



# HS TFTP for Embedded Systems v1.2 User Manual

Revision: 1.0

Date: 19 June 2007

<b>HS TFTP FOR EMBEDDED SYSTEMS V1.2 USER MANUAL</b>	<b>1</b>
<b>REVISION: 1.0</b>	<b>1</b>
<b>DATE: 19 JUNE 2007</b>	<b>1</b>
<b>1 INTRODUCTION</b>	<b>2</b>
1.1 SOFTWARE ARCHITECTURE	2
1.2 MODEL OF OPERATION	2
1.3 TARGET SYSTEM REQUIREMENTS	3
1.3.1 HARDWARE REQUIREMENTS	3
1.3.2 SOFTWARE REQUIREMENTS	3
1.4 OVERVIEW OF PRODUCT FEATURES	3
<b>2 HS TFTP API</b>	<b>4</b>
2.1 HSTFTPINIT	4
2.2 HSTFTPDESTROY	5
2.3 HSTFTPSTARTSERVER	6
2.4 HSTFTPSTARTRECEIVE	7
2.5 HSTFTPSTARTSEND	9
2.6 HSTFTPREJECTRQ	10
2.7 HSTFTPTRANSFER	11
2.8 HSTFTPTIMEREXPIRED	12
2.9 HSTFTPABORT	12
<b>3 APPLICATION NOTES</b>	<b>13</b>
3.1 SENDING FILE CONSIDERATIONS	13
3.2 RECEIVING FILE CONSIDERATIONS	13
3.3 PORTING CONSIDERATIONS	13
3.3.1 INTERFACING WITH TARGET SYSTEM ETHERNET DRIVER	13
3.3.1.1 Frame transmission to Ethernet driver:	13
3.3.1.2 Frame reception from Ethernet driver into HS TFTP	13
3.3.2 INTERFACING WITH TARGET SYSTEM TIMER SERVICES	13
3.3.3 INTERFACING WITH TARGET SYSTEM FILE SERVICES	13

# 1 Introduction

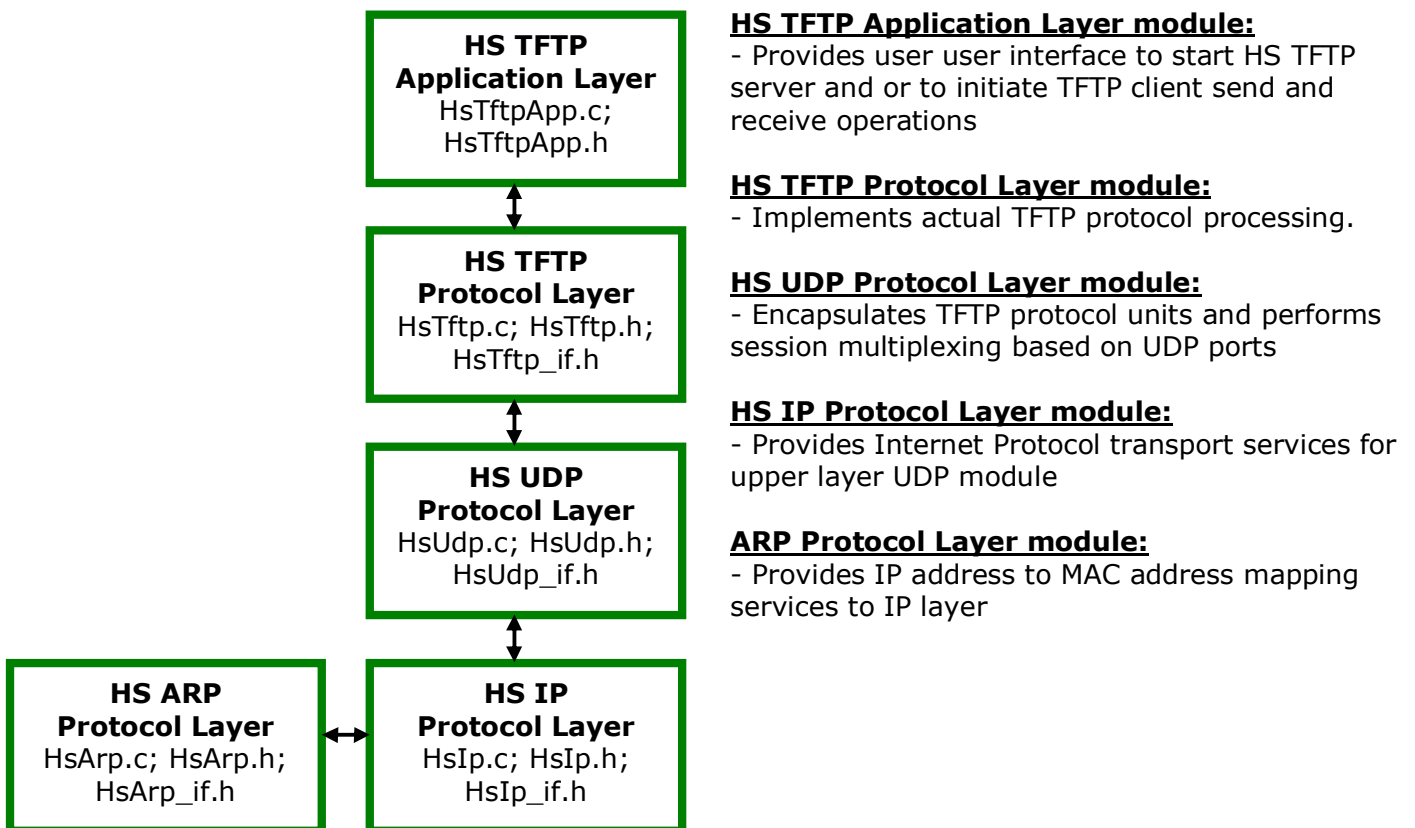
HS TFTP is a software library in C (supplied with full source code) which implements Trivial File Transfer Protocol (TFTP) over UDP/IP according to RFC 1350.

HS TFTP includes integrated UDP/IP stack necessary to transport TFTP protocol and is ideal self-contained solution for embedded systems, which may not otherwise have its own UDP/IP layer.

Current Version of HS TFTP supports Big-endian memory model only

HS TFTP is supplied in a form of uncompiled ANSI C source code and would require minimal integration effort by the customer, assisted by Hillstone Software after sales support

## 1.1 Software Architecture



## 1.2 Model of Operation

The HS TFTP Application layer initialises HS TFTP module, providing interface callbacks for the services used by HS TFTP protocol module: timer management, memory management, file operations and event processing callbacks. HS TFTP module handles all protocol information flow, error recovery, acknowledgements, timeouts, retransmissions etc. 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.

At a lower layer HS TFTP interfaces to HS UDP module which provides UDP transport and session multiplexing services. In turn HS UDP talks to HS IP module which offers IP datagram transmission and reception service. HS IP calls HS ARP module to resolve IP address to MAC address and also communicates directly with the ethernet driver on the target system.

## 1.3 Target System Requirements

### 1.3.1 Hardware Requirements

- Target System must have physical ethernet interface
- Target System memory model must be Big-endian
- RAM requirements: With default number of contexts 10, approx 200 kilobytes of RAM is required in total.
- Flash / ROM / Code size requirements: code size of HS TFTP depends on target platform and on used compiler. As an rough guide, when compiled for Powerquicc MPC860 architecture, the code size is approx 115 kilobytes

### 1.3.2 Software Requirements

- HS TFTP is designed for integration into C language based software
- Standard C Libraries: HS TFTP uses the following standard C libraries: string.h; malloc.h; stdio.h; ctype.h
- Existing target software must provide functions for Ethernet transmission and reception.
- Existing target software must provide timer services to HS TFTP
- Existing target software must provide File system level services for opening, closing reading and writing files.

## 1.4 Overview of Product Features

Module	Feature Description
TFTP	<ul style="list-style-type: none"> <li>• RFC 1350 Trivial File Transfer protocol compliant implementation</li> <li>• Simultaneous concurrent server and client sessions (defined at maximum 10 sessions, but the define can be changed to suit customer requirement</li> <li>• 512 byte block size</li> </ul>
UDP	<ul style="list-style-type: none"> <li>• Minimum required implementation according to RFC768 User Datagram Protocol</li> <li>• Incoming / outgoing session multiplexing</li> <li>• UDP headers plus data checksum validation</li> </ul>
IP	<ul style="list-style-type: none"> <li>• Minimum working implementation of Internet Protocol (RFC791)</li> <li>• IP header checksum protection and validation</li> <li>• Protocol fields validation</li> <li>• Fragmentation or Reassembly are NOT supported</li> </ul>
ARP	<ul style="list-style-type: none"> <li>• Minimum working implementation of Address Resolution Protocol (RFC826)</li> <li>• Maintenance of ARP table of 50 entries</li> </ul>

## 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
unsigned char local_ip_addr_str[16]	Zero terminated ASCII string representing local IP address, eg "10.1.1.9"
hs_tftp_get_buf_t *get_tx_buffer hs_tftp_get_buf_t *get_rx_buffer	<p><b>Prototype</b> unsigned char *hs_tftp_get_buf_t(long handle, unsigned int *length, int *cmd);</p> <p><b>Parameters:</b> 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><b>Return:</b> 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><b>Description:</b> 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><b>Prototype:</b> long hs_tftp_start_timer_t(long handle, unsigned long secs);</p> <p><b>Parameters:</b> handle – application (user) layer context handle secs – number of seconds to timeout after.</p> <p><b>Return:</b> Timer handle. Currently the same as application (user) context handle</p> <p><b>Description:</b> 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><b>Prototype:</b> void hs_tftp_stop_timer_t(long handle);</p> <p><b>Parameters:</b> Handle – application (user) layer context handle</p> <p><b>Return:</b> No return</p> <p><b>Description:</b> This function will be used by HS TFTP module to stop (destroy) a timer previously started with start_timer_fn.</p>

**Return values:**

<b>Value</b>	<b>Description</b>
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

**Sample usage:**

```
// local ip address
static unsigned char our_ip_addr[] = "10.1.1.9";

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

    memcpy(init.local_ip_addr_str, our_ip_addr, strlen(our_ip_addr));
    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;

    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      tftp_port       // initial TFTP port of server
);
```

### Summary:

Starts server operation of HS TFTP module.

### Parameters:

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

handle – currently unused

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 SOCK_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(long handle, 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, 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\_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 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 = FLASHOpen(filename, "w");
    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\_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_SOCKET_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 = FLASHOpen(filename, "r");
if (pCtx->hFile)
FLASHInfo(filename, &fsize, NULL, NULL, NULL, 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
    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\_TMOUT – 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

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 tftp_handle);
```

**Summary:**

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

**Parameters:**

tftp\_handle - TFTP module session handle

**Return values:**

None

**Sample usage:**

```
/*
 * os timer callback
 */
static void TimerProc(void *context)
{
    server_conn_ctx_t *pCtx = (server_conn_ctx_t *)context;

    if (!pCtx) return;

    TIMERDelete(pCtx->Timer);

    HsTftpTimerExpired((long)pCtx->tftp_handle);
}
```

**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_handle);
```

## 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

### 3.3 Porting Considerations

#### 3.3.1 Interfacing with target system Ethernet driver

##### 3.3.1.1 Frame transmission to Ethernet driver:

HsIp and HsArp module transmit frame to Ethernet driver calling external function EthTransmitFrame. This function call will have to be replaced to call an existing API to send a frame out Ethernet interface

##### 3.3.1.2 Frame reception from Ethernet driver into HS TFTP

Whenever a frame is received from Ethernet, the existing Ethernet driver must call HsIp function HsEthernetReceivePacket in order to pass the received frame for processing to HS TFTP.

#### 3.3.2 Interfacing with target system Timer services

HS TFTP relies on existing software to provide timer start, stop and expired services. To start and stop a timer, HS TFTP calls callback function pointers start\_timer\_fn, stop\_timer\_fn.

When calling HsTftpInit, the user code must set these function pointers to local functions which start and cancel the timers respectively.

In the supplied HsTftpApp.c (example application layer) timer management functions are TIMERCreate, TIMERStartOneShot, TIMERDelete.

When a timer started by HS TFTP expires, the application layer must call HsTftpTimerExpired function

#### 3.3.3 Interfacing with target system File services

In order to create, open, close, read and write flash files, the application layer HsTftpApp.c supplied with the library uses external functions FLASHOpen, FLASHClose, FLASHRead, FLASHWrite. These function calls will have to be replaced with calls to existing file management functions.