * @brief Searches SDP records on a remote device for RFCOMM
     services with a given UUID.
 */
void sdp_query_rfcomm_channel_and_name_for_uuid(bd_addr_t remote,
    uint16_t uuid);

/**
 * @brief Searches SDP records on a remote device for RFCOMM
     services with a given service search pattern.
 */
void sdp_query_rfcomm_channel_and_name_for_search_pattern(bd_addr_t
    remote, uint8_t * des_serviceSearchPattern);
```

Appendix H. GATT Client API

```
typedef struct gatt_complete_event{
    uint8_t   type;
    uint16_t handle;
    uint16_t attribute_handle;
    uint8_t status;
} gatt_complete_event_t;

typedef struct le_service{
    uint16_t start_group_handle;
    uint16_t end_group_handle;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} le_service_t;

typedef struct le_service_event{
    uint8_t  type;
    uint16_t handle;
    le_service_t service;
} le_service_event_t;

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

typedef struct le_characteristic_event{
    uint8_t  type;
    uint16_t handle;
    le_characteristic_t characteristic;
} le_characteristic_event_t;

typedef struct le_characteristic_value_event{
    uint8_t   type;
    uint16_t  handle;
    uint16_t  value_handle;
    uint16_t  value_offset;
    uint16_t  blob_length;
    uint8_t * blob;
} le_characteristic_value_event_t;

typedef struct le_characteristic_descriptor{
    uint16_t handle;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} le_characteristic_descriptor_t;
```

```
typedef struct le_characteristic_descriptor_event{
    uint8_t  type;
    uint16_t handle;
    le_characteristic_descriptor_t characteristic_descriptor;
    uint16_t value_length;
    uint16_t value_offset;
    uint8_t * value;
} le_characteristic_descriptor_event_t;

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

/**
 * @brief Register callback (packet handler) for GATT client.
    Returns GATT client ID.
 */
uint16_t gatt_client_register_packet_handler (gatt_client_callback_t
    callback);

/**
 * @brief Unregister callback (packet handler) for GATT client.
 */
void gatt_client_unregister_packet_handler(uint16_t gatt_client_id);

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

/**
 * @brief Returns if the GATT client is ready to receive a query. It
     is used with daemon.
 */
int gatt_client_is_ready(uint16_t handle);

/**
 * @brief Discovers all primary services. For each found service, an
     le_service_event_t with type set to GATT_SERVICE_QUERY_RESULT
    will be generated and passed to the registered callback. The
    gatt_complete_event_t, with type set to GATT_QUERY_COMPLETE,
    marks the end of discovery.
 */
le_command_status_t gatt_client_discover_primary_services(uint16_t
   gatt_client_id, uint16_t con_handle);

/**
```

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

/**
 * @brief Finds included services within the specified service. For
     each found included service, an le_service_event_t with type
     set to GATT_INCLUDED_SERVICE_QUERY_RESULT will be generated and
      passed to the registered callback. The gatt_complete_event_t
     with type set to GATT_QUERY_COMPLETE, marks the end of
     discovery. Information about included service type (primary/
     secondary) can be retrieved either by sending an ATT find
     information request for the returned start group handle (
     returning the handle and the UUID for primary or secondary
     service) or by comparing the service to the list of all primary
      services.
 */
le_command_status_t gatt_client_find_included_services_for_service(
    uint16_t gatt_client_id, uint16_t con_handle, le_service_t *
    service);

/**
 * @brief Discovers all characteristics within the specified service
     . For each found characteristic, an le_characteristics_event_t
     with type set to GATT_CHARACTERISTIC_QUERY_RESULT will be
     generated and passed to the registered callback. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     marks the end of discovery.
 */
le_command_status_t gatt_client_discover_characteristics_for_service
    (uint16_t gatt_client_id, uint16_t con_handle, le_service_t *
    service);

/**
 * @brief The following four functions are used to discover all
     characteristics within the specified service or handle range,
     and return those that match the given UUID. For each found
     characteristic, an le_characteristic_event_t with type set to
     GATT_CHARACTERISTIC_QUERY_RESULT will be generated and passed
     to the registered callback. The gatt_complete_event_t with type
      set to GATT_QUERY_COMPLETE, marks the end of discovery.
 */
```

```
le_command_status_t
    gatt_client_discover_characteristics_for_handle_range_by_uuid16(
    uint16_t gatt_client_id, uint16_t con_handle, uint16_t
    start_handle, uint16_t end_handle, uint16_t uuid16);
le_command_status_t
    gatt_client_discover_characteristics_for_handle_range_by_uuid128
    (uint16_t gatt_client_id, uint16_t con_handle, uint16_t
    start_handle, uint16_t end_handle, uint8_t  * uuid);
le_command_status_t
    gatt_client_discover_characteristics_for_service_by_uuid16 (
    uint16_t gatt_client_id, uint16_t con_handle, le_service_t  *
    service, uint16_t  uuid16);
le_command_status_t
    gatt_client_discover_characteristics_for_service_by_uuid128(
    uint16_t gatt_client_id, uint16_t con_handle, le_service_t  *
    service, uint8_t  * uuid128);

/**
 * @brief Discovers attribute handle and UUID of a characteristic
     descriptor within the specified characteristic. For each found
     descriptor, an le_characteristic_descriptor_event_t with type
     set to GATT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT will be
     generated and passed to the registered callback. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     marks the end of discovery.
 */
le_command_status_t gatt_client_discover_characteristic_descriptors(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_t  *characteristic);

/**
 * @brief Reads the characteristic value using the characteristic's
     value handle. If the characteristic value is found, an
     le_characteristic_value_event_t with type set to
     GATT_CHARACTERISTIC_VALUE_QUERY_RESULT will be generated and
     passed to the registered callback. The gatt_complete_event_t
     with type set to GATT_QUERY_COMPLETE, marks the end of read.
 */
le_command_status_t gatt_client_read_value_of_characteristic(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_t  *characteristic);
le_command_status_t
    gatt_client_read_value_of_characteristic_using_value_handle(
    uint16_t gatt_client_id, uint16_t con_handle, uint16_t
    characteristic_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, an
     le_characteristic_value_event_t with type set to
     GATT_CHARACTERISTIC_VALUE_QUERY_RESULT and updated value offset
      will be generated and passed to the registered callback. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     mark the end of read.
 */
le_command_status_t gatt_client_read_long_value_of_characteristic(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_t  *characteristic);
le_command_status_t
    gatt_client_read_long_value_of_characteristic_using_value_handle
    (uint16_t gatt_client_id, uint16_t con_handle, uint16_t
    characteristic_value_handle);

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

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

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

/**
```

```
 * @brief Writes of the long characteristic value using the
     characteristic's value handle. It uses server response to
     validate that the write was correctly received. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE
     marks the end of write. The write is successfully performed, if
      the event's status field is set to 0.
 */
le_command_status_t
    gatt_client_reliable_write_long_value_of_characteristic(uint16_t
     gatt_client_id, uint16_t con_handle, uint16_t
    characteristic_value_handle, uint16_t length, uint8_t  * data);

/**
 * @brief Reads the characteristic descriptor using its handle. If
     the characteristic descriptor is found, an
     le_characteristic_descriptor_event_t with type set to
     GATT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT will be generated
     and passed to the registered callback. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     marks the end of read.
 */
le_command_status_t gatt_client_read_characteristic_descriptor(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_descriptor_t  * descriptor);

/**
 * @brief Reads the long characteristic descriptor using its handle.
      It will be returned in several blobs. For each blob, an
     le_characteristic_descriptor_event_t with type set to
     GATT_CHARACTERISTIC_DESCRIPTOR_QUERY_RESULT will be generated
     and passed to the registered callback. The
     gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     marks the end of read.
 */
le_command_status_t gatt_client_read_long_characteristic_descriptor(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_descriptor_t  * descriptor);

/**
 * @brief Writes the characteristic descriptor using its handle. The
      gatt_complete_event_t with type set to GATT_QUERY_COMPLETE,
     marks the end of write. The write is successfully performed, if
      the event's status field is set to 0.
 */
le_command_status_t gatt_client_write_characteristic_descriptor(
    uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_descriptor_t  * descriptor, uint16_t length,
    uint8_t  * data);
le_command_status_t gatt_client_write_long_characteristic_descriptor
    (uint16_t gatt_client_id, uint16_t con_handle,
    le_characteristic_descriptor_t  * descriptor, uint16_t length,
    uint8_t  * data);

/**
```

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

Appendix I. PAN API

```
/**
 * @brief Creates SDP record for PANU BNEP service in provided empty
     buffer.
 * @note Make sure the buffer is big enough.
 *
 * @param service is an empty buffer to store service record
 * @param security_desc
 * @param name if NULL, the default service name will be assigned
 * @param description if NULL, the default service description will
    be assigned
 */
void pan_create_panu_service(uint8_t *service, 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 security_desc
 * @param name if NULL, the default service name will be assigned
 * @param description if NULL, the default service description will
    be assigned
 * @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_service(uint8_t *service, 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 name if NULL, the default service name will be assigned
 * @param security_desc
 * @param description if NULL, the default service description will
    be assigned
 * @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_service(uint8_t *service, 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);
```

Appendix J. BNEP API

```
/**
 * @brief Set up BNEP.
 */
void bnep_init(void);

/**
 * @brief Check if a data packet can be send out.
 */
int bnep_can_send_packet_now(uint16_t bnep_cid);

/**
 * @brief Send a data packet.
 */
int bnep_send(uint16_t bnep_cid, uint8_t *packet, uint16_t len);

/**
 * @brief Set the network protocol filter.
 */
int bnep_set_net_type_filter(uint16_t bnep_cid, bnep_net_filter_t *
    filter, uint16_t len);

/**
 * @brief Set the multicast address filter.
 */
int bnep_set_multicast_filter(uint16_t bnep_cid, bnep_multi_filter_t
    *filter, uint16_t len);

/**
 * @brief Set security level required for incoming connections, need
     to be called before registering services.
 */
void bnep_set_required_security_level(gap_security_level_t
    security_level);

/**
 * @brief Register packet handler.
 */
void bnep_register_packet_handler(void (*handler)(void * connection,
     uint8_t packet_type, uint16_t channel, uint8_t *packet,
    uint16_t size));

/**
 * @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(void * connection, bd_addr_t addr, uint16_t
    l2cap_psm, 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.
 */
void bnep_register_service(void * connection, uint16_t service_uuid,
    uint16_t max_frame_size);

/**
 * @brief Unregister BNEP service.
 */
void bnep_unregister_service(uint16_t service_uuid);
```

Appendix K. GAP API

```
/**
 * @brief Enable/disable bonding. Default is enabled.
 * @param enabled
 */
void gap_set_bondable_mode(int enabled);

/**
 * @brief Start dedicated bonding with device. Disconnect after
     bonding.
 * @param device
 * @param request MITM protection
 * @return error, if max num acl connections active
 * @result GAP_DEDICATED_BONDING_COMPLETE
 */
int gap_dedicated_bonding(bd_addr_t device, int
    mitm_protection_required);

gap_security_level_t gap_security_level_for_link_key_type(
    link_key_type_t link_key_type);
gap_security_level_t gap_security_level(hci_con_handle_t con_handle)
    ;

void gap_request_security_level(hci_con_handle_t con_handle,
    gap_security_level_t level);
int  gap_mitm_protection_required_for_security_level(
    gap_security_level_t level);

/**
 * @brief Sets local name.
 * @note has to be done before stack starts up
 * @param name is not copied, make sure memory is accessible during
     stack startup
 */
void gap_set_local_name(const char * local_name);
```

Appendix L. SM API

```
/**
 * @brief Security Manager event
 */
typedef struct sm_event {
    uint8_t   type;                 ///< See <btstack/hci_cmds.h>
        SM_...
    uint8_t   addr_type;
    bd_addr_t address;
    uint32_t  passkey;              ///< only used for
        SM_PASSKEY_DISPLAY_NUMBER
    uint16_t  le_device_db_index;   ///< only used for
        SM_IDENTITY_RESOLVING_...
    uint8_t   authorization_result; ///< only use for
        SM_AUTHORIZATION_RESULT
} sm_event_t;

/**
 * @brief Initializes the Security Manager, connects to L2CAP
 */
void sm_init();

/**
 * @brief Set secret ER key for key generation as described in Core
    V4.0, Vol 3, Part G, 5.2.2
 * @param er
 */
void sm_set_er(sm_key_t er);

/**
 * @brief Set secret IR key for key generation as described in Core
    V4.0, Vol 3, Part G, 5.2.2
 */
void sm_set_ir(sm_key_t ir);

/**
 *
 * @brief Registers OOB Data Callback. The callback should set the
    oob_data and return 1 if OOB data is availble
 * @param get_oob_data_callback
 */
void sm_register_oob_data_callback( int (*get_oob_data_callback)(
   uint8_t addres_type, bd_addr_t addr, uint8_t * oob_data));

/**
 *
 * @brief Registers packet handler. Called by att_server.c
 */
void sm_register_packet_handler(btstack_packet_handler_t 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
 * @param OR combination of SM_AUTHREQ_ flags
 */
void sm_set_authentication_requirements(uint8_t auth_req);

/**
 * @brief Sets the available IO Capabilities
 * @param IO_CAPABILITY_
 */
void sm_set_io_capabilities(io_capability_t io_capability);

/**
 * @brief Let Peripheral request an encrypted connection right after
     connecting
 * @note Not used normally. Bonding is triggered by access to
     protected attributes in ATT Server
 */
void sm_set_request_security(int enable);

/**
 * @brief Trigger Security Request
 * @note Not used normally. Bonding is triggered by access to
     protected attributes in ATT Server
 */
void sm_send_security_request(uint16_t handle);

/**
 * @brief Decline bonding triggered by event before
 * @param addr_type and address
 */
void sm_bonding_decline(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Confirm Just Works bonding
 * @param addr_type and address
 */
```

```
void sm_just_works_confirm(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Reports passkey input by user
 * @param addr_type and address
 * @param passkey in [0..999999]
 */
void sm_passkey_input(uint8_t addr_type, bd_addr_t address, uint32_t
    passkey);

/**
 *
 * @brief Get encryption key size.
 * @param addr_type and address
 * @return 0 if not encrypted, 7-16 otherwise
 */
int sm_encryption_key_size(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Get authentication property.
 * @param addr_type and address
 * @return 1 if bonded with OOB/Passkey (AND MITM protection)
 */
int sm_authenticated(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Queries authorization state.
 * @param addr_type and address
 * @return authorization_state for the current session
 */
authorization_state_t sm_authorization_state(uint8_t addr_type,
   bd_addr_t address);

/**
 * @brief Used by att_server.c to request user authorization.
 * @param addr_type and address
 */
void sm_request_authorization(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Report user authorization decline.
 * @param addr_type and address
 */
void sm_authorization_decline(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Report user authorization grant.
 * @param addr_type and address
 */
void sm_authorization_grant(uint8_t addr_type, bd_addr_t address);

/**
 * @brief Support for signed writes, used by att_server.
```

```
 * @note Message and result are in little endian to allows passing
     in ATT PDU without flipping them first.
 */
int  sm_cmac_ready();
void sm_cmac_start(sm_key_t k, uint16_t message_len, uint8_t *
    message, uint32_t sign_counter, void (*done_handler)(uint8_t
    hash[8]));

/**
 * @brief Identify device in LE Device DB.
 * @param handle
 * @return index from le_device_db or -1 if not found/identified
 */
int sm_le_device_index(uint16_t handle );
```

Appendix M. Events and Errors

L2CAP events and data packets are delivered to the packet handler specified by *l2cap_register_service* resp. *l2cap_create_channel_internal*. 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_internal* and *l2cap_decline_connection_internal* respectively.
- L2CAP_EVENT_CREDITS - emitted when there is a chance to send a new L2CAP packet. BTstack does not buffer packets. Instead, it requires the application to retry sending if BTstack cannot deliver a packet to the Bluetooth module. In this case, the l2cap_send_internal will return an error.
- L2CAP_EVENT_SERVICE_REGISTERED - Status not equal zero indicates an error. Possible errors: service is already registered; MAX_NO_L2CAP_SERVICES (defined in config.h) already registered.

All RFCOMM events and data packets are currently delivered to the packet handler specified by *rfcomm_register_packet_handler*. Data packets have the RFCOMM_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_internal* and *rfcomm_decline_connection_internal* respectively.
- RFCOMM_EVENT_CHANNEL_CLOSED - emitted when channel is closed. No status information is provided.
- RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE - sent if channel establishment is done. Status not equal zero indicates an error. Possible errors: an L2CAP error, out of memory.
- RFCOMM_EVENT_CREDITS - The application can resume sending when this even is received. See Section 4.3.1 for more on RFCOMM credit-based flow-control.
- RFCOMM_EVENT_SERVICE_REGISTERED - Status not equal zero indicates an error. Possible errors: service is already registered; MAX_NO_RFCOMM_SERVICES (defined in config.h) already registered.

Table 4. L2CAP Events

Event / Event Parameters (size in bits)	Event Code
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)*	0x70
L2CAP_EVENT_CHANNEL_CLOSED *event (8), len(8), channel(16)*	0x71
L2CAP_EVENT_INCOMING_CONNECTION *event(8), len(8), address(48), handle(16), psm (16), local_cid(16), remote_cid (16)*	0x72
L2CAP_EVENT_CREDITS *event(8), len(8), local_cid(16), credits(8)*	0x74
L2CAP_EVENT_SERVICE_REGISTERED *event(8), len(8), status(8), psm(16)*	0x75

Table 5. RFCOMM Events

Event / Event Parameters (size in bits)	Event Code
RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE *event(8), len(8), status(8), address(48), handle(16), server_channel(8), rfcomm_cid(16), max_frame_size(16)*	0x80
RFCOMM_EVENT_CHANNEL_CLOSED *event(8), len(8), rfcomm_cid(16)*	0x81
RFCOMM_EVENT_INCOMING_CONNECTION *event(8), len(8), address(48), channel (8), rfcomm_cid(16)*	0x82
RFCOMM_EVENT_CREDITS *event(8), len(8), rfcomm_cid(16), credits(8)*	0x84
RFCOMM_EVENT_SERVICE_REGISTERED *event(8), len(8), status(8), rfcomm server channel_id(8)*	0x85

Table 6. 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
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_CHANNEL_ALREADY_REGISTERED	0x71
RFCOMM_NO_OUTGOING_CREDITS	0x72
SDP_HANDLE_ALREADY_REGISTERED	0x80

Appendix N. Revision History

Rev	Date	Comments
1.x	April 27, 2015	Added more platforms. Replaced Recipes with Protocols and Profiles. Added more examples.
1.3	November 6, 2014	Introducing GATT client and server. Work in progress.
1.2	November 1, 2013	Explained Secure Simple Pairing in "Pairing of devices".
1.1	August 30, 2013	Introduced SDP client. Updated Quick Recipe on "Query remote SDP service".