exactLE Protocol Stack Porting Guide

Document 2010-0014
Revision 1.1
September 25, 2015

Table of Contents

1 Introduction

This document is the porting guide for Wicentric's exactLE Bluetooth low energy protocol stack. The porting process typically consists of two main steps:

1. Porting WSF interfaces and services to the target OS and software system.
2. Porting HCI to the target system and writing a transport driver, if applicable.

2 Porting WSF

2.1 About WSF

WSF is a simple OS wrapper, porting layer, and general-purpose software service used by the stack and embedded software system. The goal of WSF is to stay small and lean, supporting only the basic services required by the stack. It consists of the following:

- Event handler service with event and message passing.
- Timer service.
- Queue and buffer management service.
- Portable data types.
- Critical sections and task locking.
- Trace and assert diagnostic services.
- Security interfaces for encryption and random number generation.

WSF does not define any tasks but defines some interfaces to tasks. It relies on the target OS to implement tasks and manage the timer and event handler services from target OS tasks. WSF can also act as a simple standalone OS in software systems without an existing OS.

For a complete description of the WSF API see [1].

2.2 Porting Steps

Porting WSF typically consists of the following steps:

1. Create common data types for the target compiler.
2. Interface to a system timer to receive timer updates.
3. Implement WSF OS wrapper functions and interfaces.
4. Implement WSF diagnostics.

2.3 File Organization

WSF source code files are organized as shown below:

🗁wsf
  🗁common
  🗁include
  🗁generic

<target name>

wsf_assert.h	Assert interface
wsf_cs.h	Critical section interface
wsf_os.c	WSF OS wrapper implementation
wsf_os_int.h	Target-specific WSF interface
wsf_trace.h	Trace interface
wsf_types.h	Common data types

A new directory, typically named after the target system, is created in the WSF directory. This new directory contains the files required to implement the WSF port.

The 'common' and 'include' directories contain platform-independent files which typically do not need to be modified when porting. The 'generic' directory contains a generic port to ARM Cortex-M CPUs with WSF acting as a simple standalone OS. Files in this directory may be useful when porting to other ARM Cortex-M based platforms.

2.4 Common Data Types

The following common data types must be defined in file wsf_types.h:

Name	Description
int8_t	8 bit signed integer
uint8_t	8 bit unsigned integer
int16_t	16 bit signed integer
uint16_t	16 bit unsigned integer
int32_t	32 bit signed integer
uint32_t	32 bit unsigned integer
bool_t	Boolean integer

Note that these integer data types match the names used in C99. If C99 is used in the target system then include "stdint.h" in wsf_types.h instead of creating type definitions for the above types.

In addition, the following macros must be defined in wsf_types.h:

Name	Description
NULL	0
TRUE	1
FALSE	0

2.5 System Timer Interface

WSF has a timer service that is used by the protocol stack.

2.5.1 Initialization

The WSF timer service keeps time based on "ticks". The number of milliseconds per tick is configurable; recommended values are 10-100ms per tick. The ms per tick value is set via function WsfTimerInit().

2.5.2 Keeping Time

The target system updates the WSF timer service from the target system's own timing mechanisms. Function WsfTimerUpdate() is called to update the WSF timer service with the number of elapsed ticks.

One way to implement this is to configure a system timer to expire every tick and call WsfTimerUpdate() when the system timer expires.

2.5.3 Next Expiration

The WSF timer service provides an interface to read the number of ticks until the next WSF timer expiration, WsfTimerNextExpiration(). Use of this function is optional. This function is useful when implementing a 'tickless' timer port. For example: On sleep, call WsfTimerNextExpiration() and set a platform timer to expire at this time. On wakeup, call WsfTimerUpdate() with the elapsed time.

2.6 OS Interfaces

2.6.1 Critical Sections and Task Schedule Locking

WSF uses critical sections and task schedule locking to allow for the stack to operate in a pre-emptive multitasking environment with interrupts. Critical sections disable interrupts while task schedule locking prevents a task context switch.

Only certain WSF functions are designed to be called from interrupt context: Buffer management functions (wsf_buf.h), queue functions (wsf_queue.h), and WsfSetEvent(). Other WSF functions must be called from task context. Note that all stack API functions must only be called from task context.

The following critical section macros must be implemented in file wsf_cs.h:

Name	Description
WSF_CS_INIT()	Initialize critical section.
WSF_CS_ENTER()	Enter a critical section.
WSF_CS_EXIT()	Exit a critical section.

The following task schedule locking functions must be implemented:

Name	Description
WsfTaskLock()	Lock task scheduling.
WsfTaskUnock()	Unlock task scheduling.

Critical sections and task schedule locking may not be necessary depending on how WSF and the stack are used in the target system:

1. If no WSF functions are executed in interrupt context, then the critical section macros can be defined to call the task schedule locking functions.

2. If the target OS does not use pre-emptive multitasking then the task schedule locking functions can be implemented as empty functions.

2.6.2 WSF Event Handlers and Target OS Tasks

WSF defines and event handler service which can receive events and messages. An event is an integer bit mask set to an event handler by WsfSetEvent(). A message is a buffer containing data that is sent to an event handler by WsfMsgSend(). WSF event handlers must be executed by the target system when an event handler receives a message, event, or a timer expires for the event handler.

The target system must provide WSF certain interfaces into the target OS task service. These interfaces are in the form of task event macros and data types defined in file wsf_os_int.h plus certain functions that the target system must implement: WsfSetEvent(), WsfTaskSetReady() and WsfTaskMsgQueue().

Certain macros are passed to function WsfTaskSetReady(). The following macros must be defined in file wsf_os_int.h:

Name	Example Value	Description
WSF_MSG_QUEUE_EVENT	0x01	Message queued for event handler.
WSF_TIMER_EVENT	0x02	Timer expired for event handler.
WSF_HANDLER_EVENT	0x04	Event set for event handler.

WSF allows event handlers to run in separate target OS tasks. The handler ID is used to map a handler to a task. The following macros must be defined in file wsf_os_int.h:

Name	Description
WSF_TASK_FROM_ID(handlerID)	Derive task from handler ID.
WSF_HANDLER_FROM_ID(handlerID)	Derive handler from handler ID.

The following data types must be implemented in file wsf_os_int.h:

Name	Description
wsfHandlerId_t	Event handler ID data type.
wsfEventMask_t	Event handler event mask data type.
wsfTaskId_t	Task ID data type.
wsfTaskEvent_t	Task event mask data type.

2.6.3 WsfSetEvent()

This function sets an event for an event handler. This function must be implemented by the target system. The implementation of this function typically sets the passed event value in a data structure for the event handler and then calls WsfTaskSetReady(). An example implementation is shown below:

```
void WsfSetEvent(wsfHandlerId_t handlerId, wsfEventMask_t event)
{
  WSF_CS_INIT(cs);

  WSF_CS_ENTER(cs);
  wsfOs.task.handlerEventMask[handlerId] |= event;
  WSF_CS_EXIT(cs);

  WsfTaskSetReady(handlerId, WSF_HANDLER_EVENT);
}
```

2.6.4 WsfTaskSetReady()

This function notifies a target OS task that it is ready to run. The implementation of this function typically calls a target OS function to set a pending event for the task.

2.6.5 WsfTaskMsgQueue()

This function returns the message queue used by a given event handler. If a single message queue is used for all event handlers (a typical case) then this function can be implemented as shown below:

```
wsfQueue_t *WsfTaskMsgQueue(wsfHandlerId_t handlerId)
{
  /* return global WSF message queue */
  return &(wsfOs.task.msgQueue);
}
```

2.6.6 Initialization

WSF and the stack require a specific initialization sequence. This sequence is typically implemented in a target system initialization function that is executed once on system startup. The initialization sequence initializes WSF services, sets up event handlers, and initializes stack subsystems. An example initialization sequences is shown below. Note that each event handlers is assigned a unique ID.

```
static void mainStackInit(void)
{
  wsfHandlerId_t handlerId;

  /* initialize WSF services */
  WsfSecInit();
  WsfSecAesInit();

  /* initialize HCI */
  handlerId = WsfOsSetNextHandler(HciHandler);
  HciHandlerInit(handlerId);

  /* initialize DM */
  handlerId = WsfOsSetNextHandler(DmHandler);
  DmAdvInit();
  DmConnInit();
  DmConnSlaveInit();
  DmSecInit();
  DmHandlerInit(handlerId);
```

```
  /* initialize L2CAP */
  handlerId = WsfOsSetNextHandler(L2cSlaveHandler);
  L2cSlaveHandlerInit(handlerId);
  L2cInit();
  L2cSlaveInit();

  /* initialize ATT */
  handlerId = WsfOsSetNextHandler(AttHandler);
  AttHandlerInit(handlerId);
  AttsInit();
  AttsIndInit();


  /* initialize SMP */
  handlerId = WsfOsSetNextHandler(SmpHandler);
  SmpHandlerInit(handlerId);
  SmprInit();

  /* initialize App Framework */
  handlerId = WsfOsSetNextHandler(AppHandler);
  AppHandlerInit(handlerId);

  /* initialize application */
  handlerId = WsfOsSetNextHandler(FitHandler);
  FitHandlerInit(handlerId);
}
```

2.6.7 Servicing Event Handlers

WSF event handlers must be executed by the target system when an event handler receives a message, event, or a timer expires for the event handler. This is typically done from a target OS task or other dispatcher code that executes when WsfTaskSetReady() is called.

An example implementation for servicing WSF event handlers is shown below:

```
if (taskEventMask & WSF_MSG_QUEUE_EVENT)
{
  /* service message queue */
  while ((pMsg = WsfMsgDeq(&pTask->msgQueue, &handlerId)) != NULL)
  {
       /* execute event handler */
       (*pTask->handler[handlerId])(0, pMsg);

       /* free message buffer */
       WsfMsgFree(pMsg);
  }
}

if (taskEventMask & WSF_TIMER_EVENT)
{
  /* service timers */
  while ((pTimer = WsfTimerServiceExpired(0)) != NULL)
  {
    /* execute event handler */
    (*pTask->handler[pTimer->handlerId])(0, &pTimer->msg);
  }
}

if (taskEventMask & WSF_HANDLER_EVENT)
{
  /* service events */
  for (i = 0; i < WSF_MAX_HANDLERS; i++)
  {
    if ((pTask->eventMask[i] != 0) && (pTask->handler[i] != NULL))
    {
      /* clear event mask */
      WSF_CS_ENTER(cs);
      eventMask = pTask->eventMask[i];
      pTask->eventMask[i] = 0;
      WSF_CS_EXIT(cs);

      /* execute event handler */
      (*pTask->handler[i])(eventMask, NULL);
    }
  }
}
```

2.7 Diagnostics

WSF provides macros for interfacing to asserts and trace messages. Assert macros are defined in file wsf_assert.h. Trace macros are defined in file wsf_trace.h.

The target system must define all the macros in these files. If asserts are trace macros are not used then these macros can be defined to be empty.

2.8 Security

WSF provides interfaces for the following security functions: Random number generations, AES encryption, AES CMAC algorithm, and ECC algorithm.

2.8.1 Random Number Generation

Function WsfSecRand() is the interface for random number generation. The example implementation in /sw/wsf/common/wsf_sec.c uses the standard HCI command for random number generation. This works well for typical systems that implement standard HCI commands.

2.8.2 AES Encryption

Function WsfSecAes() is the interface to AES encryption. The example implementation in /sw/wsf/common/wsf_sec_aes.c uses the standard HCI command for AES encryption. This works well for typical systems that implement standard HCI commands. Alternatively this function could be mapped to a hardware or software AES implementation in the target system.

2.8.3 AES CMAC algorithm

Function WsfSecCmac() is the interface to the AES CMAC algorithm. The example implementation in /sw/wsf/common/wsf_sec_cmac.c uses the standard HCI command for AES encryption. This works well for typical systems that implement standard HCI commands. Alternatively this function could be mapped to a hardware or software AES or CMAC implementation in the target system.

2.8.4 ECC Algorithm

Functions WsfSecEccGenKey() and WsfSecEccGenSharedSecret() are the interfaces to the ECC algorithm. The example implementation in /sw/wsf/common/wsf_sec_ecc_debug.c always returns debug values instead of actually executing the ECC algorithm. The example implementation in /sw/wsf/uecc/wsf_sec_ecc.c interfaces to the open source micro-ecc code. For more information on micro-ecc see https://github.com/kmackay/micro-ecc.

3 Porting HCI

Wicentric's HCI layer is designed to be portable and support different transport and chip configurations. The porting process depends on the chip configuration: If the stack is ported to a single-chip system then a "thin HCI" porting process is used. If the stack is ported to a two-chip system with wired HCI transport then a transport based porting process is used.

3.1 File Organization

The HCI code is organized as follows:

🗁hci	configuration-specific hci files
🗁include	common interface files
🗁generic	common platform-independent files
🗁dual-chip	dual-chip platform files
🗁exactle	thin HCI port to exactLE link layer
🗁<target name>	target-specific implementation

```
🗁stack
  🗁hci                    platform and configuration-independent hci files
```

A new directory, typically named after the target system, is created in the HCI directory. This new directory contains the files required to implement the HCI port.

The other directories contain platform-independent or configuration-independent files which typically do not need to be modified when porting.

3.2 Porting Thin HCI

The "thin HCI" porting process is used in a single-chip system where the stack and the link layer run on the same CPU. The porting process involves adapting the stack's HCI interface to the link layer's interface. If the link layer uses a functional interface similar to that defined by the Bluetooth HCI specification then porting is rather straightforward exercise. There are three parts of the functional interface: The HCI command interface, event interface, and ACL data interface.

3.2.1 Command Interface

The stack uses a functional interface very similar to the interface defined by the Bluetooth HCI specification. The details of the stack's HCI command API are described in [2].

Porting the command interface involves implementing the HCI command API functions to call the target's link layer or HCI controller API. Depending on the target system implementation, a typical function may simply directly call the target API function or it may send a message to the link layer task.

3.2.2 Event Interface

The stack uses an optimized event interface based on the interface defined by the Bluetooth HCI specification. The details of the stack's HCI event API are described in [2].

Porting the event interface involves executing the HCI event callback with the event IDs and their associated data structures. The general procedure for interfacing events from the target link layer to the stack's HCI callback is as follows:

1. Copy the link layer event data to a WSF message buffer and queue it to the HCI RX queue.
2. In function hciEvtProcessMsg(), convert link layer event data to stack HCI data types and execute the HCI event callback.

3.2.3 ACL Data Interface

The stack sends and receives data using WSF buffers containing ACL data packets in the standard format.

For transmit data, the function HciSendAclData() must be implemented to send ACL data to the target. The function is responsible for deallocating the buffer after the data is transmitted.

For receive data, a WSF message buffer containing an ACL data packet is queued to the stack's HCI RX queue. The stack is responsible for deallocating the buffer.

3.3 Porting Transport-Based HCI

The transport-based porting process is used in a dual-chip system where the CPU running the stack is connected to a HCI controller chip via a wired interface. If the transport is UART or SPI, then the porting process involves implementing functions to send and receive data using target system's driver interface.

3.3.1 Sending Data and Commands

The target system must implement function hciDrvWrite() to send HCI data and commands. In a typical implementation this function copies data contained in a WSF buffer to the target driver interface.

3.3.2 Receiving Data and Commands

Received HCI events and ACL data must be copied into a WSF buffer and passed to function hciCoreRecv(). This function queues the buffer to the stack. Alternatively, function hciTrSerialRxIncoming() can be used to reassemble a received byte stream of data into HCI event and data packets, which are then passed to the stack.

4 References

1. Wicentric, "Software Foundation API", 2009-0003.
2. Wicentric, "HCI API", 2009-0006.

5 Definitions

HCI	Host Controller Interface
LE	(Bluetooth) Low Energy
WSF	Wicentric Software Foundation software service and porting layer