ARM® Cordio WSF

ARM-EPM-115975 1.0

Software Foundation API

Confidential

ARM® Wireless Software Foundation API

Reference Manual

Release Information

The following changes have been made to this book:

Document History

Date	Issue	Confidentiality	Change
25 September 2015	-	Non-Confidential	First Wicentric release for 1.3 as 2009-0003.
1 March 2016	A	Confidential Draft	First ARM release for 1.3.
24 August 2016	A	Confidential	AUSPEX #

Proprietary Notice

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Contents

1 Preface

This preface introduces the Wireless Software Foundation API Reference Manual.

1.1 About this book

This document describes the Wireless Software Foundation (WSF) API and lists the API functions and their parameters.

1.1.1 Intended audience

This book is written for experienced software engineers who might or might not have experience with ARM products. Such engineers typically have experience of writing Bluetooth applications but might have limited experience of the Cordio software stack.

It is also assumed that the readers have access to all necessary tools.

1.1.2 Using this book

This book is organized into the following chapters:

- **Introduction**
  Read this for an overview of the API.
- **Portable Data Types**
  Read this for a list of data types used in the API.
- **Buffers**
  Read this for a description of the buffer service functions.
- **Queues**
  Read this for a description of the queue service functions.
- **Messages**
  Read this for a description of the message service used to pass messages to WSF event functions.
- **Timers**
  Read this for a description of the timer service functions.
- **Event Handlers**
  Read this for a description of the WSF event handlers receive events, message, and timer expirations from other components in the service.
- **Critical Sections**
  Read this for a description of the critical section macros used in code which might be executed in an interrupt context.
- **Task Schedule Locking**
  Read this for a description of the interfaces for locking and unlocking task scheduling.
- **Assert**
  Read this for a description of the macros used for testing and debugging.
- **Trace**
  Read this for a description of the trace macros used for trace diagnostics.
- **Security**
  Read this for a description of the security service functions.
- **Revisions**
  Read this chapter for descriptions of the changes between document versions.

1.1.3 Terms and abbreviations

For a list of ARM terms, see the ARM glossary.

Terms specific to the Cordio software are listed below:

Term	Description
ACL	Asynchronous Connectionless data packet
AD	Advertising Data
ARQ	Automatic Repeat reQuest
ATT	Attribute Protocol, also attribute protocol software subsystem
ATTC	Attribute Protocol Client software subsystem
ATTS	Attribute Protocol Server software subsystem
CCC or CCCD	Client Characteristic Configuration Descriptor
CID	Connection Identifier
CSRK	Connection Signature Resolving Key
DM	Device Manager software subsystem
GAP	Generic Access Profile
GATT	Generic Attribute Profile
HCI	Host Controller Interface
IRK	Identity Resolving Key
JIT	Just In Time
L2C	L2CAP software subsystem
L2CAP	Logical Link Control Adaptation Protocol
LE	(Bluetooth) Low Energy
LL	Link Layer
LLPC	Link Layer Control Protocol
LTK	Long Term Key
MITM	Man In The Middle pairing (authenticated pairing)
OOB	Out Of Band data
SMP	Security Manager Protocol, also security manager protocol software subsystem
SMPI	Security Manager Protocol Initiator software subsystem
SMPR	Security Manager Protocol Responder software subsystem
STK	Short Term Key
WSF	Wireless Software Foundation software service and porting layer.

1.1.4 Conventions

The following table describes the typographical conventions:

Typographical conventions

Style	Purpose
Italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`MONOSPACE`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`MONO`SPACE	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace` *`italic`*	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

1.1.5 Additional reading

This section lists publications by ARM and by third parties.

See *Infocenter* for access to ARM documentation.

Other publications

This section lists relevant documents published by third parties:

- Bluetooth SIG, "*Specification of the Bluetooth System*", Version 4.2, December 2, 2015.

1.2 Feedback

ARM welcomes feedback on this product and its documentation.

1.2.1 Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM-EPM-115156.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note: ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

2 Introduction

This document describes the Wireless Software Foundation (WSF) API.

WSF is a simple OS wrapper, porting layer, and general-purpose software service used by the Cordio embedded software system.

The goal of WSF is to stay small and lean, supporting only the basic services required by the system. 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.

3 Portable Data Types

WSF defines the following portable data types in file `wsf_types.h`. These data `types` are used throughout the software system.

Table 1 Integer types

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
`uint64_t`	64 bit unsigned integer
`bool_t`	Boolean integer

4 Buffers

The WSF buffer management service is a pool-based dynamic memory allocation service. The buffer service interface is defined in file `wsf_buf.h`.

4.1 Data Types

4.1.1 wsfBufPoolDesc_t

This is buffer pool descriptor structure. It is used by function `WsfBufInit()`.

Type	Name	Description
`uint16_t`	`len`	Length of buffers in pool.
`uint8_t`	`num`	Number of buffers in pool.

4.2 Functions

4.2.1 WsfBufInit()

Initialize the buffer pool service. This function should only be called once upon system initialization.

Syntax:

```
uint16_t WsfBufInit(uint16_t bufMemLen, uint8_t *pBufMem, uint8_t numPools,
         wsfBufPoolDesc_t *pDesc)
```

Where:

- `bufMemLen`: Length in bytes of memory pointed to by pBufMem.
- `pBufMem`: Memory in which to store the pools used by the buffer pool service.
- `numPools`: Number of buffer pools.
- `pDesc`: Array of buffer pool descriptors, one for each pool

This function returns the amount of pBufMem used or 0 for failures.

4.2.2 WsfBufAlloc()

Allocate a buffer.

Syntax:

```
void *WsfBufAlloc(uint16_t len)
```

Where:

- `len`: Length of buffer to allocate.

This function returns a pointer to the buffer or NULL if allocation fails.

4.2.3 WsfBufFree()

Free a buffer.

Syntax:

```
void WsfBufFree(void *pBuf)
```

Where:

- `pBuf`: Buffer to free.

4.3 Diagnostic Macros

The following macros are used for diagnostic purposes.

Table 2 Diagnostic macros

Name	Value	Description
`WSF_BUF_FREE_CHECK`	TRUE, FALSE	Assert if trying to free a buffer that is already free.
`WSF_BUF_ALLOC_FAIL_ASSERT`	TRUE, FALSE	Set to TRUE to assert on buffer allocation failure.
`WSF_BUF_STATS`	TRUE, FALSE	Set to TRUE to collect buffer allocation statistics.

4.4 Diagnostic Functions

4.4.1 WsfBufGetMaxAlloc()

Diagnostic function to get maximum allocated buffers from a pool.

Syntax:

```
uint8_t WsfBufGetMaxAlloc(uint8_t pool)
```

Where:

- `pool`: Buffer pool number.

This function returns the number of allocated buffers.

4.4.2 WsfBufGetNumAlloc()

Diagnostic function to get the number of currently allocated buffers in a pool.

Syntax:

```
uint8_t WsfBufGetNumAlloc(uint8_t pool)
```

Where:

- `pool`: Buffer pool number.

This function returns the number of allocated buffers.

4.4.3 WsfBufGetAllocStats()

Diagnostic function to get the buffer allocation statistics.

The statistics contain a count of each call to `WsfBufAlloc()` for the requested buffer length.

Syntax:

```
uint8_t *WsfBufGetAllocStats(void)
```

The function returns a 128-byte array indexed by the length passed to `WsfBufAlloc()` with each element containing the total number of calls to `WsfBufAlloc()` for that length.

4.4.4 WsfBufGetPolStats()

Get statistics for each pool.

Syntax:

```
uint8_t WsfBufGetPolStats(WsfBufPoolStat_t *pStat, uint8_t numPool)
```

Where:

- `pStat`: Buffer to store statistics.
- `numPool`: Number of pool elements.

This function returns the pool statistics in variable `pStat`.

5 Queues

The WSF queue service is a general purpose queue service that is used throughout the software system. The queue service interface is defined in function `wsf_queue.h`.

5.1 Data Types

5.1.1 wsfQueue_t

Table 3 Queue data structure

Type	Name	Description
`void *`	`pHead`	Head of queue.
`void *`	`pTail`	Tail of queue.

5.2 Functions

5.2.1 WSF_QUEUE_INIT()

This macro initializes a queue structure.

Syntax:

```
WSF_QUEUE_INIT(pQueue)
```

Where:

- `pBuf`: Pointer to queue.

5.2.2 WsfQueueEnq()

Enqueue an element to the tail of a queue.

Syntax:

```
void WsfQueueEnq(wsfQueue_t *pQueue, void *pElem)
```

Where:

- `pQueue`: Pointer to queue.
- `pElem`: Pointer to element.

5.2.3 WsfQueueDeq()

Dequeue an element from the head of a queue.

Syntax:

```
void *WsfQueueDeq(wsfQueue_t *pQueue)
```

Where:

- `pQueue`: Pointer to queue.

This function returns a pointer to the element that has been dequeued or NULL if the queue is empty.

5.2.4 WsfQueuePush()

Push an element to the head of a queue.

Syntax:

```
void WsfQueuePush(wsfQueue_t *pQueue, void *pElem)
```

Where:

- `pQueue`: Pointer to queue.
- `pElem`: Pointer to element.

5.2.5 WsfQueueInsert()

Insert an element into a queue.

This function is typically used when iterating over a queue.

Syntax:

```
void WsfQueueInsert(wsfQueue_t *pQueue, void *pElem, void *pPrev)
```

Where:

- `pQueue`: Pointer to queue.
- `pElem`: Pointer to element to be inserted.
- `pPrev`: Pointer to previous element in the queue before element to be inserted.
  Note: set `pPrev` to NULL if `pElem` is first element in queue.

5.2.6 WsfQueueRemove()

Remove an element from a queue. This function is typically used when iterating over a queue.

Syntax:

```
void WsfQueueRemove(wsfQueue_t *pQueue, void *pElem, void *pPrev)
```

Where:

- `pQueue`: Pointer to queue.
- `pElem`: Pointer to element to be inserted.
- `pPrev`: Pointer to previous element in the queue before element to be removed.

5.2.7 WsfQueueCount()

Count the number of elements in a queue.

Syntax:

```
uint16_t WsfQueueCount(wsfQueue_t *pQueue)
```

Where:

- `pQueue`: Pointer to queue.

This function returns the number of elements in the queue.

5.2.8 WsfQueueEmpty()

Test if queue is empty.

Syntax:

```
bool_t WsfQueueEmpty(wsfQueue_t *pQueue)
```

Where:

- `pQueue`: Pointer to queue.

This function returns TRUE if queue is empty, FALSE otherwise.

6 Messages

The WSF message service is used to pass messages to WSF event handlers.

The WSF message service is defined in file `wsf_msg.h`.

6.1 Functions

6.1.1 WsfMsgAlloc()

Allocate a message buffer to be sent with `WsfMsgSend()`.

Syntax:

```
void *WsfMsgAlloc(uint16_t len)
```

Where:

- `len`: Message length in bytes.

This function returns a pointer to the message buffer or NULL if allocation failed.

6.1.2 WsfMsgFree()

Free a message buffer allocated with `WsfMsgAlloc()`.

Syntax:

```
void WsfMsgFree(void *pMsg)
```

Where:

- `pMsg`: Pointer to message buffer.

6.1.3 WsfMsgSend()

Send a message to an event handler.

Syntax:

```
void WsfMsgSend(wsfHandlerId_t handlerId, void *pMsg)
```

Where:

- `handlerId`: Event handler ID.
- `pMsg`: Pointer to message buffer.

6.1.4 WsfMsgEnq()

Enqueue a message.

Syntax:

```
void WsfMsgEnq(wsfQueue_t *pQueue, wsfHandlerId_t handlerId, void *pMsg)
```

Where:

- `pQueue`: Pointer to queue.

- `handerId`: Set message handler ID to this value.
- `pElem`: Pointer to message buffer.

6.1.5 WsfMsgDeq()

Dequeue a message.

Syntax:

```
void *WsfMsgDeq(wsfQueue_t *pQueue, wsfHandlerId_t *pHandlerId)
```

Where:

- `pQueue`: Pointer to queue.
- `pHandlerId`: Handler ID of returned message; this is a return parameter.

This function returns a pointer to the message that has been dequeued or NULL if the queue is empty.

6.1.6 WsfMsgPeek ()

Get the next message without removing it from the queue.

Syntax:

```
void *WsfMsgPeek (wsfQueue_t *pQueue, wsfHandlerId_t *pHandlerId)
```

Where:

- `pQueue`: Pointer to queue.
- `pHandlerId`: Handler ID of returned message; this is a return parameter.

This function returns a pointer to the next message on the queue or NULL if the queue is empty.

7 Timers

The WSF timer service is used by WSF event handlers.

When a timer expires, the event handler associated with that timer is executed.

7.1 Data Types

This section describe the timer data types.

7.1.1 wsfTimer_t

Table 4 Timer data structure.

Type	Name	Description
`wsfTimer_t *`	`pNext`	Pointer to next timer in queue.
`wsfTimerTicks_t`	`ticks`	Number of ticks until expiration.
`wsfHandlerId_t`	`handlerId`	Event handler for this timer.
`bool_t`	`isStarted`	TRUE if timer has been started.
`wsfMsgHdr_t`	`msg`	Application-defined timer event parameters.

7.2 Functions

This section describe the timer functions.

7.2.1 WsfTimerInit()

Initialize the timer service. This function should only be called once upon system initialization.

Syntax:

```
void WsfTimerInit (void)
```

7.2.2 WsfTimerStartSec()

Start a timer in units of seconds.

Before this function is called parameter `pTimer->handlerId` must be set to the event handler for this timer and parameter `pTimer->msg` must be set to any application-defined timer event parameters.

Syntax:

```
void WsfTimerStartSec(wsfTimer_t *pTimer, wsfTimerTicks_t sec)
```

Where:

- `pTimer`: Pointer to timer.
- `sec`: Seconds until expiration.

7.2.3 WsfTimerStartMs()

Start a timer in units of milliseconds.

Syntax:

```
void WsfTimerStartMs(wsfTimer_t *pTimer, wsfTimerTicks_t ms)
```

Where:

- `pTimer`: Pointer to timer.
- `ms`: Milliseconds until expiration.

7.2.4 WsfTimerStop()

Stop a timer.

Syntax:

```
void WsfTimerStop(wsfTimer_t *pTimer)
```

Where:

- `pTimer`: Pointer to timer.

7.2.5 WsfTimerUpdate()

Update the timer service with the number of elapsed ticks.

This function is typically called only from WSF timer porting code.

Syntax:

```
void WsfTimerUpdate(wsfTimerTicks_t ticks)
```

Where:

- `ticks`: Number of ticks since last update.

7.2.6 WsfTimerNextExpiration()

Return the number of ticks until the next timer expiration.

Note: This function can return zero even if a timer is running, indicating the timer has expired but has not yet been serviced.

Syntax:

```
wsfTimerTicks_t WsfTimerNextExpiration(bool_t *pTimerRunning)
```

Where:

- `pTimerRunning`: Returns TRUE if a timer is running, FALSE if no timers running.

This function returns the number of ticks until the next timer expiration.

7.2.7 WsfTimerServiceExpired()

Service expired timers for the given task.

This function is typically called only from WSF OS porting code.

Syntax:

```
wsfTimer_t *WsfTimerServiceExpired(wsfTaskId_t taskId)
```

Where:

- `taskId`: OS Task ID of task servicing timers.

This function returns a pointer to next expired timer or NULL if there are no expired timers.

8 Event Handlers

WSF event handlers receive WSF events, messages, and timer expirations from other components in the software system. Event handlers are used by the main protocol subsystems of the stack.

The event handler interface is defined in file `wsf_os.h`.

8.1 Data Types

This section describe the event handler data types.

8.1.1 wsfMsgHdr_t

This is the common message structure passed to event handlers.

Table 5 Event handler message

Type	Name	Description
`uint16_t`	`param`	General purpose parameter passed to event handler.
`uint8_t`	`event`	General purpose event value passed to event handler.
`uint8_t`	`status`	General purpose status value passed to event handler.

8.2 Functions

This section describe the event handler functions.

8.2.1 (*wsfEventHandler_t)()

This is the data type for event handler callback functions.

Syntax:

```
void (*wsfEventHandler_t)(wsfEventMask_t event, wsfMsgHdr_t *pMsg)
```

Where:

- `event`: Mask of events set for the event handler.
- `pMsg`: Pointer to message for the event handler.

8.2.2 WsfSetEvent()

Set an event to an event handler.

Syntax:

```
void WsfSetEvent(wsfHandlerId_t handlerId, wsfEventMask_t event)
```

Where:

- `handlerId`: Handler ID.
- `event`: Event or events to set.

8.2.3 WsfOsSetNextHandler()

Set the next WSF handler function in the WSF OS handler array.

This function should only be called as part of the OS initialization procedure.

Syntax:

```
wsfHandlerId_t WsfOsSetNextHandler(wsfEventHandler_t handler)
```

Where:

- `handler`: WSF handler function.

This function returns the WSF handler ID for this handler.

9 Critical Sections

WSF provides critical section macros that are used in code which might be executed in interrupt context to protect global data. The critical section interface is defined in file `wsf_cs.h`.

9.1 Macros

This section describe the macros.

9.1.1 WSF_CS_INIT()

Initialize critical section. This macro may define a variable.

Syntax:

```
WSF_CS_INIT(cs)
```

Where:

- `cs`: Critical section variable to be defined.

9.1.2 WSF_CS_ENTER()

Enter a critical section.

Syntax:

```
WSF_CS_ENTER(cs)
```

Where:

- `cs`: Critical section variable.

9.1.3 WSF_CS_EXIT()

Exit a critical section.

Syntax:

```
WSF_CS_EXIT(cs)
```

Where:

- `cs`: Critical section variable.

10 Task Schedule Locking

WSF provides interfaces for locking and unlocking task scheduling. This allows for operation in pre-emptive multi-tasking environments. The task schedule locking interface is defined in file `wsf_os.h`.

10.1 Functions

This section describe the task schedule functions.

10.1.1 WsfTaskLock()

Lock task scheduling.

Syntax:

```
void WsfTaskLock(void)
```

10.1.2 WsfTaskUnlock()

Unlock task scheduling.

Syntax:

```
void WsfTaskUnlock(void)
```

11 Assert

WSF defines assert macros that are used for testing and debugging purposes. The assert interface is defined in file `wsf_assert.h`.

11.1 Macros

This section describe the assert macros.

11.1.1 WSF_ASSERT()

Run-time assert macro. The assert executes when the expression is FALSE.

Syntax:

```
WSF_ASSERT(expr)
```

Where:

- `expr`: Boolean expression to be tested.

11.1.2 WSF_CT_ASSERT()

Compile-time assert macro. This macro causes a compiler error when the expression is FALSE. Note that this macro is generally used at file scope to test constant expressions.

Errors may result if it is used in executing code.

Syntax:

```
WSF_CT_ASSERT(expr)
```

Where:

- `expr`: Boolean expression to be tested.

12 Trace

WSF defines trace macros that are used throughout the software system for diagnostic purposes. A separate set of trace macros is used for each software subsystem (for example, WSF, HCI, DM, and ATT). This allows trace messages to be compiled in/out for each subsystem. Within each set of subsystem trace macros there are separate macros for different types of trace messages:

- `INFO`: Informational messages.
- `WARN`: Warning messages.
- `ERR`: Error messages.
- `ALLOC`: Memory or other resource is allocated.
- `FREE`: Memory or other resource is freed.
- `MSG`: WSF event handler message is sent.

13 Security

WSF provides interfaces to encryption and random number generation algorithms. These algorithms are used by the stack to perform various Bluetooth LE security procedures.

13.1 Data Types

This section describe the security data types.

13.1.1 wsfSecMsg_t

Table 6 AES security callback parameters structure

Type	Name	Description
wsfMsgHdr_t	hdr	Message header.
uint8_t	*pCiphertext	Pointer to 16 bytes of ciphertext data.

13.1.2 wsfSecEccKey_t

Table 7 ECC Security callback parameters structure

Type	Name	Description
uint8_t	pubKey_x[WSF_ECC_KEY_LEN]	Public key X.
uint8_t	pubKey_y[WSF_ECC_KEY_LEN]	Public key Y.
uint8_t	privKey[WSF_ECC_KEY_LEN]	Private key.

13.1.3 wsfSecEccSharedSec_t

Table 8 ECC shared secret structure

Type	Name	Description
uint8_t	secret[WSF_ECC_KEY_LEN]	Shared secret.

13.1.4 wsfSecEccMsg_t

Table 9 ECC Security callback parameters structure

Type	Name	Description
wsfSecEccSharedSec_t	sharedSecret	Shared secret.
wsfSecEccKey_t	key	ECC key structure.

13.2 Functions

This section describe the security functions.

13.2.1 WsfSecInit()

Initialize the security service.

This function should only be called once upon system initialization.

Syntax:

```
void WsfSecInit(void)
```

13.2.2 WsfSecRandInit()

Initialize the random number service.

This function should only be called once upon system initialization.

Syntax:

```
void WsfSecRandInit(void)
```

13.2.3 WsfSecAesInit()

Initialize the AES service.

This function should only be called once upon system initialization.

Syntax:

```
void WsfSecAesInit (void)
```

13.2.4 WsfSecCmacInit()

Called to initialize CMAC security.

This function should only be called once upon system initialization.

Syntax:

```
void WsfSecCmacInit (void)
```

13.2.5 WsfSecEccInit()

Called to initialize ECC security.

This function should only be called once upon system initialization.

Syntax:

```
void WsfSecEccInit(void)
```

13.2.6 WsfSecAes()

Execute an AES calculation.

When the calculation completes, a WSF message will be sent to the specified handler.

Syntax:

```
uint8_t WsfSecAes(uint8_t *pKey, uint8_t *pPlaintext, wsfHandlerId_t handlerId,
        uint16_t param, uint8_t event)
```

Where:

- `pKey`: Pointer to 16 byte key.
- `pPlaintext`: Pointer to 16 byte plaintext.
- `handlerId`: WSF handler ID.
- `param`: Client-defined parameter returned in message.
- `event`: Event for client's WSF handler.

This function returns a token value that the client can use to match calls to this function with messages.

13.2.7 WsfSecCmac()

Execute the CMAC algorithm.

Syntax:

```
uint8_t WsfSecCmac(const uint8_t *pKey, uint8_t *pPlaintext, uint8_t textLen,
        wsfHandlerId_t handlerId, uint16_t param, uint8_t event)
```

Where:

- `pKeyKey`: used in CMAC operation.
- `pPlaintext`: Data to perform CMAC operation over
- `len`: Size of pPlaintext in bytes.
- `handlerId`: WSF handler ID for client.
- `param`: Optional parameter sent to client's WSF handler.
- `event`: Event for client's WSF handler.

This function returns TRUE if successful, FALSE otherwise.

13.2.8 WsfSecEccGenKey()

Generate an ECC key.

Syntax:

```
uint8_t WsfSecEccGenKey(wsfHandlerId_t handlerId, uint16_t param, uint8_t event)
```

Where:

- `handlerId`: WSF handler ID for client.
- `param`: Optional parameter sent to client's WSF handler.
- `event`: Event for client's WSF handler.

This function returns TRUE if successful, FALSE otherwise.

13.2.9 WsfSecEccGenSharedSecret()

Generate an ECC shared secret from the input ECC keys.

Syntax:

```
uint8_t WsfSecEccGenSharedSecret(wsfSecEccKey_t *pKey, wsfHandlerId_t handlerId,
        uint16_t param, uint8_t event)
```

Where:

- `pKey`: ECC Key structure.

- `handlerId`: WSF handler ID for client.
- `param`: Optional parameter sent to client's WSF handler.
- `event`: Event for client's WSF handler.

This function returns TRUE if successful, FALSE otherwise.

13.2.10 WsfSecRand()

This function returns up to 16 bytes of random data to a buffer provided by the client.

Syntax:

```
void WsfSecRand(uint8_t *pRand, uint8_t randLen)
```

Where:

- `pRand`: Pointer to returned random data.
- `randLen`: Length of random data.