File Transfer Protocol (FTP)

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 FTP

The File Transfer Protocol (FTP) is a protocol designed for file transfers. FTP utilizes reliable Transmission Control Protocol (TCP) services to perform its file transfer function. Because of this, FTP is a highly reliable file transfer protocol. FTP is also high-performance. The actual FTP file transfer is performed on a dedicated FTP connection.

FTP Requirements

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

The FTP Server portion of the NetX FTP package has several additional requirements. First, it requires complete access to TCP *well-known port 21* for handling all Client FTP command requests and *well-known port 20* for handling all Client FTP data transfers. The FTP 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.

FTP Constraints

The FTP standard has many options regarding the representation of file data. Similar to Unix implementations, NetX FTP assumes the following file format constraints:

File Type:	**Binary**
File Format:	**Nonprint Only**
File Structure:	**File Structure Only**
Transmission Mode:	**Stream Mode Only**

FTP File Names

FTP 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 for the size of FTP file names in the NetX FTP implementation. However, the packet pool payload size should be able to accommodate the maximum path and/or file name.

FTP Client Commands

The FTP has a simple mechanism for opening connections and performing file and directory operations. There is basically a set of standard FTP commands that are issued by the Client after a connection has been successfully established on the TCP *well-known port 21*. The following shows some of the basic FTP commands:

FTP Command	**Meaning**
CWD path	*Change working directory*
DELE filename	*Delete specified file name*
LIST directory	*Get directory listing*
MKD directory	*Make new directory*
NLST directory	*Get directory listing*
NOOP	*No operation, returns success*
PASS password	*Provide password for login*
PASV	*Request passive transfer mode*
PWD path	*Pickup current directory path*
QUIT	*Terminate Client connection*
RETR filename	*Read specified file*
RMD directory	*Delete specified directory*
RNFR oldfilename	*Specify file to rename*
RNTO newfilename	*Rename file to supplied file name*
STOR filename	*Write specified file*
TYPE I	*Select binary file image*
USER username	*Provide username for login*
PORT ip_address,port	*Provide IP address and Client data port*

These ASCII commands are used internally by the NetX FTP Client software to perform FTP operations with the FTP Server.

FTP Server Responses

The FTP Server utilizes the *well-known TCP port 21* to field Client command requests. Once the FTP Server processes the Client command, it returns a 3-digit numeric response in ASCII followed by an optional ASCII string. The numeric response is used by the FTP Client software to determine whether the operation succeeded or failed. The following lists various FTP Server responses to Client commands:

First Numeric Field	Meaning
1xx	*Positive preliminary status – another reply coming*.
2xx	*Positive completion status*.
3xx	*Positive preliminary status – another command must be sent*.
4xx	*Temporary error condition*.
5xx	*Error condition.*

Second Numeric Field	Meaning
x0x	Syntax error in command.
x1x	Informational message.
x2x	Connection related.
x3x	Authentication related.
x4x	Unspecified.
x5x	File system related.

For example, a Client request to disconnect an FTP connection with the QUIT command will typically be responded with a “221” code from the Server – if the disconnect is successful.

FTP Passive Transfer Mode

By default, the NetX FTP Client uses the active transport mode to exchange data over the data socket with the FTP server. The problem with this arrangement is that it requires the FTP Client to open a TCP server socket for the FTP Server to connect to. This represents a possible security risk and may be blocked by the Client firewall. Passive transfer mode differs from active transport mode by having the FTP server create the TCP server socket on the data connection. This eliminates the security risk (for the FTP Client).

To enable passive data transfer, the application calls *nx_ftp_client_passive_mode_set* on a previously created FTP Client with the second argument set to NX_TRUE. Thereafter, all subsequent NetX FTP Client services for transferring data (NLST, RETR, STOR) are attempted in the passive transport mode.

The FTP Client first sends the PASV command (no arguments). If the FTP server supports this request it will return the 227 “OK” response. Then the Client sends the request e.g. RETR. If the server refuses passive transfer mode, the NetX FTP Client service returns an error status.

To disable passive transport mode and return to active transport mode, the application calls *nx_ftp_client_passive_mode_set* with the second argument set to NX_FALSE.

PASV only supports IPv4 connections. For IPv6, passive mode transfer uses the EPSV command which is not supported in the current NetX FTP Client release.

Refer to the demo program, `demo_netxduo_ftp_client_passive.c` for how to use the passive mode feature.

FTP Communication

The FTP Server utilizes the *well-known TCP port 21* to field Client requests. FTP Clients may use any available TCP port. The general sequence of FTP events is as follows:

FTP Read File Requests:

1. Client issues TCP connect to Server port 21.
2. Server sends “220” response to signal success.
3. Client sends “USER” message with “username.”
4. Server sends “331” response to signal success.
5. Client sends “PASS” message with “password.”
6. Server sends “230” response to signal success.
7. Client sends “TYPE I” message for binary transfer.
8. Server sends “200” response to signal success.
9. Client sends “PORT” message with IP address and port.
10. Server sends “200” response to signal success.
11. Client sends “RETR” message with file name to read.

12. Server creates data socket and connects with client data port specified in the “PORT” command.
13. Server sends “125” response to signal file read has started.
14. Server sends contents of file through the data connection. This process continues until file is completely transferred.
15. When finished, Server disconnects data connection.
16. Server sends “250” response to signal file read is successful.
17. Clients sends “QUIT” to terminate FTP connection.
18. Server sends “221” response to signal disconnect is successful.
19. Server disconnects FTP connection.

As mentioned previously, the only difference between FTP running over IPv4 and IPv6 is the PORT command is replaced with the EPRT command for IPv6

If the FTP Client makes a read request in the passive transfer mode, the command sequence is as follows (**bolded** lines indicates a different step from active transfer mode):

1. Client issues TCP connect to Server port 21.
2. Server sends “220” response to signal success.
3. Client sends “USER” message with “username.”
4. Server sends “331” response to signal success.
5. Client sends “PASS” message with “password.”
6. Server sends “230” response to signal success.
7. Client sends “TYPE I” message for binary transfer.
8. Server sends “200” response to signal success.
9. **Client sends “PASV” message.**
10. **Server sends “227” response, and IP address and port for the Client to connect to, to signal success.**
11. Client sends “RETR” message with file name to read.
12. **Server creates data server socket and listens for the Client connect request on this socket using the port specified in the “227” response.**
13. **Server sends “150” response on the control socket to signal file read has started.**
14. Server sends contents of file through the data connection. This process continues until file is completely transferred.
15. When finished, Server disconnects data connection.
16. **Server sends “226” response on the control socket to signal file read is successful.**
17. Client sends “QUIT” to terminate FTP connection.
18. Server sends “221” response to signal disconnect is successful.
19. Server disconnects FTP connection.

FTP Write Requests:

1. Client issues TCP connect to Server port 21.
2. Server sends “220” response to signal success.
3. Client sends “USER” message with “username.”
4. Server sends “331” response to signal success.
5. Client sends “PASS” message with “password.”
6. Server sends “230” response to signal success.
7. Client sends “TYPE I” message for binary transfer.
8. Server sends “200” response to signal success.
9. Client sends “PORT” message with IP address and port.
10. Server sends “200” response to signal success.
11. Client sends “STOR” message with file name to write.
12. Server creates data socket and connects with client data port specified in the “PORT” command.
13. Server sends “125” response to signal file write has started.
14. Client sends contents of file through the data connection. This process continues until file is completely transferred.
15. When finished, Client disconnects data connection.
16. Server sends “250” response to signal file write is successful.
17. Clients sends “QUIT” to terminate FTP connection.
18. Server sends “221” response to signal disconnect is successful.
19. Server disconnects FTP connection.

If the FTP Client makes a write request in the passive transfer mode, the command sequence is as follows (**bolded** lines indicates a different step from active transfer mode):

1. Client issues TCP connect to Server port 21.
2. Server sends “220” response to signal success.
3. Client sends “USER” message with “username.”
4. Server sends “331” response to signal success.
5. Client sends “PASS” message with “password.”
6. Server sends “230” response to signal success.
7. Client sends “TYPE I” message for binary transfer.
8. Server sends “200” response to signal success.
9. **Client sends “PASV” message.**
10. **Server sends “227” response, and IP address and port for the Client to connect to, to signal success.**
11. Client sends “STOR” message with file name to write.
12. **Server creates data server socket and listens for the**

Client connect request on this socket using the port specified in the “227” response.

13. **Server sends “150” response on the control socket to signal file write has started.**
14. Client sends contents of file through the data connection. This process continues until file is completely transferred.
15. When finished, Client disconnects data connection.
16. **Server sends “226” response on the control socket to signal file write is successful.**
17. Client sends “QUIT” to terminate FTP connection.
18. Server sends “221” response to signal disconnect is successful.
19. Server disconnects FTP connection.

FTP Authentication

Whenever an FTP connection takes place, the Client must provide the Server with a *username* and *password*. Some FTP sites allow what is called *Anonymous FTP*, which allows FTP access without a specific username and password. For this type of connection, “anonymous” should be supplied for username and the password should be a complete e-mail address.

The user is responsible for supplying NetX FTP with login and logout authentication routines. These are supplied during the ***nx_ftp_server_create*** function and called from the password processing. If the *login* function returns NX_SUCCESS, the connection is authenticated and FTP operations are allowed. Otherwise, if the *login* function returns something other than NX_SUCCESS, the connection attempt is rejected.

FTP Multi-Thread Support

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

FTP RFCs

NetX FTP is compliant with RFC959 and related RFCs.

Chapter 2

Installation and Use of FTP

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

Product Distribution

FTP 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_ftp.h	Header file for FTP for NetX
nx_ftp_client.c	C Source file for FTP Client for NetX
nx_ftp_server.c	C Source file for FTP Server for NetX
filex_stub.h	Stub file if FileX is not present
nx_ftp.pdf	PDF description of FTP for NetX
demo_netx_ftp.c	FTP demonstration system
demo_netxduo_ftp_client_passive.c	FTP demonstration of file download (read) and upload (write) in passive transfer mode

FTP Installation

In order to use FTP 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_ftp.h*, *nx_ftp_client.c*, and *nx_ftp_server.c* files should be copied into this directory.

Using FTP

Using FTP for NetX is easy. Basically, the application code must include *nx_ftp.h* after it includes *tx_api.h, fx_api.h,* and *nx_api.h*, in order to use ThreadX, FileX, and NetX, respectively. Once *nx_ftp.h* is included, the application code is then able to make the FTP function calls specified later in this guide. The application must also include *nx_ftp_client.c* and *nx_ftp_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 FTP.

Note that since FTP utilizes NetX TCP services, TCP must be enabled with the *nx_tcp_enable* call prior to using FTP.

Small Example System

An example of how easy it is to use NetX FTP is described in Figure 1.1 that appears below.

Note this is for a host device with a single network interface.

In this example, the FTP include file *nx_ftp_client.h* and *nx_ftp_server.h* are brought at line 10 and 11. Next, the FTP Server is created in "*tx_application_define*" at line 134. Note that the FTP Server control block "*Server*" was defined as a global variable at line 31 previously. After successful creation, an FTP Server is started at line 363. At line 183 the FTP Client is created. And finally, the Client writes the file at line 229 and reads the file back at line 318.

```
1 /* This is a small demo of NetX FTP on the high-performance NetX TCP/IP stack.
This demo
  2    relies on ThreadX, NetX, and FileX to show a simple file transfer from the
client
  3    and then back to the server.  */
  4
  5
  6
  7 #include    "tx_api.h"
  8 #include    "fx_api.h"
  9 #include    "nx_api.h"
 10 #include    "nx_ftp_client.h"
 11 #include    "nx_ftp_server.h"
 12
 13 #define     DEMO_STACK_SIZE         4096
 14
 15
 16
 17 /* Define the ThreadX, NetX, and FileX object control blocks...  */
 18
 19 TX_THREAD               server_thread;
 20 TX_THREAD               client_thread;
 21 NX_PACKET_POOL          server_pool;
 22 NX_IP                   server_ip;
 23 NX_PACKET_POOL          client_pool;
 24 NX_IP                   client_ip;
 25 FX_MEDIA                ram_disk;
 26
 27
 28 /* Define the NetX FTP object control blocks.  */
 29
 30 NX_FTP_CLIENT           ftp_client;
 31 NX_FTP_SERVER           ftp_server;
 32
 33
 34 /* Define the counters used in the demo application...  */
 35
 36 ULONG                   error_counter = 0;
 37
 38
 39 /* Define the memory area for the FileX RAM disk.  */
 40
 41 UCHAR                   ram_disk_memory[32000];
 42 UCHAR                   ram_disk_sector_cache[512];
 43
 44
```

```
 45 #define FTP_SERVER_ADDRESS  IP_ADDRESS(1,2,3,4)
 46 #define FTP_CLIENT_ADDRESS  IP_ADDRESS(1,2,3,5)
 47
 48 extern UINT  _fx_media_format(FX_MEDIA *media_ptr, VOID (*driver)(FX_MEDIA
*media), VOID *driver_info_ptr, UCHAR *memory_ptr, UINT memory_size,
 49                         CHAR *volume_name, UINT number_of_fats, UINT
directory_entries, UINT hidden_sectors,
 50                         ULONG total_sectors, UINT bytes_per_sector, UINT
sectors_per_cluster,
 51                         UINT heads, UINT sectors_per_track);
 52
 53 /* Define the FileX and NetX driver entry functions.  */
 54 VOID    _fx_ram_driver(FX_MEDIA *media_ptr);
 55
 56 /* Replace the 'ram' driver with your own Ethernet driver. */
 57 VOID    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
 58
 59
 60 void    client_thread_entry(ULONG thread_input);
 61 void    thread_server_entry(ULONG thread_input);
 62
 63
 64
 65
 66 /* Define server login/logout functions.  These are stubs for functions that
would
 67    validate a client login request.   */
 68
 69 UINT    server_login(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, ULONG
client_ip_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info);
 70 UINT    server_logout(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, ULONG
client_ip_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info);
71
 72
 73 /* Define main entry point.  */
 74
 75 int main()
 76 {
 77
 78     /* Enter the ThreadX kernel.  */
 79     tx_kernel_enter();
 80     return(0);
 81 }
 82
 83
 84 /* Define what the initial system looks like.  */
 85
 86 void    tx_application_define(void *first_unused_memory)
 87 {
 88
 89 UINT    status;
 90 UCHAR   *pointer;
 91
 92
 93     /* Setup the working pointer.  */
 94     pointer =  (UCHAR *) first_unused_memory;
 95
 96     /* Create a helper thread for the server. */
 97     tx_thread_create(&server_thread, "FTP Server thread", thread_server_entry,
0,
 98                      pointer, DEMO_STACK_SIZE,
 99                      4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
100
101     pointer =  pointer + DEMO_STACK_SIZE;
102
103     /* Initialize NetX.  */
104     nx_system_initialize();
105
106     /* Create the packet pool for the FTP Server.  */
107     status = nx_packet_pool_create(&server_pool, "NetX Server Packet Pool",
256, pointer, 8192);
108     pointer = pointer + 8192;
109
110     /* Check for errors.  */
111     if (status)
112         error_counter++;
113
114     /* Create the IP instance for the FTP Server.  */
115     status = nx_ip_create(&server_ip, "NetX Server IP Instance",
FTP_SERVER_ADDRESS, 0xFFFFFF00UL,
```

```
116                                         &server_pool, _nx_ram_network_driver,
pointer, 2048, 1);
117     pointer = pointer + 2048;
118
119     /* Check status.  */
120     if (status != NX_SUCCESS)
121     {
122         error_counter++;
123         return;
124     }
125
126     /* Enable ARP and supply ARP cache memory for server IP instance.  */
127     nx_arp_enable(&server_ip, (void *) pointer, 1024);
128     pointer = pointer + 1024;
129
130     /* Enable TCP.  */
131     nx_tcp_enable(&server_ip);
132
133     /* Create the FTP server.  */
134     status =  nx_ftp_server_create(&ftp_server, "FTP Server Instance",
&server_ip, &ram_disk, pointer, DEMO_STACK_SIZE, &server_pool,
135                                    server_login, server_logout);
136     pointer =  pointer + DEMO_STACK_SIZE;
137
138     /* Check status.  */
139     if (status != NX_SUCCESS)
140     {
141         error_counter++;
142         return;
143     }
144
145     /* Now set up the FTP Client. */
146
147     /* Create the main FTP client thread.  */
148     status = tx_thread_create(&client_thread, "FTP Client thread ",
client_thread_entry, 0,
149             pointer, DEMO_STACK_SIZE,
150             6, 6, TX_NO_TIME_SLICE, TX_AUTO_START);
151     pointer = pointer + DEMO_STACK_SIZE ;
152
153     /* Check status.  */
154     if (status != NX_SUCCESS)
155     {
156         error_counter++;
157         return;
158     }
159
160     /* Create a packet pool for the FTP client.  */
161     status =  nx_packet_pool_create(&client_pool, "NetX Client Packet Pool",
256, pointer, 8192);
162     pointer =  pointer + 8192;
163
164     /* Create an IP instance for the FTP client.  */
165     status = nx_ip_create(&client_ip, "NetX Client IP Instance",
FTP_CLIENT_ADDRESS, 0xFFFFFF00UL,
166                                                 &client_pool,
_nx_ram_network_driver, pointer, 2048, 1);
167     pointer = pointer + 2048;
168
169     /* Enable ARP and supply ARP cache memory for the FTP Client IP.  */
170     nx_arp_enable(&client_ip, (void *) pointer, 1024);
171
172     pointer = pointer + 1024;
173
174     /* Enable TCP for client IP instance.  */
175     nx_tcp_enable(&client_ip);
176
177     return;
178
179 }
180
181 /* Define the FTP client thread.  */
182
183 void    client_thread_entry(ULONG thread_input)
184 {
185
186 NX_PACKET   *my_packet;
187 UINT        status;
188
189
190
```

```
191     /* Format the RAM disk - the memory for the RAM disk was defined above.
*/
192     status = _fx_media_format(&ram_disk,
193                               _fx_ram_driver,                 /* Driver entry
*/
194                               ram_disk_memory,                /* RAM disk
memory pointer     */
195                               ram_disk_sector_cache,          /* Media buffer
pointer        */
196                               sizeof(ram_disk_sector_cache),  /* Media buffer
size           */
197                               "MY_RAM_DISK",                  /* Volume Name
*/
198                               1,                              /* Number of FATs
*/
199                               32,                             /* Directory
Entries           */
200                               0,                              /* Hidden sectors
*/
201                               256,                            /* Total sectors
*/
202                               128,                            /* Sector size
*/
203                               1,                              /* Sectors per
cluster         */
204                               1,                              /* Heads
*/
205                               1);                             /* Sectors per
track           */
206
207     /* Check status.  */
208     if (status != NX_SUCCESS)
209     {
210         error_counter++;
211         return;
212     }
213
214     /* Open the RAM disk.  */
215     status = fx_media_open(&ram_disk, "RAM DISK", _fx_ram_driver,
ram_disk_memory, ram_disk_sector_cache, sizeof(ram_disk_sector_cache));
216
217     /* Check status.  */
218     if (status != NX_SUCCESS)
219     {
220         error_counter++;
221         return;
222     }
223
224     /* Let the IP threads and driver initialize the system.    */
225     tx_thread_sleep(100);
226
227
228     /* Create an FTP client.  */
229     status =  nx_ftp_client_create(&ftp_client, "FTP Client", &client_ip, 2000,
&client_pool);
230
231     /* Check status.  */
232     if (status != NX_SUCCESS)
233     {
234
235         error_counter++;
236         return;
237      }
238
239     printf("Created the FTP Client\n");
240
241
242     /* Now connect with the NetX FTP (IPv4) server. */
243     status =  nx_ftp_client_connect(&ftp_client, FTP_SERVER_ADDRESS, "name",
"password", 100);
244
245     /* Check status.  */
246     if (status != NX_SUCCESS)
247     {
248
249         error_counter++;
250         return;
251      }
252
253     printf("Connected to the FTP Server\n");
254
```

```
255     /* Open a FTP file for writing.  */
256     status =  nx_ftp_client_file_open(&ftp_client, "test.txt",
NX_FTP_OPEN_FOR_WRITE, 100);
257
258     /* Check status.  */
259     if (status != NX_SUCCESS)
260     {
261
262         error_counter++;
263         return;
264      }
265
266     printf("Opened the FTP client test.txt file\n");
267
268     /* Allocate a FTP packet.  */
269     status =  nx_packet_allocate(&client_pool, &my_packet, NX_TCP_PACKET, 100);
270
271     /* Check status.  */
272     if (status != NX_SUCCESS)
273     {
274
275         error_counter++;
276         return;
277      }
278
279     /* Write ABCs into the packet payload!  */
280     memcpy(my_packet -> nx_packet_prepend_ptr, "ABCDEFGHIJKLMNOPQRSTUVWXYZ  ",
28);
281
282     /* Adjust the write pointer.  */
283     my_packet -> nx_packet_length =  28;
284     my_packet -> nx_packet_append_ptr =  my_packet -> nx_packet_prepend_ptr +
28;
285
286     /* Write the packet to the file test.txt.  */
287     status =  nx_ftp_client_file_write(&ftp_client, my_packet, 100);
288
289     /* Check status.  */
290     if (status != NX_SUCCESS)
291     {
292         error_counter++;
293     }
294     else
295         printf("Wrote to the FTP client test.txt file\n");
296
297
298     /* Close the file.  */
299     status =  nx_ftp_client_file_close(&ftp_client, 100);
300
301     /* Check status.  */
302     if (status != NX_SUCCESS)
303         error_counter++;
304     else
305         printf("Closed the FTP client test.txt file\n");
306
307
308     /* Now open the same file for reading.  */
309     status =  nx_ftp_client_file_open(&ftp_client, "test.txt",
NX_FTP_OPEN_FOR_READ, 100);
310
311     /* Check status.  */
312     if (status != NX_SUCCESS)
313         error_counter++;
314     else
315         printf("Reopened the FTP client test.txt file\n");
316
317     /* Read the file.  */
318     status =  nx_ftp_client_file_read(&ftp_client, &my_packet, 100);
319
320     /* Check status.  */
321     if (status != NX_SUCCESS)
322         error_counter++;
323     else
324     {
325             printf("Reread the FTP client test.txt file\n");
326             nx_packet_release(my_packet);
327     }
328
329     /* Close this file.  */
330     status =  nx_ftp_client_file_close(&ftp_client, 100);
331
```

```
332     if (status != NX_SUCCESS)
333         error_counter++;
334
335     /* Disconnect from the server.  */
336     status =  nx_ftp_client_disconnect(&ftp_client, 100);
337
338     /* Check status.  */
339     if (status != NX_SUCCESS)
340         error_counter++;
341
342
343     /* Delete the FTP client.  */
344     status =  nx_ftp_client_delete(&ftp_client);
345
346     /* Check status.  */
347     if (status != NX_SUCCESS)
348         error_counter++;
349 }
350
351
352 /* Define the helper FTP server thread.  */
353 void    thread_server_entry(ULONG thread_input)
354 {
355
356 UINT            status;
357
358     /* Wait till the IP thread and driver have initialized the system. */
359     tx_thread_sleep(100);
360
361
362     /* OK to start the FTP Server.   */
363     status = nx_ftp_server_start(&ftp_server);
364
365     if (status != NX_SUCCESS)
366         error_counter++;
367
368     printf("Server started!\n");
369
370     /* FTP server ready to take requests! */
371
372     /* Let the IP threads execute.    */
373     tx_thread_relinquish();
374
375     return;
376 }
377
378
379 UINT  server_login(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, ULONG
client_ip_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info)
380 {
381
382     printf("Logged in!\n");
383     /* Always return success.  */
384     return(NX_SUCCESS);
385 }
386
387 UINT  server_logout(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, ULONG
client_ip_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info)
388 {
389     printf("Logged out!\n");
390
391     /* Always return success.  */
392     return(NX_SUCCESS);
393 }
```

Figure 1.1 Example of FTP Client and Server with NetX
(Single network interface host)

Configuration Options

There are several configuration options for building FTP for NetX. The following list describes each in detail:

Define	Meaning
NX_FTP_SERVER_PRIORITY	The priority of the FTP Server thread. By default, this value is defined as 16 to specify priority 16.
NX_FTP_MAX_CLIENTS	The maximum number of Clients the Server can handle at one time. By default, this value is 4 to support 4 Clients at once.
NX_FTP_SERVER_MIN_PACKET_PAYLOAD	The minimum size of the Server packet pool payload in bytes, including TCP, IP and network frame headers plus HTTP data. The default value is 256 (maximum length of filename in FileX) + 12 bytes for file information, and NX_PHYSICAL_TRAILER.
NX_FTP_NO_FILEX	Defined, this option provides a stub for FileX dependencies. The FTP Client will function without any change if this option is defined. The FTP 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_FTP_CONTROL_TOS	Type of service required for the FTP TCP control requests. By default, this value is defined as NX_IP_NORMAL to indicate normal IP packet service. This define can be set by the

	application prior to inclusion of *nx_ftp.h*.
NX_FTP_DATA_TOS	Type of service required for the FTP TCP data requests. By default, this value is defined as NX_IP_NORMAL to indicate normal IP packet service. This define can be set by the application prior to inclusion of *nx_ftp.h*.
NX_FTP_FRAGMENT_OPTION	Fragment enable for FTP TCP requests. By default, this value is NX_DONT_FRAGMENT to disable FTP TCP fragmenting. This define can be set by the application prior to inclusion of *nx_ftp.h*.
NX_FTP_CONTROL_WINDOW_SIZE	Control socket window size. By default, this value is 400 bytes. This define can be set by the application prior to inclusion of *nx_ftp.h*.
NX_FTP_DATA_WINDOW_SIZE	Data socket window size. By default, this value is 2048 bytes. This define can be set by the application prior to inclusion of *nx_ftp.h*.
NX_FTP_TIME_TO_LIVE	Specifies the number of routers this packet can pass before it is discarded. The default value is set to 0x80, but can be redefined prior to inclusion of *nx_ftp.h.*
NX_FTP_SERVER_TIMEOUT	Specifies the number of ThreadX ticks that internal services will suspend for. The default value is set to 100, but can be redefined prior to inclusion of *nx_ftp.h.*

NX_FTP_USERNAME_SIZE	Specifies the number of bytes allowed in a client supplied *username*. The default value is set to 20, but can be redefined prior to inclusion of *nx_ftp.h.*
NX_FTP_PASSWORD_SIZE	Specifies the number of bytes allowed in a client supplied *password*. The default value is set to 20, but can be redefined prior to inclusion of *nx_ftp.h.*
NX_FTP_ACTIVITY_TIMEOUT	Specifies the number of seconds a client connection is maintained if there is no activity. The default value is set to 240, but can be redefined prior to inclusion of *nx_ftp.h.*
NX_FTP_TIMEOUT_PERIOD	Specifies the number of seconds between the Server checking for client inactivity. The default value is set to 60, but can be redefined prior to inclusion of *nx_ftp.h.*

Chapter 3

Description of FTP Services

This chapter contains a description of all NetX FTP 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_ftp_client_connect
Connect to FTP Server

nx_ftp_client_create
Create an FTP Client instance

nx_ftp_client_delete
Delete an FTP Client instance

nx_ftp_client_directory_create
Create a directory on Server

nx_ftp_client_directory_default_set
Set default directory on Server

nx_ftp_client_directory_delete
Delete a directory on Server

nx_ftp_client_directory_listing_get
Get directory listing from Server

nx_ftp_client_directory_listing_continue
Continue directory listing from Server

nx_ftp_client_disconnect
Disconnect from FTP Server

nx_ftp_client_file_close
Close Client file

nx_ftp_client_file_delete

Delete file on Server

nx_ftp_client_file_open
Open Client file

nx_ftp_client_file_read
Read from file

nx_ftp_client_file_rename
Rename file on Server

nx_ftp_client_file_write
Write to file

nx_ftp_server_create
Create FTP Server

nx_ftp_server_delete
Delete FTP Server

nx_ftp_server_start
Start FTP Server

nx_ftp_server_stop
Stop FTP Server

nx_ftp_client_connect

Connect to an FTP Server

Prototype

```
UINT nx_ftp_client_connect(NX_FTP_CLIENT *ftp_client_ptr,
            ULONG server_ip, CHAR *username, CHAR *password,
            ULONG wait_option);
```

Description

This service connects the previously created FTP Client instance to the FTP Server at the supplied IP address.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

server_ip IP address of FTP Server.

username Client username for authentication.

password Client password for authentication.

wait_option Defines how long the service will wait for the FTP Client connection. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS (0x00) Successful FTP connection.
NX_TFTP_EXPECTED_22X_CODE

	(0xDB)	Did not get a 22X (ok) response
NX_FTP_EXPECTED_23X_CODE		
	(0xDC)	Did not get a 23X (ok) response
NX_FTP_EXPECTED_33X_CODE		
	(0xDE)	Did not get a 33X (ok) response
NX_FTP_NOT_DISCONNECTED		
	(0xD4)	Client is already connected**.**
NX_PTR_ERROR	(0x07)	Invalid pointer inout.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.

Allowed From

Threads

Example

```
/* Connect the FTP Client instance “my_client” to the FTP Server at
   IP address 1.2.3.4.  */
status =  nx_ftp_client_connect(&my_client, IP_ADDRESS(1,2,3,4), NULL, NULL, 100);

/* If status is NX_SUCCESS an FTP Client instance was successfully
   connected to the FTP Server.  */
```

nx_ftp_client_create

Create an FTP Client instance

Prototype

```
UINT  nx_ftp_client_create(NX_FTP_CLIENT *ftp_client_ptr,
            CHAR *ftp_client_name, NX_IP *ip_ptr, ULONG window_size,
            NX_PACKET_POOL *pool_ptr);
```

Description

This service creates an FTP Client instance.

Input Parameters

ftp_client_ptr	Pointer to FTP Client control block.
ftp_client_name	Name of FTP Client.
ip_ptr	Pointer to previously created IP instance.
window_size	Advertised window size for TCP sockets of this FTP Client.
pool_ptr	Pointer to the default packet pool for this FTP Client. Note that the minimum packet payload must be large enough to hold complete path and the file or directory name.

Return Values

NX_SUCCESS	(0x00)	Successful FTP Client create.
NX_PTR_ERROR	(0x16)	Invalid FTP, IP pointer, or packet pool pointer. password pointer.

Allowed From

Initialization and Threads

Example

```
/* Create the FTP Client instance “my_client.” */
status =  nx_ftp_client_create(&my_client, "My Client", &client_ip,
                                                    2000, &client_pool);


/* If status is NX_SUCCESS the FTP Client instance was successfully
   created.  */
```

nx_ftp_client_delete

Delete an FTP Client instance

Prototype

```
UINT nx_ftp_client_delete(NX_FTP_CLIENT *ftp_client_ptr);
```

Description

This service deletes an FTP Client instance.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Successful FTP Client delete.
NX_FTP_NOT_DISCONNECTED	(0xD4)	FTP Client delete error.
NX_PTR_ERROR	(0x16)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete the FTP Client instance "my_client." */
status =  nx_ftp_client_delete(&my_client);

/* If status is NX_SUCCESS the FTP Client instance was successfully
   deleted.  */
```

nx_ftp_client_directory_create

Create a directory on FTP Server

Prototype

```
UINT nx_ftp_client_directory_create(NX_FTP_CLIENT *ftp_client_ptr,
            CHAR *directory_name, ULONG wait_option);
```

Description

This service creates the specified directory on the FTP Server that is connected to the specified FTP Client.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

directory_name Name of directory to create.

wait_option Defines how long the service will wait for the FTP directory create. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP directory create.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Create the directory “my_dir” on the FTP Server connected to
   the FTP Client instance “my_client.” */
status =  nx_ftp_client_directory_create(&my_client, “my_dir”, 200);

/* If status is NX_SUCCESS the directory “my_dir” was successfully
   created.  */
```

nx_ftp_client_directory_default_set

Set default directory on FTP Server

Prototype

```
UINT nx_ftp_client_directory_default_set(NX_FTP_CLIENT *ftp_client_ptr,
                                         CHAR *directory_path, ULONG wait_option);
```

Description

This service sets the default directory on the FTP Server that is connected to the specified FTP Client. This default directory applies only to this client's connection.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

directory_path Name of directory path to set.

wait_option Defines how long the service will wait for the FTP default directory set. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP default set.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.

NX_CALLER_ERROR (0x11) Invalid caller of this service.

Allowed From

Threads

Example

```
/* Set the default directory to “my_dir” on the FTP Server connected to
   the FTP Client instance “my_client.” */
status =  nx_ftp_client_directory_default_set(&my_client, “my_dir”, 200);

/* If status is NX_SUCCESS the directory “my_dir” is the default directory.  */
```

nx_ftp_client_directory_delete

Delete directory on FTP Server

Prototype

```
UINT nx_ftp_client_directory_delete(NX_FTP_CLIENT *ftp_client_ptr,
            CHAR *directory_name, ULONG wait_option);
```

Description

This service deletes the specified directory on the FTP Server that is connected to the specified FTP Client.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

directory_name Name of directory to delete.

wait_option Defines how long the service will wait for the FTP directory delete. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP directory delete.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete directory “my_dir” on the FTP Server connected to
   the FTP Client instance “my_client.” */
status =  nx_ftp_client_directory_delete(&my_client, “my_dir”, 200);

/* If status is NX_SUCCESS the directory “my_dir” is deleted.  */
```

nx_ftp_client_directory_listing_get

Get directory listing from FTP Server

Prototype

```
UINT nx_ftp_client_directory_listing_get(NX_FTP_CLIENT *ftp_client_ptr,
            CHAR *directory_name, NX_PACKET **packet_ptr,
            ULONG wait_option);
```

Description

This service gets the contents of the specified directory on the FTP Server that is connected to the specified FTP Client. The supplied packet pointer will contain one or more directory entries. Each entry is separated by a <cr/lf> combination. The ***nx_ftp_client_directory_listing_continue*** should be called to complete the directory get operation.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

directory_name Name of directory to get contents of.

packet_ptr Pointer to destination packet pointer. If successful, the packet payload will contain one or more directory entries.

wait_option Defines how long the service will wait for the FTP directory listing. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP directory listing.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_NOT_ENABLED	(0x14)	Service (IPv6) not enabled
NX_FTP_EXPECTED_1XX_CODE	(0xD9)	Did not get a 1XX (ok) response
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Get the contents of directory “my_dir” on the FTP Server connected to
   the FTP Client instance “my_client.” */
status =  nx_ftp_client_directory_listing_get(&my_client, “my_dir”, &my_packet,
                                              200);


/* If status is NX_SUCCESS, one or more entries of the directory “my_dir”
   can be found in “my_packet”.  */
```

nx_ftp_client_directory_listing_continue

Continue directory listing from FTP Server

Prototype

```
UINT nx_ftp_client_directory_listing_continue(NX_FTP_CLIENT
            *ftp_client_ptr, NX_PACKET **packet_ptr,
            ULONG wait_option);
```

Description

This service continues getting the contents of the specified directory on the FTP Server that is connected to the specified FTP Client. It should have been immediately preceded by a call to ***nx_ftp_client_directory_listing_get***. If successful, the supplied packet pointer will contain one or more directory entries. This routine should be called until an NX_FTP_END_OF_LISTING status is received.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

packet_ptr Pointer to destination packet pointer. If successful, the packet payload will contain one or more directory entries, separated by a <cr/lf>.

wait_option Defines how long the service will wait for the FTP directory listing. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP directory listing.
NX_FTP_END_OF_LISTING	(0xD8)	No more entries in this directory.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Continue getting the contents of directory “my_dir” on the FTP Server
   connected to the FTP Client instance “my_client.” */
status =  nx_ftp_client_directory_listing_continue(&my_client, &my_packet,
                                                   200);


/* If status is NX_SUCCESS, one or more entries of the directory “my_dir”
   can be found in “my_packet”.  */
```

nx_ftp_client_disconnect

Disconnect from FTP Server

Prototype

```
UINT nx_ftp_client_disconnect(NX_FTP_CLIENT *ftp_client_ptr,
                                                ULONG wait_option);
```

Description

This service disconnects a previously established FTP Server connection with the specified FTP Client.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

wait_option Defines how long the service will wait for the FTP Client disconnect. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP disconnect.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Disconnect “my_client” from the FTP Server. */
status =  nx_ftp_client_disconnect(&my_client, 200);

/* If status is NX_SUCCESS, “my_client” has been disconnected.  */
```

nx_ftp_client_file_close

Close Client file

Prototype

```
UINT nx_ftp_client_file_close(NX_FTP_CLIENT *ftp_client_ptr,
                                                ULONG wait_option);
```

Description

This service closes a previously opened file on the FTP Server.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

wait_option Defines how long the service will wait for the FTP Client file close. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file close.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_NOT_OPEN	(0xD5)	File not open; cannot close it
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Close previously opened file of client “my_client” on the FTP Server. */
status =  nx_ftp_client_file_close(&my_client, 200);

/* If status is NX_SUCCESS, the file opened previously in the “my_client” FTP
   connection has been closed.  */
```

nx_ftp_client_file_delete

Delete file on FTP Server

Prototype

```
UINT nx_ftp_client_file_delete(NX_FTP_CLIENT *ftp_client_ptr,
                  CHAR *file_name, ULONG wait_option);
```

Description

This service deletes the specified file on the FTP Server.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

file_name Name of file to delete.

wait_option Defines how long the service will wait for the FTP Client file delete. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file delete.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_2XX_CODE	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete the file “my_file.txt” on the FTP Server using the previously
   connected client “my_client.” */
status =  nx_ftp_client_file_delete(&my_client, “my_file.txt”, 200);


/* If status is NX_SUCCESS, the file “my_file.txt” on the FTP Server is
   deleted.  */
```

nx_ftp_client_file_open

Opens file on FTP Server

Prototype

```
UINT nx_ftp_client_file_open(NX_FTP_CLIENT *ftp_client_ptr,
            CHAR *file_name, UINT open_type, ULONG wait_option);
```

Description

This service opens the specified file – for reading or writing – on the FTP Server previously connected to the specified Client instance.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

file_name Name of file to open.

open_type Either **NX_FTP_OPEN_FOR_READ** or **NX_FTP_OPEN_FOR_WRITE**.

wait_option Defines how long the service will wait for the FTP 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file open.
NX_OPTION_ERROR	(0x0A)	Invalid open type.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.

NX_FTP_NOT_CLOSED	(0xD6)	FTP Client is already opened.
NX_NO_FREE_PORTS	(0x45)	No TCP ports available to assign
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

.

Allowed From

Threads

Example

```
/* Open the file “my_file.txt” for reading on the FTP Server using the previously
   connected client “my_client.” */
status =  nx_ftp_client_file_open(&my_client, “my_file.txt”, NX_FTP_OPEN_FOR_READ,
                                  200);


/* If status is NX_SUCCESS, the file “my_file.txt” on the FTP Server is
   open for reading.  */
```

nx_ftp_client_file_read

Read from file

Prototype

```
UINT nx_ftp_client_file_read(NX_FTP_CLIENT *ftp_client_ptr,
                        NX_PACKET **packet_ptr, ULONG wait_option);
```

Description

This service reads a packet from a previously opened file. It should be called repetitively until a status of NX_FTP_END_OF_FILE is received.

Note that the caller does not allocate a packet for this service. It need only supply a pointer to a packet pointer. This service will update that packet pointer to point to a packet retrieved from the socket receive queue. If a successful status is returned, that means there was a packet available, and it is the caller's responsibility to release the packet when it is done with it.

If an non-zero status (either an error status or NX_FTP_END_OF_FILE) is returned, the caller does not release the packet. Otherwise, an error is generated when if the packet pointer is NULL.

Input Parameters

ftp_client_ptr	Pointer to FTP Client control block.
packet_ptr	Pointer to destination for the data packet pointer retrieved from the queue. If successful, the packet data contains some or all of the file.
wait_option	Defines how long the service will wait for the FTP Client file read. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file read.
NX_FTP_NOT_OPEN	(0xD5)	FTP Client is not opened.
NX_FTP_END_OF_FILE	(0xD7)	End of file condition.
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
NX_PACKET   *my_packet;

/* Read a packet of data from the file “my_file.txt” that was previously opened
   from the client “my_client.” */
status =  nx_ftp_client_file_read(&my_client, &my_packet, 200);


/* Check status.  */
if (status != NX_SUCCESS)
{
    error_counter++;
}
else
{
    /* Release packet when done with it. */
    nx_packet_release(my_packet);
}


/* If status is NX_SUCCESS, the packet pointer, “my_packet” points to the packet
   that contains the next bytes from the file. If the file is completely
   downloaded, an NX_FTP_END_OF_FILE status is returned (no packet retrieved). */
```

nx_ftp_client_file_rename

Rename file on FTP Server

Prototype

```
UINT nx_ftp_client_file_rename(NX_FTP_CLIENT *ftp_ptr, CHAR *filename,
                        CHAR *new_filename, ULONG wait_option);
```

Description

This service renames a file on the FTP Server.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

filename Current name of file.

new_filename New name for file.

wait_option Defines how long the service will wait for the FTP Client file rename. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file rename.
NX_FTP_NOT_CONNECTED	(0xD3)	FTP Client is not connected.
NX_FTP_EXPECTED_3XX_CODE	(0XDD)	Did not receive 3XX (ok) response

NX_FTP_EXPECTED_2XX_CODE		
	(0xDA)	Did not get a 2XX (ok) response
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Rename file “my_file.txt” to “new_file.txt” on the previously connected
   Client instance “my_client.” */
status =  nx_ftp_client_file_rename(&my_client, “my_file.txt”, “new_file.txt”,
                        200);

/* If status is NX_SUCCESS, the file has been renamed.  */
```

nx_ftp_client_file_write

Write to file

Prototype

```
UINT nx_ftp_client_file_write(NX_FTP_CLIENT *ftp_client_ptr,
                        NX_PACKET *packet_ptr, ULONG wait_option);
```

Description

This service writes a packet of data to the previously opened file on the FTP Server.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

packet_ptr Packet pointer containing data to write.

wait_option Defines how long the service will wait for the FTP Client file write. 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 FTP 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 FTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful FTP file write.
NX_FTP_NOT_OPEN	(0xD5)	FTP Client is not opened.
NX_PTR_ERROR	(0x07)	Invalid FTP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Write the data contained in “my_packet” to the previously opened file
   “my_file.txt”.  */
status =  nx_ftp_client_file_write(&my_client, my_packet, 200);

/* If status is NX_SUCCESS, the file has been written to.  */
```

nx_ftp_client_passive_mode_set

Enable or disable passive transfer mode

Prototype

```
UINT nx_ftp_client_passive_mode_set(NX_FTP_CLIENT *ftp_client_ptr,
                                    UINT passive_mode_enabled);
```

Description

This service enables passive transfer mode if the `passive_mode_enabled` input is set to NX_TRUE on a previously created FTP Client instance such that subsequent calls to read or write files (RETR, STOR) or download a directory listing (NLST) are done in transfer mode. To disable passive mode transfer and return to the default behavior of active transfer mode, call this function with the `passive_mode_enabled` input set to NX_FALSE.

Input Parameters

ftp_client_ptr Pointer to FTP Client control block.

passive_mode_enabled
If set to NX_TRUE, passive mode is enabled.
If set to NX_FALSE, passive mode is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful passive mode set.
NX_PTR_ERROR	(0x16)	Invalid FTP pointer.
NX_INVALID_PARAMETERS	(0x4D)	Invalid non pointer input

Allowed From

Threads

Example

```
/* Enable the FTP Client to exchange data with the FTP server in passive mode. */
status =  nx_ftp_client_passive_mode_set(&my_client, NX_TRUE);

/* If status is NX_SUCCESS, the FTP client is in passive transfer mode.  */
```

nx_ftp_server_create

Create FTP Server

Prototype

```
UINT nx_ftp_server_create(NX_FTP_SERVER *ftp_server_ptr,
            CHAR *ftp_server_name, NX_IP *ip_ptr,
            FX_MEDIA *media_ptr, VOID *stack_ptr, ULONG stack_size,
            NX_PACKET_POOL *pool_ptr,
            UINT (*ftp_login)(struct NX_FTP_SERVER_STRUCT
                   *ftp_server_ptr, ULONG client_ip_address,
                  UINT client_port, CHAR *name, CHAR *password,
                  CHAR *extra_info),
            UINT (*ftp_logout)(struct NX_FTP_SERVER_STRUCT
                  *ftp_server_ptr,   ULONG client_ip_address,
                  UINT client_port, CHAR *name, CHAR *password,
                  CHAR *extra_info));
```

Description

This service creates an FTP Server instance on the specified and previously created NetX IP instance. Note the FTP Server needs to be started with a call to ***nx_ftp_server_start*** for it to begin operation.

Input Parameters

ftp_server_ptr	Pointer to FTP Server control block.
ftp_server_name	Name of FTP Server.
ip_ptr	Pointer to associated NetX IP instance. Note there can only be one FTP Server for an IP instance.
media_ptr	Pointer to associated FileX media instance.
stack_ptr	Pointer to memory for the internal FTP Server thread's stack area.
stack_size	Size of stack area specified by *stack_ptr*.
pool_ptr	Pointer to default NetX packet pool. Note the payload size of packets in the pool must be large enough to accommodate the largest filename/path.
ftp_login	Function pointer to application's login function. This function is supplied the username and password from the Client requesting a connection. If this is

valid, the application's login function should return NX_SUCCESS.

ftp_logout Function pointer to application's logout function. This function is supplied the username and password from the Client requesting a disconnection. If this is valid, the application's login function should return NX_SUCCESS.

Return Values

NX_SUCCESS	(0x00)	Successful FTP Server create.
NX_PTR_ERROR	(0x16)	Invalid FTP pointer.

Allowed From

Initialization and Threads

Example

```
/* Create the FTP Server "my_server" on the IP instance "ip_0" using the
   "ram_disk" media.  */
status =  nx_ftp_server_create(&my_server, "My Server Name", &ip_0, &ram_disk,
                        stack_ptr, stack_size, &pool_0,
                        my_login, my_logout);

/* If status is NX_SUCCESS, the FTP Server has been created.  */
```

nx_ftp_server_delete

Delete FTP Server

Prototype

```
UINT nx_ftp_server_delete(NX_FTP_SERVER *ftp_server_ptr);
```

Description

This service deletes a previously created FTP Server instance.

Input Parameters

ftp_server_ptr Pointer to FTP Server control block.

Return Values

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

Allowed From

Threads

Example

```
/* Delete the FTP Server "my_server". */
status =  nx_ftp_server_delete(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been deleted.  */
```

nx_ftp_server_start

Start FTP Server

Prototype

```
UINT nx_ftp_server_start(NX_FTP_SERVER *ftp_server_ptr);
```

Description

This service starts a previously created FTP Server instance.

Input Parameters

ftp_server_ptr Pointer to FTP Server control block.

Return Values

NX_SUCCESS	(0x00)	Successful FTP Server start.
NX_PTR_ERROR	(0x16)	Invalid FTP pointer.

Allowed From

Initialization and Threads

Example

```
/* Start the FTP Server “my_server”. */
status =  nx_ftp_server_start(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been started.  */
```

nx_ftp_server_stop

Stop FTP Server

Prototype

```
UINT nx_ftp_server_stop(NX_FTP_SERVER *ftp_server_ptr);
```

Description

This service stops a previously created and started FTP Server instance.

Input Parameters

ftp_server_ptr Pointer to FTP Server control block.

Return Values

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

Allowed From

Threads

Example

```
/* Stop the FTP Server "my_server". */
status =  nx_ftp_server_stop(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been stopped.  */
```