

Telnet-32 Library

**Telnet Services
Dynamic Link Library
for Microsoft© Windows™**

Version 5.2

**Copyright © 1988 - 2003 by Distinct Corporation
All rights reserved**

Table of Contents

1 Overview	5
1.1 Introduction	5
1.2 Registry Entries	6
1.3 Error Codes	6
1.4 D32-TNET.H.....	7
1.5 Function Summary	10
2 Reference.....	11
2.1 tnet_abort ().....	13
2.2 tnet_binary ().....	14
2.3 tnet_close ().....	15
2.4 tnet_error ()	16
2.5 tnet_fw_open ()	17
2.6 tnet_fw_x_open ()	19
2.7 tnet_getc ().....	22
2.8 tnet_local ()	23
2.9 tnet_oob_read ()	24
2.10 tnet_oob_status ()	25
2.11 tnet_oob_write ().....	26
2.12 tnet_open ()	27
2.13 tnet_option_status ().....	29
2.14 tnet_putc ()	30
2.15 tnet_read ()	31
2.16 tnet_remote ()	32
2.17 tnet_status ()	33
2.18 tnet_write ().....	34
2.19 tnet_x_open ()	35

1 Overview

1.1 Introduction

The Distinct Telnet-32 library provides developers with an API to manage one or more Telnet connections. A new connection is established by calling **tnet_open**. This function allows the caller to connect to any TCP port on the given host (not just to the default Telnet port). The caller can also specify the terminal type that will be supported. Once the **tnet_open** call returns successfully, a TCP connection with the remote host has been established. The Telnet-32 library takes care of all necessary option negotiations according to the Telnet protocol definition and filters out and replies to Telnet escape sequences in the incoming data stream. This ensures that the calling application only deals with actual user data. If the application requires more control over the various connection characteristics then it should use **tnet_x_open**. This function allows the caller to specify the network timeout value. The caller can also control if urgent data is handled inline or out of band (OOB). The user can also specify the option negotiation buffer size and a callback function to handle option negotiations. In case of an error, the **tnet_error** function can be called to determine the exact error code. To establish a connection using a firewall server, use the **tnet_fw_(x)_open** function.

The Telnet-32 library uses a message based mechanism to notify the application that data has arrived on one of its connections. In the **tnet_open** or **tnet_x_open** call, the caller specifies a window handle and the message the window should receive when data is available or some other network event takes place. By choosing different messages and/or using different window handles, an application can easily and efficiently maintain several Telnet connections. Although this is not recommended, an application can also poll a Telnet connection regularly for the availability of data by calling **tnet_status**. Normally **tnet_status** should be used after receiving a message indicating incoming data to determine how much data is there to be read.

Whenever an application calls **tnet_read**, any Telnet option negotiation requests and other escape sequences are filtered out by the Telnet-32 library. If the filtered sequence requires that an answer be sent to the remote host, then the answer is put into the TCP send queue of the associated socket at this point. Because messages are generated whenever any data arrives regardless of the type of data (user data or Telnet sequence), **tnet_read** may return with less bytes than the number returned by **tnet_status**. Immediately after a connection has been established, **tnet_read** may even return zero bytes one or more times. This is normal behavior and simply indicates that Telnet negotiations are being processed. If a callback function is specified in **tnet_x_open** then that function may be called from within **tnet_read** to process the option negotiations.

An application will be notified of any change in the state of a particular Telnet connection option with a message with the low word (LOWORD) of lParam containing the TNET_UPCALL_STATE flag and the high word (HIWORD) of lParam containing the option which has changed. The application should call **tnet_option_status** to find out the exact state of the option. The application can also use **tnet_local**, **tnet_remote** and **tnet_binary** to change the state of the Telnet connection options.

Once a Telnet connection is no longer needed, **tnet_close** can be called to release the resources allocated for the connection. This function properly closes down the TCP socket which was used to connect to the remote port. It does not perform a logout. An application must call **tnet_close** for all connections it has allocated by calling **tnet_open**. The **tnet_close** function must be called even if the application has received a message indicating that the remote host has closed the connection.

The Distinct Telnet-32 library also provides an option for generating debug information. The debug log information and the level of debug information can be controlled through a registry entry (refer to the following section for more details).

1.2 Registry Entries

Various parameters used to control the behavior of the Distinct Telnet-32 library are stored under the following registry path.

HKEY_LOCAL_MACHINE\SOFTWARE\Distinct\DLLS\TNET32

The following entries specify various Distinct Telnet-32 library settings.

Timeout **REG_DWORD** *Number*

Specifies the default timeout in seconds used for the Telnet connection. This is the timeout value used for various network operations. The default value is 20 seconds.

BufferSize **REG_DWORD** *128-8192 bytes*

Specifies the default buffer size in bytes used for option negotiations. The default value is 1024 bytes.

DebugLevel **REG_DWORD** *0-4*

Specifies what debug information should be logged into the D32-TNET.DBG file in the directory from which D32-TNET.DLL was loaded. The following values are supported.

Value	Meaning
0	Disable logging.
1	Log all errors.
2	Log all option negotiations and state changes.
3	Log all function calls and return codes.
4	Log network traffic.

A higher debug level automatically includes all lower debug levels. The default value is 0.

1.3 Error Codes

The **tnet_open** and **tnet_x_open** functions return NULL if the Telnet connection could not be established. The exact error code can be determined by calling the **tnet_error** function. The various error codes are listed below. For all other function calls, the **tnet_error** function will return the Windows Sockets error code if a network error occurred.

Value	Meaning
TNET_BAD_GLOBALHANDLE	Invalid connection handle.
TNET_DSTNCT32_ERROR	Copy protection violation.
TNET_WSASTARTUP_ERROR	Error in initializing Windows Sockets.
TNET_SYSTEM_ERROR	Error in allocating system resources.
TNET_INVALID_HOSTNAME	Specified host name is not valid.
TNET_SOCKET_ERROR	Error in creating socket.
TNET_BIND_ERROR	Error in binding to local port.
TNET_CONNECT_ERROR	Error in connecting to remote host.
TNET_ASYNC_ERROR	Error in enabling asynchronous notification.
TNET_MAX_CONNECTIONS	Cannot open any more connections.
TNET_OOB_ERROR	Error in handling urgent data options.

1.4 D32-TNET.H

```
/*
*****/
/*
    Copyright (C) 1991 - 1996 Distinct Corporation
    Program name   : d32-tnet.dll
    File name      : d32-tnet.h
    Description    : main header file
*/
/*
*****/

#ifndef _TNET32_H_INCLUDED_
#define _TNET32_H_INCLUDED_

/* maximum concurrent connections - per process */
#define MAX_NO_CONN 16

/* maximum number of telnet options supported */
#define MAX_OPTIONS 32

/* maximum length of terminal string */
#define MAX_TERM_TYPE 40

/* CmdBuffer length limits */
#define MAX_CMD_BUFFER_SIZE 8192
#define DEF_CMD_BUFFER_SIZE 1024
#define MIN_CMD_BUFFER_SIZE 128

/* default timeout */
#define DEF_TIMEOUT 20

/* connection status */
#define NOTCONNECTED 0
#define TRYING 1
#define CONNECTED 2

/* standard server telnet port */
#define TELNET_PORT 23

/* error return values, should not conflict with ftp error codes*/
#define TNET_BAD_GLOBALHANDLE -1
#define TNET_SUCCESS 0
#define TNET_DSTNCT32_ERROR 1000
#define TNET_WSASTARTUP_ERROR 1001
#define TNET_SYSTEM_ERROR 1002
#define TNET_INVALID_HOSTNAME 1003
#define TNET_SOCKET_ERROR 1004
#define TNET_BIND_ERROR 1005
#define TNET_CONNECT_ERROR 1006
#define TNET_ASYNC_ERROR 1007
#define TNET_MAX_CONNECTIONS 1008
#define TNET_OOB_ERROR 1009

/* urgent data types */
#define TNET_URGENT_INLINE 0
#define TNET_URGENT_OOB 1
```

```

/* upcall messages */
#define TNET_UPCALL_CLOSE      0x1
#define TNET_UPCALL_RESET     0x2
#define TNET_UPCALL_OOB       0x4
#define TNET_UPCALL_STATE     0x8

/* TELNET negotiation modes and states */
#define NORMALMODE      0
#define CURRENT_WILL    0x1
#define CURRENT_DO      0x2
#define WANT_WILL       0x4
#define WANT_DO         0x8
#define ALLOW_WILL      0x10
#define ALLOW_DO        0x20

/* telnet commands */
#define IAC              255      /* 0xff - interpret as command */
#define DONT             254      /* 0xfe - do not use this option */
#define DO               253      /* 0xfd - please use this option */
#define WONT            252      /* 0xfc - I won't be using this option */
#define WILL            251      /* 0xfb - I will be using this option */
#define SBNT           250      /* 0xfa - start of sub-negotiation */
#define GA              249      /* 0xf9 - go ahead */
#define EL              248      /* 0xf8 - erase line*/
#define EC              247      /* 0xf7 - erase character */
#define AYT            246      /* 0xf6 - are you there */
#define AO              245      /* 0xf5 - abort output */
#define INTPT          244      /* 0xf4 - interrupt process */
#define BRK            243      /* 0xf3 - break character */
#define DM             242      /* 0xf2 - data mark */
#define NOP            241      /* 0xf2 - NOP */
#define SENVT          240      /* 0xf0 - end of sub-negotiation */
#define EOR            239      /* 0xef - end of record (transparent mode) */

/* telnet options */
#define OPT_EOR         25      /* 0x19 */
#define OPT_TERMTYPE   24      /* 0x18 */
#define OPT_SPGA        3      /* 0x03 */
#define OPT_ECHO        1      /* 0x01 */
#define OPT_BINARY      0      /* 0x00 */

#define SEND            1      /* 0x01 */
#define IS              0      /* 0x00 */

// callback function declaration
#ifdef STRICT
typedef int (CALLBACK *TNETPROC) (GLOBALHANDLE, char far *, int);
#else
typedef FARPROC TNETPROC;
#endif

#ifdef __cplusplus
extern "C" {
#endif

GLOBALHANDLE WINAPI tnet_open (HWND, LPSTR, UINT, LPSTR, UINT);
GLOBALHANDLE WINAPI tnet_fw_open (HWND, LPSTR, UINT, LPVOID);
GLOBALHANDLE WINAPI tnet_x_open (HWND, LPSTR, UINT, LPSTR, UINT, \
    int, int, int, TNETPROC);
GLOBALHANDLE WINAPI tnet_fw_x_open (HWND, LPSTR, UINT, \
    int, int, int, TNETPROC, LPVOID);

```

```
int WINAPI tnet_close (GLOBALHANDLE);
int WINAPI tnet_abort (GLOBALHANDLE);
int WINAPI tnet_getc (GLOBALHANDLE);
int WINAPI tnet_putc (GLOBALHANDLE, char);
int WINAPI tnet_write (GLOBALHANDLE, char far *, int);
int WINAPI tnet_read (GLOBALHANDLE, char far *, int);
int WINAPI tnet_oob_write (GLOBALHANDLE, char far *, int);
int WINAPI tnet_oob_read (GLOBALHANDLE, char far *, int);
int WINAPI tnet_local (GLOBALHANDLE);
int WINAPI tnet_remote (GLOBALHANDLE);
int WINAPI tnet_binary (GLOBALHANDLE, BOOL);
int WINAPI tnet_status (GLOBALHANDLE, unsigned long);
BOOL WINAPI tnet_oob_status (GLOBALHANDLE);
int WINAPI tnet_option_status (GLOBALHANDLE, int);
ULONG WINAPI tnet_local_addr (GLOBALHANDLE);
int WINAPI tnet_error ();

#ifdef __cplusplus
}
#endif

#endif // _TNET32_H_INCLUDED_

/*****
/*
    end of d32-tnet.h
*/
*****/
```

1.5 Function Summary

tnet_abort

Abort the Telnet connection

tnet_binary

Enable or disable binary mode

tnet_close

Close the Telnet connection

tnet_error

Get the last error

tnet_fw_open

Establish a Telnet connection via a firewall server

tnet_fw_x_open

Establish a Telnet connection with the given settings via a firewall server

tnet_getc

Read one byte from the Telnet connection

tnet_local

Disable remote echo mode

tnet_oob_read

Read urgent data from the Telnet connection

tnet_oob_status

Check for urgent data

tnet_oob_write

Write urgent data to the Telnet connection

tnet_open

Establish a Telnet connection

tnet_option_status

Get the current state of a Telnet option

tnet_putc

Write one byte to the Telnet connection

tnet_read

Read data from the Telnet connection

tnet_remote

Set remote echo mode

tnet_status

Get number of bytes available to read

tnet_write

Write data to the Telnet connection

tnet_x_open

Establish a Telnet connection with the given settings

2 Reference

2.1 tnet_abort ()

Description

Abort the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_abort (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_abort** function will abort the Telnet connection associated with the connection handle *hTnet* by sending a TCP reset (RST) and all the buffered data will be lost. All the resources that have been allocated for the connection by **tnet_open** or **tnet_x_open** will be freed and the connection handle *hTnet* will no longer be valid. A message with the TNET_UPCALL_CLOSE and TNET_UPCALL_RESET flags set in IParam will be sent to the application window.

Return Value

The function returns TNET_SUCCESS if the connection is aborted. If the connection handle is invalid the function returns TNET_BAD_GLOBALHANDLE.

2.2 tnet_binary ()

Description

Enable or disable binary mode.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_binary (hTnet, DoBinary)
```

GLOBALHANDLE *hTnet*; Telnet connection handle

BOOL *DoBinary* Flag indicating the desired mode

Remarks

The **tnet_binary** function sends a request to the remote host to enable or disable binary mode. If the *DoBinary* flag is set to TRUE then binary mode is enabled. If it is set to FALSE then binary mode is disabled. If the connection is already in the correct mode or the library has already sent a request to go into the desired mode then this function will return success without sending a request to the remote host. The application should use the **tnet_option_status** function to determine the binary mode status before calling this function. This function should only be called to change the state. If the connection is already in the desired state then this function should not be called. Since the function does not wait for a response from the remote host, a successful return does not indicate that the remote host has agreed to change the current mode. Once the response from the remote host is received, a message with the TNET_UPCALL_STATE flag set in lParam is sent to the application window to notify it of the status change.

Return Value

The function returns 1 if the request was sent to the remote host or if the connection is already in the desired state, 0 if the request could not be sent or TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.3 tnet_close ()

Description

Close the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_close (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_close** function closes the Telnet connection associated with the connection handle *hTnet*. If there is data in the send buffer, this function will block until the data is sent or the timeout expires. If the timeout expires before the send buffer has been flushed then the connection is aborted. All the resources that have been allocated for the connection by **tnet_open** or **tnet_x_open** will be freed and the connection handle *hTnet* will no longer be valid. A message with the TNET_UPCALL_CLOSE flag set in lParam is sent to the application window. By the time the application gets this message, *hTnet* will no longer be a valid connection handle.

Return Value

The function returns TNET_SUCCESS if the connection is successfully closed. If the connection handle is invalid the function returns TNET_BAD_GLOBALHANDLE.

2.4 tnet_error ()

Description

Get the last error.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_error ()
```

Remarks

The **tnet_error** function returns the last error that occurred. This function should be used immediately after a function fails to determine the exact cause of the failure. Normally it is used after **tnet_open** or **tnet_x_open** to determine the error code, since these calls do not return an error code. The various error codes are listed below. For all other function calls, if a network error occurs, **tnet_error** will return the Windows Sockets error code.

Value	Meaning
TNET_BAD_GLOBALHANDLE	Invalid connection handle.
TNET_DSTNCT32_ERROR	Copy protection violation.
TNET_WSASTARTUP_ERROR	Error in initializing Windows Sockets.
TNET_SYSTEM_ERROR	Error in allocating system resources.
TNET_INVALID_HOSTNAME	The host name specified is not valid.
TNET_SOCKET_ERROR	Error in creating the socket.
TNET_BIND_ERROR	Error in binding to the local port.
TNET_CONNECT_ERROR	Error in connecting to the remote host.
TNET_ASYNC_ERROR	Error in enabling asynchronous notification.
TNET_MAX_CONNECTIONS	Cannot open any more connections.
TNET_OOB_ERROR	Error in handling urgent data options.

Return Value

This function returns the last error code.

2.5 tnet_fw_open ()

Description

Establish a Telnet connection via a firewall server.

```
#include <windows.h>
#include <d32-fw.h>
#include <d32-tnet.h>
```

GLOBALHANDLE WINAPI tnet_fw_open (*hWin, term_type, msg, fw*)

HWND <i>hWin</i> ;	Application window handle
LPSTR <i>term_type</i> ;	Terminal type (e.g. "vt100" or "tty")
UINT <i>msg</i> ;	Message to send to hWin for event notifications
fw_param <i>fw</i> * <i>fw</i> ;	Firewall parameters

Remarks

The **tnet_fw_open** function opens a Telnet connection with the remote host. The **tnet_fw_open** function establishes a connection with a Telnet service via a firewall server. The firewall information is passed through the **fw_param** structure which is defined as follows:

```
typedef struct fw_parameters
{
    char far * methods;                different types of methods.
    unsigned char n_methods;          the number of methods
    unsigned char add_type;           address type of the destination address
    char * domain_name;              the domain name
    char far * fw_host;               IP address of the firewall server
    char far * dest_host;             IP address of the SMTP server
    unsigned short fw_port;           firewall port
    unsigned short dest_port;         destination port
    char far * username;              valid user id
    char far * passwd;                user password
    int socks_ver;                    the socks version;
    char reserved [256 - 6 * sizeof (char far *) - 2 * sizeof (unsigned char) - 2 * sizeof (unsigned short) - sizeof (int)];
} fw_param;
```

Distinct firewall library supports both SOCKS version 4 and version 5 firewall server. The *socks_ver* field of the **fw_param** structure should specify the SOCKS version. The *socks_ver* field should be any or a combination of the following values:

FW_VERSION5	The Socks version is 5
FW_VERSION4	The Socks version is 4

If the *socks_ver* field contains both FW_VERSION5 and FW_VERSION4 then the Distinct firewall library will try to automatically determine the firewall server version and send appropriate information to the firewall server to establish a connection.

The *methods* field can point to string "0", "1", or "2" or any combination of "0", "1", "2", for example "01" or "12". Method 0 means no authentication is required, 1 means GSSAPI and 2 means a *username* and *passwd* is required. *n_methods* is the number of methods that appear in the *method* field. Note that only methods 0 and 2 are supported currently and if method 2 is specified then the application must specify a *username* and *passwd* for authentication. Note that the *methods*,

n_methods, and *passwd* fields are required only if the application is trying to connect to a version 5 firewall server.

The field *addr_type* specifies the type of address of the remote host. The *addr_type* can be any one of the following types.

Type	Meaning
FW_ADDR_IP4	address is a version 4 IP address
FW_ADDR_DNS	address is a DNS style domain name
FW_ADDR_IP6	address is a version 6 IP address

If the *addr_type* field is of type FW_ADDR_DNS then the *domain_name* field must point to a DNS-style domain name. If the value of the *addr_type* field is FW_ADDR_IP4 or FW_ADDR_IP6 then the *dest_host* field must contain the IP address of the Telnet server. Note that if the SOCKS version is 4 then the *addr_type* field should always be FW_ADDR_IP4 or FW_ADDR_DNS.

The *fw_host* field should contain the address of the firewall server. The port of the firewall server must be specified in the *fw_port* field, if this field is 0 then the firewall library will use the default port 1080. The field *dest_port* must contain the Telnet service port (generally 23). Both *fw_port* and the *dest_port* must be in host byte order.

The *username* should contain a valid user id if the application is trying to connect to a SOCKS version 4 firewall server or if the application has specified method 2 for authentication in case of SOCKS version 5.

Note that any string pointer of the *fw_param* structure that is not used should be set to NULL.

The function allocates all the resources for the new connection and returns the connection handle if the connection attempt succeeds.

The *hWin* parameter must be a valid application window handle that can receive messages. Whenever new data arrives or the application needs to be notified of some event, the message specified by *msg* is sent to the window identified by *hWin*. The *wParam* parameter of the message will be set to the connection handle associated with this connection. The *lParam* parameter can have the following flags set to indicate various events.

Value	Meaning
TNET_UPCALL_CLOSE	Connection was closed.
TNET_UPCALL_RESET	Connection was aborted. If this flag is set, TNET_UPCALL_CLOSE is also set.
TNET_UPCALL_OOB	Urgent data in the data stream. This message will be posted only if urgent data is being received out of band.
TNET_UPCALL_STATE	Connection state has changed. In this case HIWORD of <i>lParam</i> will contain the option whose state has changed. The application should use tnet_option_status to find out the actual state of the option.

If no flag is set then the message indicates incoming data. After receiving a message indicating that the connection was closed, the application still has to call **tnet_close** to free resources allocated for the Telnet connection. The *term_type* parameter specifies the terminal type to be used for the terminal type option negotiation. This should be a NULL terminated string and should not be more than 40 bytes long, including the terminating NULL character. The remote host is allowed to initiate terminal type option negotiations. In addition, it is allowed to enable remote echoing and suppress go ahead signals. All other options are disabled.

Return Value

The function returns a GLOBALHANDLE that is unique for every connection if successful. A return value of NULL indicates that either the connection could not be established or an invalid argument was specified. In case of an error, the **tnet_error** function can be used to retrieve the actual error code.

2.6 tnet_fw_x_open ()

Description

Establish a Telnet connection with the given settings via a firewall server.

```
#include <windows.h>
#include <d32-fw.h>
#include <d32-tnet.h>
```

GLOBALHANDLE WINAPI tnet_fw_x_open (*hWin, term_type, msg, timeout, BufferSize, UrgentData, tnetproc, fw*)

HWND <i>hWin</i> ;	Application window handle
LPSTR <i>term_type</i> ;	Terminal type (e.g. "vt100" or "tty")
UINT <i>msg</i> ;	Message to send to hWin for event notifications
int <i>timeout</i> ;	Timeout in seconds
int <i>BufferSize</i> ;	Buffer size in bytes for option negotiation
int <i>UrgentData</i> ;	Flag indicating how the urgent data is to be treated
TNETPROC <i>tnetproc</i> ;	Callback function for option negotiation
fw_param <i>fw</i> ;	Firewall parameters

Remarks

The **tnet_fw_x_open** function opens a Telnet connection on the remote host. The **tnet_fw_x_open** function establishes a connection with a Telnet service via a firewall server. The firewall information is passed through the **fw_param** structure which is defined as follows:

```
typedef struct fw_parameters
{
    char far * methods;           different types of methods.
    unsigned char n_methods;      the number of methods
    unsigned char add_type;       address type of the destination address
    char * domain_name;          the domain name
    char far * fw_host;           IP address of the firewall server
    char far * dest_host;         IP address of the SMTP server
    unsigned short fw_port;       firewall port
    unsigned short dest_port;     destination port
    char far * username;          valid user id
    char far * passwd;           user password
    int socks_ver;                the socks version;
    char reserved [256 - 6 * sizeof(char far *) - 2 * sizeof(unsigned
    short) - sizeof(int)];
} fw_param;
```

Distinct firewall library supports both SOCKS version 4 and version 5 firewall server. The *socks_ver* field of the **fw_param** structure should specify the SOCKS version. The *socks_ver* field should be any or a combination of the following values:

FW_VERSION5	The Socks version is 5
FW_VERSION4	The Socks version is 4

If the *socks_ver* field contains both FW_VERSION5 and FW_VERSION4 then the Distinct firewall library will try to automatically determine the firewall server version and send appropriate information to the firewall server to establish a connection.

The *methods* field can point to string "0", "1", or "2" or any combination of "0", "1", "2", for example "01" or "12". Method 0 means no authentication is required, 1 means GSSAPI and 2 means a *username* and *passwd* is required. *n_methods* is the number of methods that appear in the *method* field. Note that only methods 0 and 2 are supported currently and if method 2 is specified then the application must specify a *username* and *passwd* for authentication. Note that the *methods*, *n_methods*, and *passwd* fields are required only if the application is trying to connect to a version 5 firewall server.

The field *add_type* specifies the type of address of the remote host. The *add_type* can be any one of the following types.

Type	Meaning
FW_ADDR_IP4	address is a version 4 IP address
FW_ADDR_DNS	address is a DNS style domain name
FW_ADDR_IP6	address is a version 6 IP address

If the *addr_type* field is of type FW_ADDR_DNS then the *domain_name* field must point to a DNS-style domain name. If the value of the *addr_type* field is FW_ADDR_IP4 or FW_ADDR_IP6 then the *dest_host* field must contain the IP address of the Telnet server. Note that if the SOCKS version is 4 then the *add_type* field should always be FW_ADDR_IP4 or FW_ADDR_DNS.

The *fw_host* field should contain the address of the firewall server. The port of the firewall server must be specified in the *fw_port* field, if this field is 0 then the firewall library will use the default port 1080. The field *dest_port* must be the Telnet service port (generally 23). Both *fw_port* and the *dest_port* must be in host byte order.

The *username* should contain a valid user id if the application is trying to connect to a SOCKS version 4 firewall server or if the application has specified method 2 for authentication in case of SOCKS version 5.

Note that any string pointer of the *fw_param* structure that is not used should be set to NULL.

The function allocates all the resources for the new connection and returns the connection handle if the connection attempt succeeds.

The *hWin* parameter must be a valid application window handle that can receive messages. Whenever new data arrives or the application needs to be notified of some event, the message specified by *msg* is sent to the window identified by *hWin*. The *wParam* parameter of the message will be set to the connection handle associated with this connection. The *lParam* parameter can have the following flags set to indicate various events.

Value	Meaning
TNET_UPCALL_CLOSE	Connection was closed.
TNET_UPCALL_RESET	Connection was aborted. If this flag is set, TNET_UPCALL_CLOSE is also set.
TNET_UPCALL_OOB	Urgent data in the data stream. This message will be posted only if urgent data is being received out of band.
TNET_UPCALL_STATE	Connection state has changed. In this case HIWORD of <i>lParam</i> will contain the option whose state has changed. The application should use tnet_option_status to find out the actual state of the option.

If no flag is set then the message indicates incoming data. After receiving an upcall indicating that the connection was closed, the application still has to call **tnet_close** to free resources allocated for the Telnet connection.

The *term_type* parameter specifies the terminal type to be used for the terminal type option negotiation. This should be a NULL terminated string and should not be more than 40 bytes long including the terminating NULL character. The remote host is allowed to initiate terminal type option negotiations. In addition it is allowed to enable remote echoing and suppress go ahead signals. All other options are disabled. If the application wants to handle some of the option negotiations, it should specify the callback function to be used for option negotiations in *tnetproc*. The callback function is called every time data for option negotiations is received. The callback function should be defined as follows.

```
int CALLBACK TnetProc (hTnet, buf, len)

GLOBALHANDLE hTnet;          /* Telnet connection handle */
char far * buf;              /* Option negotiation data */
int len;                     /* Length of the data in bytes */
```

If the callback function handles a particular option then it should return TRUE. It should return FALSE if it does not handle a specific option, so that it can be handled internally. If *tnetproc* is set to NULL then no callback function is used.

The *timeout* parameter specifies the timeout in seconds to be used for various network operations. If the *timeout* is 0, the default timeout value defined in the registry is used. *BufferSize* specifies the buffer size in bytes used for option negotiations. The minimum value allowed is 128 bytes and the maximum value allowed is 8192 bytes. If the *BufferSize* is 0, the default value defined in the registry is used.

The *UrgentData* flag indicates how urgent data will be handled. The application should set it to `TNET_URGENT_INLINE` if urgent data is to be received as a part of the regular data stream. The application will not receive any separate notification for urgent data and it can use the **`tnet_oob_status`** function to check for urgent data. The application should set *UrgentData* to `TNET_URGENT_OOB` if urgent data is to be received Out Of Band (OOB). In this case the application will receive the `TNET_UPCALL_OOB` message to indicate the arrival of urgent data and the **`tnet_oob_read`** function must be used to read the urgent data. In most cases the `TNET_URGENT_INLINE` option should be sufficient. `TNET_URGENT_OOB` should only be used if the application needs to treat urgent data as OOB data.

Return Value

The function returns a `GLOBALHANDLE` that is unique for every connection if successful. A return value of `NULL` indicates that either the connection could not be established or an invalid argument was specified. In case of an error the **`tnet_error`** function can be used to find the actual error code.

2.7 tnet_getc ()

Description

Read one byte from the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_getc (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_getc** function reads one character from the connection associated with the connection handle *hTnet*, if any data is available. The function filters all the data corresponding to Telnet option negotiations and only returns a character from the normal data stream. The character is contained in the lower eight bits of the integer returned. This function should be called only after receiving a message which indicates that some data has arrived for this connection. A message indicating incoming data does not guarantee that this function will actually return a character since the incoming data could contain only Telnet option negotiation data. If an option negotiation callback function has been specified then it may be called from within this function. This function cannot be used if the incoming data contains **IAC** (-1) as a part of the regular data stream. In general it is not a good policy to read one character at a time. Instead, **tnet_read** should be used to read the entire incoming data at one time. After a message indicating incoming data has been posted, no more messages for incoming data will be posted until the application reads some data. If the function fails with the error code **TNET_BAD_GLOBALHANDLE** then the application may not receive any more messages notifying it of incoming data.

Return Value

The function returns the incoming character. If there is no data or the connection handle is invalid, the function returns **TNET_BAD_GLOBALHANDLE**.

2.8 tnet_local ()

Description

Disable remote echo mode.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_local (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_local** function sends a request to the remote host to discontinue remote echo. After the remote host stops remote echo, the application is responsible for local echo. If the connection already has remote echoing disabled or has sent a request to disable remote echoing, this function will return success without sending a request to the remote side. The application should use the **tnet_option_status** function to determine the status of the echo mode before calling this function. This function should be called to change the connection echo mode only if the CURRENT_DO and WANT_DO flags are set in the option status returned by **tnet_option_status** function, otherwise it means that the connection is already in the desired state. Since the function does not wait for a response from the remote host, a successful return does not indicate that the host has agreed to stop remote echo. When the response from the remote host is received, a message with the TNET_UPCALL_STATE flag set in lParam is sent to the application window.

Return Value

The function returns 1 if the request was sent to the remote host or if the remote echo mode has already been disabled, 0 if the request could not be sent and TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.9 tnet_oob_read ()

Description

Read urgent data from the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_oob_read (hTnet, buf, maxlen)
```

GLOBALHANDLE *hTnet*; Telnet connection handle

char far **buf*; Data buffer

int *maxlen*; Maximum number of bytes to read

Remarks

The **tnet_oob_read** function reads up to *maxlen* bytes of urgent data from the connection specified by the connection handle *hTnet* into the user supplied buffer *buf*. This function should be used to read the urgent data only if the urgent data is being received Out Of Band (OOB). The function filters all the data corresponding to Telnet option negotiations and only returns OOB data from the normal data stream. This function should be called only after receiving a message with the TNET_UPCALL_OOB flag set. If an option negotiation callback function has been specified, it may be called from within this function.

Return Value

The function returns the number of characters read (which may be zero) or TNET_BAD_GLOBALHANDLE to indicate an error.

2.10 tnet_oob_status ()

Description

Check for urgent data.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_oob_status (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_oob_status** function checks the incoming data stream for urgent data. It returns TRUE if there is urgent data in the data stream, FALSE if there is no urgent data. This option should be used only if urgent data is being received inline as a part of the normal data stream. The urgent data will be read as a part of the normal data stream using **tnet_read** or **tnet_getc**.

Return Value

The function returns TRUE if there is urgent data, FALSE otherwise.

2.11 tnet_oob_write ()

Description

Write urgent data to the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_oob_write (hTnet, buf, len)
```

GLOBALHANDLE *hTnet*; Telnet connection handle

char far **buf*; Data buffer

int *len*; Number of bytes to send

Remarks

The **tnet_oob_write** function sends *len* bytes of urgent data to the remote host over the connection associated with the connection handle *hTnet*. If the send buffer is full, this function will block until space becomes available in the send buffer or the timeout expires. The number of bytes sent may be less than the number of bytes specified by *len*.

Return Value

The function returns the number of bytes sent if successful or **TNET_BAD_GLOBALHANDLE** to indicate that an invalid connection handle was specified.

2.12 tnet_open ()

Description

Establish a Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

GLOBALHANDLE WINAPI tnet_open (*hWin, host, port, term_type, msg*)

HWND <i>hWin</i> ;	Application window handle
LPSTR <i>host</i> ;	Remote host name or IP address
UINT <i>port</i> ;	Port on the remote host (0 to use the default)
LPSTR <i>term_type</i> ;	Terminal type (e.g. "vt100" or "tty")
UINT <i>msg</i> ;	Message to send to hWin for event notifications

Remarks

The **tnet_open** function opens a Telnet connection with the remote host. The *host* parameter can either specify the name of a host or its internet address (in the format "a.b.c.d"). The *port* parameter can specify the port on the remote host to connect to in host byte order. If the *port* specified is 0, the port number in the services file for the "telnet" service is used. If there is an error in reading the services file, the default value of 23 is used. The function allocates all the resources for the new connection and returns the connection handle if the connection attempt succeeds.

The *hWin* parameter must be a valid application window handle that can receive messages. Whenever new data arrives or the application needs to be notified of some event, the message specified by *msg* is sent to the window identified by *hWin*. The wParam parameter of the message will be set to the connection handle associated with this connection. The lParam parameter can have the following flags set to indicate various events.

Value	Meaning
TNET_UPCALL_CLOSE	Connection was closed.
TNET_UPCALL_RESET	Connection was aborted. If this flag is set, TNET_UPCALL_CLOSE is also set.
TNET_UPCALL_OOB	Urgent data in the data stream. This message will be posted only if urgent data is being received out of band.
TNET_UPCALL_STATE	Connection state has changed. In this case HIWORD of lParam will contain the option whose state has changed. The application should use tnet_option_status to find out the actual state of the option.

If no flag is set then the message indicates incoming data. After receiving a message indicating that the connection was closed, the application still has to call **tnet_close** to free resources allocated for the Telnet connection. The *term_type* parameter specifies the terminal type to be used for the terminal type option negotiation. This should be a NULL terminated string and should not be more than 40 bytes long, including the terminating NULL character. The remote host is allowed to initiate terminal type option negotiations. In addition, it is allowed to enable remote echoing and suppress go ahead signals. All other options are disabled.

Return Value

The function returns a GLOBALHANDLE that is unique for every connection if successful. A return value of NULL indicates that either the connection could not be established or an invalid argument was specified. In case of an error, the **tnet_error** function can be used to retrieve the actual error code.

2.13 tnet_option_status ()

Description

Get the current state of a Telnet option.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_option_status (hTnet, option)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

```
    int option;            Telnet option
```

Remarks

The **tnet_option_status** function is used to get the current state of a Telnet option. The *option* parameter specifies the Telnet option whose state is to be determined (such as Binary, Echo, etc.). Please check the D32-TNET.H header file for the various options. The option state is returned as an integer with various flags indicating the different states.

Value	Meaning
CURRENT_WILL	The option is enabled locally.
CURRENT_DO	The option is enabled on the remote host.
WANT_WILL	Trying to enable the option locally.
WANT_DO	Trying to get the remote host to enable this option.
ALLOW_WILL	Allow the remote side to set this option locally.
ALLOW_DO	Allow the remote host to enable this option.

The **d32-tnet.h** header file contains macros that can be used to determine the individual option states.

Return Value

The function returns the option state or TNET_BAD_GLOBALHANDLE to indicate an invalid parameter.

2.14 tnet_putc ()

Description

Write one byte to the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_putc (hTnet, c)
```

GLOBALHANDLE <i>hTnet</i> ;	Telnet connection handle
char <i>c</i> ;	Character to send

Remarks

The **tnet_putc** function sends a character to the remote host over the connection associated with the connection handle *hTnet*. If the send buffer is full, this function will block until space becomes available in the send buffer or the timeout expires.

Return Value

The function returns 1 if the character is successfully sent, 0 if the character could not be sent or TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.15 tnet_read ()

Description

Read data from the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_read (hTnet, buf, maxlen)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

```
    char far *buf;          Data buffer
```

```
    int maxlen;             Maximum number of bytes to read
```

Remarks

The **tnet_read** function reads up to *maxlen* bytes from the connection specified by the connection handle *hTnet* into the user supplied buffer *buf*. The function filters all the data corresponding to Telnet option negotiations and only returns characters from the normal data stream. This function should be called only after receiving a message, which indicates that some data has arrived for this connection. The maximum number of bytes available can be obtained by calling **tnet_status**. This will be useful in allocating the correct amount of space for the read buffer. The number of bytes actually read may be less than that indicated by **tnet_status**, since the incoming data may contain Telnet option negotiation data. If an option negotiation callback function has been specified, it may be called from within this function. After a message indicating incoming data has been posted, no more messages for incoming data will be posted until the application reads the data. If the function fails with the error code TNET_BAD_GLOBALHANDLE, the application may not receive any more messages notifying it of incoming data.

Return Value

The function returns the number of characters read (which may be zero) or TNET_BAD_GLOBALHANDLE to indicate an error.

2.16 tnet_remote ()

Description

Set remote echo mode.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_remote (hTnet)
```

```
    GLOBALHANDLE hTnet;    Telnet connection handle
```

Remarks

The **tnet_remote** function sends a request to the remote host to start remote echo. After the remote host starts remote echo, the application is no longer responsible for local echo. If the connection already has remote echoing enabled or has sent a request to enable remote echoing, this function will return success without sending a request to the remote side. The application should use the **tnet_option_status** function to determine the status of the echo mode before calling this function. This function should be called to change the connection echo mode only if the CURRENT_DO and WANT_DO flags are not set in the option status returned by **tnet_option_status**, otherwise it means that the connection is already in the desired state. Since the function does not wait for a response from the remote host, a successful return does not indicate that the host has agreed to do remote echo. When the response from the remote host is received, a message with the TNET_UPCALL_STATE flag set in lParam is sent to the application window.

Return Value

The function returns 1 if the request was sent to the remote host or if the remote echo mode has already been enabled, 0 if the request could not be sent and TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.17 tnet_status ()

Description

Get number of bytes available to read.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_status (hTnet, seconds)
```

GLOBALHANDLE *hTnet*; Telnet connection handle

unsigned long *seconds*; Number of seconds for timeout

Remarks

The **tnet_status** function retrieves the number of bytes of data that are buffered in the receive buffer for the connection specified by the connection handle *hTnet*. If there is no data in the receive buffer, the function will block until the timeout specified by *seconds* (which may be 0) expires. The number of bytes that can be read using **tnet_read** or **tnet_getc** may be less than the value returned by **tnet_status** since the incoming data might contain Telnet option negotiation data which will be filtered out.

Return Value

The function returns the number of bytes available, 0 if the timeout expired or TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.18 tnet_write ()

Description

Write data to the Telnet connection.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

```
int WINAPI tnet_write (hTnet, buf, len)
```

GLOBALHANDLE *hTnet*; Telnet connection handle

char far **buf*; Data buffer

int *len*; Number of bytes to send

Remarks

The **tnet_write** function sends *len* bytes to the remote host over the connection associated with the connection handle *hTnet*. If the send buffer is full, this function will block until space becomes available in the send buffer or the timeout expires. The number of bytes sent may be less than the number of bytes specified by *len*.

Return Value

The function returns the number of bytes sent if successful or TNET_BAD_GLOBALHANDLE to indicate that an invalid connection handle was specified.

2.19 tnet_x_open ()

Description

Establish a Telnet connection with the given settings.

```
#include <windows.h>
```

```
#include <d32-tnet.h>
```

GLOBALHANDLE WINAPI tnet_x_open (*hWin, host, port, term_type, msg, timeout, BufferSize, UrgentData, tnetproc*)

HWND <i>hWin</i> ;	Application window handle
LPSTR <i>host</i> ;	Remote host name or IP address
UINT <i>port</i> ;	Port on the remote host (0 to use the default)
LPSTR <i>term_type</i> ;	Terminal type (e.g. "vt100" or "tty")
UINT <i>msg</i> ;	Message to send to hWin for event notifications
int <i>timeout</i>	Timeout in seconds
int <i>BufferSize</i>	Buffer size in bytes for option negotiation
int <i>UrgentData</i>	Flag indicating how the urgent data is to be treated
TNETPROC <i>tnetproc</i>	Callback function for option negotiation

Remarks

The **tnet_x_open** function opens a Telnet connection on the remote host. The *host* parameter can either specify the name of a host or its internet address (in the format "a.b.c.d"). The *port* parameter can specify the port on the remote host to connect to in host byte order. If the *port* specified is 0, the port number in the services file for the "telnet" service is used. If there is an error in reading the services file, the default value of 23 is used. The function allocates all the resources for the new connection and returns the connection handle if the connection attempt succeeds.

The *hWin* parameter must be a valid application window handle that can receive messages. Whenever new data arrives or the application needs to be notified of some event, the message specified by *msg* is sent to the window identified by *hWin*. The wParam parameter of the message will be set to the connection handle associated with this connection. The lParam parameter can have the following flags set to indicate various events.

Value	Meaning
TNET_UPCALL_CLOSE	Connection was closed.
TNET_UPCALL_RESET	Connection was aborted. If this flag is set, TNET_UPCALL_CLOSE is also set.
TNET_UPCALL_OOB	Urgent data in the data stream. This message will be posted only if urgent data is being received out of band.
TNET_UPCALL_STATE	Connection state has changed. In this case HIWORD of lParam will contain the option whose state has changed. The application should use tnet_option_status to find out the actual state of the option.

If no flag is set then the message indicates incoming data. After receiving an upcall indicating that the connection was closed, the application still has to call **tnet_close** to free resources allocated for the Telnet connection.

The *term_type* parameter specifies the terminal type to be used for the terminal type option negotiation. This should be a NULL terminated string and should not be more than 40 bytes long including the terminating NULL character. The remote host is allowed to initiate terminal type option negotiations. In addition it is allowed to enable remote echoing and suppress go ahead

signals. All other options are disabled. If the application wants to handle some of the option negotiations, it should specify the callback function to be used for option negotiations in *tnetproc*. The callback function is called every time data for option negotiations is received. The callback function should be defined as follows.

```
int CALLBACK TnetProc (hTnet, buf, len)

GLOBALHANDLE hTnet;          /* Telnet connection handle */
char far * buf;              /* Option negotiation data */
int len;                     /* Length of the data in bytes */
```

If the callback function handles a particular option then it should return TRUE. It should return FALSE if it does not handle a specific option, so that it can be handled internally. If *tnetproc* is set to NULL then no callback function is used.

The *timeout* parameter specifies the timeout in seconds to be used for various network operations. If the *timeout* is 0, the default timeout value defined in the registry is used. *BufferSize* specifies the buffer size in bytes used for option negotiations. The minimum value allowed is 128 bytes and the maximum value allowed is 8192 bytes. If the *BufferSize* is 0, the default value defined in the registry is used.

The *UrgentData* flag indicates how urgent data will be handled. The application should set it to TNET_URGENT_INLINE if urgent data is to be received as a part of the regular data stream. The application will not receive any separate notification for urgent data and it can use the **tnet_oob_status** function to check for urgent data. The application should set *UrgentData* to TNET_URGENT_OOB if urgent data is to be received Out Of Band (OOB). In this case the application will receive the TNET_UPCALL_OOB message to indicate the arrival of urgent data and the **tnet_oob_read** function must be used to read the urgent data. In most cases the TNET_URGENT_INLINE option should be sufficient. TNET_URGENT_OOB should only be used if the application needs to treat urgent data as OOB data.

Return Value

The function returns a GLOBALHANDLE that is unique for every connection if successful. A return value of NULL indicates that either the connection could not be established or an invalid argument was specified. In case of an error the **tnet_error** function can be used to find the actual error code.

