Trivial File Transfer Protocol (TFTP)

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-1052
Revision 5.12

Contents

Chapter 1

Introduction to TFTP

The Trivial File Transfer Protocol (TFTP) is a lightweight protocol designed for file transfers. Unlike more robust protocols, TFTP does not perform extensive error checking and can also have limited performance because it is a stop-and-wait protocol. After a TFTP data packet is sent, the sender waits for an ACK to be returned by the recipient. Although this is simple, it does limit the overall TFTP throughput.

TFTP Requirements

In order to function properly, the NetX TFTP package requires that a NetX IP instance has already been created. In addition, UDP must be enabled on that same IP instance. The TFTP Client portion of the NetX TFTP package has no further requirements.

The TFTP Server portion of the NetX TFTP package has several additional requirements. First, it requires complete access to UDP *well known port 69* for handling all client TFTP requests. The TFTP Server is also designed for use with the FileX embedded file system. If FileX is not available, the user may port the portions of FileX used to their own environment. This is discussed in later sections of this guide.

TFTP File Names

TFTP file names should be in the format of the target file system (usually FileX). They should be NULL terminated ASCII strings, with full path information if necessary. There is no specified limit in the size of TFTP file names in the NetX TFTP implementation.

TFTP Messages

The TFTP has a very simple mechanism for opening, reading, writing, and closing files. There are basically 2-4 bytes of TFTP header underneath the UDP header. The definition of the TFTP file open messages has the following format:

ooooof…f0OCTET0

Where:

oooo 2-byte Opcode field

0x0001 -> Open for read
0x0002 -> Open for write

f…f n-byte Filename field

0 1-byte NULL termination character

OCTET ASCII “OCTET” to specify binary transfer

0 1-byte NULL termination character

The definition of the TFTP write, ACK, and error messages are slightly different and are defined as follows:

oooobbbbd…d

Where:

oooo 2-byte Opcode field

0x0003 -> Data packet
0x0004 -> ACK for last read
0x0005 -> Error condition

bbbb 2-byte Block Number field (1-n)

d…d n-byte Data field

Opcode	**Filename**	**NULL**	**Mode**	**NULL**
0x0001 (read)	File Name	0	OCTET	0
0x0002 (write)	File Name	0	OCTET	0

TFTP Communication

The TFTP Server utilizes the well-known UDP port 69 to field client requests. TFTP Clients may use any available UDP port. Data packets are fixed at 512 bytes, until the last packet. A packet containing fewer than 512 bytes signals the end of file. The general sequence of events is as follows:
TFTP Read File Requests:

1. Client Issues “Open For Read” request with the File Name and waits for a packet from Server.
2. Server sends the first 512 bytes of the file.
3. Client receives data, sends ACK, and waits for the next packet if the last packet had 512 bytes.
4. The sequence ends when a packet containing fewer than 512 bytes is received.

TFTP Write Requests:

1. Client Issues “Open for Write” request with the File Name and waits for an ACK with a block number of 0 from the Server.
2. When the Server is ready to write the file, it sends an ACK with a block number of zero.
3. Client sends the first 512 bytes of the file to the Server and waits for an ACK.
4. Server sends ACK after the bytes are written.
5. The sequence ends when the Client completes writing a packet containing fewer than 512 bytes.

TFTP Server Session Timer

The TFTP Server has a limited number of client request slots. If a client session appears to be dropped, that slot cannot be available for re-use. However if the NX_TFTP_SERVER_RETRANSMIT_ENABLE option is enabled, the NetX TFTP Server creates an session timer that monitors the timeout on each of its client sessions. When a session timeout expires it is terminated and any open files are closed. Thus the ‘slot’ becomes available for another TFTP Client request.

To set the timeout, adjust the configuration option NX_TFTP_SERVER_RETRANSMIT_TIMEOUT which by default is 200 timer ticks. The interval between which session timeouts are checked is

set by the NX_TFTP_SERVER_TIMEOUT_PERIOD which is 20 timer ticks by default.

TFTP Multi-Thread Support

The NetX TFTP Client services can be called from multiple threads simultaneously. However, read or write requests for a particular TFTP client instance should be done in sequence from the same thread.

TFTP RFCs

NetX TFTP is compliant with RFC1350 and related RFCs.

Chapter 2

Installation and Use of TFTP

This chapter contains a description of various issues related to installation, setup, and usage of the NetX TFTP component.

Product Distribution

TFTP for NetX is shipped on a single CD-ROM compatible disk. The package includes two source files and a PDF file that contains this document, as follows:

nx_tftp_client.h	Header file for TFTP Client for NetX
nx_tftp_client.c	C Source file for TFTP Client for NetX
nx_tftp_server.c	C Source file for TFTP Server for NetX
nx_tftp_server.h	Header file for TFTP Server for NetX
filex_stub.h	Stub file if FileX is not present
nx_tftp.pdf	PDF description of TFTP for NetX
demo_netx_tftp.c	NetX TFTP demonstration

TFTP Installation

In order to use TFTP for NetX, the entire distribution mentioned previously should be copied to the same directory where NetX is installed. For example, if NetX is installed in the directory "*threadx**arm7**green*" then the *nx_tftp_client.h*, *nx_tftp_client.c*, *nx_tftp_server.c and nx_tftp_server.h* files should be copied into this directory.

Using TFTP

To run a TFTP application, the application code must include the header files after it includes *tx_api.h, fx_api.h,* and *nx_api.h*, in order to use ThreadX, FileX, and NetX, respectively. Once the header files are included, the application code is then able to make the TFTP function calls specified later in this guide. The application must also include *nx_tftp_client.c* and *nx_tftp_server.c* in the build process. These files must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX TFTP.

Note that since TFTP utilizes NetX UDP services, UDP must be enabled with the *nx_udp_enable* call prior to using TFTP.

Small Example System

An example of how easy it is to use NetX TFTP is described in Figure 1.1 that appears below. In this example, the TFTP include file *nx_tftp_client.h and nx_tftp_server.h* are brought at line 11 and 12. Next, the TFTP server is created in "*tx_application_define*" at line 153. Note that the TFTP Server control block "*server*" was defined as a global variable at line 37 previously. After successful creation, a TFTP Server is started at line 233. At line 272 the TFTP Client is created. And finally, the client writes the file at line 299 and reads the file back at line 322.

```
 1 /* This is a small demo of TFTP on the high-performance NetX TCP/IP stack.  This demo
 2    relies on ThreadX and NetX , to show a simple file transfer from the client
 3    and then back to the server.  */
 4
 5 /* Indicate if using a NetX TFTP services. To port a NetX TFTP application to NetX TFTP
 6    undefine this term.  */
 7
 8
 9 #include    "tx_api.h"
10 #include    "nx_api.h"
11 #include    "nx_tftp_client.h"
12 #include    "nx_tftp_server.h"
13 #ifndef     NX_TFTP_NO_FILEX
14 #include    "fx_api.h"
15 #endif
16
17
18 #define     DEMO_STACK_SIZE         4096
19
20 /* To use another file storage utility define this symbol:
21 #define NX_TFTP_NO_FILEX
22 */
23
24 /* Define the ThreadX, NetX, and FileX object control blocks...  */
25
26 TX_THREAD               server_thread;
27 TX_THREAD               client_thread;
28 NX_PACKET_POOL          server_pool;
29 NX_IP                   server_ip;
30 NX_PACKET_POOL          client_pool;
31 NX_IP                   client_ip;
32 FX_MEDIA                ram_disk;
33
34 /* Define the NetX TFTP object control blocks.  */
35
36 NX_TFTP_CLIENT          client;
37 NX_TFTP_SERVER          server;
38
39 /* Define the application global variables */
40
41 #define                 CLIENT_ADDRESS  IP_ADDRESS(1, 2, 3, 5)
42 #define                 SERVER_ADDRESS  IP_ADDRESS(1, 2, 3, 4)
43
44
45 UINT                    error_counter = 0;
46
47 /* Define buffer used in the demo application.  */
48 UCHAR                   buffer[255];
49 ULONG                   data_length;
50
51
52 /* Define the memory area for the FileX RAM disk.  */
53 #ifndef NX_TFTP_NO_FILEX
54 UCHAR                   ram_disk_memory[32000];
55 UCHAR                   ram_disk_sector_cache[512];
56 #endif
57
58
59 /* Define function prototypes.  */
60
61 VOID    _fx_ram_driver(FX_MEDIA *media_ptr);
62 VOID    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
63 void    client_thread_entry(ULONG thread_input);
64 void    server_thread_entry(ULONG thread_input);
65
66
67 /* Define main entry point.  */
68
69 int main()
70 {
71
72     /* Enter the ThreadX kernel.  */
73     tx_kernel_enter();
74 }
75
76
77 /* Define what the initial system looks like.  */
78
79 void    tx_application_define(void *first_unused_memory)
80 {
81
82 UINT    status;
83 UCHAR   *pointer;
84
85
86     /* Setup the working pointer.  */
87     pointer =  (UCHAR *) first_unused_memory;
88
89
```

```
    /* Create the main TFTP server thread.  */
    status = tx_thread_create(&server_thread, "TFTP Server Thread", server_thread_entry, 0,
                              pointer, DEMO_STACK_SIZE,
                              4,4, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer += DEMO_STACK_SIZE ;

    /* Check for errors.  */
    if (status)
        error_counter++;


    /* Create the main TFTP client thread at a slightly lower priority.  */
    status = tx_thread_create(&client_thread, "TFTP Client Thread", client_thread_entry, 0,
                              pointer, DEMO_STACK_SIZE,
                              5, 5, TX_NO_TIME_SLICE, TX_DONT_START);

    pointer += DEMO_STACK_SIZE ;

    /* Check for errors.  */
    if (status)
        error_counter++;

    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Note: The data portion of a packet is exactly 512 bytes, but the packet payload size must
       be at least 580 bytes. The remaining bytes are used for the UDP, IP, and Ethernet
       headers and byte alignment requirements. */

    status =  nx_packet_pool_create(&server_pool, "TFTP Server Packet Pool", NX_TFTP_PACKET_SIZE, pointer, 8192);
    pointer = pointer + 8192;

    /* Check for errors.  */
    if (status)
        error_counter++;

    /* Create the IP instance for the TFTP Server.  */
    status = nx_ip_create(&server_ip, "NetX Server IP Instance", SERVER_ADDRESS, 0xFFFFFF00UL,
                                      &server_pool, _nx_ram_network_driver, pointer, 2048, 1);
    pointer = pointer + 2048;

    /* Check for errors.  */
    if (status)
        error_counter++;

    /* Enable ARP and supply ARP cache memory for IP Instance 0.  */
    status =  nx_arp_enable(&server_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

    /* Check for errors.  */
    if (status)
        error_counter++;

    /* Enable UDP.  */
    status =  nx_udp_enable(&server_ip);

    /* Check for errors.  */
    if (status)
        error_counter++;


    /* Create the TFTP server.  */
    status =  nx_tftp_server_create(&server, "TFTP Server Instance", &server_ip, &ram_disk,
                                    pointer, DEMO_STACK_SIZE, &server_pool);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Check for errors for the server.  */
    if (status)
        error_counter++;

    /* Create a packet pool for the TFTP client.  */

    /* Note: The data portion of a packet is exactly 512 bytes, but the packet payload size must
       be at least 580 bytes. The remaining bytes are used for the UDP, IP, and Ethernet
       headers and byte alignment requirements. */

    status =  nx_packet_pool_create(&client_pool, "TFTP Client Packet Pool", NX_TFTP_PACKET_SIZE, pointer, 8192);
    pointer =  pointer + 8192;

    /* Create an IP instance for the TFTP client.  */
    status = nx_ip_create(&client_ip, "TFTP Client IP Instance", CLIENT_ADDRESS, 0xFFFFFF00UL,
                                      &client_pool, _nx_ram_network_driver, pointer, 2048, 1);
    pointer = pointer + 2048;

    /* Enable ARP and supply ARP cache memory for IP Instance 1.  */
    status =  nx_arp_enable(&client_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

    /* Enable UDP for client IP instance.  */
    status |=  nx_udp_enable(&client_ip);
    status |= nx_icmp_enable(&client_ip);

    tx_thread_resume(&client_thread);
}

void server_thread_entry(ULONG thread_input)
{

UINT        status, running;


    /* Allow time for the network driver and NetX to get initialized. */
    tx_thread_sleep(100);

#ifndef  NX_TFTP_NO_FILEX

    /* Format the RAM disk - the memory for the RAM disk was defined above.  */
    status = fx_media_format(&ram_disk,
                             _fx_ram_driver,                /* Driver entry             */
                             ram_disk_memory,               /* RAM disk memory pointer  */
                             ram_disk_sector_cache,         /* Media buffer pointer     */
                             sizeof(ram_disk_sector_cache), /* Media buffer size        */
                             "MY_RAM_DISK",                 /* Volume Name              */
                             1,                             /* Number of FATs           */
                             32,                            /* Directory Entries        */
                             0,                             /* Hidden sectors           */
                             256,                           /* Total sectors            */
                             128,                           /* Sector size              */
                             1,                             /* Sectors per cluster      */
                             1,                             /* Heads                    */
                             1);                            /* Sectors per track        */

    /* Check for errors.  */
    if (status != FX_SUCCESS)
    {
        return;
    }
```

```
219
220     /* Open the RAM disk.  */
221     status = fx_media_open(&ram_disk, "RAM DISK", _fx_ram_driver, ram_disk_memory, ram_disk_sector_cache,
sizeof(ram_disk_sector_cache));
222
223     /* Check for errors.  */
224     if (status != FX_SUCCESS)
225     {
226         return;
227     }
228
229 #endif /*  NX_TFTP_NO_FILEX */
230
231
232     /* Start the NetX TFTP server.  */
233     status =  nx_tftp_server_start(&server);
234
235     /* Check for errors.  */
236     if (status)
237     {
238         error_counter++;
239         return;
240     }
241
242     /* Run for a while */
243     running = NX_TRUE;
244     while(running)
245         tx_thread_sleep(200);
246
247     nx_tftp_server_delete(&server);
248
249
250 }
251
252 /* Define the TFTP client thread.  */
253
254 void    client_thread_entry(ULONG thread_input)
255 {
256
257 NX_PACKET   *my_packet;
258 UINT        status;
259 UINT        all_done = NX_FALSE;
260
261
262     /* Allow time for the network driver and NetX to get initialized. */
263     tx_thread_sleep(100);
264
265
266
267     /* The TFTP services used below include the NetX equivalent service which will work with
268        NetX TFTP.
269     */
270
271     /* Create a TFTP client.  */
272     status =  nx_tftp_client_create(&client, "TFTP Client", &client_ip, &client_pool);
273
274     /* Check status.  */
275     if (status)
276         return;
277
278     /* Open a TFTP file for writing.  */
279     status =  nx_tftp_client_file_open(&client, "test.txt", SERVER_ADDRESS, NX_TFTP_OPEN_FOR_WRITE, 100);
280
281     /* Check status.  */
282     if (status)
283         return;
284
285     /* Allocate a TFTP packet.  */
286     status =  nx_tftp_client_packet_allocate(&client_pool, &my_packet, 100);
287     /* Check status.  */
288     if (status)
289         error_counter++;
290
291     /* Write ABCs into the packet payload!  */
292     memcpy(my_packet -> nx_packet_prepend_ptr, "ABCDEFGHIJKLMNOPQRSTUVWXYZ  ", 28);
293
294     /* Adjust the write pointer.  */
295     my_packet -> nx_packet_length =  28;
296     my_packet -> nx_packet_append_ptr =  my_packet -> nx_packet_prepend_ptr + 28;
297
298     /* Write this packet to the file via TFTP.  */
299     status =  nx_tftp_client_file_write(&client, my_packet, 100);
300
301     /* Check status.  */
302     if (status)
303         error_counter++;
304
305     /* Close this file.  */
306     status =  nx_tftp_client_file_close(&client);
307
308     /* Check status.  */
309     if (status)
310         error_counter++;
311
312     /* Open the same file for reading.  */
313     status =  nx_tftp_client_file_open(&client, "test.txt", SERVER_ADDRESS, NX_TFTP_OPEN_FOR_READ, 100);
314
315     /* Check status.  */
316     if (status)
317         error_counter++;
318     do
319     {
320
321     /* Read the file back.  */
322         status =  nx_tftp_client_file_read(&client, &my_packet, 100);
323         /* Check for retranmission/dropped packet error. Benign. Try again... */
324         if (status == NX_TFTP_INVALID_BLOCK_NUMBER)
325         {
326
327             continue;
328         }
329         else if (status == NX_TFTP_END_OF_FILE)
330         {
331
332             /* All done. */
333             all_done = NX_TRUE;
334         }
335         else if (status != NX_SUCCESS)
336         {
337
338             /* Internal error, invalid packet or error on read. */
339             break;
340         }
341
342
343         /* Do something with the packet data and release when done. */
344         nx_packet_data_retrieve(my_packet, buffer, &data_length);
345         buffer[data_length] = 0;
346         printf("Receive data: %s\n", buffer);
```

```
347
348         printf("release packet in demo.\n");
349
350         nx_packet_release(my_packet);
351
352     } while (all_done == NX_FALSE);
353
354     /* Close the file again.  */
355     status =  nx_tftp_client_file_close(&client);
356
357     /* Check status.  */
358     if (status)
359         error_counter++;
360
361     /* Delete the client.  */
362     status =  nx_tftp_client_delete(&client);
363
364     /* Check status.  */
365     if (status)
366         error_counter++;
367
368     return;
369 }
```

Figure 1.1 Example of TFTP use with NetX

Configuration Options

There are several configuration options for building TFTP for NetX. The following list describes each in detail. Unless otherwise specified, these options are found in *nx_tftp_client.h* and *nx_tftp_server.h*.

Define	Meaning
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic TFTP error checking. It is typically used after the application has been debugged.
NX_TFTP_SERVER_PRIORITY	The priority of the TFTP server thread. By default, this value is defined as 16 to specify priority 16.
NX_TFTP_SERVER_TIME_SLICE	The time slice for the TFTP Server to run before yielding to other threads of the same priority. The default value is 2.
NX_TFTP_MAX_CLIENTS	The maximum number of clients the server can handle at one time. By default, this value is 10 to support 10 clients at once.
NX_TFTP_ERROR_STRING_MAX	The maximum number of characters in the error string. By default, this value is 64.
NX_TFTP_NO_FILEX	Defined, this option provides a stub for FileX dependencies. The TFTP Client will function without any change if this option is defined. The TFTP Server will need to either be modified or the user will have to create a handful of FileX services in order to function properly.
NX_TFTP_TYPE_OF_SERVICE	Type of service required for the TFTP UDP requests. By default, this value is defined as

NX_IP_NORMAL to indicate normal IP packet service.

NX_TFTP_FRAGMENT_OPTION Fragment enable for TFTP UDP requests. By default, this value is NX_DONT_FRAGMENT to disable TFTP UDP fragmenting.

NX_TFTP_TIME_TO_LIVE Specifies the number of routers this packet can pass before it is discarded. The default value is set to 0x80.

NX_TFTP_SOURCE_PORT This option allows a TFTP Client application to specify the TFTP Client UDP socket port. It is defaulted to NX_ANY_PORT.

NX_TFTP_SERVER_RETRANSMIT_ENABLE

Enables the TFTP server's timer to check each TFTP client session with for recent activity (either an ACK or data packet). When the session timeout expires after the maximum number of times, it is assumed the connection was lost. The Server clears the Client request, closes any open files and makes the connection request available for the next Client. The default setting is disabled.

NX_TFTP_SERVER_TIMEOUT_PERIOD

Specifies the interval when the TFTP server timer entry function checks Client connections for receiving any packets. The default value is 20 (timer ticks).

NX_TFTP_SERVER_RETRANSMIT_TIMEOUT

This is the timeout for receiving a valid ACK or data packet from

	the Client. The default value is 200 (timer ticks).
NX_TFTP_SERVER_MAX_RETRIES	Specifies the maximum number of times the Client session retransmit timeout is renewed. Thereafter, the session is closed by the Server.
NX_TFTP_MAX_CLIENT_RETRANSMITS	Specifies the maximum number of times the Server receives a duplicate ACK or data packet from the Client (which it drops) without sending an error message to the Client and closing the session. Has no effect if NX_TFTP_SERVER_RETRANSMIT_ENABLE is defined.

Chapter 3

Description of TFTP Services

This chapter contains a description of all NetX TFTP services (listed below) in alphabetic order.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **NX_DISABLE_ERROR_CHECKING** define that is used to disable API error checking, while non-bold values are completely disabled.

nx_tftp_client_create
Create a TFTP client instance

nx_tftp_client_delete
Delete a TFTP client instance

nx_tftp_client_error_info_get
Get client error information

nx_tftp_client_file_close
Close client file

nx_tftp_client_file_open
Open client file

nx_tftp_client_file_read
Read a block from client file

nx_tftp_client_file_write
Write block to client file

nx_tftp_client_packet_allocate
Allocate packet for client file write

nx_tftp_client_set_interface
Set the physical interface for TFTP requests

nx_tftp_server_create
Create TFTP server

nx_tftp_server_delete
Delete TFTP server

nx_tftp_server_start
Start TFTP Server

nx_tftp_server_stop
Stop TFTP Server

nx_tftp_client_create

Create a TFTP client instance

Prototype

```
UINT nx_tftp_client_create(NX_TFTP_CLIENT *tftp_client_ptr,
      CHAR *tftp_client_name, NX_IP *ip_ptr, NX_PACKET_POOL *pool_ptr);
```

Description

This service creates a TFTP client instance for the previously created IP instance.

Important Note: The application must make certain the supplied IP and packet pool are already created. In addition, UDP must be enabled for the IP instance prior to calling this service.

Input Parameters

tftp_client_ptr	Pointer to TFTP client control block.
tftp_client_name	Name of this TFTP client instance
ip_ptr	Pointer to previously created IP instance.
pool_ptr	Pointer to packet pool TFTP client instance.

Return Values

NX_SUCCESS	**(0x00)**	**Successful TFTP create.**
NX_TFTP_INVALID_SERVER_ADDRESS	**(0x08)**	**Invalid Server IP address received**
NX_TFTP_NO_ACK_RECEIVED	**(0x09)**	**Server ACK not received**
NX_PTR_ERROR	(0x16)	Invalid IP, pool, or TFTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization and Threads

Example

```
/* Create a TFTP client instance.  */
  status =  nx_tftp_client_create(&my_tftp_client, "My TFTP Client",
                                                    &my_ip, &pool_ptr);

/* If status is NX_SUCCESS a TFTP client instance was successfully created.  */
```

nx_tftp_client_delete

Delete a TFTP client instance

Prototype

```
UINT nx_tftp_client_delete(NX_TFTP_CLIENT *tftp_client_ptr);
```

Description

This service deletes a previously created TFTP client instance.

Input Parameters

tftp_client_ptr	Pointer to previously created TFTP client instance.

Return Values

NX_SUCCESS	(0x00)	Successful TFTP client delete.
NX_PTR_ERROR	(0x16)	Invalid TFTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete a TFTP client instance.  */
status =  nx_tftp_client_delete(&my_tftp_client);

/* If status is NX_SUCCESS the TFTP client instance was successfully
   deleted.  */
```

nx_tftp_client_error_info_get

Get client error information

Prototype

```
UINT nx_tftp_client_error_info_get(NX_TFTP_CLIENT *tftp_client_ptr,
                        UINT *error_code, CHAR **error_string);
```

Description

This service returns the last error code received and sets the pointer to the client's internal error string. In error conditions, the user can view the last error sent by the server. A null error string indicates no error is present.

Input Parameters

tftp_client_ptr	Pointer to previously created TFTP client instance.
error_code	Pointer to destination area for error code
error_string	Pointer to destination for error string

Return Values

NX_SUCCESS	(0x00)	Successful TFTP error info get.
NX_PTR_ERROR	(0x16)	Invalid TFTP client pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Get error information for client.  */
status =  nx_tftp_client_error_info_get(&my_tftp_client, &error_code,
                        &error_string_ptr);

/* If status is NX_SUCCESS the error code and error string are available.  */
```

nx_tftp_client_file_close

Close client file

Prototype

```
UINT nx_tftp_client_file_close(NX_TFTP_CLIENT *tftp_client_ptr);
```

Description

This service closes the previously opened file by this TFTP client instance. A TFTP client instance is allowed to have only one file open at a time.

Input Parameters

tftp_client_ptr	Pointer to previously created TFTP client instance.

Return Values

NX_SUCCESS	(0x00)	Successful TFTP file close.
status		Actual NetX completion status
NX_PTR_ERROR	(0x16)	Invalid TFTP client pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Close the previously opened file associated with “my_client”.  */
status =  nx_tftp_client_file_close(&my_tftp_client);

/* If status is NX_SUCCESS the TFTP file is closed.  */
```

nx_tftp_client_file_open

Open client file

Prototype

```
UINT nx_tftp_client_file_open(NX_TFTP_CLIENT *tftp_client_ptr,
            CHAR *file_name, ULONG server_ip_address, UINT open_type,
            ULONG wait_option);
```

Description

This service attempts to open the specified file on the TFTP server at the specified IP address. The file will be opened for either reading or writing.

Input Parameters

tftp_client_ptr Pointer to TFTP control block.

file_name ASCII file name, NULL-terminated and with appropriate path information.

server_ip_address IP address of TFTP Server.

open_type Type of open request, either:

NX_TFTP_OPEN_FOR_READ (0x01)
NX_TFTP_OPEN_FOR_WRITE (0x02)

wait_option Defines how long the service will wait for the TFTP client file open. The wait options are defined as follows:

timeout value (0x00000001 through 0xFFFFFFFE)
TX_WAIT_FOREVER (0xFFFFFFFF)

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a TFTP server responds to the request.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the TFTP server response.

Return Values

NX_SUCCESS	(0x00)	Successful TFTP client file open
NX_TFTP_NOT_CLOSED	(0xC3)	Client already has file open
NX_INVALID_TFTP_SERVER_ADDRESS	**(0x08)**	Invalid server address received
NX_TFTP_NO_ACK_RECEIVED	**(0x09)**	No ACK received from server
NX_TFTP_INVALID_SERVER_ADDRESS	**(0x08)**	**Invalid Server IP received**
NX_TFTP_CODE_ERROR	**(0x05)**	**Received error code** from Server
NX_PTR_ERROR	(0x16)	Invalid TFTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_IP_ADDRESS_ERROR	(0x21)	Invalid TFTP Server IP address
NX_OPTION_ERROR	(0x0a)	Invalid open type

Allowed From

Threads

Example

```
/* Open file “test.txt” for reading on the TFTP server at 202.2.2.13.  */
status =  nx_tftp_client_file_open(&my_tftp_client, “test.txt”,
                IP_ADDRESS(202,2,2,13), NX_TFTP_OPEN_FOR_READ, 200);

/* If status is NX_SUCCESS the “test.txt” file is now open for reading.  */
```

nx_tftp_client_file_read

Read a block from client file

Prototype

```
UINT nx_tftp_client_file_read(NX_TFTP_CLIENT *tftp_client_ptr,
                        NX_PACKET **packet_ptr, ULONG wait_option);
```

Description

This service reads a 512-byte block from the previously opened TFTP client file. A block containing fewer than 512 bytes signals the end of the file.

Input Parameters

tftp_client_ptr	Pointer to TFTP client control block.
packet_ptr	Destination for packet containing the block read from the file.
wait_option	Defines how long the service will wait for the read to complete. The wait options are defined as follows: **timeout value** (0x00000001 through 0xFFFFFFFE) **TX_WAIT_FOREVER** (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the TFTP server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the TFTP server to send a block of the file.

Return Values

NX_SUCCESS	(0x00)	Successful block read
NX_TFTP_NOT_OPEN	(0xC3)	Specified Client file is not open for reading
NX_NO_PACKET	(0x01)	No Packet received from Server.

NX_INVALID_TFTP_SERVER_ADDRESS
(0x08) Invalid server address received
NX_TFTP_NO_ACK_RECEIVED
(0x09) No ACK received from Server
NX_TFTP_END_OF_FILE
(0xC5) End of file detected (not an error).
NX_TFTP_CODE_ERROR (0x05) Received error code
NX_TFTP_FAILED (0xC2) Unknown TFTP code received
NX_TFTP_INVALID_BLOCK_NUMBER
(0x0A) Invalid block number received
NX_PTR_ERROR (0x16) Invalid pointer input.
NX_CALLER_ERROR (0x11) Invalid caller of this service

Allowed From

Threads

Example

```
/* Read a block from a previously opened file of “my_client”.  */
status =  nx_tftp_client_file_read(&my_tftp_client, &return_packet_ptr, 200);

/* If status is NX_SUCCESS a block of the TFPT file is in the payload of
   “return_packet_ptr”.  */
```

nx_tftp_client_file_write

Write block to client file

Prototype

```
UINT nx_tftp_client_file_write(NX_TFTP_CLIENT *tftp_client_ptr,
            NX_PACKET *packet_ptr, ULONG wait_option);
```

Description

This service writes a 512-byte block to the previously opened TFTP client file. Specifying a block containing fewer than 512 bytes signals the end of the file.

Input Parameters

tftp_client_ptr	Pointer to TFTP client control block.
packet_ptr	Packet containing the block to write to the file.
wait_option	Defines how long the service will wait for the write to complete. The wait options are defined as follows: **timeout value** (0x00000001 through 0xFFFFFFFE) **TX_WAIT_FOREVER** (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the TFTP server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the TFTP server to send an ACK for the write request.

Return Values

NX_SUCCESS	(0x00)	Successful Client block write
NX_TFTP_NOT_OPEN	(0xC3)	Specified Client file is not open for writing

NX_TFTP_TIMEOUT	(0xC1)	Timeout waiting for Server ACK
NX_INVALID_TFTP_SERVER_ADDRESS	**(0x08)**	Invalid server address received
NX_TFTP_NO_ACK_RECEIVED	**(0x09)**	No ACK received from server
NX_INVALID_TFTP_SERVER_ADDRESS	(0x08)	Invalid server address received
NX_TFTP_CODE_ERROR	(0x05)	Received error code

NX_PTR_ERROR	(0x16)	Invalid pointer input.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Write a block to the previously opened file of “my_client”.  */
status =  nx_tftp_client_file_write(&my_tftp_client, packet_ptr, 200);

/* If status is NX_SUCCESS the block in the payload of “packet_ptr” was
   written to the TFPT file opened by “my_client”.  */
```

nx_tftp_client_packet_allocate

Allocate packet for client file write

Prototype

```
UINT nx_tftp_client_packet_allocate(NX_PACKET_POOL *pool_ptr,
                        NX_PACKET **packet_ptr, ULONG wait_option)
```

Description

This service allocates a UDP packet from the specified packet pool and makes room for the 4-byte TFTP header before the packet is returned to the caller. The caller can then build a buffer for writing to a client file.

Input Parameters

pool_ptr Pointer to packet pool.

packet_ptr Destination for pointer to allocated packet.

wait_option Defines how long the service will wait for the packet allocate to complete. The wait options are defined as follows:

timeout value (0x00000001 through 0xFFFFFFFE)
TX_WAIT_FOREVER (0xFFFFFFFF)

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the allocation completes.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the packet allocation.

Return Values

NX_SUCCESS	(0x00)	Successful packet allocate
NX_PTR_ERROR	(0x16)	Invalid TFTP client pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Allocate a packet for TFTP file write.  */
status =  nx_tftp_packet_allocate(&my_pool, &packet_ptr, 200);

/* If status is NX_SUCCESS “packet_ptr” contains the new packet.  */
```

nx_tftp_client_set_interface

Set physical interface for TFTP requests

Prototype

```
UINT nx_tftp_client_set_interface(NX_TFTP_CLIENT *tftp_client_ptr,
                                  UINT if_index)
```

Description

This service uses the input interface index to set the physical interface for the TFTP Client to send and receive TFTP packets. The default value is zero, for the primary interface.

Input Parameters

tftp_client_ptr	Pointer to TFTP Client instance
if_index	Index of physical interface to use

Return Values

NX_SUCCESS	(0x00)	Successfully set interface
NX_TFTP_INVALID_INTERFACE	(0x0B)	Invalid interface input
NX_PTR_ERROR	(0x16)	Invalid pointer input.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Specify the primary interface for TFTP requests.  */
status =  nx_tftp_client_set_interface(&client, 0);

/* If status is NX_SUCCESS the primary interface will be use for TFTP
communications.  */
```

nx_tftp_server_create

Create TFTP server

Prototype

```
UINT nx_tftp_server_create(NX_TFTP_SERVER *tftp_server_ptr,
            CHAR *tftp_server_name, NX_IP *ip_ptr, FX_MEDIA *media_ptr,
            VOID *stack_ptr, ULONG stack_size,
            NX_PACKET_POOL *pool_ptr);
```

Description

This service creates a TFTP server that responds to TFTP client requests on port 69. The server must be started by a subsequent call to *nx_tftp_server_start*.

Important Note: The application must make certain the supplied IP, packet pool, and FileX media instance are already created. In addition, UDP must be enabled for the IP instance prior to calling this service.

Input Parameters

tftp_server_ptr	Pointer to TFTP server control block.
tftp_server_name	Name of this TFTP server instance
ip_ptr	Pointer to previously created IP instance.
media_ptr	Pointer to FileX media instance.
stack_ptr	Pointer to stack area for TFTP server thread.
stack_size	Number of bytes in the TFTP server stack.
pool_ptr	Pointer to TFTP packet pool. Note that the supplied pool must have packet payloads at least 560 bytes in size.[1]

Return Values

NX_SUCCESS	(0x00)	Successful TFTP server create

[1] The data portion of a packet is exactly 512 bytes, but the packet payload size must be at least 560 bytes. The remaining bytes are used for the UDP, IP, and Ethernet headers.

NX_TFTP_POOL_ERROR	(0xC6)	Packet pool has packet size of less than 560 bytes
NX_PTR_ERROR	(0x16)	Invalid pointer input

Allowed From

Initialization, Threads

Example

```
/* Create a TFTP server called “my_server”.  */
status =  nx_tftp_server_create(&my_server, “My TFTP Server”, &server_ip,
              &ram_disk, stack_ptr, 2048, pool_ptr);

/* If status is NX_SUCCESS the TFTP server is created.  */
```

nx_tftp_server_delete

Delete TFTP server

Prototype

```
UINT nx_tftp_server_delete(NX_TFTP_SERVER *tftp_server_ptr);
```

Description

This service deletes a previously created TFTP server.

Input Parameters

tftp_server_ptr	Pointer to TFTP server control block.

Return Values

NX_SUCCESS	(0x00)	Successful Server delete
NX_PTR_ERROR	(0x16)	Invalid Server pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller

Allowed From

Threads

Example

```
/* Delete the TFTP server called “my_server”.  */
status =  nx_tftp_server_delete(&my_server);

/* If status is NX_SUCCESS the TFTP server is deleted.  */
```

nx_tftp_server_start

Start TFTP server

Prototype

```
UINT nx_tftp_server_start(NX_TFTP_SERVER *tftp_server_ptr);
```

Description

This service starts the previously created TFTP server.

Input Parameters

tftp_server_ptr	Pointer to TFTP server control block.

Return Values

NX_SUCCESS	(0x00)	Successful TFTP server start
NX_PTR_ERROR	(0x16)	Invalid TFTP server pointer.

Allowed From

Initialization, threads

Example

```
/* Start the TFTP server called “my_server”.  */
status =  nx_tftp_server_start(&my_server);

/* If status is NX_SUCCESS the TFTP server is started.  */
```

nx_tftp_server_stop

Stop TFTP server

Prototype

```
UINT nx_tftp_server_stop(NX_TFTP_SERVER *tftp_server_ptr);
```

Description

This service stops the previously created TFTP server.

Input Parameters

tftp_server_ptr	Pointer to TFTP server control block.

Return Values

NX_SUCCESS	(0x00)	Successful TFTP server stop
NX_PTR_ERROR	(0x16)	Invalid TFTP server pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Stop the TFTP server called “my_server”.  */
status =  nx_tftp_server_stop(&my_server);

/* If status is NX_SUCCESS the TFTP server is stopped.  */
```