

HS TFTP v1.3.2 User Manual

Revision: 1.2
Date: 15 June 2011

Table of Contents

1	INTRODUCTION.....	2
1.1	SOFTWARE ARCHITECTURE.....	2
1.2	MODEL OF OPERATION.....	2
2	HS TFTP API.....	3
2.1	HS TFTP INIT.....	3
2.2	HS TFTP DESTROY.....	5
2.3	HS TFTP START SERVER.....	6
2.4	HS TFTP SERVER START RECEIVE.....	8
2.5	HS TFTP SERVER START SEND.....	9
2.6	HS TFTP REJECT RQ.....	11
2.7	HS TFTP TRANSFER.....	11
2.8	HS TFTP TIMER EXPIRED.....	12
2.9	HS TFTP ABORT.....	13
2.10	HS TFTP ERR STR.....	13
3	APPLICATION NOTES.....	14
3.1	SENDING FILE CONSIDERATIONS.....	14
3.2	RECEIVING FILE CONSIDERATIONS.....	14

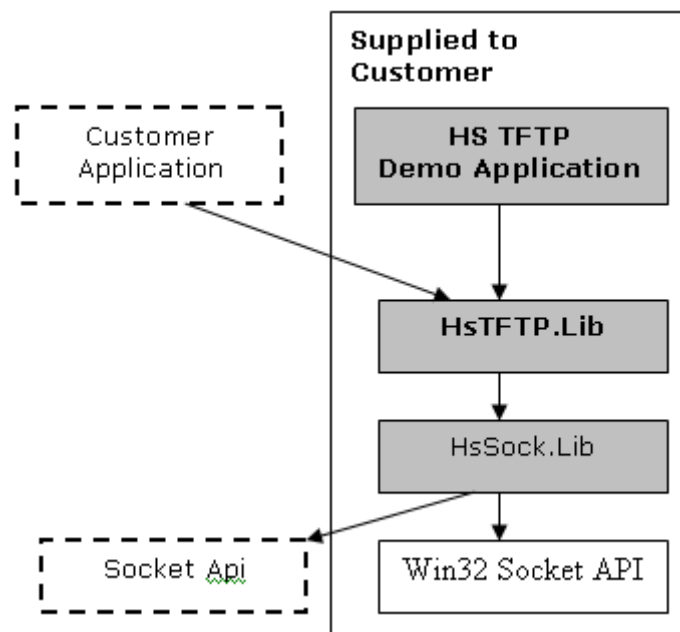
1 Introduction

HS TFTP is a software library in C (supplied with full source code) which implements Trivial File Transfer Protocol (TFTP) over UDP Windows socket layer according to RFC 1350. HS TFTP also implements RFC 1782 (TFTP Option Extension). In version 1.3.1, the only supported TFTP option is block size as defined in RFC 1783 (TFTP Blocksize Option).

HS TFTP v1.3.1 implements concurrent TFTP server and TFTP client operation.

1.1 Software Architecture

The library is a stand-alone module which links directly to customer application:



1.2 Model of Operation

When user application initialises Hs TFTP library, it provides interface callbacks for the services used by HS TFTP protocol module: timer management, memory management and event callbacks. This architecture makes it easy to port HS TFTP protocol module to any environment. HS TFTP internally at a lower layer interfaces to HS Sock library which provides UDP transport services. User application need not worry about Winsock - HS TFTP does all transmission, reception and event handling over socket layer.

TFTP module handles all protocol information flow, error recovery, acknowledgements, timeouts and so on. When it is appropriate to send next block of data HS TFTP will get next memory block from user application. Similarly, when data has been received HS TFTP module will get the next block of memory from user application to store data into.

2 HS TFTP API

2.1 HsTftpInit

Declaration:

```
extern int HsTftpInit(tftp_init_t *init);
```

Summary:

This function initialises HS TFTP Library and must be called first before any other functions are called. Init structure contains function pointers which must be initialised with function addresses in application layer. HS TFTP module will call these functions when it needs to manage timers and memory.

Parameters:

hs_tftp_init_t *init - Pointer to initialisation structure, defined as follows:

parameter	Description
hs_tftp_get_buf_t *get_tx_buffer hs_tftp_get_buf_t *get_rx_buffer	<p>Prototype unsigned char *hs_tftp_get_buf_t(long handle, unsigned int *length, int *cmd);</p> <p>Parameters: handle – application (user) layer context handle *length – Length of memory block. HS TFTP will pass number of bytes requested here. The user code sets the value to the actual number of bytes granted. While most of the time number of requested bytes equals number of given bytes, for example the last block may be shorter due to end of file, so the given length is less than requested. *cmd – this is intended for exchange of additional information or commands between user code and HS TFTP module – currently not used.</p> <p>Return: pointer to memory buffer in user code or NULL if no memory available or nothing to give (end of transmission or end of file)</p> <p>Description: These functions shall be called from HS TFTP module when it needs to send next block of data (on reception of ACK from remote peer) or when data has been successfully received and now needs to be copied to user space from HS TFTP space.</p>
hs_tftp_start_timer_t *start_timer_fn;	<p>Prototype: long hs_tftp_start_timer_t(long handle, unsigned long secs);</p> <p>Parameters: handle – TFTP module context handle secs – number of seconds to timeout after.</p> <p>Return: Timer handle</p> <p>Description: This function in user code will be called from HS TFTP code to start a timer with a specified number of seconds.</p>
hs_tftp_stop_timer_t *stop_timer_fn	<p>Prototype: void hs_tftp_stop_timer_t(long handle);</p> <p>Parameters: Handle – timer handle</p> <p>Return: No return</p> <p>Description: This function will be used by HS TFTP module to stop (destroy) a timer previously started with start_timer_fn.</p>

<p>hs_create_lock_t *create_lock_fn</p>	<p>Prototype: typedef void *hs_create_lock_t(void);</p> <p>Parameters: none</p> <p>Return: Synchronization lock handle</p> <p>Description: This function will be used by HS TFTP to create a synchronization lock for concurrent access to objects that are shared between TFTP sessions (session list). User application should implement this function depending on OS to create a lockm for example on Windows: InitializeCriticalSection</p>
<p>hs_delete_lock_t *delete_lock_fn</p>	<p>Prototype: typedef void hs_delete_lock_t(void *lock);</p> <p>Parameters: Synchronization lock handle (obtained with create_lock_fn)</p> <p>Return: none</p> <p>Description: This function will be used by HS TFTP to destroy the synchronization, created with create_lock_fn. User application should implement this function depending on OS to delete the lock, for example on Windows: DeleteCriticalSection</p>
<p>hs_enter_lock_t *enter_lock_fn;</p>	<p>Prototype: typedef void hs_enter_lock_t(void *lock);</p> <p>Parameters: Synchronization lock handle (obtained with create_lock_fn)</p> <p>Return: none</p> <p>Description: This function will be used by HS TFTP to enter the synchronization lock, User application should implement this function depending on OS, for example on Windows: EnterCriticalSection</p>
<p>hs_leave_lock_t *leave_lock_fn;</p>	<p>Prototype: typedef void hs_leave_lock_t(void *lock);</p> <p>Parameters: Synchronization lock handle (obtained with create_lock_fn)</p> <p>Return: none</p> <p>Description: This function will be used by HS TFTP to exit the synchronization lock, User application should implement this function depending on OS, for example on Windows: LeaveCriticalSection</p>
<p>unsigned short int max_blksize</p>	<p>Maximum possible TFTP data block size, 0=use default, otherwise 8 to 65464 is the valid range. This is a global setting which will affect all TFTP sessions. If TFTP blocksize option negotiation is used, HS TFTP will not negotiate or use blocksize above this value. This setting also affects the size of allocated buffer for receiving data from socket layer.</p>

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_INIT	HS TFTP already initialised
HS_TFTP_RC_INVALID_PAR	Invalid parameter
HS_TFTP SOCK_STARTUP_FAIL	UDP layer initialisation failed
HS_TFTP_RC_NO_MEM	No free contexts or not enough memory for HS TFTP structures

Sample usage:

```

/*
 * initiase TFTP library
 */
int init_tftp_library(void)
{
    tftp_init_t init = {0};
    int rc;

    init.get_tx_buffer = tftp_get_buf_tx_cb;
    init.get_rx_buffer = tftp_get_buf_rx_cb;
    init.start_timer_fn = tftp_start_timer_cb;
    init.stop_timer_fn = hs_tftp_stop_timer_cb;
    init.create_lock_fn = hs_create_lock;
    init.delete_lock_fn = hs_delete_lock;
    init.enter_lock_fn = hs_enter_lock;
    init.leave_lock_fn = hs_leave_lock;

    init.max_blksize = 65464; // max blkoption option size value from RFC 2348

    rc = HsTftpInit(&init);

    if (rc == HS_TFTP_RC_OK)
        tftp_initialised = 1;

    return rc;
}

```

2.2 HsTftpDestroy

Declaration:

```
extern void HsTftpDestroy(void);
```

Summary:

De-allocates resources and closes HS TFTP services.

Parameters:

None

Return values:

None

Sample usage:

```
HsTftpDestroy();
```

2.3 HsTftpStartServer

Declaration:

```
extern int HsTftpStartServer(
    hs_tftp_srv_ev_fn_t *callback_fn,           // event callback (used for server mode)
    unsigned short int  blocksize,            // 0= default (don't support blksize option)
                                                // or blocksize to use in option negotiation,
                                                // valid // values: 8 to 65464
    unsigned short      tftp_port            // initial TFTP port of server
);
```

Summary:

Starts server operation of HS TFTP module.

Parameters:

unsigned short int blocksize – controls the handling of blocksize option received in WRQ and RRQ requests (file read and write requests) from clients. Set to 0 to not support the blocksize option and always use default block size of 512 bytes. Set to maximum supported block size in the range from 8 to 65464. Please note that this value also cannot exceed the value of max_blksize set in init structure when HsTftpInit was called. If client request contains TFTP blocksize option, the request will be acknowledged by HS TFTP server with OACK packet containing either the requested blocksize value or a lower value if blocksize parameter in this call is lower.

unsigned short tftp_port – UDP port number the HS TFTP module listens for incoming TFTP client commands, recommended default value is 69.

hs_tftp_srv_ev_fn_t *callback_fn – function pointer to callback function in application layer which receives events related to the status of the server operations:

```
typedef long hs_tftp_srv_ev_fn_t(int ev_code, long arg1, long arg2);
```

ev_code – one of the following values:

HS_TFTP_EV_WRITE_REQ – write request received from remote TFTP client (client wants to send file)

Arg1 – pointer to null terminated filename string

Arg2 – 32 bit remote IP address (address of TFTP client sending this request)

HS_TFTP_EV_READ_REQ - read request received from remote TFTP client (client wants to get file)

Arg1 – pointer to null terminated filename string

Arg2 – 32 bit remote IP address (address of TFTP client sending this request)

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_UDP_SOCKET_OPEN	UDP layer failed to open session on specified UDP port
HS_TFTP_RC_INVALID_PAR	Invalid parameter

Sample usage:

```
/*
 * Event callback from TFTP library (server mode)
 */
long hs_tftp_ev_server_handler(int ev_code, long arg1, long arg2)
{
    unsigned char *file;

    if (!tftp_initialised)
        return 0;
```

```

    if (!handle)
        return 0;

    switch (ev_code)
    {
    case HS_TFTP_EV_WRITE_REQ:
        file = (unsigned char *)arg1;
        process_file_write_request(file, arg2);
        break;

    case HS_TFTP_EV_READ_REQ:
        file = (unsigned char *)arg1;
        process_file_read_request(file, arg2);
        break;
    }

    return 0;
}

/* Start server mode */
void StartServer(void)
{
    int rc;

    if (!tftp_initialised)
    {
        rc = init_tftp_library();
        if (rc != HS_TFTP_RC_OK)
        {
            printf("HsTFTP init failed. Error %d\n", rc);
            return;
        }
    }

    if (!server_started)
    {
        rc = HsTftpStartServer(hs_tftp_ev_server_handler, 65464, TFTP_PORT);
        if (rc != HS_TFTP_RC_OK)
        {
            printf("Server failed to start. HS TFTP Error: %d\n", rc);
            return;
        }
        server_started = TRUE;
    }
    else
    {
        printf("Server mode already running\n");
        return;
    }

    testing_hs_tftp = 1;

    printf("Server mode started OK\n");
}

```

2.4 HsTftpServerStartReceive

Declaration:

```
extern
int HsTftpServerStartReceive(
    hs_tftp_ev_fn_t *callback_fn, // event callback (used for indication of completion or error)
    long *handle, // Connection handle returned after transfer initiated
    long user_handle); // upper layer context handle
```

Summary:

Start receiving requested file from remote peer (in Server mode). This function is called in response to HS_TFTP_EV_WRITE_REQ event, which occurs when a write request for a file is received from remote TFTP client

Parameters:

hs_tftp_ev_fn_t *callback_fn – function pointer to a callback function in application (user) layer code which receives notifications related to this transfer session.

```
typedef long hs_tftp_ev_fn_t(long handle, int ev_code, long arg1, long arg2);
```

handle – application layer handle

ev_code – event, one of the following:

HS_TFTP_EV_ERR_TMOUT – session timed out and closed

HS_TFTP_RX_COMPLETE – receive transfer completed successfully

Arg1 and arg2 are currently unused

long *handle – pointer to long variable to received TFTP module connection handle after file transfer is initiated.

long user_handle – application layer context handle. This handle is saved into corresponding TFTP session context and is returned unchanged as a parameter in various notification callbacks from HS TFTP to application layer code.

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_UDP SOCK_OPEN	UDP layer failed to open session
HS_TFTP_RC_INVALID_PAR	Invalid parameter

Sample usage:

```
/* server mode - process file write request */
void process_file_write_request(unsigned char *filename, long arg2)
{
    int rc;
    unsigned char ipstr[20] = {0};
    server_conn_ctx_t *pCtx;

    HsSockInetNtoa(arg2, ipstr);
    printf("file write request from (%s) %s\n", ipstr, filename);

    pCtx = tftp_alloc_srvconn_ctx();
    if (!pCtx)
    {
        printf("rejected: no free contexts\n");
        HsTftpRejectRq(HS_TFTP_EV_WRITE_REQ,
                     HS_TFTP_SERVER_ERR_USER, "no free contexts");
        return;
    }
}
```

```

pCtx->hFile = CreateFile(filename,          // file to open
    GENERIC_WRITE,                          // open for writing
    0,                                       // no sharing
    NULL,                                    // default security
    OPEN_ALWAYS,                            // overwrite existing file
    FILE_ATTRIBUTE_NORMAL,                  // normal file
    NULL);

if (pCtx->hFile == INVALID_HANDLE_VALUE)
    pCtx->hFile = NULL;

if (!pCtx->hFile)
{
    printf("rejected: file open error\n");

    tftp_free_srvconn_ctx(pCtx);

    /* Reject request */
    HsTftpRejectRq(HS_TFTP_EV_WRITE_REQ, HS_TFTP_SERVER_ERR_FIO, NULL);
    return;
}

pCtx->is_server_session = TRUE;
pCtx->is_send = FALSE;

tftp_add_srvconn(pCtx);

rc = HsTftpServerStartReceive(hs_tftp_server_ev_handler, &pCtx->tftp_handle, (long)pCtx);
if (rc != HS_TFTP_RC_OK)
{
    printf("Server Start Receive failed RC (%u)\n", rc);
    hs_tftp_cleanup_ctx(pCtx);
    return;
}

printf("Server receive transfer started\n");
}

```

2.5 HsTftpServerStartSend

Declaration:

```

extern int HsTftpServerStartSend(
    hs_tftp_ev_fn_t    *callback_fn, // event callback (used for infication of completion or error)
    long               *handle,      // Connection handle returned after transfer initiated
    long               user_handle); // upper layer context handle

```

Summary:

Start sending requested file to remote peer (in Server mode). This function is called in response to HS_TFTP_EV_READ_REQ event, which occurs when a read request for a file is received from remote TFTP client

Parameters:

hs_tftp_ev_fn_t *callback_fn – function pointer to a callback function in application (user) layer code which receives notifications related to this transfer session.

typedef long hs_tftp_ev_fn_t(long handle, int ev_code, long arg1, long arg2);

handle – application layer handle

ev_code – event, one of the following:

HS_TFTP_EV_ERR_TMOU – session timed out and closed

HS_TFTP_RX_COMPLETE – receive transfer completed successfully
Arg1 and arg2 are currently unused

long *handle – pointer to long variable to received TFTP module connection handle after file transfer is initiated.

long user_handle – application layer context handle. This handle is saved into corresponding TFTP session context and is returned unchanged as a parameter in various notification callbacks from HS TFTP to application layer code.

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_UDP SOCK_OPEN	UDP layer failed to open session
HS_TFTP_RC_INVALID_PAR	Invalid parameter

Sample usage:

```

/* server mode - process file read request */
void process_file_read_request(unsigned char *filename, long arg2)
{
    int rc;
    long fsize = 0;
    unsigned char ipstr[20] = {0};
    server_conn_ctx_t *pCtx;

    HsSockInetNtoa(arg2, ipstr);
    printf("file read request from (%s) %s\n", ipstr, filename);

    pCtx = tftp_alloc_srvconn_ctx();
    if (!pCtx)
    {
        printf("rejected: no free contexts\n");
        HsTftpRejectRq(HS_TFTP_EV_READ_REQ,
                      HS_TFTP_SERVER_ERR_USER, "no free contexts");
        return;
    }

    pCtx->hFile = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ, NULL,
                            OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (pCtx->hFile == INVALID_HANDLE_VALUE)
        pCtx->hFile = NULL;

    fsize = (long)GetCompressedFileSize(filename, NULL);
    if ((!pCtx->hFile) || (!fsize))
    {
        printf("rejected: file open error\n");
        HsTftpRejectRq(HS_TFTP_EV_READ_REQ, HS_TFTP_SERVER_ERR_FNOTFOUND, NULL);
        hs_tftp_cleanup_ctx(pCtx);
        return;
    }

    pCtx->fblock = pCtx->rxbuf;

    pCtx->is_server_session = TRUE;
    pCtx->is_send = TRUE;

    tftp_add_srvconn(pCtx);

```

```

rc = HsTftpServerStartSend(hs_tftp_server_ev_handler, &pCtx->tftp_handle, (long)pCtx);
if (rc != HS_TFTP_RC_OK)
{
    printf("Server Start Send failed RC (%u)\n", rc);
    hs_tftp_cleanup_ctx(pCtx);
    return;
}

pCtx->total_blocks = fsize / TFTP_BLK_SIZE;

printf("Server send transfer started\n");
}

```

2.6 HsTftpRejectRq

Declaration:

```
extern void HsTftpRejectRq(int rq, int reason, unsigned char *str);
```

Summary:

Sends TFTP ERROR packet (in server mode) to remote TFTP client with specified reason code and descriptive ASCII string. This function may be called in response to HS_TFTP_EV_WRITE_REQ and HS_TFTP_EV_READ_REQ events.

Parameters:

rq – currently unused

reason – reason code, one of the following:

HS_TFTP_SERVER_ERR_FIO - FILE I/o error

HS_TFTP_SERVER_ERR_FNOTFOUND - file not found

HS_TFTP_SERVER_ERR_USER - user defined error, send supplied error string

Str – pointer to zero terminated ASCII string to send with the ERROR packet, only valid if reason is HS_TFTP_SERVER_ERR_USER.

Return values:

None

Sample usage:

```
HsTftpRejectRq(HS_TFTP_EV_WRITE_REQ, HS_TFTP_SERVER_ERR_USER, "no free contexts");
```

2.7 HsTftpTransfer

Declaration:

extern

```

int HsTftpTransfer(
    int          operation,          // packet type
    unsigned long dest_ip,          // remote end IP address
    unsigned char *filename,        // filename (0 terminated)
    hs_tftp_ev_fn_t *callback_fn,   // event callback (used for indication of completion or error)
    unsigned short tftp_port,       // initial TFTP port of server
    unsigned short int blocksize,   // 0=block size option not supported (default 512
                                     bytes // used) or blocksize to use in option negotiation,
                                     valid // values 8 to 65464
    long         *handle,           // Connection handle returned after transfer initiated
    long         user_handle);      // upper layer context

```

Summary:

This function is used to initiate client mode Send or Receive file transfer.

Parameters:

operation – type of transfer to start:

TFTP_OP_SEND_FILE – start sending file to remote TFTP server

TFTP_OP_GET_FILE – start receiving file from remote TFTP server

dest_ip – 32 bit remote TFTP server IP address

filename – name of file to send or receive (pointer to zero terminated ASCII string)

callback_fn – pointer to event callback used for indication of completion or errors.

Callback function prototype:

```
typedef long hs_tftp_ev_fn_t(long handle, int ev_code, long arg1, long arg2);
```

handle – user application layer context

ev_code – event, one of the following:

HS_TFTP_EV_ERR_TMOU – TFTP transfer timed out, session closed

HS_TFTP_RX_COMPLETE – receive transfer complete

HS_TFTP_EV_ERR_RXED – remote TFTP server send aborted transfer, arg2 – pointer to zero terminated error string from received TFTP ERROR packet

tftp_port – TFTP port at remote server to send the request to (normally 69)

*handle – pointer to long variable to receive TFTP session handle after the function returns

unsigned short int blocksize – TFTP data payload size to be used for this TFTP client session. Set to 0 to not to use TFTP blocksize option negotiation and to use default block size of 512. Set to desired block size (valid values are 8 to 65464) to attempt blocksize option negotiation with the server. If the server supports blocksize TFTP option, the negotiated blocksize shall be used for the transfer. If the server does not support TFTP blocksize option, still default 512 blocksize is used.

user_handle – application layer context handle to be stored by HS TFTP module in this session context.

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_UDP SOCK_OPEN	UDP layer failed to open session
HS_TFTP_RC_INVALID_PAR	Invalid parameter
HS_TFTP_RC_NO_FREE_CTX	HS TFTP module has no more free session contexts
HS_TFTP_RC_UDP SOCK_SEND	UDP layer failed to send TFTP packet

Sample usage:

```
rc = HsTftpTransfer(TFTP_OP_SEND_FILE, dest_ip, &filename[i],
    hs_tftp_ev_handler, TFTP_PORT, &pCtx->tftp_handle, (long)pCtx);
```

2.8 HsTftpTimerExpired**Declaration:**

```
extern void HsTftpTimerExpired(long handle);
```

Summary:

Function called from user code when timer previously started by HS TFTP has expired

Parameters:

handle – TFTP session handle, this handle is passed as a parameter to callback function

hs_tftp_start_timer_t when HS TFTP starts a timer

Return values:

None

2.9 HsTftpAbort

Declaration:

```
extern int HsTftpAbort(long handle);
```

Summary:

Abort current operation and cleanly disconnect remote end based on passed connection handle. This function is going to send TFTP ERROR packet to remote end with the string "Aborted by user" and cleanup local TFTP session context.

Parameters:

handle – TFTP module session handle

Return values:

Value	Description
HS_TFTP_RC_OK	Success
HS_TFTP_RC_NOT_INIT	TFTP module not initialised
HS_TFTP_RC_INVALID_PAR	Invalid parameter

Sample usage:

```
HsTftpAbort(pCtx->tftp_handl);
```

2.10 HsTftpErrStr

Declaration:

```
extern unsigned char *HsTftpErrStr(int rc);
```

Summary:

Use this function to get a pointer to a null terminated string which describes HS TFTP return code rc.

Parameters:

rc - HS TFTP integer return code

Return values:

Pointer to a null terminated string which describes HS TFTP return code rc.

Sample usage:

```
printf("HsTftpTransfer failed, Error: %s", HsTftpErrStr(rc));
```

3 Application Notes

3.1 Sending File Considerations

Client mode: To send file in client mode, setup all callbacks and call HsTftpTransfer. Every time, HS TFTP receives an acknowledgement from remote end, it will call get_tx_buffer callback function – give it the next data block to transmit. After reception of acknowledge from remote end for last data block, HS TFTP is going to call the get_tx_buffer callback again – give it NULL pointer and zero length and consider transfer complete.

Server mode: The procedure for sending files in server mode is similar to client mode, except it is initiated not with HsTftpStart transfer, but with HsTftpServerStartSend

3.2 Receiving File Considerations

Client mode: To receive file in client mode, setup all callbacks and call HsTftpTransfer. Every time HS TFTP receives, correctly processes and acknowledges a packet from remote end, it will call get_rx_buffer callback function – give it the pointer of the next space to store the received data into. When HS TFTP receives the last data packet from remote end it will call event callback function with HS_TFTP_RX_COMPLETE return code. At this point consider receive transfer complete.

Server mode: The procedure for receiving files in server mode is similar to client mode, except it is initiated not with HsTftpStart transfer, but with HsTftpServerStartReceive