VERSION 1.1
August 30, 2013

BTstack Getting Started

Using MSP430 Examples

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

Contents

Thanks for checking out BTstack! In this manual, we first provide the usual 'quick starter guide' before highlighting BTstack's main design choices and going into more details with a few examples. 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 D on page 50.

1. Get started with BTstack and MSP-EXP430F5438 + CC256x

1.1. 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[1].

1.2. General Tools. 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[2].

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

On Windows, you need to install and configure Subversion, mspgcc, GNU Make, and Python manually:

- Subversion[3] for Windows.
- Optionally Tortoise SVN[4]: This is a GUI front-end for Subversion that makes checkouts and other operations easier by integrating them into the Windows Explorer. 1.2.1). For example, for one Python installation the path is `C:\Python27`.
- mspgcc[5] for Windows: Download and extract to `C:\mspgcc`. Add `C:\mspgcc\bin` folder to the Windows Path in Environment variable as explained in Section 1.2.1.
- GNU Make[6] 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[7] for Windows: Add Python installation folder to the Windows Path in Environment Variables.

1.2.1. *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:\mspgcc\bin` for mspgcc.
- Ensure that there is a semicolon before and after `C:\mspgcc\bin`.

[1]http://processors.wiki.ti.com/index.php/PAN1315EMK_User_Guide#RF3_Connector
[2]http://sourceforge.net/apps/mediawiki/mspgcc/index.php?title=MSPGCC_Wiki
[3]http://www.sliksvn.com/en/download
[4]http://tortoisesvn.net/downloads.html
[5]http://sourceforge.net/projects/mspgcc/files/Windows/mingw32/
[6]http://gnuwin32.sourceforge.net/packages/make.htm
[7]http://www.python.org/getit/

1.3. Getting BTstack from SVN.
Use Subversion to check out the latest version. There are two approaches:

- On Windows: Use Tortoise SVN to checkout using the URL:

```
http://btstack.googlecode.com/svn/trunk/
```

- Use Subversion in a shell: Navigate to a folder where you would like to checkout BTstack, then type:

```
svn checkout http://btstack.googlecode.com/svn/trunk/
```

In both cases, Subversion will create the `btstack` folder and place the code there.

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

- Download the BTS file[8] for your PAN13xx 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 PAN13xx module you're using, you'll need to update `bluetooth_init_cc25...` in the Makefile to match the downloaded file.

1.5. Compiling the Examples.
Go to `btstack/MSP-EXP430F5438-CC256x/example` folder in command prompt and run make. If all the paths are correct, it will generate several .hex files. These .hex files are the firmware for the MSP430 and can be loaded onto the device, as described in the next section.

1.6. Loading Firmware.
To load firmware files onto the MSP430 MCU, you need a programmer like the MSP430 MSP-FET430UIF debugger or something similar. Now, you can use one of following software tools:

- MSP430Flasher software[9] (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[10]: An example session with the MSP-FET430UIF connected on OS X is given in following listing:

[8] http://processors.wiki.ti.com/index.php/CC256x_Downloads
[9] http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer
[10] http://mspdebug.sourceforge.net/

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

1.7. **Run the Example.** As a first test, we recommend the SPP Counter example (see Section 5.3). During the startup, the LEDs flash rapidly while the init script is transferred to the CC256x chipset. After that, the Experimenter 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. The SPP Counter doesn't use the display to keep the memory footprint small.

The HID demo has a fancier user interface - it uses a display to show the discovery process and connection establishment with a Bluetooth keyboard, as well as the text as you type.

After this quick intro, the main manual starts now.

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

Figure 1 shows the general architecture of a BTstack-based 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 7.2.

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

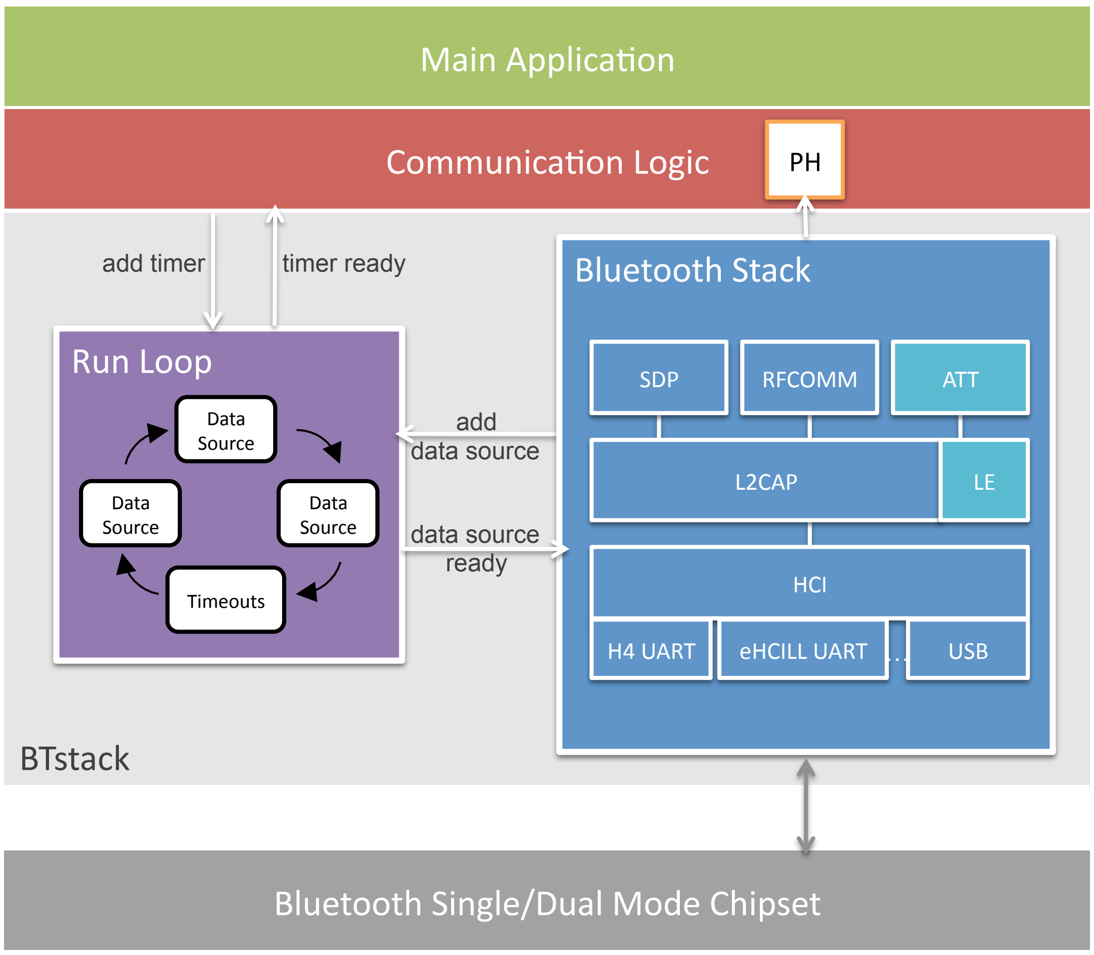

Figure 1. BTstack-based single-threaded application. The Main Application contains the application logic, e.g., reading a sensor value and providing it via the Communication Logic as a SPP Server. The Communication Logic is often modeled as a finite state machine with events and data coming from either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

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 provided protocols and services, as well as of the memory management and the run loop, 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 5.

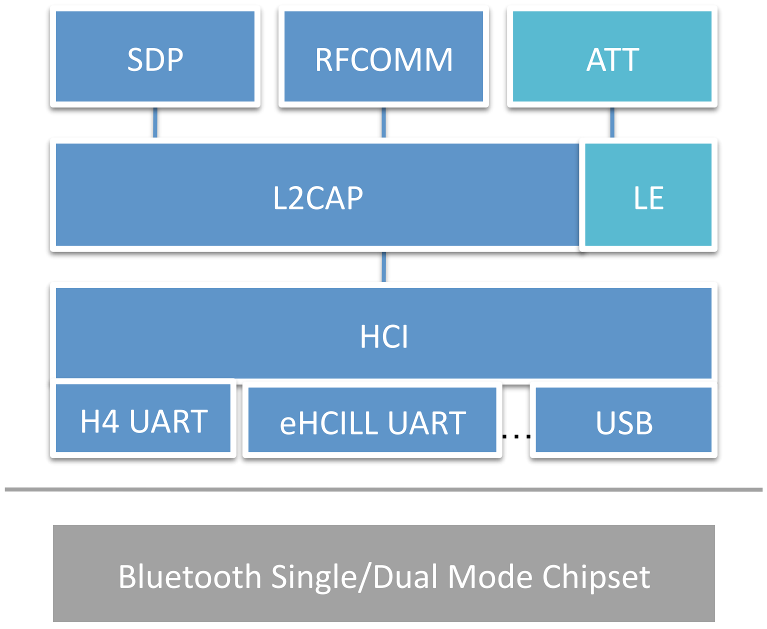

Figure 2. BTstack Protocol Architecture

3.1. **Protocols and services.** Figure 2 depicts protocols that BTstack implements: HCI, L2CAP, L2CAP-LE, RFCOMM, SDP, and ATT. The Host Controller Interface (HCI) provides a command interface to the Bluetooth chipset. The Logical Link Control and Adaptation Protocol (L2CAP) supports higher level protocol multiplexing and reassembly. The L2CAP Low Energy (LE) variant is optimized for connectionless data used by Bluetooth Low Energy devices. It is the base for the Attribute Protocol (ATT) of Bluetooth LE, which provides access to Services and Characteristics. The Radio frequency communication (RFCOMM) protocol provides emulation of serial ports over the L2CAP protocol. The Service Discovery Protocol (SDP) allows to discover services provided by a Bluetooth device. BTstack's API for HCI, L2CAP, RFCOMM and SDP is provided in Appendix A.

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.

```
#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 1. Memory configuration for an SPP service with a minimal L2CAP MTU.

3.2. **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 1.
- 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();
```

3.3. **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 structs *data_source_t* and *timer_source_t* respectively. Each of these structs contain a link 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, see Section 4.1 for an example. Note that BTstack expects to get called periodically to keep its time, see Section 6.1 for more on the tick hardware abstraction.

The Run loop API is provided in Appendix C. 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 23. 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);
```

3.4. **BTstack initialization.** To initialize BTstack you need to initialize the memory and the run loop as explained in Sections 3.2 and 3.3 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 6.3.

```
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 6.4.

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

Table 1. 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*

After these are ready, HCI is initialized like this:

```
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.

3.5. **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 B 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).
- 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 1.

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.

3.6. **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. Quick Recipes

4.1. **Periodic time handler.** As timers in BTstack are single shot, a periodic timer, e.g., to implement a counter or to periodically sample a sesor, is implemented by re-registering the *timer_source* in the *timer_handler* callback function, as shown in Listing 2.

4.2. **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[11] - Core Version 4.0, Volume 2, Part E, Chapter 5.4. In a HCI command, the OpCode is followed by parameter total length, and the actual parameters.

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

BTstack provides the *hci_cmd_t* struct as a compact format to define HCI command packets, see Listing 3, and `include/btstack/hci_cmds.h` file in the source code. The OpCode of a command can be calculated using the OPCODE macro.

```
#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 2. Periodic counter

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

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

extern const hci_cmd_t hci_write_local_name;
...
```

Listing 3. hci_cmds.h defines HCI command template.

```
#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 4. hci.h defines possible OGFs used for creation of a HCI command.

Table 2. Supported Format Specifiers of HCI Command Parameter

Format Specifier	Description
"1"	8 bit value
"2"	16 bit value
"H"	HCI handle
"3"	24 bit value
"4"	32 bit value
"B"	Bluetooth address
"D"	8 byte data block
"E"	Extended Inquiry Information 240 octets
"N"	UTF8 string, null terminated
"P"	16 byte PIN code or link key
"A"	31 bytes advertising data
"S"	Service Record (Data Element Sequence)

Listing 4 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.

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 2 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.

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

Listing 6 illustrates how to 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. All higher level functions in BTstack for the L2CAP and RFCOMM APIs manage this automatically. The main use of HCI commands in application is during the startup phase. At this time, no L2CAP or higher level data is sent, and the setup is usually done in the packet handler where the reception of the last command complete event triggers sending of the next command, hereby asserting that the Bluetooth module is ready and the outgoing buffer is free as shown in Listing 7 taken from `MSP-EXP430F5438-CC256x/example-ble/ble_server.c`.

```
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.

```
void packet_handler (uint8_t packet_type, uint16_t channel, uint8_t
   *packet, uint16_t size){
    ...
    switch (event) {
        ..
        case HCI_EVENT_COMMAND_COMPLETE:
            ...
            if (COMMAND_COMPLETE_EVENT(packet,
                hci_read_local_supported_features)){
                hci_send_cmd(&hci_set_event_mask, 0xffffffff, 0
                    x20001fff);
                break;
            }
            if (COMMAND_COMPLETE_EVENT(packet, hci_set_event_mask)){
                hci_send_cmd(&hci_write_le_host_supported, 1, 1);
                break;
            }
            if (COMMAND_COMPLETE_EVENT(packet,
                hci_write_le_host_supported)){
                hci_send_cmd(&hci_le_set_event_mask, 0xffffffff, 0
                    xffffffff);
                break;
            }
            if (COMMAND_COMPLETE_EVENT(packet, hci_le_set_event_mask
                )){
                hci_send_cmd(&hci_le_read_buffer_size);
                break;
            }
            ...
            break;
        ...
    }
}
```

Listing 7. Example of sending a sequence of HCI Commands

4.4. **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 8. 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 9. 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 8, we show how to resend data packets when credits or outgoing buffers become available.

```
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 INIT:
               if (packet[2] == HCI_STATE_WORKING) {
                    hci_send_cmd(&hci_write_local_name, "BTstack SPP
                        Counter");
                    state = W4_CONNECTION;
                }
               break;
          case W4_CHANNEL_COMPLETE:
               // if connection is successful, make device
                   undiscoverable
               hci_discoverable_control(0);
          ...
     }
 }
```

Listing 10. Setting device as discoverable. OFF by default.

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 9 for a RFCOMM example.

4.5. **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 sending the *hci_write_local_name* command. To save energy, you may set the device as undiscoverable again, once a connection is established. See Listing 10 for an example.

4.6. **Discover remote devices.** To scan for remote devices, the *hci_inquiry* command is used. After that, the Bluetooth devices actively scans for other devices and reports these as 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 11.

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 5.2.

```
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");
               break;
          ...
     }
}
```

Listing 12. Answering authentication request with PIN 0000.

4.7. **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). Bluetooth authentication is normally done with PIN codes. A PIN code is an ASCII string up to 16 characters in length. User is required to enter the same PIN code on both devices. The described above procedure is called pairing. See Listing 12 for providing a PIN.

Once the user has entered the PIN code, both devices will generate a link key. The link key can be stored either in the Bluetooth module themself or in a persistent storage, see Section 6.4. The next time, both devices will use

```
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 11. Discovering remote Bluetooth devices.

previously generated link key. Please note that the pairing must be repeated if the link key is lost by one device.

4.8. **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 13. 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 13.

4.9. **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 14. Providing an L2CAP service.

```
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 15. RFCOMM handler for outgoing RFCOMM channel.

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 14 provides L2CAP service example code.

4.10. **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

```
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 16. Providing RFCOMM service.

rfcomm_create_channel_intern-al 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 4.12.

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 15.

4.11. **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 16 provides the RFCOMM service example code.

4.12. **Slowing down RFCOMM data reception.** RFCOMM has a mandatory credit-based flow-control that 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 17, 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);
    rfcomm_register_service_internal(NULL, rfcomm_channel_nr, 100);
}
```

LISTING 17. RFCOMM service with automatic credit management.

```
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 18. RFCOMM service with manual credit management.

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

Listing 19. Providing new credits

```
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 20. Creating record with the data element (*de_**) functions.

If the management of credits is manual, credits are provided by the application such that it can manage its receive buffers explicitly, see Listing 18.

Manual credit management is recommended when received RFCOMM data cannot be processed immediately. In the SPP flow control example in Section 5.5, 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 19. 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.

4.13. **Create SDP records.** BTstack contains a complete SDP server and allows to register SDP records. An SDP record is a list of SDP Attribute {*ID, Value*} pairs that are stored in a Data Element Sequence (DES). The Attribute ID is a 16-bit number, the value can be of other simple types like integers or strings or can 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 20 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*.

4.14. **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 A.5. 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 21.

```
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 21. Searching RFCOMM services on a remote device.

5. Examples

The `MSP-EXP430F5438-CC256x` folder in BTstack repository currently includes the following examples for the MSP430F5438 Experimenter Board:

- UART example:
  - *led_counter*: provides UART and timer interrupt without Bluetooth.
- GAP example:
  - *gap_inquiry*: uses GAP to discover surrounding Bluetooth devices and then requests their remote name.
- SPP Server examples :
  - *spp_counter*: provides a virtual serial port via SPP and a periodic timer over RFCOMM.
  - *spp_accel*: provides a virtual serial port via SPP. On connect, it sends the current accelerometer values as fast as possible.
  - *spp_flowcontrol*: provides a virtual serial port via SPP with manual RFCOMM credit management. Delayed processing of received data is simulated with the help of a periodic timer.
- HID Host example:
  - *hid_demo*: on start, the device does a device discovery and connects to the first Bluetooth keyboard it finds, pairs, and allows to type on the little LCD screen.
- Low Energy example:
  - *ble_server*: provides a ready-to-run example for a test Peripheral device. It assumes that a PAN1323 or 1326 module with a CC2564 chipset is used.

In all examples the debug UART port is configured at 57600 bps speed.

5.1. led_counter: UART and timer interrupt without Bluetooth.

The example demonstrates how to setup hardware, initialize BTstack without Bluetooth, provide a periodic timer to toggle an LED and print number of toggles as a minimal BTstack test.

5.1.1. *Periodic Timer Setup.*

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

    // toggle LED
    LED_PORT_OUT = LED_PORT_OUT ^ LED_2;

    // re-register timer
    run_loop_register_timer(ts, HEARTBEAT_PERIOD_MS);
}
```

Listing 22. Periodic counter

```
void timer_setup(){
    // set one-shot timer
    heartbeat.process = &timer_handler;
    run_loop_register_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
}

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

    // go!
    run_loop_execute();

    // happy compiler!
    return 0;
}
```

Listing 23. Run loop execution.

As timers in BTstack are single shot, the periodic counter is implemented by re-registering the *timer_source* in the *heartbeat_handler* callback function. The general setup is explained in Section 4.1. Listing 22 shows *heartbeat_handler* adapted to periodically toggle an LED and print number of toggles.

5.1.2. *Turn On and Go.* Listing 23 shows how to setup and start the run loop. For hardware and BTstack setup, please check the source code.

5.2. **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.

5.2.1. *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, W4_INQUIRY_MODE_COMPLETE, and ACTIVE.

In INIT, the application enables the extended inquiry mode, which includes RSSI values, and transits to W4_INQUIRY_MODE_COMPLETE state.

In W4_INQUIRY_MODE_COMPLETE, after the inquiry mode was set, 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 please check Section 4.6 and the source code.

```
void btstack_setup(void){
    btstack_memory_init();
    run_loop_init(RUN_LOOP_EMBEDDED);

    // init HCI
    hci_transport_t    * transport = hci_transport_h4_dma_instance();
    bt_control_t       * control   = bt_control_cc256x_instance();
    hci_uart_config_t * config  = hci_uart_config_cc256x_instance();
    remote_device_db_t * remote_db = (remote_device_db_t *) &
        remote_device_db_memory;
    hci_init(transport, config, control, remote_db);
    hci_register_packet_handler(packet_handler);

    // init L2CAP
    l2cap_init();
    l2cap_register_packet_handler(packet_handler);

    // init RFCOMM
    rfcomm_init();
    rfcomm_register_packet_handler(packet_handler);
    rfcomm_register_service_internal(NULL, rfcomm_channel_nr, 100);

    // 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");
    sdp_register_service_internal(NULL, service_record_item);
}
```

Listing 24. SPP service setup

5.3. 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.

5.3.1. *SPP Service Setup.* SPP is based on RFCOMM, a Bluetooth protocol that emulates RS-232 serial ports. To access an RFCOMM serial port on a remote device, a client has to query its Service Discovery Protocol (SDP) server. The SDP response for an SPP service contains the RFCOMM channel number. To provide an SPP service, you need to initialize memory (Section 3.2) and the run loop (Section 3.3), setup HCI (Section 3.4) and L2CAP, then register an RFCOMM service and provide its RFCOMM channel number as part of the Protocol List attribute of the SDP record . Example code for SPP service setup is provided in Listing 24. The SDP record created by *sdp_create_spp_service* consists of a basic SPP definition that uses provided RFCOMM channel ID and service name. For more details, please have a look at it in `include/btstack/sdp_util.c`. The SDP record is created on the fly in RAM and is deterministic. To preserve valuable RAM, the result can be stored as constant data inside the ROM.

5.3.2. *Periodic Timer Setup.* The heartbeat handler increases the real counter every second, as shown in Listing 25. The general setup is explained in Section 4.1.

```
#define HEARTBEAT_PERIOD_MS 1000

void  theartbeat_handler(timer_source_t *ts){
    real_counter++;
    // re-register timer
    run_loop_register_timer(ts, HEARTBEAT_PERIOD_MS);
}
```

Listing 25. Periodic counter

5.3.3. *Bluetooth logic.* The Bluetooth logic is implemented as a state machine within the packet handler, see Listing 26. In this example, the following states are passed sequentially: INIT, W4_CONNECTION, W4_CHANNEL_COMPLETE, and ACTIVE.

In INIT, upon successful startup of BTstack, the local Bluetooth name is set, and the state machine transits to W4_CONNECTION.

The W4_CONNECTION state handles authentication and accepts incoming RFCOMM connections. It uses a fixed PIN code "0000" for authentication. An incoming RFCOMM connection is accepted, and the state machine progresses to W4_CHANNEL_COMPLETE. 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.

In W4_CHANNEL_COMPLETE state, an error in the channel establishment fails (rare case, e.g., client crashes), the application returns to the W4_CONNECTION state. On successful connection, the RFCOMM channel ID and MTU for

```
void prepareData(void){
   counter_to_send++;
}

void tryToSend(void){
     // see Quick Recipe 5.4, Listing 8
}

void packet_handler (uint8_t packet_type, uint8_t *packet, uint16_t
    size){
     ...
     switch(state){
          case INIT:
               if (packet[2] == HCI_STATE_WORKING) {
                    hci_send_cmd(&hci_write_local_name, "BTstack Demo");
                    state = W4_CONNECTION;
               }
               break;

          case W4_CONNECTION:
               switch (event) {
                    case HCI_EVENT_PIN_CODE_REQUEST:
                         // see Quick Recipe 5.7
                         break;
                    case RFCOMM_EVENT_INCOMING_CONNECTION:
                         // see Quick Recipe 5.11
                         state = W4_CHANNEL_COMPLETE;
                         break;
               }

          case W4_CHANNEL_COMPLETE:
               if (event != RFCOMM_EVENT_OPEN_CHANNEL_COMPLETE) break;
               // see Quick Recipe 5.11
               // state: W4_CONNECTION on failure, otherwise ACTIVE
               break;

          case ACTIVE:
               // see Quick Recipe 5.4, Listing 9
               // state: W4_CONNECTION on channel closed
               break;
          ...
     }
}
```

Listing 26. SPP Server - Heartbeat Counter over RFCOMM.

this channel are made available to the heartbeat counter and the state machine transits to ACTIVE.

While in the ACTIVE state, the communication between client and the application takes place. In this example, the timer handler increases the real counter every second. The packet handler tries to send this information when an RFCOMM credit (RFCOMM_EVENT_CREDITS) or an HCI packet sent event (DAEMON_EVENT_HCI_PACKET_SENT) are received. These two events represent two orthogonal mechanisms that deal with flow control. A packet can only be sent when an RFCOMM credit is available and the internal BTstack outgoing packet buffer is free.

5.4. **spp_accel: SPP Server - Accelerator Values.**

In this example, the server tries to send the current accelerometer values. It does not use a periodic timer, instead, it sends the data as fast as possible.

5.5. **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.

5.5.1. *SPP Service Setup.*

Listing 18 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.

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

Listing 27. Heartbeat handler with manual credit management.

```
void packet_handler (void * connection, uint8_t packet_type,
    uint16_t channel, uint8_t *packet, uint16_t size){
    ...
    if (packet_type == RFCOMM_DATA_PACKET){
        packet[size] = 0;
        puts( (const char *) packet);
        rfcomm_send_credit = 1;
        return;
    }
    ...
}
```

Listing 28. Packet handler with manual credit management.

5.5.2. *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. 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 27. The general setup is explained in Section 4.1.

5.5.3. *Bluetooth logic.* The packet handler now additionally handles RFCOMM data packets and sets a flag for granting a new credit, see Listing 28.

6. Porting to Other Platforms

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

6.1. **Tick Hardware Abstraction Layer.** BTstack requires a way to learn about passing time. In an embedded configuration, the following functions have to be provided. The *hal_tick_init* and the *hal_tick_set_handler* functions 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);
```

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

Overall, the struct *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.

6.3. **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.

6.3.1. *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.

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

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

6.4. **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 29.

```
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 29. Persistent Storage Interface.

7. 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.

7.1. **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.

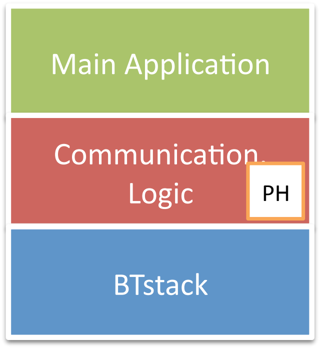

Figure 3. BTstack in single-threaded environment.

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.

7.2. **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:

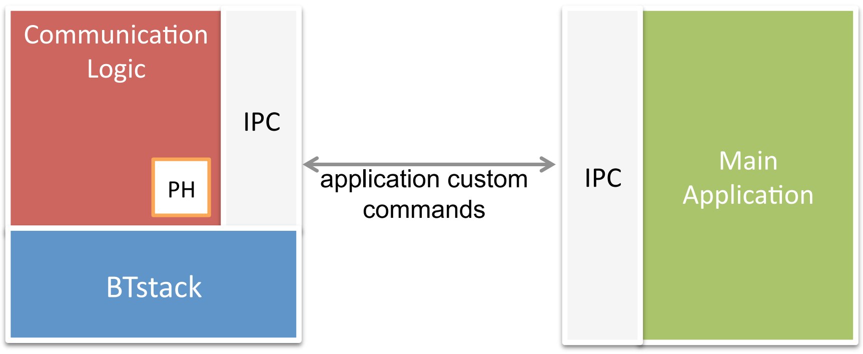

FIGURE 4. BTstack in multi-threaded environment - monolithic solution.

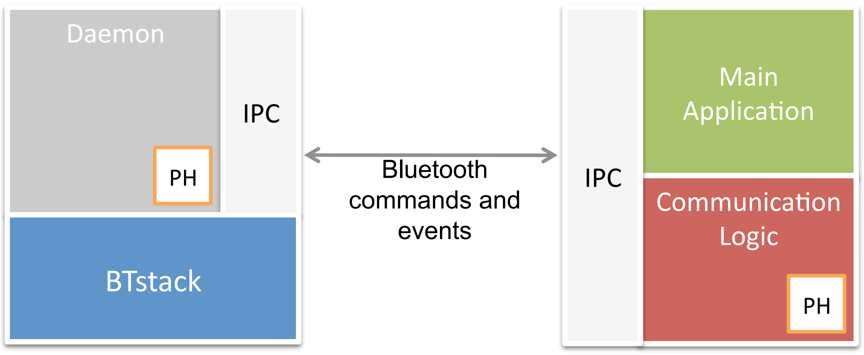

FIGURE 5. BTstack in multi-threaded environment - solution with daemon.

- 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 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. BTstack Protocol API

A.1. **Host Controller Interface (HCI) API.**

```
 // Set up HCI.
void hci_init(hci_transport_t *transport, void *config, bt_control_t
    *control, remote_device_db_t const* remote_device_db);

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

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

// Allows to control if device is dicoverable. OFF by default.
void hci_discoverable_control(uint8_t enable);

// 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, ...);

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

A.2. **L2CAP API.**

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

// 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));

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

// Disconencts L2CAP channel with given identifier.
void l2cap_disconnect_internal(uint16_t local_cid, uint8_t reason);

// 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);

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

// 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);

// Unregisters L2CAP service with given PSM.  On embedded systems,
// use NULL for connection parameter.
void l2cap_unregister_service_internal(void *connection, uint16_t
    psm);

// 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);
```

A.3. **RFCOMM API.**

```
// Set up RFCOMM.
void rfcomm_init(void);

// 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));

// 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 connection is an RFCOMM channel. The channel will
// automatically provide enough credits to the remote side.
void rfcomm_create_channel_internal(void *connection, bd_addr_t *
    addr, uint8_t channel);

// 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 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);

// Disconencts RFCOMM channel with given identifier.
void rfcomm_disconnect_internal(uint16_t rfcomm_cid);

// 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 automatically provide
// enough credits to the remote side.
void rfcomm_register_service_internal(void *connection, uint8_t
    channel, uint16_t max_frame_size);

// 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);

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

// Accepts/Deny incoming RFCOMM connection.
void rfcomm_accept_connection_internal(uint16_t rfcomm_cid);
void rfcomm_decline_connection_internal(uint16_t rfcomm_cid);
```

```
// 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);

// 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);
```

A.4. **SDP API.**

```
// Set up SDP.
void sdp_init(void);

// Register service record internally - this version does not copy
// the record therefore it must be forever accessible.
// Preconditions:
//     - AttributeIDs are in ascending order;
//     - ServiceRecordHandle is first attribute and valid.
// @returns ServiceRecordHandle or 0 if registration failed.
uint32_t sdp_register_service_internal(void *connection,
    service_record_item_t * record_item);

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

A.5. **SDP Client API.**

```
/* SDP Client */

// 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);



/* SDP Parser */

// Basic SDP Parser event type.
typedef enum sdp_parser_event_type {
    SDP_PARSER_ATTRIBUTE_VALUE = 1,
    SDP_PARSER_COMPLETE,
} sdp_parser_event_type_t;

typedef struct sdp_parser_event {
    uint8_t type;
} sdp_parser_event_t;

// SDP Parser event to deliver an attribute value byte by byte.
typedef struct sdp_parser_attribute_value_event {
    uint8_t type;
    int record_id;
    uint16_t attribute_id;
    uint32_t attribute_length;
    int data_offset;
    uint8_t data;
} sdp_parser_attribute_value_event_t;

// SDP Parser event to indicate that parsing is complete.
typedef struct sdp_parser_complete_event {
    uint8_t type;
    uint8_t status; // 0 == OK
} sdp_parser_complete_event_t;

// Registers a callback to receive attribute value data and parse
// complete event.
void sdp_parser_register_callback(void (*sdp_callback)(
    sdp_parser_event_t* event));
```

```
/* SDP Queries */

// Basic SDP Query event type.
typedef struct sdp_query_event {
    uint8_t type;
} sdp_query_event_t;

// SDP Query event to indicate that query is complete.
typedef struct sdp_query_complete_event {
    uint8_t type;
    uint8_t status; // 0 == OK
} sdp_query_complete_event_t;



/* SDP Query for RFCOMM */

// 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;

// 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);

// 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);

// 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);
```

Appendix B. 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.

Table 3. 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

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 3.6 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. 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 5. 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 C. Run Loop API

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

// 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));

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

// 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);

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

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

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

// Sets how many milliseconds has one tick.
uint32_t embedded_ticks_for_ms(uint32_t time_in_ms);

// Queries the current time in ticks.
uint32_t embedded_get_ticks(void);

// 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);
```

Listing 30. Run Loop API.

Appendix D. Revision History

Rev	Date	Comments
1.1	August 30, 2013	Introduced SDP client. Updated Quick Recipe on "Query remote SDP service".