Wicentric Software Foundation API

Document 2009-0003
Revision 1.3
September 25, 2015

Table of Contents

1 Introduction

This document describes the Wicentric Software Foundation (WSF) API. WSF is a simple OS wrapper, porting layer, and general-purpose software service used by the Wicentric 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.

2 Portable Data Types

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

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

3 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.

3.1 Data Types

3.1.1 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.

3.2 Functions

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

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

- **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.

3.2.2 void *WsfBufAlloc(uint16_t len)

Allocate a buffer.

- **len**: Length of buffer to allocate.

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

3.2.3 void WsfBufFree(void *pBuf)

Free a buffer.

- **pBuf**: Buffer to free.

3.3 Diagnostic Macros

The following macros are used for diagnostic purposes.

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.

3.4 Diagnostic Functions

3.4.1 uint8_t WsfBufGetMaxAlloc(uint8_t pool)

Diagnostic function to get maximum allocated buffers from a pool.

- **pool**: Buffer pool number.

This function returns the number of allocated buffers.

3.4.2 uint8_t WsfBufGetNumAlloc(uint8_t pool)

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

- **pool**: Buffer pool number.

This function returns the number of allocated buffers.

3.4.3 uint8_t *WsfBufGetAllocStats(void)

Diagnostic function to get the buffer allocation statistics. These statistics contain a count of each call to WsfBufAlloc() for the requested buffer length. 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.

3.4.4 uint8_t WsfBufGetPolStats(WsfBufPoolStat_t *pStat, uint8_t numPool)

Get statistics for each pool.

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

This function returns the pool statistics in variable pStat.

4 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.

4.1 Data Types

4.1.1 wsfQueue_t

Queue data structure.

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

4.2 Functions

4.2.1 WSF_QUEUE_INIT(pQueue)

This macro initializes a queue structure.

- **pBuf**: Pointer to queue.

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

Enqueue an element to the tail of a queue.

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

4.2.3 void *WsfQueueDeq(wsfQueue_t *pQueue)

Dequeue an element from the head of a queue.

- **pQueue**: Pointer to queue.

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

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

Push an element to the head of a queue.

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

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

Insert an element into a queue. This function is typically used when iterating over a queue.

- **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.

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

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

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

4.2.7 uint16_t WsfQueueCount(wsfQueue_t *pQueue)

Count the number of elements in a queue.

- **pQueue**: Pointer to queue.

This function returns the number of elements in the queue.

4.2.8 bool_t WsfQueueEmpty(wsfQueue_t *pQueue)

Return TRUE if queue is empty.

- **pQueue**: Pointer to queue.

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

5 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.

5.1 Functions

5.1.1 void *WsfMsgAlloc(uint16_t len)

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

- **len**: Message length in bytes.

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

5.1.2 void WsfMsgFree(void *pMsg)

Free a message buffer allocated with WsfMsgAlloc().

- **pMsg**: Pointer to message buffer.

5.1.3 void WsfMsgSend(wsfHandlerId_t handlerId, void *pMsg)

Send a message to an event handler.

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

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

Enqueue a message.

- **pQueue**: Pointer to queue.
- **handerId**: Set message handler ID to this value.
- **pElem**: Pointer to message buffer.

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

Dequeue a message.

- **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.

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

Get the next message without removing it from the queue.

- **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.

6 Timers

The WSF timer service is used by WSF event handlers. When a timer expires, the event handler associated with that timer is executed.

6.1 Data Types

6.1.1 wsfTimer_t

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.

6.2 Functions

6.2.1 void WsfTimerInit(void)

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

6.2.2 void WsfTimerStartSec(wsfTimer_t *pTimer, wsfTimerTicks_t sec)

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.

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

6.2.3 void WsfTimerStartMs(wsfTimer_t *pTimer, wsfTimerTicks_t ms)

Start a timer in units of milliseconds.

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

6.2.4 void WsfTimerStop(wsfTimer_t *pTimer)

Stop a timer.

- **pTimer**: Pointer to timer.

6.2.5 void WsfTimerUpdate(wsfTimerTicks_t ticks)

Update the timer service with the number of elapsed ticks. This function is typically called only from WSF timer porting code.

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

6.2.6 wsfTimerTicks_t WsfTimerNextExpiration(bool_t *pTimerRunning)

Return the number of ticks until the next timer expiration. Note that this function can return zero even if a timer is running, indicating the timer has expired but has not yet been serviced.

- **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.

6.2.7 wsfTimer_t *WsfTimerServiceExpired(wsfTaskId_t taskId)

Service expired timers for the given task. This function is typically called only from WSF OS porting code.

- **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.

7 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.

7.1 Data Types

7.1.1 wsfMsgHdr_t

This is the common message structure passed to event handlers.

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.

7.2 Functions

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

This is the data type for event handler callback functions.

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

7.2.2 void WsfSetEvent(wsfHandlerId_t handlerId, wsfEventMask_t event)

Set an event to an event handler.

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

7.2.3 wsfHandlerId_t WsfOsSetNextHandler(wsfEventHandler_t handler)

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.

- **handler**: WSF handler function.

This function returns the WSF handler ID for this handler.

8 Critical Sections

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

8.1 Macros

8.1.1 WSF_CS_INIT(cs)

Initialize critical section. This macro may define a variable.

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

8.1.2 WSF_CS_ENTER(cs)

Enter a critical section.

- **cs**: Critical section variable.

8.1.3 WSF_CS_EXIT(cs)

Exit a critical section.

- **cs**: Critical section variable.

9 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.

9.1 Functions

9.1.1 void WsfTaskLock(void)

Lock task scheduling.

9.1.2 void WsfTaskUnlock(void)

Unlock task scheduling.

10 Assert

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

10.1 Macros

10.1.1 WSF_ASSERT(expr)

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

- **expr**: Boolean expression to be tested.

10.1.2 WSF_CT_ASSERT(expr)

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.

- **expr**: Boolean expression to be tested.

11 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 (e.g. WSF, HCI, DM, ATT, etc.). 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.

12 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.

12.1 Data Types

12.1.1 wsfSecMsg_t

AES security callback parameters structure.

Type	Name	Description
wsfMsgHdr_t	hdr	Message header.

uint8_t	*pCiphertext	Pointer to 16 bytes of ciphertext data.

12.1.2 wsfSecEccKey_t

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.

12.1.3 wsfSecEccSharedSec_t

ECC shared secret structure.

Type	Name	Description
uint8_t	secret[WSF_ECC_KEY_LEN]	Shared secret.

12.1.4 wsfSecEccMsg_t

ECC Security callback parameters structure.

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

12.2 Functions

12.2.1 void WsfSecInit(void)

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

12.2.2 void WsfSecRandInit(void)

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

12.2.3 void WsfSecAesInit(void)

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

12.2.4 void WsfSecCmacInit(void)

Called to initialize CMAC security. This function should only be called once upon system initialization.

12.2.5 void WsfSecEccInit(void)

Called to initialize ECC security. This function should only be called once upon system initialization.

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

Execute an AES calculation. When the calculation completes, a WSF message will be sent to the specified handler. This function returns a token value that the client can use to match calls to this function with messages.

- **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.

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

Execute the CMAC algorithm.

- **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.

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

Generate an ECC key.

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

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

Generate an ECC shared secret from the input ECC keys.

- **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.

12.2.10 void WsfSecRand(uint8_t *pRand, uint8_t randLen)

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

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