HCI Software Design Document

Document 2009-0011
Revision 1.1
September 25, 2015

Copyright © 2009-2015 Wicentric, Inc. All rights reserved.

Wicentric Confidential

Table of Contents

1 Introduction

This document describes the software design of the host controller interface (HCI) subsystem of the Wicentric Bluetooth LE protocol stack.

2 Design Considerations

2.1 Support Different HCI Topologies

The HCI subsystem is designed to support both single-chip and dual-chip HCI topologies.

2.2 Flexible Feature Configuration

The HCI subsystem is designed to allow the inclusion and exclusion of features such as fragmentation and reassembly, admission control, and data queueing.

2.3 Platform and Transport Independence

The HCI subsystem is designed to be easily portable to different platforms and different transports.

3 System Context

Figure 1 shows the context of the HCI subsystem in the Bluetooth LE stack.

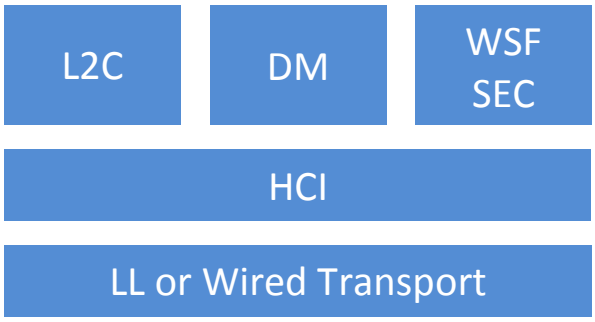

Figure 1. Bluetooth LE stack software system.

HCI interfaces to L2C to send and receive ACL data packets. HCI interfaces to DM for most HCI commands and events. HCI interfaces to the WSF security interface for encryption and random number generation. HCI sits above and communicates with either the link layer (LL) in a single chip system or a wired transport driver in a dual-chip system.

4 Subsystem Architecture

Figure 2 shows the different modules that make up the HCI subsystem.

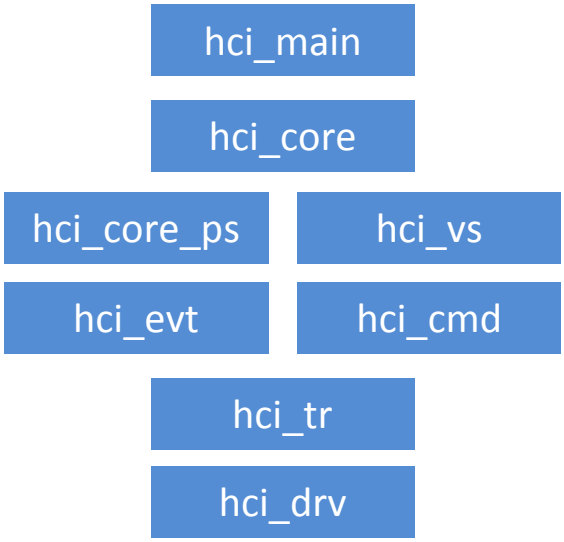

Figure 2. Subsystem architecture.

Modules are as follows:

- **hci_main**: This is the main module of the HCI subsystem. It contains the API functions for initialization and registration. This module is platform independent.
- **hci_core**: This module implements core platform independent HCI features for transmit data path, fragmentation, reassembly, and connection management.
- **hci_core_ps**: This module implements core platform-dependent HCI features for transmit data path, receive data path, the "optimization" API, and the main event handler. This module contains separate implementations for dual chip and single chip. The dual chip implementation is platform independent.
- **hci_vs**: This module implements vendor-specific features for the reset sequence and vendor-specific HCI commands and events. The example "generic" dual chip implementation is platform independent.
- **hci_evt**: This module implements parsing and translation of HCI event data structures. This module contains separate implementations for dual chip and single chip. The dual chip implementation is platform independent.
- **hci_cmd**: This module builds and translates HCI command data structures. It also implements command flow control. This module contains separate implementations for dual chip and single chip. The dual chip implementation is platform independent.
- **hci_tr**: This module implements the transport layer of HCI. In the transmit path this module sends HCI commands and ACL packets to wired transport or the link layer. In the receive path this module performs reassembly (if necessary) and sends complete ACL packets and HCI events to the upper layers of HCI. This module is platform dependent.

- **hci_drv**: This module contains driver APIs for dual chip wired transport. This module is platform dependent.

4.1 ACL Transmit Data Path

The ACL transmit data path covers the ACL packet data flow from the stack to the wired transport or link layer. The data path implements several HCI procedures which may optionally be included depending on the requirements of the platform. Figure 3 shows the typical operation of this data path.

Data packets passed to function HciSendAclData() are queued. If the packet is longer than the maximum ACL packet buffer size of the controller then the packet is fragmented. Then packets are passed to an admission control mechanism which implements HCI packet flow control, processing of HCI Number of Completed Packets events, and sending of flow control events to the HCI client. Function HciTrSendAclData() is a transport-specific function that sends a single ACL packet to the wired transport or link layer.

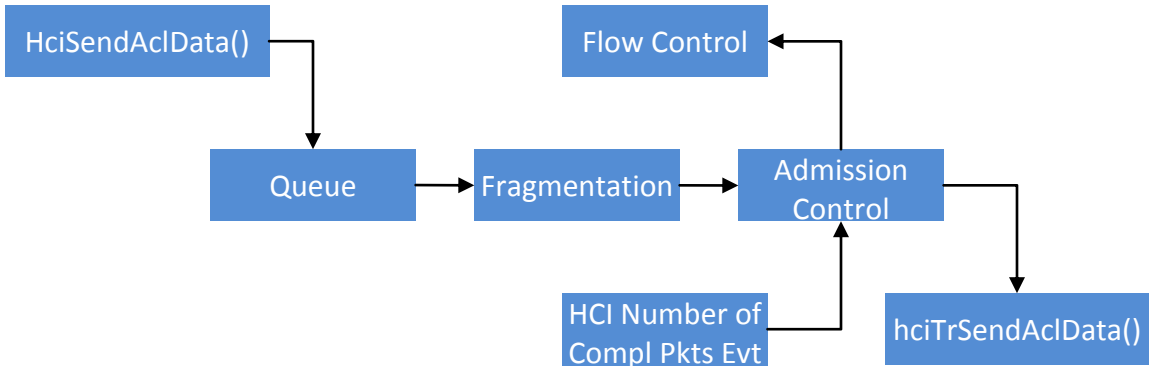

Figure 3. Typical transmit data path.

4.2 ACL Receive Data Path

The ACL receive data path covers ACL packet flow from the link layer or wired transport to the stack. The data path implements several HCI procedures which may optionally be included depending on the requirements of the platform. Figure 4 shows the typical operation of this data path.

First the packet is received from the wired transport or link layer. If the packet is fragmented it is reassembled. Then the packet is queued to the HCI event handler. When the HCI event handler runs it processes the queue and calls the client callback function to send the packet to the stack.

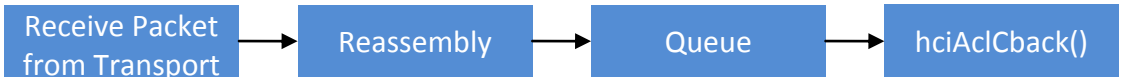

Figure 4. Typical receive data path.

4.3 Command Data Path

The command data path covers commands sent from the stack to the wired transport or link layer. Figure 5 shows the typical operation of this data path. The stack calls an HCI API function to send an HCI command. If a command is already pending the command is queued. When a Command Complete or Command Status event is received function hciTrSendCmd() is called to send the next command to the wired transport or link layer.

For wired transport, a timeout handles a non-responsive controller that does not send back an event. If a timeout occurs, an HCI event for the corresponding command is internally generated with an error status.

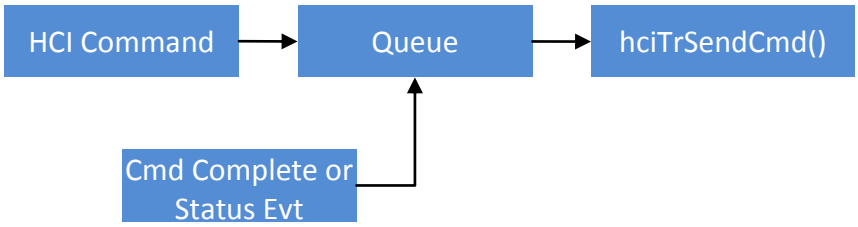

Figure 5. Typical command data path.

4.4 Event Data Path

The event data path covers events sent from the link layer or wired transport to the stack. Figure 6 shows the typical operation of this data path.

After an HCI event is received from the wired transport or link layer it is sent in a message to the HCI event handler. When the HCI event handler executes it processes the message and calls the HCI client callback, if applicable.

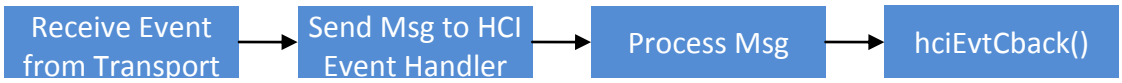

Figure 6. Typical event data path.

4.5 Directory Structure

The code is implemented in the directory structure shown below.

```
sw
  hci
    include
```

```
        hci_cmd.h, hci_core.h, hci_drv.h, hci_evt.h, hci_tr.h
    🗁dual-chip
        hci_core_ps.c, hci_core_ps.h, hci_cmd.c, hci_evt.c, hci_vs.c
    🗁generic
        hci_core.c, hci_tr.c, hci_drv.c
    🗁exactle
        hci_cmd.c, hci_core_ps.c, hci_core_ps.h, hci_evt.c, hci_tr.c, hci_vs.c
    🗁someplatform

🗁stack
    🗁hci
        hci_main.c, hci_main.h
    🗁include
        hci_api.h, hci_defs.h
```

Directory /sw/hci contains platform and transport dependent code. Directory /sw/stack/hci contains platform independent code.

5 Detailed Design

This section describes the platform and transport independent portion of the design.

5.1 Main Event Handler

A WSF event handler is used to process events and messages such as the HCI command timeout event (if applicable) and received HCI events and ACL data. An example event handler for the dual chip implementation is in function HciCoreHandler() in file hci_core_ps.c. This function does the following:

- Handle HCI command timeout
- Process incoming data in the HCI rx queue
- Handle the reset sequence during HCI event processing
- Execute the reassembly function for incoming ACL data.
- Pass reassembled ACL packets to the stack via callback.

5.2 Reset Sequence

The reset sequence procedure is executed on system initialization. Its purpose is to send an HCI reset to the controller or link layer and also read static information from the controller or link layer. The reset sequence procedure typically sends an HCI Reset command followed by several HCI commands in sequence. For a dual chip implementation, this function sends a sequence of HCI commands which may be specific to the particular manufacturer of the controller. It also parses received HCI Command Complete events and uses these events to trigger transmission of the next command in the sequence.

The dual chip example implementation is in function hciCoreResetSequence() in file hci_vs.c.

5.3 Optimization API

The optimization API is an optimized interface for certain HCI commands which simply read a value. The stack uses these functions rather than their corresponding functions in the command interface. The functions in the optimization API are described in more detail in [2]. The example dual chip implementation is in file hci_core_ps.c.

5.4 Connection Management

HCI needs to store state for each connection in order to manage fragmentation and reassembly and to properly account for controller ACL packet buffers. The following connection management functions are implemented in file hci_core.c:

- hciCoreConnOpen()
- hciCoreConnClose()
- hciCoreConnAlloc()
- hciCoreConnFree()
- hciCoreConnByHandle()
- hciCoreNextConnFragment()

5.5 Data Path Configuration

There are two functions in hci_core.c used to configure the receive and transmit data paths. Function HciSetAclQueueWatermarks() sets the high and low watermarks used for flow control in the transmit data path. When the number of queued buffers reaches the high watermark flow control is asserted. When the number of buffers reaches the low watermark flow control is released.

Function HciSetMaxRxAclLen() is used to set the maximum reassembled ACL packet size. The minimum value is set to 27, which is also the default value. To receive larger ACL packets, for example when SMP LE secure connections is used or larger ATT MTU sizes are used, this function must be called to set a higher value.

5.6 Fragmentation

ACL packet fragmentation is performed by functions in file hci_core.c. When a packet is transmitted and it is larger than the controller ACL packet size then the fragmentation procedure is started. The larger packet is broken into multiple smaller ACL packets up to the controller packet size in length. Fragments are sent from the original large ACL packet buffer so no new buffer allocation or data copy is required for fragmentation. However this does require special consideration for deallocation the large ACL buffer. When the transmission of each ACL fragment packet is complete (or upon transmission of an unfragmented ACL packet) function hciCoreTxAclComplete() must be called by the transport layer. This function frees the ACL packet buffer when fragmentation is complete.

5.7 Reassembly

ACL packet reassembly is performed by function hciCoreAclReassembly() in file hci_core.c. This function allocates a large buffer that will contain the entire reassembled packet and then copies received packet

fragments to this buffer to reassemble the packet. This function also performs a number of protocol and length checks to verify the received packet fragments are valid.

5.8 Vendor-specific Commands and Events

The HCI code is designed to accommodate vendor specific HCI commands and events. Functions to handle these commands and events can be added to hci_vs.c. The example implementation for dual chip contains the following placeholder functions:

- hciCoreVSCmdCmplRcvd(): Handle vendor specific HCI command complete events.
- hciCoreVsEvtRcvd(): Handle vendor specific HCI events.
- hciCoreHwErrorRcvd(): Perform internal HCI processing for hardware error events.
- HciVsInit(): Vendor-specific controller initialization.

6 Detailed Design, Dual Chip

6.1 Sending Commands

file hci_cmd.c implements the "command interface" API functions, which send HCI commands. These functions all have the same general procedure: Allocate a buffer, build an HCI command in the buffer, and call hciCmdSend() to send the command.

HCI commands are queued in queue hciCmdCb.cmdQueue and sent one at a time when each HCI event for the previous command has been received. Commands are sent to the transport layer by calling function hciTrSendCmd(). A timer is started when each command is sent to check for command timeout.

When a command timeout occurs…

6.1.1.1 void HciCmdRecvCmpl(uint8_t numCmdPkts)

This function is called when an HCI Command Complete or Command Status event is received. It stops the HCI command timer, increments hciCmdCb.numCmdPkts and then calls hciCmdSend() sends the next queued command.

*6.1.1.2 void hciCmdSend(uint8_t *pData)*

This function sends an HCI command. It operates according to the following pseudocode:

```
if pData != NULL
  wsfMsgEnq(&hciCmdCb.cmdQueue, pData)

if hciCmdCb.numCmdPkts > 0
  if p = wsfMsgDeq(&hciCmdCb.cmdQueue, &handlerId) != NULL
    hciCmdCb.numCmdPkts--
    hciCmdCb.cmdOpcode = (command opcode in buffer p)
    WsfTimerStartSec(&hciCmdCb.cmdTimer, timeout value)
   hciTrSendCmd(p)
```

6.2 Receiving Events

This function parses byte-oriented HCI event messages. It first parses the event type and length and decodes the event type into four categories: Command Status events, Command Complete events, LE events, and other events. It then calls different functions to process each of the four categories.

6.3 Transmit Data Path

*6.3.1.1 void hciCoreNumCmplPkts(hciNumCmplPktsEvt_t *pMsg)*

Process HCI Number of Completed Packets event. TBD.

6.4 Receive Data Path

6.4.1.1

*6.4.1.2 void hciCoreRecv(uint8_t msgType, uint8_t *pMsg)*

This function takes a received HCI event or ACL data packet and sends it to the HCI event handler. Parameter pData points to a WSF msg buffer. Parameter msgType is set to HCI_ACL_TYPE or HCI_EVT_TYPE. This function queues the buffer in hciCb.rxQueue and sets event HCI_EVT_RX for the HCI event handler

7 Detailed Design, UART Transport

This section describes the UART transport design. It describes the design in a general manner-- it does not cover platform-specific details of the design.

UART transport is implemented according to the Bluetooth HCI UART transport specification in [3]. According to the specification, each command, event, or ACL packet is preceded by a byte which indicates the message type:

Name	Value	Description
HCI_CMD_TYPE	1	HCI command.
HCI_ACL_TYPE	2	HCI ACL data.
HCI_EVT_TYPE	4	HCI event.

7.1.1 Data Types

7.1.1.1 Transport Receive Structure

The UART transport receive procedure typically requires a data structure to store several parameters:

- Received message type.
- Received message payload length.
- Received message header.
- Received message length so far.
- Pointer to received message buffer.
- Received message state.

7.1.2 Functions

*7.1.2.1 void hciTrSendAclData(uint8_t *pData)*

This function sends a complete HCI ACL packet to the transport. It reads the ACL packet payload length from the ACL packet header and calculates the total data length to send: The payload length plus the header length (HCI_ACL_HDR_LEN). It then sends the message type byte followed by the ACL packet to the UART driver by calling function hciDrvWrite(). If the complete packet is written to the driver function hciCoreTxAclComplete() is called to free the packet buffer.

*7.1.2.2 void hciTrSendCmd(uint8_t *pData)*

This function sends a complete HCI command to the transport. It reads the command payload length from the command header and calculates the total length to send: The payload length plus the header length (HCI_CMD_HDR_LEN). It then sends the message type byte followed by the HCI command to the UART driver by calling function hciDrvWrite(). If the complete packet is written to the driver function WsfMsgFree() is called to free the packet buffer.

7.1.2.3 void hciTrRecv(void)

This function receives a complete HCI event or ACL packet from the UART driver and sends it to the HCI layer of the stack. It is executed when data has been received by the UART driver. Since received UART data may be of any length a state machine is required to receive a complete message. Typical state machine operation is as follows:

1. Receive the first byte that indicates whether this is an event or ACL packet.
    a. If an event, receive the next two bytes which contain the event type and length.
    b. If an ACL packet, receive the next four bytes which contain the handle and length.
2. Allocate a WSF msg buffer for the received message and write received header bytes to it.
3. Receive message payload and write it to the buffer.
4. When the complete message is received, call hciCoreRecv() to send the message to HCI.

7.1.2.4 Driver Open and Close

There is no interface from the stack to HCI to open and close the HCI transport layer driver. Instead, these interfaces are platform-specific and called from the platform software system which is using the stack. The UART driver must be opened before the stack is initialized and can be closed when the platform stops using the stack.

8 References

1. Wicentric, “Bluetooth LE Stack System Architecture”, 2009-0005.
2. Wicentric, “HCI API”, 2009-0006.
3. Bluetooth SIG, “Specification of the Bluetooth System”, Version 4.2, December 2, 2015.

9 Definitions

ACL	Asynchronous Connectionless data packet

DM	Device Manager software subsystem
HCI	Host Controller Interface
L2C	L2CAP software subsystem
L2CAP	Logical Link Control Adaptation Protocol
LE	(Bluetooth) Low Energy
WSF	Wicentric Software Foundation software service and porting layer