

TCP/IP VxD Interface

The interface to the Microsoft® TCP/IP-32 VxD is a variant of the Transport Device Interface (TDI) defined for Microsoft® Windows NT®. The primary difference between the two interfaces is that under Windows NT, requests are made using IRPs, while in the VxD environment requests are made by pushing parameters on the stack and calling a function indirectly through a pointer. The primitives in the interface are substantially the same.

This documentation summarizes the differences, though minor, between the two environments and describes the VxD calling conventions.

This documentation also assumes that the reader has a basic familiarity with VxD programming issues such as VxD services and the use of the VMM. For information on these topics, refer to the Windows 3.1 DDK documentation.

The following topics are provided:

[Structures](#)

[Binding with the MS TCP/IP-32 VxD](#)

[Communication Using the TDI interface](#)

[Descriptions of TDI Functions](#)

Structures

There are several structures used on the interface. Some of the more important ones are described in this documentation:

[TDI_REQUEST](#)

[NDIS_BUFFER](#)

[TRANSPORT_ADDRESS](#)

[TA_ADDRESS](#)

[TDI_ADDRESS_IP](#)

[TDI_CONNECTION_INFORMATION](#)

[EventRcvBuffer](#)

[ConnectEventInfo](#)

[TdiDispatchTable](#)

The following typedefs are used:

```
typedef unsigned long ulong;
typedef unsigned short ushort;
typedef unsigned char uchar;
typedef unsigned int uint;
typedef void (*CTEReqCmplItRtn)(void *Context, TDI_STATUS FinalStatus,
uint ByteCount);
```

TDI_REQUEST

The **TDI_REQUEST** structure is defined in the file TDI.H.

```
typedef struct _TDI_REQUEST {
    union {
        HANDLE AddressHandle;
        CONNECTION_CONTEXT ConnectionContext;
        HANDLE ControlChannel;
    } Handle;
    PVOID RequestNotifyObject;
    PVOID RequestContext;
    TDI_STATUS TdiStatus;
} TDI_REQUEST, *PTDI_REQUEST;
```

Comments

A pointer to a **TDI_REQUEST** structure is passed into every VxD TDI call. The VxD TDI client must fill in the structure before invoking the VxD. The handle field will contain either an address object handle or a connection context, depending on the call. The *RequestNotifyObject* field is set to be a pointer to a client-supplied callback function that TCP will call when the request finally completes, if and only if the request returned **TDI_PENDING** when it was first invoked. Note that a callback can happen before the call to the function returns. The *RequestContext* field will be passed back unchanged as a parameter to the callback function when it is called. The *TdiStatus* field is unused.

Following is the prototype for the callback function:

```
void Callback(
    PVOID Context,
    TDI_STATUS FinalStatus,
    unsigned long ByteCount
);
```

Parameters

Context

RequestContext originally specified by the client in the **TDI_REQUEST**.

FinalStatus

Completion status of the command.

ByteCount

Number of bytes sent or received. This final parameter is unused for some commands. The type `CTEReqCmpltRtn` is defined as a pointer to a callback routine.

Comments

TDI_REQUEST structures are ephemeral. The VxD will copy any needed information out of them before the initial invocation completes, and the client may reuse them immediately

NDIS_BUFFER

The **NDIS_BUFFER** structure is defined in `NDIS.H`. In Windows NT, **NDIS_BUFFER** structures are MDLs, but in the VxD environment, they are essentially buffer descriptors consisting of a length, pointer, and a next field. Chains of **NDIS_BUFFER** structures may be used to pass in scatter/gather lists of buffers for sending and receiving. **NDIS_BUFFER** chains must be NULL terminated. Both the **NDIS_BUFFER** structure itself and the buffer pointed to by the **NDIS_BUFFER** structure must be locked down and validated before being passed to the transport. VxD TDI clients may use the macros and functions for managing buffers provided by the NDIS wrapper (`ndis.386`) and described in the NDIS 3.0 specification, or they may create and initialize **NDIS_BUFFER** structures themselves. **NDIS_BUFFER** chains submitted to the transport must not be altered while the request with which they were submitted is outstanding.

TRANSPORT_ADDRESS

The **TRANSPORT_ADDRESS** structure is used for describing transport addresses passed to the VxD transport. It is defined in `TDI.H`.

```
typedef struct _TRANSPORT_ADDRESS {  
    LONG TAddressCount;  
    TA_ADDRESS Address[1];  
} TRANSPORT_ADDRESS, *PTRANSPORT_ADDRESS;
```

Members

TAddressCount

Number of addresses available.

Address[1]

Actual number of **TAddressCount** elements.

Comments

A **TRANSPORT_ADDRESS** is a counted array of `TA_ADDRESSES`. Each [TA_ADDRESS](#) describes one address.

TA_ADDRESS

The **TA_ADDRESS** structure is used for describing transport addresses passed to the VxD transport. It is defined in TDI.H.

```
typedef struct _TA_ADDRESS {
    USHORT AddressLength;    // length in bytes of Address[] in this
    USHORT AddressType;      // type of this address
    UCHAR  Address[1];       // actually AddressLength bytes long
} TA_ADDRESS, *PTA_ADDRESS;
```

Members

AddressLength

Length in bytes of **Address[1]**.

AddressType

Type of this address.

Address[1]

Length in bytes of **AddressLength**.

Comments

For the TCP VxD, the **AddressType** field should be **TDI_ADDRESS_IP**, which is defined in TDI.H.

```
#define TDI_ADDRESS_TYPE_IP          ((USHORT)2) // internetwork: UDP, TCP,
etc.
```

Then, the **AddressLength** field of the **TA_ADDRESS** should be set to **TDI_ADDRESS_LENGTH_IP**, and the **Address** field is a structure of type [**TDI_ADDRESS_IP**](#).

TDI_ADDRESS_IP

The **TDI_ADDRESS_IP** structure is defined in TDI.H.

```
typedef struct _TDI_ADDRESS_IP {
    USHORT sin_port;
    ULONG  in_addr;
    UCHAR  sin_zero[8];
} TDI_ADDRESS_IP, *PTDI_ADDRESS_IP;

#define TDI_ADDRESS_LENGTH_IP sizeof (TDI_ADDRESS_IP)
```

Comments

The **TDI_ADDRESS_IP** structure is similar to the **sockaddr** structure used in Windows Sockets.

TDI_CONNECTION_INFORMATION

A **TDI_CONNECTION_INFORMATION** structure is used for passing in and out information used for establishing connections. They are not copied by the transport and must be preserved intact until the request with which they are issued completes. This structure is defined in TDI.H.

```
typedef struct _TDI_CONNECTION_INFORMATION {
    LONG UserDataLength;
    PVOID UserData;
    LONG OptionsLength;
    PVOID Options;
    LONG RemoteAddressLength;
    PVOID RemoteAddress;
} TDI_CONNECTION_INFORMATION, *PTDI_CONNECTION_INFORMATION;
```

Members

UserDataLength

Reserved. Do not use.

UserData

Reserved. Do not use.

OptionsLength

Length of the buffer containing IP options. If there are no options, this must be set to zero.

Options

Pointer to a buffer containing IP options. If there are no options, this must be set to NULL.

RemoteAddressLength

Length of the **TRANSPORT_ADDRESS** structure.

RemoteAddress

Pointer to the **TRANSPORT_ADDRESS** structure.

EventRcvBuffer

An **EventRcvBuffer** is a structure used to pass information about buffers and completion routines for data delivered through a receive indication. A pointer to an **EventRcvBuffer** (ERB) structure is passed to the client as a parameter to the indication routine. The client may elect to fill in the ERB with the appropriate information and return **TDI_MORE_PROCESSING**. In this case, the transport will examine the ERB and take the appropriate actions. The **EventRcvBuffer** structure is defined in TDIVXD.H.

```
typedef struct EventRcvBuffer {
    PNDIS_BUFFER      erb_buffer;
    uint              erb_size;
    CTEReqCmpltRtn    erb_rtn;
    PVOID             erb_context;
    ushort            *erb_flags;
} EventRcvBuffer;
```

Members

erb_buffer

Pointer to an **NDIS_BUFFER** chain describing the buffers that the received data will be placed in.

erb_size

Maximum amount of data that may be copied into the buffer chain. The total of the sizes of all the **NDIS_BUFFER** structures in the chain must be greater than or equal to **erb_size**. If the total is greater than **erb_size**, only **erb_size** bytes will be copied. The **NDIS_BUFFER** chain must remain intact until the request completes.

erb_rtn

Pointer to a callback routine that will be called when the request is complete. The prototype for the routine is the same as for the **Callback** routine previously described.

erb_context

The context passed to the callback routine.

erb_flags

Pointer to a short of flags. This will be altered by the transport when the request completes to provide the client with the output flags. The flags pointer must not be NULL.

For this release, **erb_flags** must point at a 16-bit value of zero. When the request completes, the flags will be set to (TDI_RECEIVE_NORMAL | TDI_RECEIVE_ENTIRE_MESSAGE) for normal data, and (TDI_RECEIVE_EXPEDITED | TDI_RECEIVE_ENTIRE_MESSAGE) for urgent data.

ConnectEventInfo

The **ConnectEventInfo** structure is a structure passed by the transport to the client during a connect event indication.

```
typedef struct ConnectEventInfo {
    CTEReqCmplRtn      cei_rtn;
    PVOID              cei_context;
    PTDI_CONNECTION_INFORMATION cei_acceptinfo;
    PTDI_CONNECTION_INFORMATION cei_conninfo;
} ConnectEventInfo;
```

Members

cei_rtn

Pointer to a callback routine. The routine will be called when the connect completes.

cei_context

Pointers to a callback context for that routine.

cei_acceptinfo

Pointer providing information for use in accepting the connection. The only fields used in this structure are the **OptionsLength** and **Options** fields. This field may be NULL.

cei_conninfo

Points to information regarding the remote peer when the connection is complete. This field may be NULL.

TdiDispatchTable

The **TdiDispatchTable** is a structure that the client gets from the VTDI VxD. This structure is a table of entry points into the TCP VxD. An entry point may be called by calling indirectly through the corresponding function pointer in the **TdiDispatchTable**. The table is defined in TDIVXD.H.

```
struct TDIDispatchTable {
    TDI_STATUS      (*TdiOpenAddressEntry)(PTDI_REQUEST, PTRANSPORT_ADDRESS,
    uint,
    PVOID);
    TDI_STATUS      (*TdiCloseAddressEntry)(PTDI_REQUEST);
    TDI_STATUS      (*TdiOpenConnectionEntry)(PTDI_REQUEST, PVOID);
    TDI_STATUS      (*TdiCloseConnectionEntry)(PTDI_REQUEST);
    TDI_STATUS      (*TdiAssociateAddressEntry)(PTDI_REQUEST, HANDLE);
    TDI_STATUS      (*TdiDisAssociateAddressEntry)(PTDI_REQUEST);
    TDI_STATUS      (*TdiConnectEntry)(PTDI_REQUEST, PVOID,
    PTDI_CONNECTION_INFORMATION,
    PTDI_CONNECTION_INFORMATION);
    TDI_STATUS      (*TdiDisconnectEntry)(PTDI_REQUEST, PVOID, ushort,
    PTDI_CONNECTION_INFORMATION,
    PTDI_CONNECTION_INFORMATION);
    TDI_STATUS      (*TdiListenEntry)(PTDI_REQUEST, ushort,
    PTDI_CONNECTION_INFORMATION,
    PTDI_CONNECTION_INFORMATION);
    TDI_STATUS      (*TdiAcceptEntry)(PTDI_REQUEST,
    PTDI_CONNECTION_INFORMATION,
    PTDI_CONNECTION_INFORMATION);
    TDI_STATUS      (*TdiReceiveEntry)(PTDI_REQUEST, ushort *, uint *,
    PNDIS_BUFFER);
    TDI_STATUS      (*TdiSendEntry)(PTDI_REQUEST, ushort, uint,
    PNDIS_BUFFER);
    TDI_STATUS      (*TdiSendDatagramEntry)(PTDI_REQUEST,
    PTDI_CONNECTION_INFORMATION,
    uint, uint *, PNDIS_BUFFER);
    TDI_STATUS      (*TdiReceiveDatagramEntry)(PTDI_REQUEST,
    PTDI_CONNECTION_INFORMATION,
    PTDI_CONNECTION_INFORMATION, uint, uint *, PNDIS_BUFFER);
    TDI_STATUS      (*TdiSetEventEntry)(PVOID, int, PVOID, PVOID);
    TDI_STATUS      (*TdiQueryInformationEntry)(PTDI_REQUEST, uint,
    PNDIS_BUFFER, uint *, uint);
    TDI_STATUS      (*TdiSetInformationEntry)(PTDI_REQUEST, uint,
    PNDIS_BUFFER, uint, uint);
    TDI_STATUS      (*TdiActionEntry)(PTDI_REQUEST, uint,
    PNDIS_BUFFER, uint);
    TDI_STATUS      (*TdiQueryInformationExEntry)(PTDI_REQUEST,
    struct TDIObjectID *, PNDIS_BUFFER, uint *, void *);
    TDI_STATUS      (*TdiSetInformationExEntry)(PTDI_REQUEST,
    struct TDIObjectID *, void *, uint);
};
typedef struct TDIDispatchTable TDIDispatchTable;
```

Binding with the MS TCP/IP-32 VxD

In order to communicate with the TCP/IP VxD, a client must first get a pointer to the **TdiDispatchTable** for the VxD. This is done by communicating with the VTDI VxD. This VxD provides a service (VTDI_Get_Info) that takes in a protocol name and returns a pointer to a **TdiDispatchTable**. VTDI is device ID 0x488. A client VxD should call this service during Device_Init time. In order to initialize properly, a client VxD should have an initialization order of at least 0xc000 + VNETBIOS_Init_Order.

VTDI_Get_Info takes the name of the protocol VxD as a parameter on the stack (as a case-sensitive NULL-terminated ASCII string), and returns the pointer to the **TdiDispatchTable** in EAX. EAX will be zero if the named protocol does not exist. The name of the MS TCP/IP-32 VxD is MSTCP.

The following assembly code illustrates how to get the dispatch table:

```
VTDI_Device_ID    equ    0488h
include    vtdi.inc
VxD_IDATA_SEG
TCPName    db    'MSTCP', 0
VxD_IDATA_ENDS
VxD_ICODE_SEG
BeginProc GetTCPDispatchTable

; Make sure VTDI is present
VxDcall    VTDI_Get_Version
jc short Failure

; VTDI is present. Get a pointer to the TCP dispatch table
push    OFFSET32 TCPName
VxDcall    VTDI_Get_Info
add     esp, 4

; EAX contains a pointer to the DispatchTable, or 0 if TCP isn't available.
ret

Failure:
; VTDI isn't present.
sub     eax, eax
ret

EndProc GetTCPDispatchTable
VxD_ICODE_END
```

Communication Using the TDI Interface

Communications through TDI centers around two abstractions: the address object and the connection. An address object is the representation of a local address (IP address/port number/protocol ID). An address object is opened through the **TdiOpenAddress** primitive, which specifies the necessary information. The IP address specified may be zero, indicating a wildcard local address. The **TdiOpenAddress** call returns a HANDLE that the client uses to identify the address object in subsequent calls.

An address object may have one or more event handlers associated with it. An event handler is a client-specified routine that the TCP VxD will call when certain events occur. The most popular event handlers are connect event handlers, receive data and receive datagram event handlers, and disconnect event handlers. Connect event handlers are routines that the TCP VxD will call when an incoming connect request (a TCP frame with the SYN bit set with no ACK) is received. The client may inspect the remote address and accept or reject the connection. Receive data and datagram handlers are routines that are called when incoming data or datagrams are received. Disconnect event handlers are called when a TCP connection is being torn down, either gracefully through receipt of a FIN or abortively due to a RST or time-out. Clients specify event handlers through the **TdiSetEvent** primitive. More information on event handlers may be found in the Windows NT TDI documentation and in [TdiSetEvent](#).

An address object handle is all that is needed for UDP traffic. A client may send and receive through UDP without opening a connection. Datagrams are sent with **TdiSendDatagram**, and may be received with either **TdiReceiveDatagram** or through a receive datagram event handler.

A connection represents the local side of a TCP connection. Connections are opened through **TdiOpenConnection**. Initially, the connection is "blank" and unassociated with any local address. Clients associate a connection with a local address through **TdiAssociateAddress**. This routine takes as input a connection context and an address object handle, and associates the connection with the local address represented by the address object. After this is done, a connection to a remote host may be established through **TdiConnect**, or the connection may be placed into the listening state through **TdiListen**. As an alternative to **TdiListen**, clients may set a connect event handler on the associated address object. This is the recommended alternative for VxD TDI clients.

After a connection is established, data may be sent with **TdiSend**. Data may be received with either **TdiReceive** or a receive data event handler. TCP connections are torn down with **TdiDisconnect**. This primitive specifies whether the disconnection is to be graceful or abortive. A graceful disconnect results in a FIN being sent. The disconnect will complete when this FIN is successfully acknowledged. The connection will not be completely gone until this remote peer has sent a FIN and this FIN has been successfully acknowledged. Receipt of a FIN from the remote side will result in a **DisconnectWait** request being complete, or a disconnect event handler being called. If there is no disconnect event handler set or **DisconnectWait** primitive outstanding, the client will not be notified. A client may receive a graceful disconnect notification before issuing a disconnect request.

An abortive disconnect destroys the connection immediately by sending a RST. Receiving a RST will result in a client being notified through event handler or **DisconnectWait** primitives.

Connections are closed through **TdiCloseConnection**. Closing an established connection will result in a RST being sent. Closing a connection is different from disconnecting a connection. A disconnected connection is still around, associated with an address, and may be connected again through **TdiConnect**. Closing a connection results in the connection object itself being destroyed.

Descriptions of TDI Functions

This section provides a description of the available TDI functions, and notes where there are differences between the VxD version and the Windows NT version. For more detailed information on these functions, refer to the Windows NT TDI documentation.

[TdiOpenAddress](#)

[TdiCloseAddress](#)

[TdiOpenConnection](#)

[TdiCloseConnection](#)

[TdiAssociateAddress](#)

[TdiDisAssociateAddress](#)

[TdiConnect](#)

[TdiDisconnect](#)

[TdiListen](#)

[TdiAccept](#)

[TdiReceive](#)

[TdiSend](#)

[TdiSendDatagram](#)

[TdiReceiveDatagram](#)

[TdiSetEvent](#)

[TDI_EVENT_CONNECT](#)

[TDI_EVENT_DISCONNECT](#)

[TDI_EVENT_ERROR](#)

[TDI_EVENT_RECEIVE](#)

[TDI_EVENT_RECEIVE_DATAGRAM](#)

[TDI_EVENT_RECEIVE_EXPEDITED](#)

[TdiSetInformation](#)

[TdiActionInformation](#)

[TdiQueryInformationEx](#)

[TdiSetInformationEx](#)

[TdiSetInformationEx](#)

TdiOpenAddress

The **TdiOpenAddress** function is called through the **TdiOpenAddressEntry** pointer in the **TdiDispatchTable** structure. This function is called to open an address object.

```
TDI_STATUS TdiOpenAddress(PTDI_REQUEST Request,  
    TRANSPORT_ADDRESS *AddrList,  
    uint Protocol,  
    void *Ptr  
);
```

Parameters

AddrList

Points to a **TRANSPORT_ADDRESS** structure.

Protocol

Protocol number for the protocol with which the address is to be associated. Protocol is 17 (decimal) for UDP and 6 (decimal) for TCP.

Ptr

Pointer to a byte string of options for the opening of this address object. *Ptr* may be NULL. The options are:

```
#define    TDI_OPTION_EOL                0  
#define    TDI_ADDRESS_OPTION_REUSE    1  
#define    TDI_ADDRESS_OPTION_DHCP     2
```

TDI_OPTION_EOL

Terminates the list.

TDI_ADDRESS_OPTION_REUSE

Allows the address to be reused, as in the manner of the Windows Sockets REUSE_ADDR option.

TDI_ADDRESS_OPTION_DHCP

Allows the client to override the usual interpretation of 0.0.0.0 as the wildcard address.

Comments

If the client specifies this option, then 0.0.0.0 will be the actual address transmitted. If this option is specified, then the client must specify 0.0.0.0 in the **AddrList** structure. If this option is not specified and the client is opening 0.0.0.0, then the open is interpreted as a wildcard open and IP will choose an appropriate IP address as needed. The address selection in this case is done on a per request basis (for example, each time a datagram send is submitted or a connection using the address object is established).

TDI_ADDRESS_OPTION_DHCP is intended for use by DHCP clients. Most clients do not need to set this option.

Opening an address object with a `sin_port` of zero causes the transport to select an unused port. This port selection is done only once, at the time the address object is opened.

This function will return an address object handle in the `Request->Handle.AddressHandle` field.

TdiCloseAddress

The **TdiCloseAddress** function is called through the **TdiCloseAddressEntry** pointer in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiCloseAddress(PTDI_REQUEST Request
);
```

Comments

This function is called to close an address object. The address object to be closed is specified in the `Request->Handle.AddressHandle` field. Closing an address object abortively disconnects any active connections associated with the address object and disassociates all associated connections. Receive datagram requests are aborted. The request does not complete until all pending datagram sends are completed. However, no new datagram sends may be submitted after a **CloseAddress** request has been submitted.

TdiOpenConnection

TdiOpenConnection is called through the **TdiOpenConnectionEntry** pointer in the **TdiDispatchTable** structure. This primitive opens a connection object. The connection object is initially unassociated with an address object.

```
TDI_STATUS TdiOpenConnection(PTDI_REQUEST Request,
    PVOID Context
);
```

Parameters

Context

Context value that is passed back to the client on receive data and disconnect events. This function returns the connection handle in the `Request->Handle.ConnectionContext` field.

TdiCloseConnection

The **TdiCloseConnection** function is called through the **TdiCloseConnectionEntry** pointer in the **TdiDispatchTable** structure.

This primitive is called to close a TDI connection. The connection to be closed is specified through the Request->Handle.*ConnectionContext* field. If a TCP connection is active through the connection being closed, it will be abortively disconnected and a RST will be sent.

```
TDI_STATUS TdiCloseConnection(PTDI_REQUEST Request  
);
```

TdiAssociateAddress

The **TdiAssociateAddress** function is called through the **TdiAssociateAddressEntry** pointer in the **TdiDispatchTable** structure.

This primitive associates a connection object with an address object. An association must be done before a TCP connection can be established through a connection object. The connection to be associated is specified in the Request->Handle.*ConnectionContext* field.

```
TDI_STATUS TdiAssociateAddress(PTDI_REQUEST Request,  
    HANDLE AddrHandle  
);
```

Parameters

AddrHandle

Identifies the address object with which the connection is to be associated. A connection object may not be associated with an address object if it is already associated with another address object.

TdiDisAssociateAddress

The **TdiDisAssociateAddress** function is called through the **TdiDisAssociateAddressEntry** pointer in the **TdiDispatchTable** structure.

This is a primitive to disassociate a connection object from an address object. The connection object must not be connected to a remote peer when this primitive is called. Request->Handle.*ConnectionContext* identifies the connection to be disassociated.

TdiConnect

The **TdiConnect** function is called through the **TdiConnectEntry** pointer in the **TdiDispatchTable** structure.

This function establishes a TCP connection through the connection object identified in `Request->Handle.ConnectionContext`. The connection object must be associated with an address object before calling this function.

```
TDI_STATUS TdiConnect(PTDI_REQUEST Request,  
    void *TO,  
    PTDI_CONNECTION_INFORMATION RequestAddr,  
    PTDI_CONNECTION_INFORMATION ReturnAddr  
);
```

Parameters

TO

Points to a time-out value indicating how long to keep trying to connect before giving up. This value must be a 32-bit unsigned integer. The units are in milliseconds. *TO* may be NULL, indicating that the transport is to pick a suitable time-out. Pointing *TO* at a value of zero is the same as setting *TO* to NULL.

RequestAddr

Specifies the remote address to be connected. It must specify a valid, unicast IP address. *RequestAddr* may also specify IP options, such as source route.

ReturnAddr

Filled in by TCP with the remote address actually connected. For TCP, this will always be the same as *RequestAddr*. *ReturnAddr* also indicates which IP options (if any) are actually in use on the established connection.

TdiDisconnect

The **TdiDisconnect** function is called through the **TdiDisconnectEntry** pointer in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiDisconnect(PTDI_REQUEST Request,  
    void *TO,  
    ushort Flags,  
    PTDI_CONNECTION_INFORMATION DiscConnInfo,  
    PTDI_CONNECTION_INFORMATION ReturnInfo  
);
```

Parameters

TO

Time-out value indicating how long to wait for the disconnection to complete. It has the same semantics as the *TO* value for **TdiConnect**.

Flags

Identifies the disconnect type. It may take on one of the following values:

TDI_DISCONNECT_ABORT

Specifies that this is to be an abortive disconnect.

TDI_DISCONNECT_WAIT

Specifies that this is a disconnect wait request. This type of request results in no protocol action. The disconnect request is in a pending state until the remote peer

issues a disconnect. This command may be used to receive disconnect notifications without a disconnect event handler, although the event handler approach is the recommended mechanism.

If neither of these flags is set, the disconnect is considered to be graceful.

DiscConnInfo

Unused and may be set to NULL.

ReturnInfo

Unused and may be set to NULL.

Comments

The **TdiDisconnect** function is called through **TdiDisconnectEntry** pointer in the **TdiDispatchTable** structure.

This is the primitive to disconnect an established connection. The connection to be disconnected is specified through `Request->Handle.ConnectionContext`. A connection may be abortively or gracefully disconnected. An abortive disconnect results in a RST being sent. A graceful disconnect results in a FIN being sent, and the graceful disconnect will not complete until the FIN is successfully acknowledged, or the disconnect times out.

If a connection is being gracefully disconnected, the disconnection is not complete until both sides have disconnected. A client may be notified of the remote peers disconnect through a **DisconnectWait** request or a disconnect event notification. After a client has been notified that the remote side has disconnected and had a graceful disconnect of its own complete, the connection is gone. An abortive disconnect, however, is unilateral. Once either side has issued an abortive disconnect, the connection is terminated.

A disconnect results in all outstanding requests on the connection completing. A graceful disconnect does not complete until all outstanding send requests have been sent and acknowledged, or until the disconnect time-out occurs.

TdiListen

The **TdiListen** is called through the **TdiListenEntry** pointer in the **TdiDispatchTable** structure.

This primitive puts a connection (specified through `Request->Handle.ConnectionContext`) into the listening state. Incoming connection requests are first matched against possible connections in the listening state. If no match is found the connection request is passed to the Connect Event handler, if there is one. Most VxD clients should use the `TDI_CONNECT_EVENT` event handler instead of using this primitive.

```
TDI_STATUS TdiListen(PTDI_REQUEST Request,
    ushort Flags,
    PTDI_CONNECTION_INFORMATION AcceptableAddr,
    PTDI_CONNECTION_INFORMATION ConnectedAddr
);
```

Parameters

Flags

May be set to zero or TDI_QUERY_ACCEPT. If TDI_QUERY_ACCEPT is set, the listen will complete with success and the **ConnectedAddr** field filled in when an acceptable connection request is received, but the connection will not actually be established. The client must then call **TdiAccept** to complete the connection or **TdiDisconnect** (with TDI_DISCONNECT_ABORT set) to reject the connection. If TDI_QUERY_ACCEPT is not set, the connection is completely established when the **TdiListen** request completes successfully.

AcceptableAddr

Specifies the possible acceptable remote addresses that might connect to the listening connection. The address portion may contain a wildcard (0.0.0.0) IP address, meaning that any remote IP address with the specified port may connect. The **Options** and **OptionsLength** fields may specify IP options to be used on this connection. These options will override any specified by the remote peer.

ConnectedAddr

Filled in with the address which connected and the IP options (if any) active on this connection if (and only if) the connection completes successfully.

TdiAccept

The **TdiAccept** function is called through the **TdiAcceptEntry** pointer in the **TdiDispatchTable** structure.

This primitive accepts a connection on a connection object that previously had a **TdiListen** request with TDI_QUERY_ACCEPT completed successfully.

```
TDI_STATUS TdiAccept(PTDI_REQUEST Request,  
    PTDI_CONNECTION_INFORMATION AcceptInfo,  
    PTDI_CONNECTION_INFORMATION ConnectedInfo  
);
```

Parameters

AcceptInfo

Used to provide IP options to the transport. These options, if present, will override any specified by the remote peer. *AcceptInfo* may be set to NULL.

ConnectedInfo

Filled in by the transport when the request completes and provides the client with the remote address connected and IP options active on the connection.

TdiReceive

The **TdiuReceive** function is called through the **TdiReceiveEntry** pointer in the **TdiDispatchTable** structure.

This request receives data on a connected connection. If no data is currently available on the connection the request will pend until there is data. A receive request may specify whether normal data, urgent data, or either may be placed in the given buffer. The connection on which a receive is issued must be connected and must not have received a FIN.

```
TDI_STATUS TdiReceive(PTDI_REQUEST Request,  
    ushort *Flags,  
    uint *RcvLength,  
    PNDIS_BUFFER Buffer  
);
```

Parameters

Flags

Used both to specify the type of data which may be placed in the buffer and to return the type of data actually placed in the buffer. On input, *Flags* may be zero, TDI_RECEIVE_NORMAL, TDI_RECEIVE_EXPEDITED, or (TDI_RECEIVE_NORMAL | TDI_RECEIVE_EXPEDITED). If *Flags* is zero, MSTCP will interpret it as TDI_RECEIVE_NORMAL. *Flags* must not be set to NULL.

If *Flags* is set to TDI_RECEIVE_EXPEDITED, only urgent data will be placed in the buffer. When the request completes successfully, *Flags* will be set to either TDI_RECEIVE_NORMAL or TDI_RECEIVE_URGENT. Also, the TDI_RECEIVE_ENTIRE_MESSAGE bit is set in *Flags* on completion. The transport will not combine urgent and normal data into one buffer chain. See the Windows NT TDI documentation for more details on the flags.

RcvLength

Specifies the total length in bytes of data that may be placed in the buffer chain pointed to by *Buffer*. This length may be less than the sum of the sizes of the individual buffers in the chain, but it may not be greater than this sum. If the *RcvLength* is smaller than the actual size of the buffer chain, only *RcvLength* bytes will be placed in the buffer.

For example, if *Buffer* points to a chain of three 100-byte buffers but *RcvLength* is set to 100 on input, only the first buffer in the chain would be used. If the **TdiReceive** requests completes immediately with success, *RcvLength* will be set by the transport to the actual number of bytes received. If the request pends or fails, *RcvLength* is unused. Thus, *RcvLength* needs to be valid only for the duration of the actual call to **TdiReceive**, and not until the time the receive completes. If the receive request pends, the number of bytes received will be returned as the *ByteCount* parameter passed to the request completion callback routine.

Buffer

Points to an **NDIS_BUFFER** chain into which data will be copied. The individual buffers in this chain must be locked before submitting them.

TdiSend

The **TdiSend** function is called through the **TdiSendEntry** pointer in the **TdiDispatchTable** structure.

This primitive causes data to be sent on the connection identified by Request->Handle.ConnectionContext. The specified connection must be established and must not have been disconnected. A send request does not complete until the remote peer has successfully acknowledged the data. The number of bytes of data sent is returned as the *ByteCount* parameter to the completion callback routine. This value will always be equal to *SendLength* if the send completes successfully.

```
TDI_STATUS TdiSend(PTDI_REQUEST Request,
    ushort Flags,
    uint SendLength,
    PNDIS_BUFFER SendBuffer
);
```

Flags

May be set to zero or TDI_SEND_EXPEDITED. If TDI_SEND_EXPEDITED is set, the specified data is sent as urgent data. By default, MSTCP uses BSD style urgent data, and thus allows only one byte of urgent data to be sent. If more than one byte is submitted as urgent, only the last byte of the send is marked as such. A connection may be configured to use RFC 1122 style urgent data, in which case all data will be sent as urgent.

SendLength

Specifies the length in bytes of the data to be sent. This value *must* be equal to the sum of the sizes of the individual **NDIS_BUFFER** structures in the *SendBuffer* chain.

SendBuffer

Points to a chain of **NDIS_BUFFER** structures to be sent. All buffers in this chain must be locked before the send is submitted and the data in the buffers must not be altered until the send completes.

TdiSendDatagram

The **TdiSendDatagram** function is called through the **TdiSendDatagramEntry** pointer in the **TdiDispatchTable** structure.

This request submits a datagram send request on the address object identified by Request->Handle.AddressHandle. The local address for the datagram is derived from the specified address object. If the address object was opened with a local IP address of 0.0.0.0 the transport will choose an appropriate valid local address. The maximum size of a datagram send is 0xffff - sizeof(UDP Header). The transport will fail a send attempt with a size larger than this. The datagram send is not buffered by the transport and does not complete until the underlying link layer is finished sending the data.

```
TDI_STATUS TdiSendDatagram(PTDI_REQUEST Request,
    PTDI_CONNECTION_INFORMATION ConnInfo,
    uint DataSize,
    uint *BytesSent,
    PNDIS_BUFFER Buffer
);
```

Parameters

ConnInfo

Specifies the remote address to which the datagram is to be sent. The address must be fully specified without wildcards. *ConnInfo* is not used to specify IP options for the send. These options may only be set on a per-address object basis, and not on a per send basis.

DataSize

Specifies the size in bytes of the data to be sent. This size must be equal to the sum of the sizes of the **NDIS_BUFFER** structures in the *Buffer* chain.

BytesSent

Pointer to where the transport may fill in the bytes of data sent in the event that the request completes immediately with TDI_SUCCESS. If the request pends for later completion, the bytes of data sent will be returned as the *ByteCount* parameter to the completion callback routine. This parameter will always be equal to *DataSize* if the request completes successfully.

Buffer

Points to a chain of **NDIS_BUFFER** structures to be sent. All buffers in this chain must be locked before the send is submitted, and the data in the buffers must not be altered until the send completes.

TdiReceiveDatagram

The **TdiReceiveDatagram** function is called through the **TdiReceiveDatagramEntry** pointer in the **TdiDispatchTable** structure.

This request submits a receive datagram request on the address object identified by Request->Handle.AddressHandle. MSTCP does not buffer incoming datagrams — if a datagram arrives and there is neither a receive datagram request posted nor a receive datagram event handler to be called, the datagram is dropped.

```
TDI_STATUS TdiReceiveDatagram(PTDI_REQUEST Request,  
    PTDI_CONNECTION_INFORMATION ConnInfo,  
    PTDI_CONNECTION_INFORMATION ReturnInfo,  
    uint RcvSize,  
    uint *BytesRcvd,  
    PNDIS_BUFFER Buffer  
);
```

Parameters

ConnInfo

Points to a structure describing remote addresses from which remote datagrams may be received. The addresses specified may contain wildcards. Incoming datagrams are only placed in receive datagram buffers that have matching remote addresses or wildcards in the **ConnInfo** structure. *ConnInfo* may be NULL, in which case any incoming datagram may use the buffer.

ReturnInfo

Points to a structure that will be filled in with the address of the remote entity that sent the datagram, as well as any IP options received with the datagram.

RcvSize

Maximum size in bytes of data that may be received. If *RcvSize* is less than the sum of the sizes of the buffers in the *Buffer* chain, only the first *RcvSize* bytes of the chain will be filled in.

BytesRcvd

Pointer to where the transport may fill in the bytes of data sent in the event that the request completes immediately with TDI_SUCCESS. If the request pends for later completion, the bytes of data sent will be returned as the *ByteCount* parameter to the completion callback routine.

Buffer

Pointer to a NULL-terminated list of **NDIS_BUFFER** structures of data to be sent. All buffers in the chain must be locked, and the chain must not be altered while the request is outstanding.

TdiSetEvent

The **TdiSetEvent** function is called through the **TdiSetEventEntry** pointer in the **TdiDispatchTable** structure.

This primitive sets a TDI event handler. These event handlers are pointers to routines that the VxD TCP stack will call when certain conditions occur, such as data arriving or a connection being disconnected. Event handlers are set on a per address object basis. The address object on which the handler is being set is identified by *Handler*.

```
TDI_STATUS TdiSetEvent(PVOID Handle,  
    int Type,  
    PVOID Handler,  
    PVOID Context  
);
```

Parameters

Type

Specifies which TDI event handler is being set. The valid types are TDI_EVENT_CONNECT, TDI_EVENT_DISCONNECT, TDI_EVENT_ERROR, TDI_EVENT_RECEIVE, TDI_EVENT_RECEIVE_DATAGRAM, and TDI_EVENT_RECEIVE_EXPEDITED.

Handler

Function pointer for the function to be called when an event of type *Type* occurs. Setting an event handler to NULL disables that event.

Context

Client-supplied value that is passed to the function identified by *Handler* when it is called. This *Context* is not used by all events.

TDI_EVENT_CONNECT

The event handler for **TDI_EVENT_CONNECT** is called when an incoming connection request arrives (a SYN frame) for a local address represented by the specified address object

and there is no connection in the listening state that could handle the request. The called client must indicate either acceptance or rejection of the connection by the return code from the call. If the client returns `TDI_MORE_PROCESSING`, the connection will be accepted. Returning any other return code will cause the connection to be rejected.

```
TDI_STATUS ConnectEvent(  
    PVOID EventContext,  
    uint AddressLength,  
    PTRANSPORT_ADDRESS Address,  
    uint UserDataLength,  
    PVOID UserData,  
    uint OptionsLength,  
    PVOID Options,  
    PVOID \*AcceptingID,  
    ConnectEventInfo \*EventInfo  
);
```

Parameters

EventContext

Context value supplied on the **TdiSetEvent** call.

AddressLength

Length in bytes of the structure pointed to by *Address*.

Address

Pointer to a **TRANSPORT_ADDRESS** structure identifying the remote peer that is attempting to connect. This structure is valid only for the duration of the call to the event handler.

UserDataLength

Length is the length in bytes of *UserData*. This is zero for MSTCP.

UserData

Pointer to connect data. This is NULL for MSTCP.

OptionsLength

Length in bytes of *Options*.

Options

Pointer to a buffer containing IP options received with the SYN. This buffer is valid only for the duration of the call to the event handler, and may be NULL.

AcceptingID

Pointer to where the called client should store the *ConnectionContext* of a connection on which to accept the incoming request, if it is to be accepted. This connection must be associated with the address object that received the connect request.

EventInfo

Pointer to a **ConnectEventInfo** structure. The client may fill in this structure with pointers to information that the transport will use in accepting the connection, and fill in with information about the remote peer when the connect completes. If the connection is accepted, the client must minimally fill in the **cei_rtn** field to a pointer to a callback routine and set the **cei_acceptinfo** and **cei_conninfo** fields to NULL.

TDI_EVENT_DISCONNECT

The **TDI_EVENT_DISCONNECT** event handler is called when a connection is disconnecting, either gracefully or abortively. If the connection is gracefully closing, the called client must issue a graceful **TdiDisconnect** to complete the disconnection, if it has not already done so. If the connection is abortively closing, the connection is gone at the time the event handler is called and the client need take no further action. An abortive disconnect may occur because an RST was received or because of a time-out.

```
TDI_STATUS DisconnectEvent(  
    PVOID EventContext,  
    PVOID ConnectionContext,  
    uint DisconnectDataLength,  
    PVOID DisconnectData,  
    uint OptionsLength,  
    PVOID Options,  
    ulong Flags  
);
```

Parameters

EventContext

Context value supplied on the TdiSetEvent call.

ConnectionContext

Context for the disconnection connection object. This context was supplied by the client as a parameter to the **TdiOpenConnection** call that opened the connection.

DisconnectDataLength

Unused and set to zero and NULL, respectively.

DisconnectData

Unused and set to zero and NULL, respectively.

OptionsLength

Length in bytes of the *Options* parameter.

Options

Pointer to a buffer containing IP options received with the frame containing the FIN or RST, if the disconnect was triggered by an incoming frame. This buffer is valid only for the duration of the call to the event handler, and may be NULL.

Flags

Identifies the type of disconnect. It is either TDI_DISCONNECT_RELEASE or TDI_DISCONNECT_ABORT.

TDI_EVENT_ERROR

The **TDI_EVENT_ERROR** event handler is unused by MSTCP and should not be set.

TDI_EVENT_RECEIVE

The **TDI_EVENT_RECEIVE** event handler is called when incoming data arrives for a connection. The called client may take all, some, or none of the indicated data during the call to the handler, and may also optionally pass back a pointer to a buffer to receive remaining or newly arrived data. There are some intricacies relating to when a receive event handler will be

called if it has recently been called or there is a receive buffer outstanding. For more information, refer to the Windows NT TDI documentation.

Since TCP is a stream-oriented protocol, there is no restriction on the minimum number of bytes that will be indicated in a receive event call.

```
TDI_STATUS RcvEvent(  
    PVOID EventContext,  
    PVOID ConnectionContext,  
    ULONG Flags,  
    UINT Indicated,  
    UINT Available,  
    UINT *Taken,  
    UCHAR *Data,  
    EventRcvBuffer *Buffer  
);
```

Parameters

EventContext

Context value supplied on the **TdiSetEvent** call.

ConnectionContext

Context for the connection object receiving the data. This context was supplied by the client as a parameter to the **TdiOpenConnection** call that opened the connection.

Flags

Supplies information about the receive indication. The most interesting flag is TDI_RECEIVE_ENTIRE_MESSAGE. If this flag is set, the PUSH bit was set on the segment containing the data being delivered. See the Windows NT TDI documentation for more details on the flags.

Indicated

Count of the number of bytes of data being indicated (for example, the number of bytes in the buffer pointed to by *Data*).

Available

Count of the total number of bytes of data available in the transport. This value will always be greater than or equal to *Indicated*. If *Available* is greater than *Indicated*, then there is data available that is not being passed up in the event handler, and the called client will need to supply a receive buffer to retrieve it. The receive buffer may be supplied either by returning a pointer in the **EventRcvBuffer** structure or by calling **TdiReceive**.

Taken

Pointer to a location where the client may fill in the total number of bytes taken on the indication. The client may set *Taken* to zero if no data was consumed. The maximum value to which *Taken* may be set is *Available*. Note that this may be greater than indicated, which allows the client to skip data that has arrived but not been indicated if it needs. *Taken* is only examined if TDI_MORE_PROCESSING or TDI_SUCCESS is returned. In these cases, the transport assumes that the first *Taken* bytes of data were consumed by the client and will not copy this data into a client buffer.

Data

Pointer to a buffer of received data. The length of the buffer is given by *Indicated*.

Buffer

Pointer to an **EventRcvBuffer** structure. The client may fill in this structure to give the transport a buffer into which to place data. This structure is only examined if the status TDI_MORE_PROCESSING is returned. The buffer returned by the client may be larger than the size indicated by *Available*. In this case, the buffer will be retained for more arriving data.

Comments

There are three relevant status codes that the client can return from the call to the event handler.

Status Code	Description
TDI_MORE_PROCESSING	Indicates that the client has taken some or all of the data and has returned a buffer for more data through the EventRcvBuffer structure.
TDI_NOT_ACCEPTED	Indicates that the client has taken none of the data and has not returned a receive buffer.
TDI_SUCCESS	Indicates that the client has taken some or all of the data, but has not returned a receive buffer.

Note If TDI_MORE_PROCESSING or TDI_SUCCESS is returned, the client must set *Taken* to be the count of the number of bytes consumed. If there is data that the client does not consume and there is no receive buffer provided, the VxD transport will make a best effort attempt to buffer the data for later consumption. However, if the transport is unable to allocate memory, the data is dropped and will need to be retransmitted.

TDI_EVENT_RECEIVE_DATAGRAM

The **TDI_EVENT_RECEIVE_DATAGRAM** event handler is called when an incoming datagram is received. As with the TDI_EVENT_RECEIVE event, the called client may take all, some, or none of the data, and may optionally pass back a buffer for unconsumed data. Unlike stream data, datagrams are not buffered, so any data the client does not consume is lost if no buffer is provided on the return. Also, MSTCP will not queue datagram receive buffers for later use if they are returned from an event handler. Thus, if a client consumes all the datagram and passes back a buffer, the buffer will be completed immediately with a length of zero.

```
TDI_STATUS RcvDGEvent(  
    PVOID EventContext,  
    uint AddressLength,  
    PTRANSPORT_ADDRESS Address,  
    uint OptionsLength,  
    PVOID Options,  
    uint Flags,  
    uint Indicated,  
    uint Available,  
    uint *Taken,  
    uchar *Data,  
    EventRcvBuffer **Buffer );
```

Parameters

EventContext

Context value supplied on the **TdiSetEvent** call.

AddressLength

Length in bytes of the structure pointed to by *Address*.

Address

Pointer to a **TRANSPORT_ADDRESS** structure identifying the remote peer that sent the datagram. This structure is valid only for the duration of the call to the event handler.

OptionsLength

Length in bytes of *Options*.

Options

Pointer to a buffer containing IP options received with the datagram. This buffer is valid only for the duration of the call to the event handler, and may be NULL if there were no options.

Flags

Set of TDI flags providing more information about the event. In the VxD environment, they are largely meaningless and may be ignored. See the Windows NT TDI documentation for more details on the flags.

Indicated

Is a cCount of the number of bytes of data being indicated (for example, the number of bytes in the buffer pointed to by *Data*).

Available

Count of the total number of bytes of data available in the transport. This value will always be greater than or equal to *Indicated*. If *Available* is greater than *Indicated*, then there is data available that is not being passed up in the event handler, and the called client will need to supply a receive buffer to retrieve it. The receive buffer must be supplied either by returning a pointer to an **EventRcvBuffer** structure. If *Available* is greater than *Indicated* and the client does not provide an **EventRcvBuffer** structure, the additional data will be lost.

Taken

Pointer to a location where the client may fill in the total number of bytes taken on the indication. The client may set *Taken* to zero if no data was consumed. The maximum value to which *Taken* may be set is *Indicated*. This is different from the **TDI_EVENT_RECEIVE** handler, where the maximum value of *Taken* is *Available*. If a client wants to consume the indicated data and discard the rest, it should return **TDI_SUCCESS**. *Taken* is only examined if **TDI_MORE_PROCESSING** is returned. In this case, the transport assumes that the first *Taken* bytes of data were consumed by the client and will not copy this data into a client buffer.

Data

Is a pointer to a buffer of received data. The length of the buffer is given by *Indicated*.

Buffer

Pointer to a pointer to an **EventRcvBuffer** structure. If the client wants to return a receive buffer, it should set *Buffer* to a pointer to a filled in **EventRcvBuffer** structure describing the buffer chain and providing a callback routine. Note that this is different from the **TDI_EVENT_RECEIVE** case, where the transport provides the **EventRcvBuffer** structure and the client merely fills it in. This structure is only examined if the status **TDI_MORE_PROCESSING** is returned. The client-provided

EventRcvBuffer structure must remain valid until the completion routine is called. The completion routine is generally called immediately.

Comments

The status codes that can be returned from a receive datagram event are **TDI_MORE_PROCESSING**, **TDI_NOT_ACCEPTED**, and **TDI_SUCCESS**. They have the same meaning as when they are returned from an ordinary **TDI_EVENT_RECEIVE** handler.

TDI_EVENT_RECEIVE_EXPEDITED

The **TDI_EVENT_RECEIVE_EXPEDITED** event is called to receive urgent data. It is nearly identical to the **TDI_EVENT_RECEIVE** event; the only difference between the two is that **TDI_EVENT_RECEIVE** receives only normal data and **TDI_EVENT_RECEIVE_EXPEDITED** receives only urgent data. The prototypes for the two event handlers are the same. Note that the two event handlers are not serialized by the transport.

TdiQueryInformation

The **TdiQueryInformation** function is called through the **TdiQueryInformationEntry** pointer in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiQueryInformation( PTDI_REQUEST Request,
    uint QueryType,
    PNDIS_BUFFER Buffer,
    uint *BufferSize,
    uint IsConn
);
```

Parameters

QueryType

Indicates which information is being queried. The following values are supported.

Value	Description
TDI_QUERY_PROVIDER_INFO	Queries the transport about provider information. A structure of type TDI_PROVIDER_INFO (see TDI.H) will be copied into the buffer chain pointed to by <i>Buffer</i> .
TDI_QUERY_ADDRESS_INFO	Queries the transport about the local transport of an address object or connection. The query is for a connection if <i>IsConn</i> is TRUE (nonzero), and for an address object if <i>IsConn</i> is FALSE (zero). The address object or connection being queried is specified by <i>Request->Handle</i> . A structure of type TDI_ADDRESS_INFO (see TDI.H) will

TDI_QUERY_CONNECTION_INFO	be copied into the buffer chain pointed to by <i>Buffer</i>. Queries MSTCP for information about the connection identified by Request->Handle.<i>ConnectionContext</i>. A structure of type TDI_CONNECTION_INFO (see TDI.H) will be copied into the buffer chain pointed to by <i>Buffer</i>.
TDI_QUERY_PROVIDER_STATISTICS	Queries the transport for certain statistics. Not all the defined statistics are kept by MSTCP. A structure of type TDI_PROVIDER_STATISTICS (see TDI.H) will be copied into the buffer chain pointed to by <i>Buffer</i>.

Buffer

Points to an **NDIS_BUFFER** chain into which the information is copied.

BufferSize

Points to the size of the input buffer chain in bytes. On return, *BufferSize* is set to the size of the information copied into *Buffer*. If the **NDIS_BUFFER** chain pointed to by *Buffer* is too small to contain the requested data, only the amount that fits is copied and TDI_BUFFER_OVERFLOW is returned.

IsConn

Boolean used to provide additional information for the TDI_QUERY_ADDRESS_INFO call. If *IsConn* is nonzero, than information about a connection is returned. If *IsConn* is zero, information about an address object is returned.

TdiSetInformation

The **TdiSetInformation** function is called through the **TdiSetInformationEntry** pointer in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiSetInformation(PTDI_REQUEST Request,
    uint SetType,
    PNDIS_BUFFER Buffer,
    uint *BufferSize,
    uint IsConn
);
```

Parameters

SetType

To be provided.

Buffer

To be provided.

BufferSize

To be provided.

IsConn

To be provided.

Comments This call is not supported in MSTCP and always fails.

TdiActionInformation

The **TdiActionInformation** function is called through the **TdiActionEntry** pointer in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiAction( PTDI_REQUEST Request,  
    uint ActionType,  
    PNDIS_BUFFER Buffer,  
    uint *BufferSize  
);
```

Parameters

ActionType

To be provided.

Buffer

To be provided.

BufferSize

To be provided.

Comments

This call is not supported in MSTCP and always fails.

TdiQueryInformationEx

The **TdiQueryInformationEx** function is called through the **TdiQueryInformationExEntry** in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiQueryInformationEx( PTDI_REQUEST Request,  
    TDIObjectID *ID,  
    PNDIS_BUFFER Buffer,  
    uint *Size,  
    void *Context  
);
```

Parameters

ID

To be provided.

Buffer

To be provided.

Size

To be provided.

Context

To be provided.

TdiSetInformationEx

The **TdiSetInformationEx** function is called through the **TdiQueryInformationExEntry** in the **TdiDispatchTable** structure.

```
TDI_STATUS TdiSetInformationEx( PTDI_REQUEST Request,  
    TDIObjectID *ID,  
    void *Buffer,  
    uint Size  
);
```

Parameters

ID

To be provided.

Buffer

To be provided.

Size

To be provided.