RW BLE Host Interface Specification

Interface Specification

RW-BLE-HOST-IS

Version 9.0

2017-03-09

RivieraWaves
http://www.rivierawaves.com/

Revision History

Version	Date	Revision Description	Author
0.1	2011-03-03	Initial release	CI
1.0	2011-12-15	Small updates post qualification process	CI
8.0	2015-06-29	Updated for BLE 4.2	FBE
9.0	2017-03-09	Updated for BLE 5	LT

Table of Contents

1 Overview

1.1 Document Overview

This document describes the RivieraWaves Bluetooth Low Energy non-standard Higher Layer interface.

Its purpose is to explain the format of the interface packets passed between the Application/Tool through the physical interface with the embedded platform running in Fully Embedded mode.

By Higher Layers please understand GAP, GATT and SM, also the Profiles implementation - the Profiles have been developed with an API for a future application layer, and for now, for test and integration reasons, are treated as Higher Layers, using GAP and GATT.

NOTE: The way the Higher Layer interface API and its interface are built is specific to RivieraWaves Bluetooth Low Energy IP design. The API is described in the referenced IS documents and also easily found in the **module_task.c/.h** *files in every higher layer module.*

1.2 Higher Layer Interface Overview

The physical transport layer for these Higher Layer interface messages between a Fully Embedded BLE platform and a controlling tool on a computer is the same as for the Host Controller Interface for a Split Embedded platform (UART for now).

The reception and sending of these interface messages is adapted from the HCI module.

For UART transport, the HCI uses a single byte at the beginning of the packet to identify the type of the packet (Command, Event, ACL or SCO data packets). In order to identify the fully embedded interface messages, the choice was made to use such a byte, different from those 4.

There was no need to have a complex classification of messages like in HCI, the same byte is used for receiving and sending messages (0x05). This suffices because the only two types of messages that exist in the Higher Layer or Profiles API are *requests* and *events* (responses or indications that were not triggered by a request). The ***request*** denomination was chosen to avoid the term command, very much used in HCI, and also *event* is not really used, but ***response*** or ***indication***. The data packet does not exist anymore because ACL packets for HCI are actually the Protocol Data Units (PDUs) that are sent by Higher Layers through L2CAP to the Lower Layers that are otherwise directly interfaced through HCI in a Split system.

In order to not have additional processing of sent and received messages, the structure of the kernel message is directly used in the controlling tool to format the packet to be sent through this higher layer interface. The kernel message and its use for this interface is described in the following chapter.

2 Kernel Message Structure

The Kernel message structure can be found in *ke_msg.h*. It contains the information passed between the different kernel tasks while the firmware is running, and when a destination task is outside of the system, the message will be handled by the HCI or Higher Layer interface module depending on the embedded system (split/fully embedded).

```
struct ke_msg
{
    ke_msg_id_t     id;         ///< Message id.
    ke_task_id_t    dest_id;    ///< Destination kernel identifier.
    ke_task_id_t    src_id;     ///< Source kernel identifier.
    uint16_t        param_len;  ///< Parameter embedded struct length.
    uint32_t        param[1];   ///< Parameter embedded struct. Must be word-aligned.
};
```

2.1 Classic HCI

The Host Controller Interface uses the kernel messages in the following manner:

- When kernel messages from Lower Layers are destined to a Higher Layer, or vice-versa, the source and destination task identifiers allow the Kernel to know that the message it is redirecting should go to a task that exists outside the system, thus requiring to be sent through HCI. A message handler in the HCI is called and given the kernel message in question as parameter, and then the kernel message is transformed into a HCI packet and sent through the interface to the other part of the system, and then freed.
- When an HCI packet is received through the physical interface, according to its header information (command opcode or event code) it will cause an allocation of a kernel message with a corresponding source and destination task identifiers and a corresponding parameter structure. The HCI packet will be unpacked correctly and the kernel message parameter structure will be filled with the unpacked HCI parameter values. This kernel message is then left to the kernel to be sent to the appropriate destination task and processed.

The packing and unpacking operations are necessary for two main reasons:

- The structures of parameters in C are sometimes padded with empty space due to unaligned parameter sizes (may depend on compilers). Thus the structure space pointer cannot be directly used for the HCI packet buffer.
- The endianness of the two system components may be different and HCI must ensure Little to Big endian parameter values in the packets.
-

2.2 Higher Layer Interface

For a rapid development, the classic HCI principles were used:

- The transport function was adapted to add the 0x05 byte before all packets to keep the HCI reception and send functions format.
- The message send handler called by the kernel to handle kernel messages destined to an outside task is greatly simplified
- There is no kernel message packing or unpacking because everything from *id* parameter -1 is sent through HCI as is

Thus, the padding issues and task identifiers are handled in the API for this interface in the testing tools.

The API message structures are described in corresponding Interface Specification files. It allows understanding how the API should be built (where to pad a packet that is built to map onto a kernel message, what task identifiers to use in the packet, etc.)

The API is built in an intuitive manner:

- using generic *command complete events* for simple requests for which the user only needs to know if they were done successfully or not
- using responses corresponding to more complex requests
- Using indication messages informing the user of an event that requires a response from the user (information, authorization, accepting an action, etc.)

The only needed information to build API messages and their interpreters is the kernel message structure and the structures of the different message parameters.

2.3 Caution

Task Identifiers are simple shifted indexes for tasks that are instantiated only once. But for those that are instantiated per connection, they are re-indexed with the connection index e.g: (**TASK_GAPC <<8) + *connection_index***.

When a kernel message is received outside the Fully Embedded system, the task identifier will be indexed with the connection index which **must thus be known** in the tool handling sending and receiving messages to the right tasks.

Generally in the first messages corresponding to a connection, the index is present so it can be recovered and used in a tool to build task identifiers for sending requests, etc.

3 Abbreviations

Abbreviation	Original Terminology
API	Application Programming Interface
BLE	Bluetooth Low Energy
GAP	Generic Access Profile
GATT	Generic Attribute Profile
RW	RivieraWaves

References

[1]	**Title**	RW BLE GAP Interface Specification		
	Reference	RW-BLE-GAP-IS.pdf		
	Version	9.0	**Date**	2017-03-09
	Source	RivieraWaves		

[2]	**Title**	RW BLE GATT Interface Specification		
	Reference	RW-BLE-GATT-IS.pdf		
	Version	9.0	**Date**	2017-03-09
	Source	RivieraWaves		

[3]	**Title**	RW BLE Find Me Profile Interface Specification		
	Reference	RW-BLE-FMP-IS.pdf		
	Version	9.0	**Date**	2017-03-09
	Source	RivieraWaves		

[4]	**Title**	RW BLE Health Thermometer Profile Interface Specification		
	Reference	RW-BLE-HTP-IS.pdf		
	Version	9.0	**Date**	2017-03-09
	Source	RivieraWaves		

[5]	**Title**	RW BLE Proximity Profile Interface Specification		
	Reference	RW-BLE-PXP-IS.pdf		
	Version	9.0	**Date**	2017-03-09
	Source	RivieraWaves		