
## File Transfer Protocol (NetX Duo 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 5.10

# Contents

# Chapter 1

# Introduction to NetX Duo 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. NetX Duo FTP accommodates both IPv4 and IPv6 networks. IPv6 does not directly change the FTP protocol, although some changes in the original NetX FTP API are necessary to accommodate IPv6 and will be described in this document.

## FTP Requirements

In order to function properly, the NetX FTP package requires NetX Duo. The host application must create an IP instance for running NetX services and periodic tasks. If running the FTP host application over an IPv6 network, IPv6, and ICMPv6 must be enabled on the IP task. TCP must be also enabled for either IPv6 or IPv4 networks. The IPv6 host application must set its linklocal and global IPv6 address using the IPv6 API and/or DHCPv6. A demo program in section “Small Example System” in **Chapter 2** demonstrates how this is done.

The FTP Server and Client are also designed to work with the FileX embedded file system. If FileX is not available, the host developer can implement or substitute their own file system along the guidelines suggested in filex_stub.h by defining each of the services listed in that file. This is discussed in later sections of this guide.

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.

## FTP Constraints

The FTP standard has many options regarding the representation of file

data. NetX FTP does not implement switch options e.g. ls –al. NetX FTP Server expects to receive requests and their arguments in a single packet rather than consecutive packets.

Similar to UNIX implementations, NetX FTP assumes the following file format constraints:

| | |
|---|---|
| File Type: | **Binary** |
| File Format: | **Nonprint Only** |
| File Structure: | **File Structure 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. Note that the only difference when FTP runs over IPv6 is that the PORT command is replaced with the EPRT command:

| **FTP Command** | **Meaning** |
|---|---|
| CWD path | *Change working directory* |
| DELE filename | *Delete specified file name* |
| EPRT ip_address, port | *Provide IPv6 address and Client data port* |
| 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* |
| PORT ip_address,port | *Provide IP address and Client data port* |
| 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* |

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

# FTP Server Responses

Once the FTP Server processes the Client request, it returns a 3-digit coded response in ASCII followed by optional ASCII text. The numeric response is used by the FTP Client software to determine whether the operation succeeded or failed. The following list shows various FTP Server responses to Client requests:

| 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 Duo 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 Duo 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 Duo 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 Duo 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. Client 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. *IPv6 applications*: Client sends “EPRT” message with IP address and port.
   *IPv4 applications*: 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 previous “EPRT” or “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. Client 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 ***nxd_ftp_server_create*** and ***nx_ftp_server_create*** services and called from the password processing. The difference between the two is the ***nxd_ftp_server_create*** input function pointers to login and logout authenticate functions expect the NetX Duo address type ***NXD_ADDRESS***. This data type holds both IPv4 or IPv6 address formats, making this function the “duo” service supporting both IPv4 and IPv6 networks. The ***nx_ftp_server_create*** input function pointers to login and logout authenticate functions expect ULONG IP address type. This function is limited to IPv4 networks. The developer is encouraged to use the “duo” service whenever possible.

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 Duo FTP is compliant with RFC 959, RFC 2428 and related RFCs.

# Chapter 2

# Installation and Use of FTP

This chapter contains a description of various issues related to installation, set up, and usage of the NetX Duo FTP services.

## Product Distribution

NetX Duo FTP 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:

| | |
|---|---|
| **nxd_ftp_client.h** | Header file for NetX Duo FTP Client |
| **nxd_ftp_client.c** | C Source file for NetX Duo FTP Client |
| **nxd_ftp_server.h** | Header file for NetX Duo FTP Server |
| **nxd_ftp_server.c** | C Source file for NetX Duo FTP Server |
| **filex_stub.h** | Stub file if FileX is not present |
| **nxd_ftp.pdf** | PDF description of FTP for NetX Duo |
| **demo_netxduo_ftp.c** | FTP demonstration system |
| **demo_netxduo_ftp_client_passive.c** | FTP demonstration of file download (read) and upload (write) in passive transfer mode |

## NetX Duo FTP Installation

In order to use the NetX Duo FTP API, the entire distribution mentioned previously should be copied to the same directory where NetX Duo is installed. For example, if NetX Duo is installed in the directory "*threadx**arm7**green*" then the *nxd_ftp_client.h* and *nxd_ftp_client.c* should be copied into this directory for FTP Client applications, and *nxd_ftp_server.h* and *nxd_ftp_server.c* files should be copied into this directory for FTP Server applications.

## Using NetX Duo FTP

Using the NetX Duo FTP API is easy. Basically, the application code must include either *nxd_ftp_client.h* for FTP Client applications or *nxd_ftp_server* for FTP Server applications, after it includes *tx_api.h, fx_api.h,* and *nx_api.h*, in order to use ThreadX, FileX, and NetX Duo, respectively. The build project must include the FTP source code and the host application file, and of course the ThreadX and NetX library files. This is all that is required to use NetX Duo FTP.

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

Note that the NetX Duo library can be enabled for IPv6 and still support IPv4 networks. However, NetX Duo cannot support IPv6 unless it is enabled. To disable IPv6 processing in NetX Duo, the **NX_DISABLE_IPV6** must be defined in the *nx_user.h* file, and that file must be included in the NetX Duo library build by defining **NX_INCLUDE_USER_DEFINE_FILE** in the *nx_port.h* file. By default, **NX_DISABLE_IPV6** is not defined (IPv6 is enabled). This is different from the *nxd_ipv6_enable* service that sets up the IPv6 protocols and services on the IP task, and requires **NX_DISABLE_IPV6** to be not defined.

# Small Example System of NetX Duo FTP

An example of how easy it is to use NetX Duo FTP is described in Figure 1.1 that appears below. In this example, both an FTP Server and an FTP Client are created. Therefore both FTP include files *nxd_ftp_client.h* and *nxd_ftp_server.h are* brought in at line 10 and 11. Next, the FTP Server is created in "*tx_application_define*" at line 99. Note that the FTP Server and Client control blocks are defined as global variables at line 26 previously.

This demo shows how to use the duo functions available in NetX Duo FTP as well as the legacy IPv4 limited FTP services. To use the IPv6 functions, the demo defines USE_IPV6 in line 16

At line 162 the FTP Server is created with ***nxd_ftp_server_create*** if the host application defines USE_IPV6 which supports both IPv4 and IPv6. If it is not, the FTP Server is created with ***nx_ftp_server_create*** on line 166 with the IPv4 limited service. Note that the 'duo' function uses different login and logout function arguments than the IPv4 service, both of which are defined at the bottom of the file on lines 534 -568.

The FTP server must then establish its IPv6 address (global and link local) with NetX Duo, starting at line 466 in the FTP server thread entry function. The FTP server is then started on line 518 and is ready for FTP client requests.

The FTP Client is created in line 316 and goes through the same process as the FTP Server to get the FTP Client IP task IPv6 enabled, and its IPv6 addresses validated starting on lines 263-313.

Then the Client connects to the FTP Server using ***nxd_ftp_client_connect*** in line 334 if it has defined USE_IPV6, or line 340 if it is using the IPv4 limited service

***nx_ftp_client_connect***. Over the course of the FTP Client thread function, it writes a file to the FTP server and reads it back before disconnecting.

```
  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    "nxd_ftp_client.h"
 11 #include    "nxd_ftp_server.h"
 12
 13 #define     DEMO_STACK_SIZE         4096
 14
 15 #ifdef FEATURE_NX_IPV6
 16 #define USE_IPV6
 17 #endif /* FEATURE_NX_IPV6 */
 18
 19
 20 /* Define the ThreadX, NetX, and FileX object control blocks...  */
 21
 22 TX_THREAD               server_thread;
 23 TX_THREAD               client_thread;
 24 NX_PACKET_POOL          server_pool;
 25 NX_IP                   server_ip;
 26 NX_PACKET_POOL          client_pool;
 27 NX_IP                   client_ip;
 28 FX_MEDIA                ram_disk;
 29
 30
 31 /* Define the NetX FTP object control blocks.  */
 32
 33 NX_FTP_CLIENT           ftp_client;
 34 NX_FTP_SERVER           ftp_server;
 35
 36
 37 /* Define the counters used in the demo application...  */
 38
 39 ULONG                   error_counter = 0;
 40
 41
 42 /* Define the memory area for the FileX RAM disk.  */
 43
 44 UCHAR                   ram_disk_memory[32000];
 45 UCHAR                   ram_disk_sector_cache[512];
 46
 47
 48 #define FTP_SERVER_ADDRESS  IP_ADDRESS(1,2,3,4)
 49 #define FTP_CLIENT_ADDRESS  IP_ADDRESS(1,2,3,5)
 50
 51 extern UINT  _fx_media_format(FX_MEDIA *media_ptr, VOID (*driver)(FX_MEDIA *media),
VOID *driver_info_ptr, UCHAR *memory_ptr, UINT memory_size,
 52                         CHAR *volume_name, UINT number_of_fats, UINT
directory_entries, UINT hidden_sectors,
 53                         ULONG total_sectors, UINT bytes_per_sector, UINT
sectors_per_cluster,
 54                         UINT heads, UINT sectors_per_track);
 55
 56 /* Define the FileX and NetX driver entry functions.  */
 57 VOID    _fx_ram_driver(FX_MEDIA *media_ptr);
 58
 59 /* Replace the 'ram' driver with your own Ethernet driver. */
 60 VOID    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
 61
 62
 63 void    client_thread_entry(ULONG thread_input);
 64 void    thread_server_entry(ULONG thread_input);
 65
 66
 67 #ifdef USE_IPV6
 68 /* Define NetX Duo IP address for the NetX Duo FTP Server and Client. */
 69 NXD_ADDRESS     server_ip_address;
 70 NXD_ADDRESS     client_ip_address;
```

```
 71 #endif
 72
 73
 74 /* Define server login/logout functions.  These are stubs for functions that would
 75    validate a client login request.   */
 76
 77 #ifdef USE_IPV6
 78 UINT    server_login6(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, NXD_ADDRESS
*client_ipduo_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info);
 79 UINT    server_logout6(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, NXD_ADDRESS
*client_ipduo_address, UINT client_port, CHAR *name, CHAR *password, CHAR *extra_info);
 80 #else
 81 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);
 82 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);
 83 #endif
 84
 85
 86 /* Define main entry point.  */
 87
 88 int main()
 89 {
 90
 91     /* Enter the ThreadX kernel.  */
 92     tx_kernel_enter();
 93     return(0);
 94 }
 95
 96
 97 /* Define what the initial system looks like.  */
 98
 99 void    tx_application_define(void *first_unused_memory)
100 {
101
102 UINT    status;
103 UCHAR   *pointer;
104
105
106     /* Setup the working pointer.  */
107     pointer =  (UCHAR *) first_unused_memory;
108
109     /* Create a helper thread for the server. */
110     tx_thread_create(&server_thread, "FTP Server thread", thread_server_entry, 0,
111                      pointer, DEMO_STACK_SIZE,
112                      4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
113
114     pointer =  pointer + DEMO_STACK_SIZE;
115
116     /* Initialize NetX.  */
117     nx_system_initialize();
118
119     /* Create the packet pool for the FTP Server.  */
120     status = nx_packet_pool_create(&server_pool, "NetX Server Packet Pool", 256,
pointer, 8192);
121     pointer = pointer + 8192;
122
123     /* Check for errors.  */
124     if (status)
125         error_counter++;
126
127     /* Create the IP instance for the FTP Server.  */
128     status = nx_ip_create(&server_ip, "NetX Server IP Instance", FTP_SERVER_ADDRESS,
0xFFFFFF00UL,
129                                         &server_pool, _nx_ram_network_driver, pointer,
2048, 1);
130     pointer = pointer + 2048;
131
132     /* Check status.  */
133     if (status != NX_SUCCESS)
134     {
135         error_counter++;
136         return;
137     }
138
139     /* Enable ARP and supply ARP cache memory for server IP instance.  */
140     nx_arp_enable(&server_ip, (void *) pointer, 1024);
141     pointer = pointer + 1024;
142
143     /* Enable TCP.  */
144     nx_tcp_enable(&server_ip);
```

```
145
146 #ifdef USE_IPV6
147
148     /* Next set the NetX Duo FTP Server and Client addresses. */
149     server_ip_address.nxd_ip_address.v6[3] = 0x105;
150     server_ip_address.nxd_ip_address.v6[2] = 0x0;
151     server_ip_address.nxd_ip_address.v6[1] = 0x0000f101;
152     server_ip_address.nxd_ip_address.v6[0] = 0x20010db8;
153     server_ip_address.nxd_ip_version = NX_IP_VERSION_V6;
154
155     client_ip_address.nxd_ip_address.v6[3] = 0x101;
156     client_ip_address.nxd_ip_address.v6[2] = 0x0;
157     client_ip_address.nxd_ip_address.v6[1] = 0x0000f101;
158     client_ip_address.nxd_ip_address.v6[0] = 0x20010db8;
159     client_ip_address.nxd_ip_version = NX_IP_VERSION_V6;
160
161     /* Create the FTP server.  */
162     status =  nxd_ftp_server_create(&ftp_server, "FTP Server Instance", &server_ip,
&ram_disk, pointer, DEMO_STACK_SIZE, &server_pool,
163                                     server_login6, server_logout6);
164 #else
165     /* Create the FTP server.  */
166     status =  nx_ftp_server_create(&ftp_server, "FTP Server Instance", &server_ip,
&ram_disk, pointer, DEMO_STACK_SIZE, &server_pool,
167                                     server_login, server_logout);
168 #endif
169     pointer =  pointer + DEMO_STACK_SIZE;
170
171     /* Check status.  */
172     if (status != NX_SUCCESS)
173     {
174         error_counter++;
175         return;
176     }
177
178     /* Now set up the FTP Client. */
179
180     /* Create the main FTP client thread.  */
181     status = tx_thread_create(&client_thread, "FTP Client thread ",
client_thread_entry, 0,
182             pointer, DEMO_STACK_SIZE,
183             6, 6, TX_NO_TIME_SLICE, TX_AUTO_START);
184     pointer = pointer + DEMO_STACK_SIZE ;
185
186     /* Check status.  */
187     if (status != NX_SUCCESS)
188     {
189         error_counter++;
190         return;
191     }
192
193     /* Create a packet pool for the FTP client.  */
194     status =  nx_packet_pool_create(&client_pool, "NetX Client Packet Pool", 256,
pointer, 8192);
195     pointer =  pointer + 8192;
196
197     /* Create an IP instance for the FTP client.  */
198     status = nx_ip_create(&client_ip, "NetX Client IP Instance", FTP_CLIENT_ADDRESS,
0xFFFFFF00UL,
199                                                 &client_pool, _nx_ram_network_driver,
pointer, 2048, 1);
200     pointer = pointer + 2048;
201
202     /* Enable ARP and supply ARP cache memory for the FTP Client IP.  */
203     nx_arp_enable(&client_ip, (void *) pointer, 1024);
204
205     pointer = pointer + 1024;
206
207     /* Enable TCP for client IP instance.  */
208     nx_tcp_enable(&client_ip);
209
210     return;
211
212 }
213
214 /* Define the FTP client thread.  */
215
216 void    client_thread_entry(ULONG thread_input)
217 {
218
219 NX_PACKET   *my_packet;
```

```
220 UINT        status;
221
222 #ifdef USE_IPV6
223 UINT        iface_index, address_index;
224 #endif
225
226
227     /* Format the RAM disk - the memory for the RAM disk was defined above.  */
228     status = _fx_media_format(&ram_disk,
229                               _fx_ram_driver,                   /* Driver entry
*/
230                               ram_disk_memory,                  /* RAM disk memory
pointer     */
231                               ram_disk_sector_cache,            /* Media buffer pointer
*/
232                               sizeof(ram_disk_sector_cache),    /* Media buffer size
*/
233                               "MY_RAM_DISK",                    /* Volume Name
*/
234                               1,                                /* Number of FATs
*/
235                               32,                               /* Directory Entries
*/
236                               0,                                /* Hidden sectors
*/
237                               256,                              /* Total sectors
*/
238                               128,                              /* Sector size
*/
239                               1,                                /* Sectors per cluster
*/
240                               1,                                /* Heads
*/
241                               1);                               /* Sectors per track
*/
242
243     /* Check status.  */
244     if (status != NX_SUCCESS)
245     {
246         error_counter++;
247         return;
248     }
249
250     /* Open the RAM disk.  */
251     status = fx_media_open(&ram_disk, "RAM DISK", _fx_ram_driver, ram_disk_memory,
ram_disk_sector_cache, sizeof(ram_disk_sector_cache));
252
253     /* Check status.  */
254     if (status != NX_SUCCESS)
255     {
256         error_counter++;
257         return;
258     }
259
260     /* Let the IP threads and driver initialize the system.    */
261     tx_thread_sleep(100);
262
263 #ifdef USE_IPV6
264
265     /* Here's where we make the FTP Client IPv6 enabled. */
266     status = nxd_ipv6_enable(&client_ip);
267
268     /* Check status.  */
269     if (status != NX_SUCCESS)
270     {
271         error_counter++;
272         return;
273     }
274
275     status = nxd_icmp_enable(&client_ip);
276
277     /* Check status.  */
278     if (status != NX_SUCCESS)
279     {
280         error_counter++;
281         return;
282     }
283
284     /* Set the Client link local and global addresses. */
285     iface_index = 0;
286
```

```
287     /* This assumes we are using the primary network interface (index 0). */
288     status = nxd_ipv6_address_set(&client_ip, iface_index, NX_NULL, 10,
&address_index);
289
290     /* Check for link local address set error.  */
291     if (status != NX_SUCCESS)
292     {
293
294         error_counter++;
295         return;
296      }
297
298      /* Set the host global IP address. We are assuming a 64
299         bit prefix here but this can be any value (< 128). */
300     status = nxd_ipv6_address_set(&client_ip, iface_index, &client_ip_address, 64,
&address_index);
301
302     /* Check for global address set error.  */
303     if (status != NX_SUCCESS)
304     {
305
306         error_counter++;
307         return;
308      }
309
310     /* Let NetX Duo validate the addresses. */
311     tx_thread_sleep(400);
312
313 #endif  /* USE_IPV6 */
314
315     /* Create an FTP client.  */
316     status =  nx_ftp_client_create(&ftp_client, "FTP Client", &client_ip, 2000,
&client_pool);
317
318     /* Check status.  */
319     if (status != NX_SUCCESS)
320     {
321
322         error_counter++;
323         return;
324      }
325
326     printf("Created the FTP Client\n");
327
328 #ifdef USE_IPV6
329
330     do
331     {
332
333         /* Now connect with the NetX Duo FTP (IPv6) server. */
334         status =  nxd_ftp_client_connect(&ftp_client, &server_ip_address, "name",
"password", 100);
335     } while (status != NX_SUCCESS);
336
337 #else
338
339     /* Now connect with the NetX FTP (IPv4) server. */
340     status =  nx_ftp_client_connect(&ftp_client, FTP_SERVER_ADDRESS, "name",
"password", 100);
341
342 #endif  /* USE_IPV6 */
343
344     /* Check status.  */
345     if (status != NX_SUCCESS)
346     {
347
348         error_counter++;
349         return;
350      }
351
352     printf("Connected to the FTP Server\n");
353
354     /* Open a FTP file for writing.  */
355     status =  nx_ftp_client_file_open(&ftp_client, "test.txt", NX_FTP_OPEN_FOR_WRITE,
100);
356
357     /* Check status.  */
358     if (status != NX_SUCCESS)
359     {
360
361         error_counter++;
```

```
362         return;
363      }
364
365     printf("Opened the FTP client test.txt file\n");
366
367     /* Allocate a FTP packet.  */
368     status =  nx_packet_allocate(&client_pool, &my_packet, NX_TCP_PACKET, 100);
369
370     /* Check status.  */
371     if (status != NX_SUCCESS)
372     {
373
374         error_counter++;
375         return;
376      }
377
378     /* Write ABCs into the packet payload!  */
379     memcpy(my_packet -> nx_packet_prepend_ptr, "ABCDEFGHIJKLMNOPQRSTUVWXYZ  ", 28);
380
381     /* Adjust the write pointer.  */
382     my_packet -> nx_packet_length =  28;
383     my_packet -> nx_packet_append_ptr =  my_packet -> nx_packet_prepend_ptr + 28;
384
385     /* Write the packet to the file test.txt.  */
386     status =  nx_ftp_client_file_write(&ftp_client, my_packet, 100);
387
388     /* Check status.  */
389     if (status != NX_SUCCESS)
390     {
391         error_counter++;
392     }
393     else
394         printf("Wrote to the FTP client test.txt file\n");
395
396
397     /* Close the file.  */
398     status =  nx_ftp_client_file_close(&ftp_client, 100);
399
400     /* Check status.  */
401     if (status != NX_SUCCESS)
402         error_counter++;
403     else
404         printf("Closed the FTP client test.txt file\n");
405
406
407     /* Now open the same file for reading.  */
408     status =  nx_ftp_client_file_open(&ftp_client, "test.txt", NX_FTP_OPEN_FOR_READ,
100);
409
410     /* Check status.  */
411     if (status != NX_SUCCESS)
412         error_counter++;
413     else
414         printf("Reopened the FTP client test.txt file\n");
415
416     /* Read the file.  */
417     status =  nx_ftp_client_file_read(&ftp_client, &my_packet, 100);
418
419     /* Check status.  */
420     if (status != NX_SUCCESS)
421         error_counter++;
422     else
423     {
424             printf("Reread the FTP client test.txt file\n");
425             nx_packet_release(my_packet);
426     }
427
428     /* Close this file.  */
429     status =  nx_ftp_client_file_close(&ftp_client, 100);
430
431     if (status != NX_SUCCESS)
432         error_counter++;
433
434     /* Disconnect from the server.  */
435     status =  nx_ftp_client_disconnect(&ftp_client, 100);
436
437     /* Check status.  */
438     if (status != NX_SUCCESS)
439         error_counter++;
440
441
```

```
442     /* Delete the FTP client.  */
443     status =  nx_ftp_client_delete(&ftp_client);
444
445     /* Check status.  */
446     if (status != NX_SUCCESS)
447         error_counter++;
448 }
449
450
451 /* Define the helper FTP server thread.  */
452 void    thread_server_entry(ULONG thread_input)
453 {
454
455 UINT            status;
456 #ifdef  USE_IPV6
457 UINT            iface_index, address_index;
458 #endif
459
460     /* Wait till the IP thread and driver have initialized the system. */
461     tx_thread_sleep(100);
462
463 #ifdef USE_IPV6
464
465     /* Here's where we make the FTP server IPv6 enabled. */
466     status = nxd_ipv6_enable(&server_ip);
467
468     /* Check status.  */
469     if (status != NX_SUCCESS)
470     {
471
472         error_counter++;
473         return;
474      }
475
476     status = nxd_icmp_enable(&server_ip);
477
478     /* Check status.  */
479     if (status != NX_SUCCESS)
480     {
481
482         error_counter++;
483         return;
484      }
485
486      /* Set the link local address with the host MAC address. */
487     iface_index = 0;
488
489     /* This assumes we are using the primary network interface (index 0). */
490     status = nxd_ipv6_address_set(&server_ip, iface_index, NX_NULL, 10,
&address_index);
491
492     /* Check for link local address set error.  */
493     if (status)
494     {
495
496         error_counter++;
497         return;
498      }
499
500     /* Set the host global IP address. We are assuming a 64
501        bit prefix here but this can be any value (< 128). */
502     status = nxd_ipv6_address_set(&server_ip, iface_index, &server_ip_address, 64,
&address_index);
503
504     /* Check for global address set error.  */
505     if (status)
506     {
507
508         error_counter++;
509         return;
510      }
511
512     /* Wait while NetX Duo validates the link local and global address. */
513     tx_thread_sleep(500);
514
515 #endif /* USE_IPV6 */
516
517     /* OK to start the FTP Server.   */
518     status = nx_ftp_server_start(&ftp_server);
519
520     if (status != NX_SUCCESS)
```

```
521         error_counter++;
522
523     printf("Server started!\n");
524
525     /* FTP server ready to take requests! */
526
527     /* Let the IP threads execute.    */
528     tx_thread_relinquish();
529
530     return;
531 }
532
533
534 #ifdef USE_IPV6
535 UINT  server_login6(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, NXD_ADDRESS
*client_ipduo_address, UINT client_port,
536                     CHAR *name, CHAR *password, CHAR *extra_info)
537 {
538     printf("Logged in6!\n");
539
540     /* Always return success.  */
541     return(NX_SUCCESS);
542 }
543
544 UINT  server_logout6(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr, NXD_ADDRESS
*client_ipduo_address, UINT client_port,
545                      CHAR *name, CHAR *password, CHAR *extra_info)
546 {
547     printf("Logged out6!\n");
548
549     /* Always return success.  */
550     return(NX_SUCCESS);
551 }
552 #else
553 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)
554 {
555
556     printf("Logged in!\n");
557     /* Always return success.  */
558     return(NX_SUCCESS);
559 }
560
561 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)
562 {
563     printf("Logged out!\n");
564
565     /* Always return success.  */
566     return(NX_SUCCESS);
567 }
568 #endif  /* USE_IPV6 */
```

Figure 1.1 Example of NetX Duo FTP

# Configuration Options

There are several configuration options for building NetX FTP and NetX Duo FTP. The default values are listed, but each define can be set by the application prior to inclusion of the specified NetX Duo FTP header file. If no header file is specified, the option is available in both *nxd_ftp_client.h* and *nxd_ftp_server.h*. 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_TIMEOUT** | Specifies the number of ThreadX ticks that internal services will suspend for. The default value is set to 1 second (1 * NX_IP_PERIODIC_RATE). |
| **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. |
| **NX_FTP_TIMEOUT_PERIOD** | Specifies the intervals in seconds when the Server checks for Client activity. The default value is set to 60. |
| **NX_FTP_SERVER_RETRY_SECONDS** | Specifies the initial timeout in seconds before retransmitting server response. The default value is 2. |
| **NX_FTP_SERVER_TRANSMIT_QUEUE_DEPTH** | |

| | |
|---|---|
| | Specifies the maximum of depth of queued transmit packets on Server socket. The default value is 20. |
| **NX_FTP_SERVER_RETRY_MAX** | Specifies the maximum retries per packet. The default value is 10. |
| **NX_FTP_SERVER_RETRY_SHIFT** | Specifies the number of bits to shift in setting the retry timeout. The default value is 2, e.g. every retry timeout is twice as long as the previous retry. |
| **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 control requests. By default, this value is defined as NX_IP_NORMAL to indicate normal IP packet service. |
| **NX_FTP_DATA_TOS** | Type of service required for the FTP data requests. By default, this value is defined as NX_IP_NORMAL to indicate normal IP packet service. |
| **NX_FTP_FRAGMENT_OPTION** | Fragment enable for FTP requests. By default, this value is NX_DONT_FRAGMENT to disable FTP TCP fragmenting. |
| **NX_FTP_CONTROL_WINDOW_SIZE** | TCP Control socket window size. By default, this value is 400 bytes. |
| **NX_FTP_DATA_WINDOW_SIZE** | TCP Data socket window size. By default, this value is 2048 bytes. |

| | |
|---|---|
| **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. |
| **NX_FTP_USERNAME_SIZE** | Specifies the number of bytes allowed in a Client supplied *username*. The default value is set to 20 *.* |
| **NX_FTP_PASSWORD_SIZE** | Specifies the number of bytes allowed in a client supplied *password*. The default value is set to 20. |

# Chapter 3

# Description of FTP Services

This chapter contains a description of all NetX FTP services (listed below) in alphabetic order (except where IPv4 and IPv6 equivalents of the same service are paired together).

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 with IPv4 only*

nxd_ftp_client_connect
*Connect to FTP Server with IPv6 and IPv4 support*

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_client_passive_mode_set
*Enable or disable passive transfer*

nx_ftp_server_create
*Create FTP Server with IPv4 support only*

nxd_ftp_server_create
*Create FTP Server with IPv4 and IPv6 support*

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 over IPv4

### 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 NetX 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.  */
```

### See Also

nx_ftp_client_create, nx_ftp_client_delete, nx_ftp_client_directory_create, nx_ftp_client_disconnect, nxd_ftp_client_connect

# nxd_ftp_client_connect

Connect to an FTP Server with IPv6 support

### Prototype

```
UINT nxd_ftp_client_connect(NX_FTP_CLIENT *ftp_client_ptr,
            NXD_ADDRESS *server_ipduo, CHAR *username, CHAR *password,
            ULONG wait_option);
```

### Description

This service connects the previously created NetX Duo FTP Client instance to the FTP Server at the supplied IP address. Both IPv4 and IPv6 networks are supported.

### Input Parameters

**ftp_client_ptr** Pointer to FTP Client control block.

**server_ipduo** IP address of the 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 an FTP Client instance to the FTP Server.  */

/* Set up an IPv6 address for the server here. Note this could also be an IPv4 address as
   well*/
server_ip_addr.nxd_ip_address.v6[3] = 0x106;
server_ip_addr.nxd_ip_address.v6[2] = 0x0;
server_ip_addr.nxd_ip_address.v6[1] = 0x0000f101;
server_ip_addr.nxd_ip_address.v6[0] = 0x20010db8;
server_ip_addr.nxd_ip_version = NX_IP_VERSION_V6;

status =  nxd_ftp_client_connect(&my_client, server_ip_addr, NULL, NULL, 100);


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

## See Also

nx_ftp_client_create, nx_ftp_client_delete, nx_ftp_client_directory_create, nx_ftp_client_disconnect, nx_ftp_client_connect

# 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 | (0x07) | Invalid pointer input. |

### 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 not disconnected |
| NX_PTR_ERROR | (0x07) | 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.

### Input Parameters

**ftp_client_ptr** Pointer to FTP Client control block.

**packet_ptr** Pointer to destination for the data packet pointer to be stored. If successful, the packet some or all the contains 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

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


/* If status is NX_SUCCESS, the packet “my_packet” contains the next bytes
   from the file.  */
```

# 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 | (0x07) | 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, and the Client |

address in the ULONG data type. 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, and the Client address in the ULONG data type. 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 | (0x07) | 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.  */
```

# nxd_ftp_server_create

Create FTP Server with IPv4 and IPv6 support

### Prototype

```
UINT nxd_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_duo)(struct NX_FTP_SERVER_STRUCT
                   *ftp_server_ptr, NXD_ADDRESS *client_ip_address,
                  UINT client_port, CHAR *name, CHAR *password,
                  CHAR *extra_info),
            UINT (*ftp_logout_duo)(struct NX_FTP_SERVER_STRUCT
                  *ftp_server_ptr,   NXD_ADDRESS *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 still needs to be started with a call to ***nx_ftp_server_start*** for it to begin operation after the Server is created.

### 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_duo** | Function pointer to application's login function. This function is supplied the username and password from the Client requesting a connection, and a pointer to the Client address in the NXD_ADDRESS data type. If this is |

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

**ftp_logout_duo** Function pointer to application's logout function. This function is supplied the username and password from the Client requesting a disconnection, and a pointer to the Client address in the NXD_ADDRESS data type. 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 | (0x07) | Invalid FTP pointer. |
| NX_CALLER_ERROR | (0x11) | Invalid caller of this service. |

### Allowed From

Initialization and Threads

### Example

```
/* Create the FTP Server "my_server" on the IP instance "ip_0" using the
   "ram_disk" media.  */
status =  nxd_ftp_server_create(&my_server, "My Server Name", &ip_0, &ram_disk,
                        stack_ptr, stack_size, &pool_0,
                        my_duo_login, my_duo_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 | (0x07) | 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 | (0x07) | Invalid FTP pointer. |

### Allowed From

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 | (0x07) | 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.  */
```