Communication Service Group

Communication Overview

AppleTalk

AppleTalk: Guides

AppleTalk: Functions

AppleTalk: Structures

Asynchronous I/O

Asynchronous I/O: Guides

Asynchronous I/O: Functions

Asynchronous I/O: Structures

BSD Sockets

BSD Socket: Guides

BSD Socket: Functions

BSD Socket: Structures

Diagnostic

Diagnostic: Guides

Diagnostic: Functions

Diagnostic: Structures

Internet Network Library

Internet Network Library: Guides

Internet Network Library: Functions

Internet Network Library: Structures

IPX/SPX

IPX/SPX: Guides

IPX: Functions

SPX: Functions

IPX/SPX and TLI IPX: Structures

Message

Message: Guides

Message: Functions

NCP Extension

NCP Extension: Guides

NCP Extension: Functions

NCP Extension: Structures

NWSIPX

NWSIPX: Guides

NWSIPX: Functions

NWSIPX: Structures

SAP

SAP: Guides

SAP: Functions

SAP: Structures

TCP/IP and TCP/IPX

TCP/IP and TCP/IPX: Guides

TLI

TLI: Guides

TLI: Functions

TLI: Structures

WinSock 2

See the http://www.stardust.com URL for more information.

AppleTalk

AppleTalk: Guides

AppleTalk: General Guide

Communication Service Group

AppleTalk: Structures

AppleTalk: Concepts

About Receiving Data in General

When it comes time to receive any kind of data from the protocol layer, call the **ATAtpGet** function. There are four events that can come up from the protocol: one for an arriving request (ATP_EVENT_RCV_REQ), one for an arriving response (ATP_EVENT_RCV_RSP), one for an error in sending a request (ATP_EVENT_SND_REQ), one for an error in sending a response (ATP_EVENT_SND_RSP). An application that uses ATP should frequently call **ATAtpGet** to receive events, then act on the events. For example, an ATP based service like AFP would call **ATAtpGet** waiting for a request (ATP_EVENT_RCV_REQ), do the action requested (for example, a disk request), then send a response with **ATAtpSendRsp**.

Parent Topic: ATP and NetWare

About Sockets and ATP

Unlike DDP, ATP accepts only one listening socket per file descriptor.

AppleTalk sockets fall into two categories: statically assigned and dynamically assigned. Statically assigned sockets have numbers 1 through 127; dynamically assigned sockets have numbers 128 through 254.

Apple Computer Inc. reserves the use of sockets 1 through 63; you can use sockets 64 through 127 **experimentally but not in released products**.

Parent Topic: ATP and NetWare

About This PAP Interface

This interface is unique among PAP interfaces in that the read function, **ATPapGet**, allows a client to do without a **look** function to see what kind of data is coming up the pipe, which is more efficient. Also, with this interface, it is possible to have a single thread servicing all incoming requests.

In terms of the amount of time it takes for PAP functions to complete, you can count on the connect, read (**ATPapGet**), and write functions to take time. The others are very quick, involving no blocking of significant duration.

Read *Inside AppleTalk* to get a full description of PAP and the basic functionality of a PAP implementation. Bear in mind that, although they are called "workstation-side" and "server-side" functions, an NLM could call either type of function.

Parent Topic: PAP and NetWare

About Transaction IDs

Remember that there are **two** transaction IDs. The first is the transaction ID on the wire (network). The second, the subject of the following discussion, is created by the ATP implementation for matching up requests and responses across the API boundary.

The transaction ID is created by the side that created the request.**The side that received the request must present the same transaction ID when it presents the reply**. The end that receives the reply can then match the response with the request. Thus, when the client receives a request from ATP (in **ATAtpGet**), it must send the response with the same transaction ID (**ATAtpSendRsp**). Likewise, when the client sends a request (**ATAtpSendReq**), it can specify a transaction ID which is returned when ATP receives a reply (in **ATAtpGet**).

Transaction IDs are also used to cancel transactions, and since the ones generated by the client may be nonunique, multiple cancellations could possibly occur. Note, however, that in ALO mode the transaction ID of received requests is the first, or wire, type. Therefore, multiple responses can be made reusing this transaction ID.

Parent Topic: ATP and NetWare

ADSP

ADSP is a connection-oriented protocol that provides reliable, full-duplex, byte-stream service between sockets. It guarantees that data bytes are delivered in the same order as they were sent, and that they are free of duplicates.

Using TLI, you can access ADSP to carry out these tasks:

Create a connection end

Bind

Request a connection with another protocol client

Wait for a connection request from another protocol client

Accept a connection request

Send and receive bytes of data over a connection by segmenting data into variable length messages

Send priority messages that do not disrupt the existing byte stream

Receive notification that a connection has been closed

Unbind

Close a connection

For information on using TLI functions to access ADSP, seeTLI Functions: AppleTalk Notes.

Parent Topic: AppleTalk Overview

ADSP Performance Notes

A number of factors affect performance when your application accesses ADSP through TLI. This chapter provides guidelines for making changes to your application in order to maximize performance.

The sections that follow provide guidelines for an application whose data flow follows one of these patterns:

The application transfers large amounts of data. Data flow in a backup or file-transfer application may follow this pattern. ADSP is presently engineered to serve this type of application best.

The application transfers variable amounts of data in alternating directions. Data flow in a database or terminal emulation application may follow this pattern.

This chapter does not apply to optimizing an application whose data transfer follows one of these patterns:

The application transfers data that must arrive within an extremely narrow window of time. Data flow in a videoconferencing application follows this pattern.

The application involves the simultaneous transfer of large amounts of identical data to a large number of remote stations. Data flow in a broadcast application follows this pattern.

Connection-oriented protocols such as ADSP are not well-suited to these types of applications.

Related Topics

Transferring Data in ADSP

Using the T_MORE Bit

Parent Topic: TLI Implementation Notes for AppleTalk

AppleTalk Byte Ordering

The AppleTalk protocol family specifies that data be represented in network byte order, in which integers are stored most-significant-byte-first. However, the NetWare platform is built on the I86 family of processors, which uses Intel byte order, in which integers are stored least-significant-byte-first.

Except for the user bytes area of the ATP header (*userdata* field of the ATAtpPass_t structure), all structures and parameters defined by this interface assume host order. On the other hand, the data areas are considered byte arrays (C char *) and data buffers are put on the wire exactly as passed in to the interface. When sending numeric information in the data area, the decision must be made whether to pass the information in host or network order. If all hosts were of the same processor type, it would be possible to use host order. In today's heterogenous networking environments, it is more typical to use network order.

Suppose you wanted to send a packet to a node residing on destination network number 10. An example of bytes whose order you wouldn't need to swap are the destination network number (*dst_net* field in the ATDdp_t structure). You would set this field to 10 and the interface would swap the bytes to reflect network ordering. On the other hand, if you had a four-byte integer in the data area of the packet, it would result in the bytes 01 00 00 00 being sent out over the wire (1 in Intel host order):

```
int *data;
data = (int*) data;    /*  data is a char *  */
*data=1;
```

Therefore, if you want to send 00 00 00 01 (1 in network order), you must add hton32(*data):

```
int *data;
data = (int*) data;  /*  data is a char *  */
*data=1;
hton32(*data);
```

Perhaps the biggest challenge in regard to byte order is presented by the ATP interface, the *userdata* field of the ATAtpPass_t structure. Most protocols will want to use this field on a byte-by-byte basis yet it is defined as data type long. Given that this field is in host order, if you simply set *userdata* to the value of 15, the four following bytes would be sent out over the wire: 00 00 00 15.

To set each individual byte of the *userdata* field, use the following C code:

```
userdata = hihibyte << 24 + hilobyte << 16 + lohibyte <<
8 + lolobyte;
```

This code will produce the following byte order on the wire:

```
hihibyte hilobyte lohibyte lolobyte
```

There is another ordering convention you must consider before sending structures across a network. Be sure to check the structure-packing conventions of your compiler.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

AppleTalk Internet Addresses

The size of an AppleTalk Phase I or Phase II internet address is always 4 bytes.

An AppleTalk internet address uses the ATInet_t structure. This structure and the type definitions it uses are defined in appletlk.h. See also ATInet_t.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

AppleTalk Notes: t_accept

NOTE: Applies to ADSP only.

The **t_accept** function enables your application to accept a connection request on a transport endpoint:

```
t_accept(int fh, int resfh, struct t_call *call)
```

If *fh* is a listening endpoint and resfh is the transport endpoint where the connection is established, *fh* cannot equal *resfh*. In other words, an endpoint listening for a connect request with **t_listen** cannot accept a connect request with **t_accept**.

In addition, ADSP does not support the transfer of data during a **t_accept** function. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set *call->udata* .buf and *call->udata.len*, **t_accept** ignores the data.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_bind

In AppleTalk, all opened transport endpoints on a node have the same network and node number. **t_bind (Function)** enables you to associate a protocol address with a specified transport endpoint in order to allocate a socket for use by it.

AppleTalk sockets fall into two categories: statically assigned sockets and dynamically assigned sockets. Statically assigned sockets are denoted by numbers 1 through 127; dynamically assigned sockets are denoted by numbers 128 through 254. Apple Computer, Inc. reserves the use of sockets 1 through 63; you can use sockets 64 through 127 experimentally, **but not in released products**.

Dynamic sockets are assigned when a transport client specifies zero for *req->addr.len* or sets req to NULL when calling **t_bind**. The transport provider returns the internet address of the socket in *ret->addr.buf*. Attempts to bind to a specific socket in the dynamic socket range result in the error TBADADDR.

If no more sockets are available, **t_open** succeeds, but **t_bind** fails and generates the error TNOADDR.

> **IMPORTANT:** If you pass in zero (0) as *qlen*, you get a connection endpoint and if you pass in a nonzero value, you get a listener. It is best to decide up front (at the point of binding) whether you want a connection endpoint or a listener. Then, if that changes later, we recommend that you unbind and rebind.

It is possible for a single socket to be used for more than one connection. However, it cannot be used for more than one listener.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_close

The AppleTalk protocols do not support an orderly release mechanism. Therefore, before you issue **t_close** for an ADSP transport endpoint, make sure that all data to be received or sent on that endpoint has been transmitted. When you issue **t_close**, the endpoint should be in the T_UNBND state. If the endpoint is in the T_DATAXFER state when you issue **t_close**, all queued data is discarded.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_connect

> **NOTE:** Applies to ADSP only.

An endpoint listening for a connect request with **t_listen** cannot issue

t_connect.

In addition, ADSP does not support the transfer of data during a call to **t_connect**. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set *call->udata.buf* and *call->udata.len*, **t_connect** ignores the data.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_getinfo

t_getinfo returns the current characteristics of the underlying transport protocol. For the values returned (in the t_info structure) by **t_getinfo** when your application is accessing DDP, see the t_info Values for DDP table. For the values returned (in the t_info structure) by **t_getinfo** when your application is accessing ADSP, see the t_info Values for ADSP table.

If the size of the option or address buffer changes for a future version of DDP or ADSP, so are the corresponding values in the t_info structure (*addr* and *options*).

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_listen

NOTE: Applies to ADSP only.

The transport client must complete the NBP registration procedure **before** calling **t_listen** in blocking mode. When **t_listen** is in blocking mode, control does not return to the application until the listening endpoint receives a connection request. However, because the listening endpoint is not registered on the network, it does not receive a connection request. It therefore listens endlessly. For information on NBP registration, seeNBP Function List.

In addition, ADSP does not support the transfer of data during a call to **t_listen**. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set *call->udata.buf* and *call->udata.len*, **t_listen** ignores the data.

When an application no longer wants to listen, it should complete the NBP deregistration procedure and **t_unbind**.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_open

Call **t_open** to initialize a transport endpoint using this syntax:

```
t_open(char *path, int oflag, struct t_info
*info)
```

Specify either of these files in the *path* argument:

"/dev/ddp"	Identifies DDP, the connectionless transport service.
"/dev/adsp"	Identifies ADSP, the connection-oriented transport service.

You can specify NULL for *info*.

If info is not NULL, it contains (on returning) the current characteristics of the underlying transport protocol. For the values returned by **t_open** when your application is accessing DDP, see the t_info Values for DDP table. For the values returned by **t_open** when your application is accessing ADSP, see the t_info Values for ADSP table.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_optmgmt

This function enables a transport client to retrieve, verify, or negotiate protocol options with the transport provider. SeeHow to Determine the ADSP Interface Version.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_rcv

NOTE: Applies to ADSP only.

When **t_rcv** receives data with the T_EXPEDITED flag set, the flag indicates that the data is part of an expedited transport service data unit (ETSDU), also known as an ADSP Attention message. Attention messages enable two transport clients to signal each other outside the regular flow of data between them. The Attention message consists of a 2-byte attention code and up to 570 bytes of data. When a client sends an Attention message with **t_snd**, the message is delivered to buffer space outside the target client's receive queue.

Only one Attention message in each direction may be outstanding.

Although the buffer space for expedited data is separate, this data may get "caught behind" normal data during processing of the data by STREAMS.

Message boundaries are supported in ADSP when an optional end-of-message (EOM) bit is set in the ADSP header. Because message boundaries are optional in ADSP, there is no limit to the size of a transport service data unit (TSDU). Therefore, it is possible that every call to **t_rcv** returns with the T_MORE flag set. The presence of this flag does not necessarily indicate that more data is available for **t_rcv**; rather, it may simply mean that the EOM indicator does not appear in the ADSP data stream.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_rcvconnect

NOTE: Applies to ADSP only.

ADSP does not support the transfer of data during a call to **t_rcvconnect**. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set call->udata.buf and call->udata.len, **t_rcvconnect** ignores the data.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_rcvdis

NOTE: Applies to ADSP only.

If a connection terminates, ADSP tells the transport client why the termination occurred by using the *reason* field in the t_discon structure.

When your application accesses ADSP, the possible values for reason are these:

ETIMEDOUT	The connection partner failed to acknowledge the connection after repeated queries from ADSP.
ECONNREFUSED	During the connection establishment phase, the target connection end refused the request to connect. This failed connection may be due to a resource problem at the target.
ECONNRESET	The connection was terminated by the connection partner.

In addition, ADSP does not support the transfer of data during a call to **t_rcvdis**. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set call->udata.buf and call->udata.len, **t_rcvdis** ignores the data.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_rcvrel

DDP and ADSP do not support an orderly release of a connection. Therefore, this function is not supported. An attempt to call**t_rcvrel** results in the error TNOTSUPPORT.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_rcvudata

NOTE: Applies to DDP only.

The maximum amount of data a transport client can receive with a single call to **t_rcvudata** is the maximum size of the data area in a DDP packet, which is 586 bytes.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_snd

NOTE: Applies to ADSP only.

When you use **t_snd** with the T_EXPEDITED flag, the data is sent as an expedited transport service data unit (ETSDU), also known as an ADSP **Attention message**. Attention messages enable two transport clients to signal each other outside the regular flow of data between them. The Attention message consists of a 2-byte attention code and up to 570 bytes of data. When a client sends an Attention message with **t_snd**, the message is delivered to buffer space outside the target client's receive queue.

Only one Attention message in each direction may be outstanding

Although the buffer space for expedited data is separate, this data may get "caught behind" normal data during processing of the data by STREAMS.

Message boundaries are supported in ADSP by setting an optional end-of-message (EOM) bit. Because message boundaries are optional in ADSP, there is no limit to the size of a transport service data unit (TSDU). Therefore, it is possible that every t_snd call sets the T_MORE flag. The

presence of this flag does not necessarily indicate that more data is available for **t_snd**; it may simply mean that the EOM indicator does not appear in the ADSP data stream.

Conversely, (as mentioned in ATInet_t (appletlk.h)) if you do not set the T_MORE flag during a call to **t_snd**, ADSP sets the EOM bit for the packet containing the data going out on that **t_snd** call. Because an EOM indicator must logically be found at the end of an ADSP packet, the use of the T_MORE flag may influence the size of each packet. For example, if you send 100 bytes five times, and you **do not** set the T_MORE flag each time, ADSP sends the data in five packets, each containing 100 bytes of user data; the EOM indicator appears at the logical end of each packet. However, if you send 100 bytes five times using the T_MORE flag on the first four sends, ADSP may send the data as a single packet; in this case, the EOM indicator appears at the logical end of the final packet. Keep in mind the performance benefits of using the T_MORE flag when using **t_snd**.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_snddis

NOTE: Applies to ADSP only.

ADSP does not support the transfer of data during a call to **t_snddis**. If **t_alloc** is used to allocate the t_call structure, no buffer is allocated for data. If you allocate a data buffer yourself and set call->udata.buf and call->udata.len, **t_snddis** ignores the data.

When a transport endpoint on one side of a connection issues **t_snddis**, any data queued for that connection or in transit over that connection is discarded. Therefore, make sure there is no longer any data to send or receive before you issue **t_snddis**. **t_snddis** sends a notification of the disconnection to the connection partner.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_sndrel

DDP and ADSP do not support an orderly release of a connection. Therefore, this function is not supported. An attempt to call **t_sndrel** results in the error TNOTSUPPORT.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_sndudata

NOTE: Applies to DDP only.

When you use **t_sndudata** with the AppleTalk protocols, keep these points in mind:

The maximum amount of data a transport client can send with a single call to **t_sndudata** is the maximum data size of a DDP packet, which is 586 bytes.

DDP supports two transport options, checksum and ddp_type.

If the *opt.buf->checksum* field is nonzero, DDP generates a checksum for the datagram. (The default is zero.)

The *opt.buf->ddp_type* field should be set to a nonzero value. The default is 128. A zero value is invalid. Also, remember that Apple* Computer, Inc. reserves numbers 1 through 15 for its own use.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Notes: t_unbind

After you disconnect a connection endpoint with **t_rcvdis** or **t_snddis**, you can re-use it, but only as a connection endpoint, not as a listener. To use a connection endpoint as a listener, or vice versa, you need to unbind and bind again. You need not close the connection endpoint with **t_close** and open it again with **t_open**, or even unbind it.

Parent Topic: TLI Functions: AppleTalk Notes

AppleTalk Overview

This guide is written for NLM programmers using Transport Level Interface (TLI) functions to communicate with Macintosh clients across the network. It focuses on the **interface** to the AppleTalk protocol stack, not on the internals of the stack. Knowledge of TLI and the AppleTalk network system are assumed. This guide describes Novell's implementation of TLI and AppleTalk. Therefore, although a glossary is provided and every attempt has been made to clarify TLI and AppleTalk concepts, you should rely on the following sources for TLI and AppleTalk information:

The TLI chapters in this manual

Inside AppleTalk (AppleTalk design and function descriptions)

AppleTalk Services is for NLM™ programmers who want to communicate over the network with Macintosh clients using the AppleTalk protocol suite. The following figure shows the general purpose of the protocols in the

AppleTalk suite. The following table shows the relative equivalency of NetWare and AppleTalk protocols.

Figure 1. NetWare and AppleTalk Protocols with OSI Layers

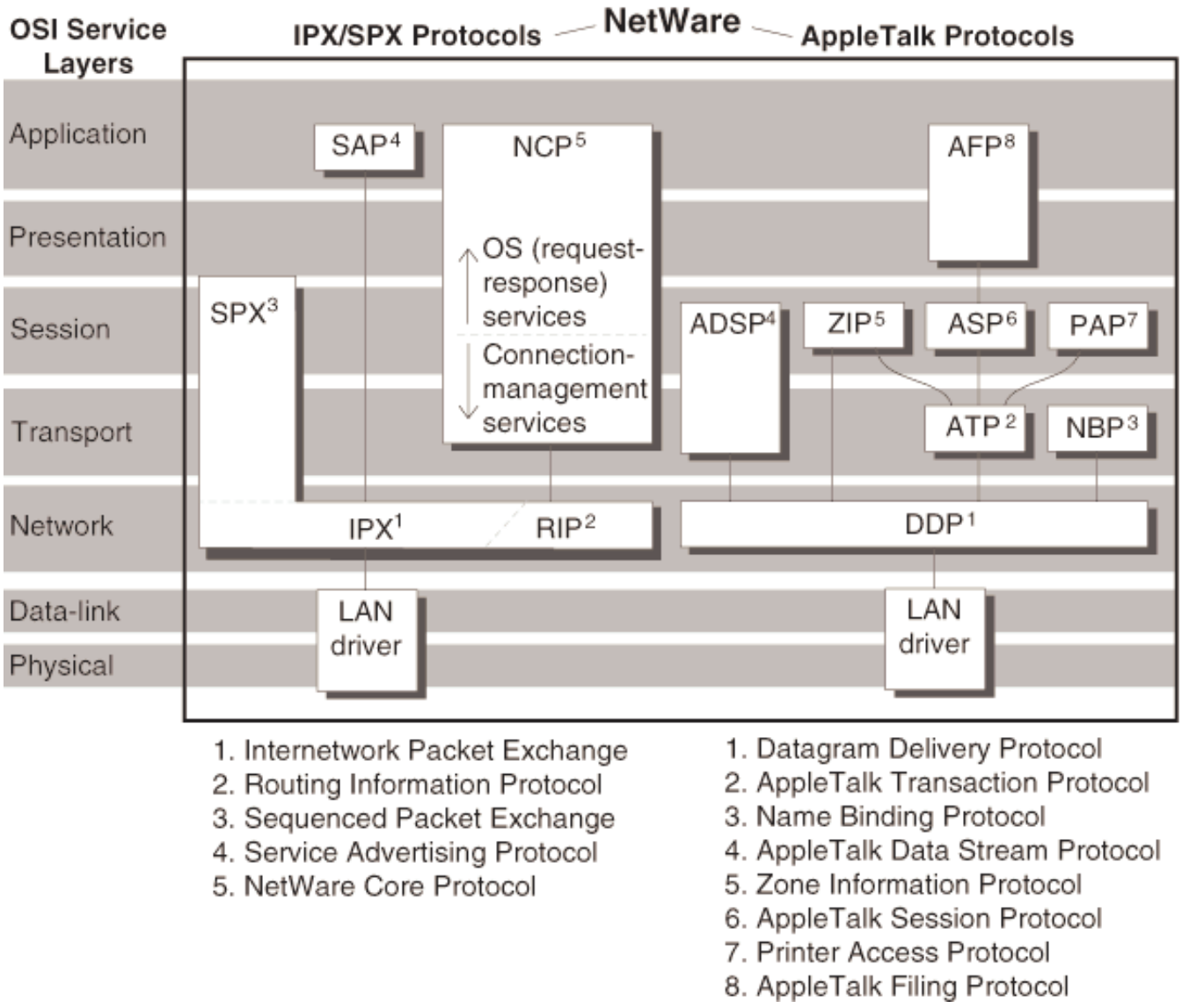


Table auto. Comparability of Protocols

NetWare	**AppleTalk**
NCP	AFP, ASP, ATP, PAP
SAP	NBP

SPX	ADSP
RIP	RTMP
IPX	DDP

Novell® provides two ways to access the AppleTalk protocols, Transport Level Interface (TLI).

TLI provides a single interface through which you can access multiple protocol families, for example IPX/SPX, TCP/IP, and, in the AppleTalk family, the following two protocols:

Datagram Delivery Protocol (DDP)

AppleTalk Data Stream Protocol (ADSP)

The native AppleTalk functions, exported by APPLETLK.NLM, provide more direct access to DDP as well as the following five protocols:

Name Binding Protocol (NBP)

Zone Information Protocol (ZIP)

AppleTalk Transaction Protocol (ATP)

AppleTalk Session Protocol (ASP)

Printer Access Protocol (PAP)

IMPORTANT: The information in this chapter provides only an introduction to the AppleTalk protocols; you are encouraged to read *Inside AppleTalk, Second Edition* for complete information on them.

Related Topics

ADSP

DDP

NBP

ZIP

ATP

ASP

PAP

Notes on the AppleTalk Interface, both TLI and Native

AppleTalk Services Restrictions on TLI

ADSP: This implementation of TLI does not support a forward-reset mechanism to allow an ADSP client to abort the delivery of outstanding data to the connection partner.

Parent Topic: TLI Implementation Notes for AppleTalk

AppleTalk TLI Structures

Seven structures are described in this section:

ATInet_t

t_optmgmt (Structure) (used with **t_optmgmt (Function)**)

netbuf

ATAdspOpt_t

ATADdpOpt_t

t_info

t_unitdata

In addition, the TLI chapters in this manual contain descriptions of the TLI structures above as well as others:

t_bind (Structure)

t_call (**t_accept**, **t_connect**, **t_listen**, **t_rcvconnect**, and **t_snddis** functions)

t_discon

t_sndudata

t_rcvudata

t_uderr (used with the **t_rcvuderr** function)

They are defined in tiuser.h.

Related Topics

Synopsis of the Structures' Functions

ATAdspOpt_t (adsp.h)

t_optmgmt (adsp.h)

netbuf (tiuser.h)

t_info (tiuser.h)

ATInet_t (appletlk.h)

ATDdpOpt_t (ddp.h)

Parent Topic: TLI Implementation Notes for AppleTalk

AppleTalk Version Function

ATAppletalkVersion, defined in the appletlk.h file, allows your NLM to request a particular level of interface service and find out which version of the native AppleTalk interface is running on the local server. The version described in this manual is 2.0. If you write your NLM to this version of the interface, call **ATAppletalkVersion** with *major* as 2 and *minor* as 0. If version 2.0 is not running on your server, the function returns 0 (FALSE). (Please note that although functions usually return 0 to indicate success, in this function 0 denotes FALSE, and 1 is TRUE.)

AppleTalk Version Functions

Your NLM™ application can now determine which versions of the AppleTalk TLI interfaces to ADSP are running on the server. By calling **t_optmgmt (Function)**, you can compare the version you wrote your NLM for to the one returned by the function. It is essential that you check the interface version on the server for the following reason: Although Novell has taken great care to leave the fields of the options structures in the same position relative to the beginning of the structures, even after they have become obsolete, as the TransThresh and TransTimerIntrvl options have, the overall size of these structures has changed. They are larger in this release than in the previous one. Therefore, before your NLM begins its normal work, have it check the interface version.

You can use the new **ATAppletalkVersion** function to find out which version of the AppleTalk Interface for NLM applications is running on the local server so that your NLM can select the right function. For example, if you are using this release of the interface (and using **ATAtpGet**), call **ATAppletalkVersion**, requesting this release (2.0). If it is not running on your server, an error is returned. (Although functions usually return a nonzero value to indicate an error, this function returns 1 to indicate**no** error.)

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

ASP

AppleTalk provides a reliable transaction service, via ATP, that can be used

for transporting workstation commands to servers. However, ATP does not provide the full range of transport functions needed by many higher-level network services. AppleTalk Session Protocol (ASP) was designed specifically for the use of these higher-level services.

ASP is a client of ATP; it adds value to ATP, providing the level of transport service needed for higher-level workstation-to-server interaction.

ASP provides the following services:

- Setting up (opening) and tearing down (closing) sessions
- Sending commands on an open session to the server and returning command replies (which might include a block of data)
- Writing blocks of data from the workstation to the server end of the session
- Sending an attention from the server to the workstation
- Retrieving service status information from the server without opening a session

The concept of a session is central to ASP. Two network entities, one in a workstation and the other in a server, can set up an ASP session between themselves. A **session** is a logical relationship (**connection**) between two network entities; it is identified by a unique **session identifier** (**session ID)**. For the duration of the session, the workstation entity can (through ASP) send a sequence of commands to the server entity. ASP ensures that the commands are delivered without duplication in the same order as they were sent and conveys the results of these commands (known as a **command reply** or **reply**) back to the workstation entity.

The process of setting up a session is always initiated by the workstation entity when it wants to use the server entity's advertised service. Once the session is established, the workstation client of ASP sends commands, and the server client of ASP replies to the commands. ASP does not allow its server client to send commands to the workstation client. However, ASP provides an attention mechanism by which the server can inform the workstation of a need for attention.

More than one workstation can establish a session with the same server at the same time. ASP uses the session ID to distinguish between commands received during these various sessions. The session ID is unique among all the sessions established with the same server.

A server entity that needs to make its service known on the AppleTalk network calls ASP to open an ASP responding socket and then calls the Name Binding Protocol (NBP) to register a unique name on this socket. ASP then starts listening on the socket for session-opening commands coming over the network. A workstation that wants to use the advertised service uses NBP to identify the service's socket address. Then the workstation client calls ASP to open a session.

Setting up a responding socket and looking for the socket's address through NBP are done outside the scope of ASP. The participation of ASP starts with the process of setting up a session.

ASP does not provide a user authentication mechanism. If needed, this mechanism must be supplied by a higher-level protocol than ASP. In addition, ASP does not provide any mechanism to allow the use of a particular session by more than one server entity. Such multiplexing of a session can be done by the ASP clients if higher-level protocols divide the function codes into ranges and manage them completely outside the scope of ASP. The use of a single session to gain access to various services on the same node is not recommended.

The **ATAspCloseSession** function has not been implemented. This is because each file descriptor is a session, so when a close is wanted, **ATAspClose** should be called on that file descriptor.

For a full description of the ASP functions, see AppleTalk: Functions.

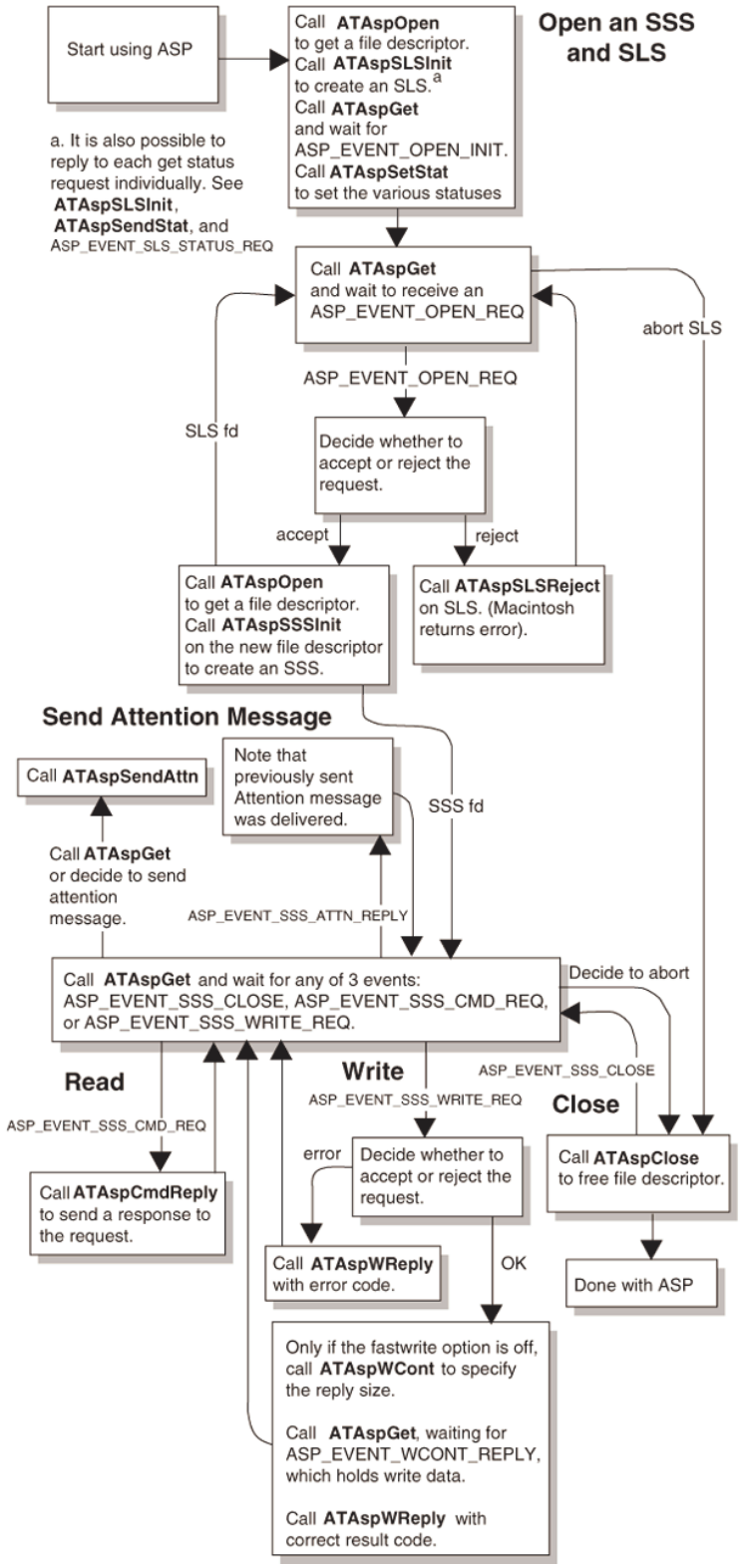
Parent Topic: AppleTalk Overview

ASP and NetWare

ASP is a transaction-oriented, reliable session protocol. Novell offers an interface to the server side of this protocol that is different from that suggested in *Inside AppleTalk*, so a brief discussion of the interface is in order. Follow along on the following figure.

IMPORTANT: An interface to the workstation side is not provided.

Figure 2. ASP Flowchart


Start using ASP
Call ATAspOpen to get a file descriptor. Call ATAspSLSInit to create an SLS.a Call ATAspGet and wait for ASP_EVENT_OPEN_INIT. Call ATAspSetStat to set the various statuses
Open an SSS and SLS
a. It is also possible to reply to each get status request individually. See ATAspSLSInit, ATAspSendStat, and ASP_EVENT_SLS_STATUS_REQ
Call ATAspGet and wait to receive an ASP_EVENT_OPEN_REQ
abort SLS
ASP_EVENT_OPEN_REQ
SLS fd
Decide whether to accept or reject the request.
accept
reject
Call ATAspOpen to get a file descriptor. Call ATAspSSSInit on the new file descriptor to create an SSS.
Call ATAspSLSReject on SLS. (Macintosh returns error).
Send Attention Message
Call ATAspSendAttn
Note that previously sent Attention message was delivered.
SSS fd
Call ATAspGet or decide to send attention message.
ASP_EVENT_SSS_ATTN_REPLY
Call ATAspGet and wait for any of 3 events: ASP_EVENT_SSS_CLOSE, ASP_EVENT_SSS_CMD_REQ, or ASP_EVENT_SSS_WRITE_REQ.
Decide to abort
Read
Write
ASP_EVENT_SSS_CLOSE
ASP_EVENT_SSS_WRITE_REQ
Close
ASP_EVENT_SSS_CMD_REQ
error
Decide whether to accept or reject the request.
Call ATAspClose to free file descriptor.
Call ATAspCmdReply to send a response to the request.
Call ATAspWReply with error code.
OK
Done with ASP
Only if the fastwrite option is off, call ATAspWCont to specify the reply size.
Call ATAspGet, waiting for ASP_EVENT_WCONT_REPLY, which holds write data.
Call ATAspWReply with correct result code.

ASP, like most session protocols, has listeners for opening sessions, and regular sockets for data transfer. Listeners are Server Listener Sockets, or SLSs, and the data transfer sockets are Server Session Sockets, or SSSs.

First you need an SLS. All sockets and sessions reside on NetWare CLIB file descriptors, so create a new ASP file descriptor by calling **ATAspOpen**.

Next decide how you want to cope with the status requests that come in. The options are to reply to each status message individually or to set the status message once and have all requests be answered with that response. Usually the latter suffices. An example of how the former could be useful would be if, for security reasons, you wanted to show a busy status to requests from a particular network, but idle status to other networks.

Having made that decision, call **ATAspSLSInit** to make this file descriptor into an SLS, and set the stat_flag parameter appropriately. Call **ATAspGet** and wait for the event ASP_EVENT_SLS_INIT to come up. Look in the *pass->ret* field to make sure the initialization succeeded. In the single-status case, now call **ATAspSetStat** with the status string you desire.

The SLS is properly initialized now. You wait for events to come up by calling **ATAspGet**. When an event comes up, service it, then go back to calling **ATAspGet** to get the next event. In this way, the interface is event-oriented. If you want to not have a process block on this file descriptor, poll for readability with the CLIB poll function. The two valid events in this state are ASP_EVENT_OPEN_REQ and ASP_EVENT_SLS_STATUS_REQ. The ASP_EVENT_SLS_STATUS_REQ event is received in cases when you indicated the desire to respond to each status request individually. You should respond with **ATAspSendStat**. When an ASP_EVENT_OPEN_REQ comes up, decide if you want to accept the connection. Possible reasons for rejecting it would be that there are too many connections already (licencing) or a connection request arrived from an unauthorized address (in *pass->U.OpenReq.addr*). If you want to reject the connection request, call **ATAspSLSReject**, handing back the address and ID that came up in the pass structure of the **ATAspGet**.

To accept the session, you create an SSS. Thus, you need to create a new file descriptor that becomes the SSS by calling **ATAspOpen**. On this new file descriptor, call **ATAspSSSInit** with the address and ID that came up in the pass structure on the **ATAspGet** call on the SLS file descriptor. The other parameter, fastwrite, refers to how write requests are handled and is explained shortly. The ASP_EVENT_SLS_OPEN_REQ becomes fulfilled, and **ATAspGet** should be called to get another event.

But now you have an SSS to process. Since ASP is an asynchronous protocol, the server side receives requests and rarely generates transactions. Thus, **ATAspGet** should be called in a loop, over and over, satisfying the requests that come in. There are two major kinds of requests, command requests (ASP_EVENT_SSS_CMD_REQ) and write requests (ASP_EVENT_SSS_WRITE_REQ). Command requests probably make up the bulk of requests and are simple to satisfy; when you know the answer to

the request, call **ATAspCmdReply** to send it to the workstation.

Write requests involve the complex inverted double transaction explained best in *Inside AppleTalk*. After receiving the ASP_EVENT_SSS_WRITE_REQ event and the associated data, you can decide if the request is in error and, if so, call **ATAspWReply** with an error code, terminating the write before the second transaction. However, if it appears to be good or if you can't tell yet if there is an error, call **ATAspWCont** with the write buffer size you are willing to accept, and call **ATAspGet**, waiting for an ASP_EVENT_SSS_WRITECONT_REPLY. This event has the write data, the second part of the write transaction. After receiving the full write data, call **ATAspWReply** with either success or failure to terminate the write. Go back to reading **ATAspGet** for the next request.

Notice that a significant optimization can be made here. The only data to the **ATAspWCont** call was the buffer size to be accepted. Most applications want this to be as large as possible, such as ASPQuantumSize (retrieved using the **ATAspGetParms** function). If this is true of your application, when you initialize the session using **ATAspSSSInit** you can activate the Fastwrite option. In this option, the protocol doesn't wait for the application to call **ATAspWCont**. It sends to the workstation the write continue request immediately, with a buffer size of ASPQuantumSize.

The action that occurs completely out of band is the Attention message. The server end is allowed to initiate actions by sending attention messages. An example of this use is to send the AFP client notification that a message has arrived that should be displayed on the screen, such as imminent shutdown of the file server. Previously, workstations had to poll the server, which could create large load on mostly idle, large node networks. The server sends an Attention message by calling **ATAspAttn**, using an SSS file descriptor. The acknowledgment that the attention arrived comes up as an ASP_EVENT_SSS_ATTN_REPLY and occurs at any time. A complication of this process arises in the write request handling. After calling **ATAspWCont**, wait for ASP_EVENT_SSS_WRITECONT_REPLY, but an ASP_EVENT_ATTN_REPLY may come up instead. Beware.

In order to close any socket, SLS or SSS, call **ATAspClose**. Calling **ATAspClose** causes rather drastic, abort-like closes. When an SLS is closed, all SSSs created from that SLS is aborted as well. On an SSS, there is no guarantee that outstanding data is delivered, and although an attempt is made to notify the client, the system doesn't wait for the client to acknowledge the close. Therefore, it is good for the application to provide close semantics as well. (AFP does this with the **AFPLogout** command.) When the workstation wants to close a session, an ASP_EVENT_SSS_CLOSE comes up. On the server side, the session isn't closed then but when **ATAspClose** is called, so call it soon.

Low Memory and AppleTalk: Low memory conditions create some noteworthy challenges. The general goal of the interface is to attempt to survive resource limitation spikes but start closing connections when the limitations become chronic. How much is enough for the interface to do in this regard is still an open issue. In general, Novell attempts to deliver

whatever possible; however, we have no control over STREAMS policies for delivering the data.

Buffer Tips: Be aware that most buffers can be passed in as zero pointers if you want to pass in no buffer. You also need to pass in a length of 0.

Making the Right ASP Call at the Right Time: Because ASP is a state-oriented protocol, making the wrong call at the wrong time could result in errors. The ASP Flowchart shows which functions to call at which time. ASP sends the message up to the Stream head and it is then out of ASP's jurisdiction; it's at that point that the state changes. It is good practice to receive events in a timely fashion, that is, to read off the Stream head quickly. Do not make assumptions about which state exists, such as "I**will** receive a message, thus I **will** be in abc state, therefore xyz function is OK to call now."

Related Topics

Parent Topic: ASP and NetWare

ASP Function List

Table auto. AppleTalk Services ASP Functions

Function	Purpose
ATAspAttn	Sends an attention request to the remote WSS (workstation session socket).
ATAspClose	Closes the session
ATAspCmdReply	Sends a command reply.
ATAspGet	Reads an event from the ASP protocol.
ATAspGetParms	Allows a client to find particular constants befpre a session is active
ATAspGetWorkstation	Returns the AppleTalk address of the workstation to which this file descriptor is connected.
ATAspOpen	Opens a file descriptor for reading and writing, but does not bind a session to this file descriptor.
ATAspSendStat	Replies to a GetStatus request received through an **ATAspGet** function.
ATAspSetStat	Sets the status string on an SLS (session listening socket) when an ATAspGetStatus packet is received.
ATAspSLSInit	Makes the specified file descriptor into an SLS (session listening socket)
ATAspSLSReject	Rejects a request to open a connection.

ATAspSSSInit	Creates a new SSS (server session socket) when a request comes in.
ATAspWCont	Sends a write continue request.
ATAspWReply	Sends a write reply.

ATAdspOpt_t (adsp.h)

```
typedef struct ATAdspOpt {
   char    Major;
   char    Minor;
   char    Revision;
   unsigned char    reserved1;
   unsigned short   TransThresh;          /* Obsolete */
   unsigned         TransTimerIntrvl;     /* Obsolete */
   unsigned char    reserved2[118];
} ATAdspOpt_t;
```

Major	AppleTalk interface major version number. See Synopsis of the Structures' Functions for details.
Minor	AppleTalk interface minor version number. See Synopsis of the Structures' Functions for details.
Revision	AppleTalk interface revision version number. See Synopsis of the Structures' Functions for details.

IMPORTANT: Please note that you must retrieve the defaults before changing any values.

Parent Topic: AppleTalk TLI Structures

ATDdpOpt_t (ddp.h)

You can set five fields in this structure, the DDP options checksum and data type and the AppleTalk version number. Checksum and data type are described below. See Synopsis of the Structures' Functions for an explanation of version number fields.

The t_unitdata and t_uderr structures have an *opt* field, which lets you access protocol options. *opt* points to a netbuf structure, which contains the **buf* field, which in turn points to the ATDdpOpt_t structure:

```
typedef struct ATDdpOpt {
   u_short    checksum;       /* optional */
```

```
    u_char      ddp_type;       /* required */
    char        major;          /* a read-only option */
    char        minor;          /* a read-only option */
    char        revision;       /* a read-only option */
} ATDdpOpt_t;
```

checksum	Enables the sending client to indicate whether DDP should calculate a checksum for a packet it sends out. The DDP header contains a checksum field, in which the value of the checksum appears.
ddp_type	The DDP header contains a DDP type field, an unsigned byte field that identifies the protocol contained in the data section of the datagram.
major	AppleTalk interface major version number. See ATAdspOpt_t (adsp.h) and Synopsis of the Structures' Functions for further explanation.
minor	AppleTalk interface minor version number. See ATAdspOpt_t (adsp.h) and Synopsis of the Structures' Functions for further explanation.
revision	AppleTalk interface revision version number. See ATAdspOpt_t (adsp.h) and Synopsis of the Structures' Functions for further explanation.

A nonzero value in the *checksum* field of the ATDdpOpt_t structure causes DDP to generate a checksum when a transport client sends a datagram. If the *checksum* field of a DDP datagram is nonzero when the datagram arrives at its destination, DDP verifies the checksum. If the checksum is valid, DDP passes the packet on to the next protocol layer. If the checksum is in error, DDP discards the packet.

The checksum option also enables a receiving client to verify whether the sender has used a checksum. Therefore, if a transport client receives data with **t_rcvudata**, and the *checksum* field of ATDdpOpt_t is nonzero, it is considered "polite" to send subsequent datagrams using a checksum.

If the *checksum* field of ATDdpOpt_t is 0, DDP does not calculate a checksum. The default value is 0.

The checksum option controls the use of checksums only on send, and does not automatically affect the use of checksums on other transport endpoints. The use of checksums is not universal, because it does add some overhead to the handling of a packet. However, we recommend the use of checksums in large internets where packets may become corrupted when crossing multiple repeaters or routers.

> **NOTE:** Even if you do not specify the use of checksums in this way, they may be turned on for the AppleTalk stack as a whole. This is true for DDP and ADSP.

Apple Computer, Inc. has established the following universally known values for the *data_type* field of the DDP packet:

0	Invalid (do not use)
1	Routing Table Maintenance Protocol (RTMP) response or data packet
2	Name Binding Protocol (NBP) packet
3	AppleTalk Transaction Protocol (ATP) packet
4	AppleTalk Echo Protocol (AEP) packet
5	RTMP request packet
6	Zone Information Protocol (ZIP) packet
7	AppleTalk Data Stream Protocol (ADSP) packet

Values 8 through 255 are valid but not universally known. Apple Computer, Inc. has reserved values 1 through 15 for its own use.

When a transport client receives a datagram with **t_rcvudata**, the *ddp_type* field of ATDdpOpt_t is set to the DDP type field in the DDP header.

When a client sends a datagram with **t_sndudata**, the DDP type field of the DDP header is set to the value specified by the *ddp_type* field of ATDdpOpt_t.

The default value is 128.

Parent Topic: AppleTalk TLI Structures

ATInet_t (appletlk.h)

For information about these address designations, see *Inside AppleTalk*.

```
typedef u_short   ATNet;
typedef u_char    ATNode;
typedef u_char    ATSocket;
typedef struct ATInet {
   ATNet       net;
   ATNode      node;
   ATSocket    socket;
} ATInet_t;
```

net	Network (cable) number
node	Node number
socket	Socket number

IMPORTANT: Remember that this structure is in Intel byte order.

Parent Topic: AppleTalk TLI Structures

ATP

ATP operates at the transport layer of the OSI model. It adds reliability to lower-layer services by providing loss-free delivery of packets from a source socket to a destination socket.

Central to ATP is the concept of a **transaction**. One socket client, called a **requester**, requests a service from another socket client using a Transaction Request (TReq) packet. The target socket client, called the **responder**, executes the request and reports the outcome to the requester in one or more Transaction Response (TResp) packets. The interaction between the requester and the responder constitutes the transaction.

Because the requester can send out several TReq packets, the socket client must have a way of distinguishing which response corresponds to each request. To identify each unique request, the requester generates a 2-byte **transaction identifier** (TID) in the header of the TReq packet.

Further, because the responder can send out multiple TResp packets as part of a single TResp message, each TResp packet contains a **sequence number** to identify the sequential position of the packet in the TResp message. The ATP header provides a 1-byte **bitmap/sequence** field to hold this sequence number. The value of this field in the TResp packet is an integer in the range 0 through 7, because up to 8 TResp packets can constitute a single TResp message.

In a TReq packet, the bitmap/sequence field contains the **transaction bitmap**. The transaction bitmap indicates the number of buffers reserved by the requester for the packets that make up the TResp message. The transaction bitmap also indicates which packets in a message the requester has received. The responder can examine the TReq packet's bitmap and determine the number of TResp packets the requester is expecting.

If a TReq is lost in the internet, or if a TResp is delayed in transit, the transaction process must have a way of recovering. ATP uses two recovery methods, each one corresponding to a different transaction type. The transaction types are these:

At-least-once (ALO) transaction

Exactly-once (XO) transaction

An ALO transaction implements a recovery mechanism that ensures a request is executed **at least once**. This recovery mechanism is appropriate for a request that can be executed more than once without serious

consequences, such as a request for a node to identify itself.

An XO transaction implements a recovery mechanism that ensures a request is executed **only once**. This recovery mechanism is appropriate for a request that cannot be executed more than once without causing serious consequences, such as a request to open a file.

In an XO transaction, the responder uses a **transactions list** in order to filter out duplicate requests. Each request is time-stamped when it enters the list. When a requester receives a TResp packet showing that the request was serviced, the requester should send a **Transaction Release** (TRel) packet to the responder. The TRel packet signals the responder to remove the request from the transactions list. If the TRel packet is not sent, or if it is lost in transit, the responder still has a mechanism for removing an obsolete request from the list. The responder checks the list periodically, and removes any request that has been in the list longer than the time specified by the **release timer**.

For a full description of the ATP functions, see AppleTalk: Functions.

Parent Topic: AppleTalk Overview

ATP and NetWare

Inside AppleTalk provides the definitive explanation of the ATP protocol. This discussion briefly describes the NetWare interface and how it differs from the one suggested in *Inside AppleTalk*.

ATP is a reliable, connectionless, transaction-oriented protocol. The basic semantics of ATP are the **request** and the **response**. ATP would be used in a connectionless environment where reliability is necessary or where the work being done is transaction-oriented. In some custom environments, the relative simplicity of implementing ATP on another platform could lead to choosing ATP over other protocols.

ATP is the foundation of the ZIP, ASP, and PAP protocols. If the ASP and PAP protocols do not fill your exact needs, it would not be difficult to create your own session-oriented protocol over ATP.

Related Topics

XO or ALO?

Opening a File Descriptor for ATP

About Receiving Data in General

Sending Requests in ATP

Sending Replies in ATP

About Transaction IDs

ATP Function List

Table auto. AppleTalk Services ATP Functions

Function	Purpose
ATAtpCancelRecvReq	Allows you to terminate a received XO request prematurely.
ATAtpCancelSendReq	Allows a request transaction generated by the local endpoint to be terminated earlier.
ATAtpClose	Closes a previously opened socket that this file descriptor is associated with.
ATAtpGet	Retrieves the next event from the protocol client.
ATAtpOpen	Opens a socket that listens for packets from the file descriptor that is returned by this function.
ATAtpSendReq	Sends a request to a remote endpoint. Its answers return in **ATAtpGet**.
ATAtpSendRsp	Sends a response to a remote requesting endpoint.

Blocking in NBP

Although it would have been possible for Novell to write a nonblocking NBP interface, it is the basic nature of NBP to wait for a certain amount of time. Therefore, all NBP functions are blocking (threads are sent to a queue) while they are waiting to receive information. Because an NBP lookup has to wait until the retry is exhausted before returning results (if the buffer hasn't filled up before that), this could be a significant amount of time.

> **WARNING: Beware of a problem inherent in a blocking interface, that is, the issue of who is able to close file descriptors. For example in Netware, if an NLM is unloaded while one of these functions is in progress, CLIB cleans up the resources in use, including the file descriptors. If this is not desirable, your NLM must not let itself be unloaded while an NBP function is in progress.**

The NBP functions described in AppleTalk: Functions are approximately (but not exactly) equal to UNIX SVR4* functions and the last version of this AppleTalk interface, v1.3.

Parent Topic: NBP and NetWare

Client-End Connection: Opening a WSS

PAP is based on CLIB, and all sessions are referred to with CLIB file descriptors, which lead to STREAMS underneath. Therefore, first you must call **ATPapOpen** to get a file descriptor.

Next, to attempt to open a connection, call **ATPapConnect** and pass in the following data: the file descriptor (*fd*), the AppleTalk address to connect to (**addr*), the waittime value (*waittime_waited*), and the time, in seconds, to attempt the connection (*retry_time*). The AppleTalk address is the address of the server, and is usually discovered with the NBP protocol. The *waittime_waited* value is a number of seconds you give to the server that the server uses to enforce fairness among those trying to connect. The principle is that those who have waited the longest time should be connected first; hence, requests with the highest *waittime_waited* values are connected before those with lower values. Set this value to the total length of time, in seconds, that you have been attempting to connect.

Next, call **ATPapGet** with a 256-byte buffer, waiting for a PAP_EVENT_OPENSESS_REPLY event. If you don't want to block your thread while waiting for this reply, use the CLIB **poll** function to determine when the file descriptor is readable. The *pass->ret* field, returned by **ATPapGet**, contains whether the connection attempt succeeded or failed, and the status message is placed in the buffer. If the function succeeded, you now have an open WSS and can continue into the data transfer phase. If the function failed, you must decide whether to continue attempting to connect, in which case you would either repeat the first part of the connection process, incrementing the *waittime* field with the amount of time you've been waiting, or give up and report an error to your user.

Finding the status of the server is part of establishing connections. Many implementations display the status to the user while attempting to connect. To get the status, call **ATPapGetStatus** on any available file descriptor. Often it is reasonable to use the file descriptor you have opened for the connection. If you are attempting to find the status on multiple printers, they can all share the same file descriptor. Use **ATPapGet** with a buffer of 256 bytes to receive the PAP_EVENT_GETSTAT_REPLY message. The buffer is filled with the status. You can de-multiplex among many outstanding replies by looking at the *pass->U.GetStat.id* and *pass->U.GetStat.addr* fields.

Parent Topic: Opening a Connection in PAP

Closing a Connection in PAP

If you want to close a connection, you might want an orderly close, which waits for all data written into PAP to be written to the remote endpoint, or you might want to abort the connection. If you want an orderly close, call

ATPapDisconnect. After this, **ATPapWrite** is no longer be valid even though PAP_EVENT_DATA messages can continue to come in. When all data has been written to the remote endpoint, the PAP_EVENT_DISCONN event flows up, indicating that it is safe and desirable to call **ATPapClose**. To effect an abortive close, simply call **ATPapClose**.

Parent Topic: PAP and NetWare

DDP

DDP operates at the network layer of the OSI model. It provides best-effort delivery of data across an AppleTalk internet.

DDP provides each protocol client in a node with an addressable entity known as a **socket**. Protocol clients can associate themselves with one or more sockets within their nodes and exchange packets through them. The packets exchanged through this DDP service are called **datagrams**.

Using the services of TLI, you can access DDP to carry out these tasks:

Create a transport endpoint (file descriptor)

Bind the transport endpoint to an AppleTalk address

Send datagrams

Receive datagrams

Receive an error associated with a sent datagram

Unbind

Close a transport endpoint

For details on using native functions to access DDP, seeDDP.

For information on using TLI functions to access DDP, seeTLI Functions: AppleTalk Notes.

Parent Topic: AppleTalk Overview

DDP and NetWare

ATDdpOpen now returns the file descriptor as a parameter, not as the return code from the function.

ATDdpOpen no longer allocates the socket. This is done with **ATDdpRegisterListener**.

Packets sent must have the *src_socket* field filled in properly.

Although the send routine has been changed from write to **ATDdpWrite**, write still works.

Related Topics

Opening a File Descriptor for DDP

Registering Listeners

Sending DDP Packets

Receiving DDP Packets

DDP Function List

Table auto. AppleTalk Services DDP Functions

Function	Purpose
ATDdpClose	Closes a socket opened by **ATDdpOpen**.
ATDdpDeregister Listener	Removes a listener on the specified file descriptor.This file descriptor stops receiving all DDP packets destined to that socket.
ATDdpNetinfo	Gets information about this interface.
ATDdpOpen	Opens a file descriptor on the local host for reading and writing but does not bind an AppleTalk socket to this file descriptor.
ATDdpRead	Reads a packet.
ATDdpRegisterListener	Allows a DDP client to begin listening on a particular socket.
ATDdpWrite	Sends a packet.

Differences in NBP between Versions 1.x and 2.0

The NBP interface has four new functions. For the functions that already existed, the changes are minimal in that they concern a small number of parameters. These changes are summarized for you below.

ATRetry_t Structure: First, the ATRetry_t structure has changed. An additional field has been added, *backoff.* It allows you to increase the interval between successive retries. And the interval between retries is now measured in milliseconds, not seconds.

Here is the new structure:

```
typedef struct ATRetry {
    u_long   interval;        /* Retry interval in
                                 milliseconds */
    short    retries;         /* Maximum number of
                                 retries */
    u_char   backoff;         /* Retry backoff, must
                                 be 0 through 4 */
} ATRetry_t;
```

However, all the functions that use the ATRetry_t structure allow the passing in of 0 instead of a pointer to the retry structure. If you pass in 0, everything still works. For a discussion of this structure, see ATRetry_t.

The more flag is used to control whether the function returns as soon as the buffer fills or waits until max+1 replies return. Input of 1 means to wait. more returns whether the buffer overflowed or not; 1 means the buffer overflowed. The **ATNbpLookup** function uses a ATRetry_t structure. Those changes are as noted above.

Parent Topic: NBP and NetWare

Fastwrite

Fastwrite is an option that simplifies the ASP write dialogue. The **ATAspWCont** function need not be called. You activate this option with the **ATAspSSSInit** function, on a per-session basis.

The only data in **ATAspWCont** is the size of the buffer expected back, but normally this is the ASP quantum size. Many clients may want to wait for the ASP_EVENT_WRITE_CONT data before committing to success or failure of the transaction, and thus rarely call **ATAspWReply** after the ASP_EVENT_SSS_WRITE_REQ. For these applications, Novell® offers Fastwrite.

The normal flow for the server side of a write transaction follows this sequence:

1. Get an ASP_EVENT_SSS_WRITE_REQ
2. Call **ATAspWCont**
3. Get an ASP_EVENT_SSS_WRITECONT_REPLY
4. Call **ATAspWReply**

With Fastwrite, the **ATAspWCont** must not be called. The size of the buffer is automatically sent to the workstation as the ASP quantum size. The **ATAspWReply** must be called after the ASP_EVENT_SSS_WRITECONT_REPLY arrives and any error can be indicated then.

Also note that since two events arrive at the same time, it is possible to receive them out of order in NetWare if there are multiple threads servicing the same file descriptor.

ASP and NetWare

How to Determine Structure Sizes for AppleTalk

You can get ADSP- or DDP-specific information by calling **t_open**, which returns a t_info structure. This structure contains the sizes of the address and options structures for each protocol.

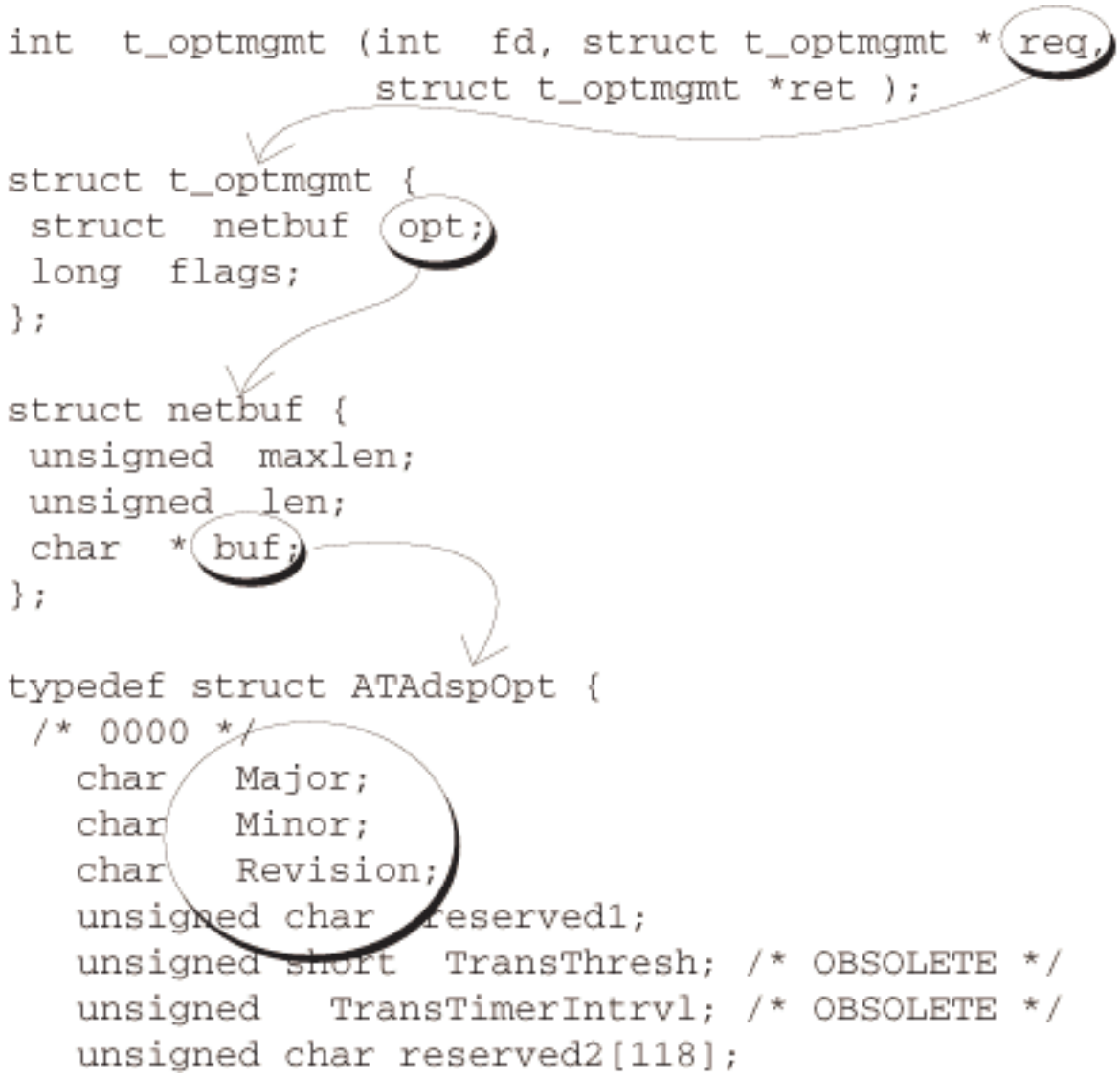
Parent Topic: Synopsis of the Structures' Functions

How to Determine the ADSP Interface Version

Because **t_optmgmt (Function)** doesn't involve any network traffic, it allows you to take care of version management prior to launching into the normal work of your NLM. This explanation uses the ATAdspOpt_t structure but the ATDdpOpt_t structure works the same way.

As shown in the following figure, the **t_optmgmt** function points to two t_optmgmt (Structure) structures, *req* and *ret*. The t_optmgmt structures include a netbuf structure and a *flags* field. The netbuf structure contains a pointer to an ATAdspOpt_t structure (adsp.h).

Figure 3. t_optmgmt Structures



The major, minor, and revision version numbers are defined as shown in the following figure.

Figure 4. Version Numbers

The major number of the ADSP interface is "revved" (increased in value) when a release contains one or more new functions, the minor number is revved when options have changed, and the revision number is revved when the size of an options buffer has changed.

The *flags* field of t_optmgmt (Structure) gives you the choice of negotiating an option, checking compatibility, or retrieving the default option. These

settings are defined in the **t_optmgmt (Function)** description as follows:

T_NEGOTIATE---This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider evaluates the requested options and negotiates the values, returning the negotiated values through *ret*.

T_CHECK---This action enables the user to verify whether the options specified in req are supported by the transport provider. On return, the flags field of *ret* has either T_SUCCESS or T_FAILURE set, to indicate to the user whether the options are supported. These flags are only meaningful for the T_CHECK request.

T_DEFAULT---This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero and the *buf* field may be NULL.

Because the version is implemented as a read-only option, the first choice is not useful. The second choice returns T_SUCCESS only if the version numbers match exactly. Therefore, use the T_DEFAULT choice.

In the netbuf of *req*, set *len* to zero and *buf* to NULL. When the function completes, *ret* still has the T_DEFAULT flag set, *len* is nonzero, and *buf* points to a char* buffer that you have to cast to the type structATAdspOpt (see adsp.h) in order to access its contents. (STREAMS and TLI are unaware of the type of information in the buffer; therefore, you need to type cast it to separate the data into the fields of ATAdspOpt_t.) Compare the *Major* and, if desired, the *Minor* and *Revision* fields returned with the following `#define` values in adsp.h:

```
#define     ADSP_MAJOR 2
#define     ADSP_MINOR 0
#define     ADSP_REVISION 0
```

Parent Topic: Synopsis of the Structures' Functions

Internationalization and AppleTalk

Internationalization is a pertinent issue in all the AppleTalk protocols, especially in the NBP and ZIP protocols, where the strings are part of the actual protocol (object and zone names). The status strings of ASP and PAP must also be considered.

Even though AppleTalk is built to transport byte strings without making sense of them, in most cases the lower 7 bits of ASCII will be valid. Therefore, special characters like * and = will be noticed correctly.

The decision of which character set to use to send out strings falls to the API client. In most cases it will be obvious that the character set called MacASCII should be used. It is specified in *Inside AppleTalk*. The other commonly used character set is MacKanji, a SHIFT-JIS implementation.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

Loaded But Unbound Stack

In this version of the interface, it is possible for the stack to be active but not yet bound to a net and node number when higher-level services try to use it. The services are allowed to bind and open sockets and file descriptors anyway, even though the stack is not bound to any node. DDP is in charge of issuing error messages to the system console, stating that services are registered but the stack is not bound.

Similarly, APPLETLK.NLM can now be directly on the wire; therefore, its node and net number can change. When the net number alone changes, the services will not be notified. When the node number changes, the stack will become unconfigured, then reconfigured again. This is the same as unbinding and then rebinding from the console. Upper-level services will be notified by their protocol layer. In general, listeners will not close, and will keep whatever NBP registration they had made in hopes that the situation is transitory. Open sessions will close, most likely with an ENETDOWN error.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

Loading ADSP.NLM

To use ADSP, you must have ADSP.NLM and TLI.NLM loaded. ADSP.NLM contains the AppleTalk Data Stream Protocol (ADSP), so in order for your application to access ADSP, you must load ADSP.NLM **before** you load your application. TLI.NLM contains the TLI interface functions that access ADSP.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

Loading APPLETLK.NLM

APPLETLK.NLM contains the AppleTalk protocol stack and the AppleTalk router. In order for your application to access the AppleTalk protocol stack, you must load APPLETLK.NLM **before** you load your application.

Further, because you must load APPLETLK.NLM with a configuration specific to each network on which it runs, your NLM should never "autoload" APPLETLK.NLM using the NLM link utility's MODULE command.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

NBP

When you develop an NLM to provide a service to Macintosh clients, you must register the service with the internet. The Name Binding Protocol (NBP) exists for this purpose.

Central to an understanding of NBP is the concept of a **network-visible entity**, or **NVE**. An NVE is any entity accessible over an AppleTalk internet. Each NVE has an **entity name**. An entity name consists of three fields---object name, type, and zone.

Each node maintains a **names table** containing name-to-internet address mappings of all entities in that node; these mappings are known as **name-address tuples**. The **names directory** is a distributed database of all the name-to-address mappings in the AppleTalk internet.

NBP uses the names directory to perform **name binding**; that is, NBP uses the names directory to map each entity name to its AppleTalk internet address. Using NBP, a socket providing services can register its entity name in the names table, or delete its name from the names table. A client can use NBP to look up a server's AppleTalk internet address before sending a request for service.

For a full description of the NBP functions, see AppleTalk: Functions.

Parent Topic: AppleTalk Overview

NBP and NetWare

See the following for information about NBP:

Differences in NBP between Versions 1.x and 2.0

NBP Structures

Blocking in NBP

NBP Function List

Table auto. AppleTalk Services NBP Functions

Function	Purpose
ATNbpConfirm	Verifies that the network address of the specified NBP entity is still valid.
ATNbpDirected	Directs a lookup query to a particular node.

ATNbpDirected Lookup	Directs a lookup query to a particular node.
ATNbpLookup	Searches of the names directory and returns a mapping of the specified entity's name to its internet address(es).
ATNbpMakeEntity	Creates a complete three-part NBP entity.
ATNbpMakeEntityXlate	Has the same uses as **ATNbpMakeEntity**, but also translates the incoming object, type, and zone strings from the local code page to the MacASCII code page.
ATNbpParseEntity	Converts a string from the "object:type@zone" form to the three-part NBP entity form.
ATNbpRegister	Registers a service by file descriptor so that it becomes visible on the internet.
ATNbpRegByAddr	Registers a service by its AppleTalk socket number alone.
ATNbpRemove	Removes an NBP registration from the local names table
ATNbpRemoveByAddr	Removes an NBP registration from the local names table.

NBP Structures

NBP has defined three data structures, an **NVE** (network visible entity), an **entity**, and a **tuple**. All these are discussed at the beginning of this chapter under "Structures."

An **NVE** is a single network string defined as follows:

Uses the MacASCII character set

Is less than or equal to 32 characters in length

Is length-preceded (P string)

An **entity** is a set of three of these strings, the object, type, and zone strings. This entity is also thought of as a service "name," and is unique across an internet. An **NBP tuple** is the combination of the entity name and the AppleTalk internet address that corresponds to the net,node,socket where that name is.

Parent Topic: NBP and NetWare

netbuf (tiuser.h)

In TLI, netbuf structures are buffers that are used for passing three types of information: addresses, options, and data. Both DDP and ADSP use the netbuf structure for addresses and options, but only DDP uses it for data. (ADSP uses function parameters.)

```
struct netbuf {
   unsigned    maxlen;
   unsigned    len;
   char        *buf;
};
```

maxlen	Maximum bytes the buffer can hold
len	Number of valid bytes of data in the buffer
buf	Points to an options structure, ATDdpOpt_t or ATAdspOpt_t

There can be two types of data, the type that is passed when connecting or disconnecting and the type that is passed when sending or receiving (**transferring**) data. AppleTalk only allows the second type.

Since DDP is a connectionless protocol, it relies upon having a maximum length of data that can be transferred. Hence it uses netbufs to transfer data. For transferring data, DDP uses the t_unitdata structure, which includes three netbufs:

```
struct t_unitdata {
   struct netbuf    addr;
   struct netbuf    opt;
   struct netbuf    udata;
};
```

To allocate netbufs, call **t_alloc**, which figures out the right maxlen for you. It allocates a buffer of size maxlen, sets maxlen to this length, and sets buf to point to the buffer. You need only fill in the *len* field to indicate the number of valid bytes in the buffer. With addresses and options, *len* usually is the same each time, but with DDP data, *len* could vary.

For all functions using addr, the data referenced by addr *buf* is in the structure ATInet_t (defined in appletlk.h). Therefore, to interpret the address, cast addr *buf* to (ATInet_t *).

On the other hand, AppleTalk did not set a maximum amount of transfer data on ADSP; connection-oriented protocols commonly have no maximum. Hence, TLI provides direct data-length parameters in its connection-mode send and receive functions.

Parent Topic: AppleTalk TLI Structures

Notes on the AppleTalk Interface, both TLI and Native

Important issues to be aware of are summarized in the following:

AppleTalk Version Functions

Loaded But Unbound Stack

AppleTalk Byte Ordering

Referencing Zones

Referencing Names

Loading APPLETLK.NLM

Loading ADSP.NLM

The Internal AppleTalk Network

AppleTalk Internet Addresses

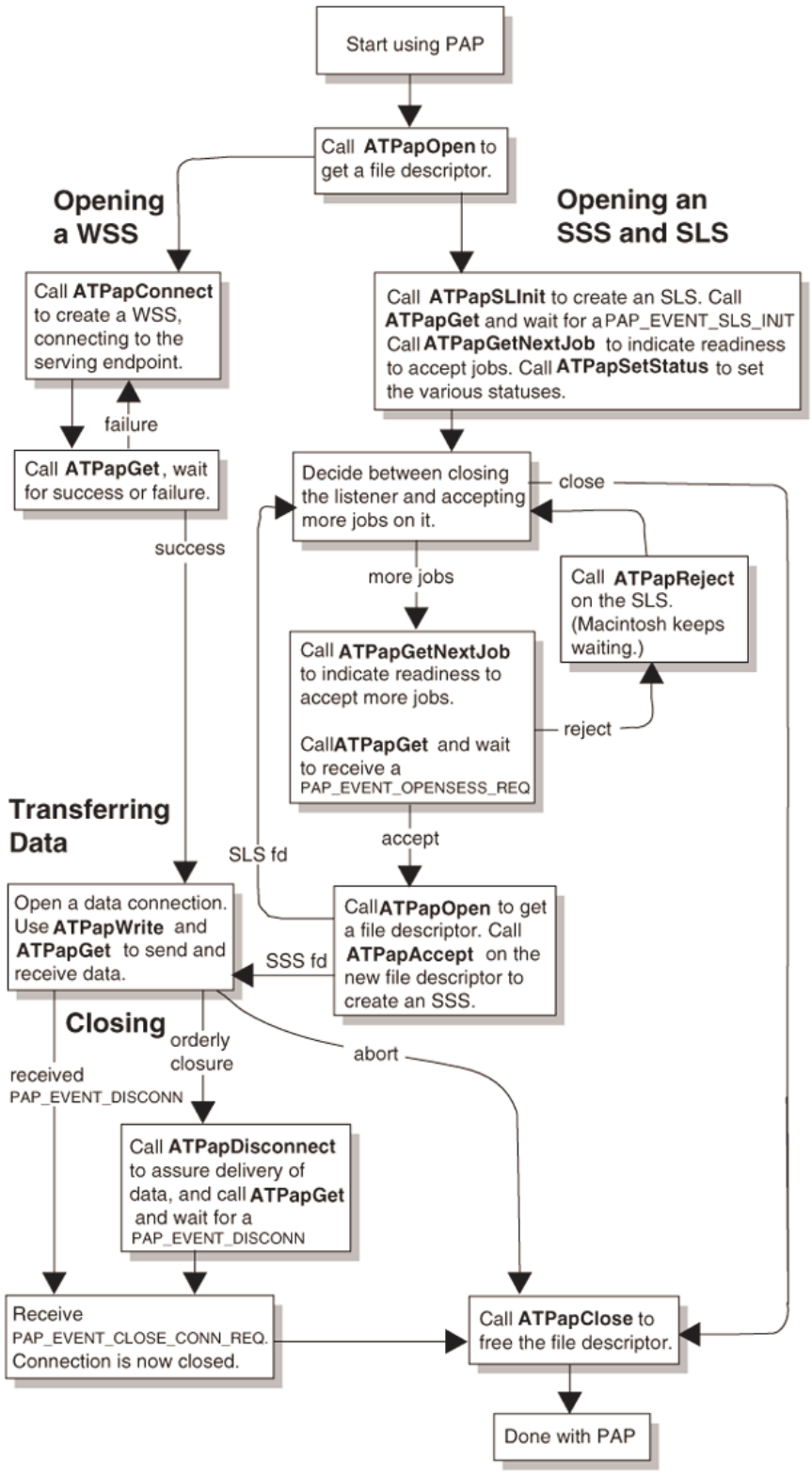
Internationalization and AppleTalk

Parent Topic: AppleTalk Overview

Opening a Connection in PAP

To open the connection dialogue, you open either a **Workstation Session Socket** (WSS) or, on the server end, a **Server Session Socket** (SSS). This discussion covers opening a WSS first, then an SSS. You can locate the points in the discussion on the following figure. Unfortunately, the details of sending and receiving data, as well as sending and receiving **ATPapGetStatus** requests, are not shown in the figure. Nevertheless, these procedures are outlined in PAP State Tables. You can refer to these tables as well.

Figure 5. PAP Flowchart


Start using PAP
Call ATPapOpen to get a file descriptor.
Opening a WSS
Opening an SSS and SLS
Call ATPapConnect to create a WSS, connecting to the serving endpoint.
failure
Call ATPapGet, wait for success or failure.
success
Call ATPapSLInit to create an SLS. Call ATPapGet and wait for a PAP_EVENT_SLS_INIT Call ATPapGetNextJob to indicate readiness to accept jobs. Call ATPapSetStatus to set the various statuses.
Decide between closing the listener and accepting more jobs on it.
close
more jobs
Call ATPapReject on the SLS. (Macintosh keeps waiting.)
Call ATPapGetNextJob to indicate readiness to accept more jobs.
CallATPapGet and wait to receive a PAP_EVENT_OPENSESS_REQ
reject
accept
Transferring Data
SLS fd
Open a data connection. Use ATPapWrite and ATPapGet to send and receive data.
CallATPapOpen to get a file descriptor. Call ATPapAccept on the new file descriptor to create an SSS.
SSS fd
Closing
orderly closure
abort
received PAP_EVENT_DISCONN
Call ATPapDisconnect to assure delivery of data, and call ATPapGet and wait for a PAP_EVENT_DISCONN
Receive PAP_EVENT_CLOSE_CONN_REQ. Connection is now closed.
Call ATPapClose to free the file descriptor.
Done with PAP

Related Topics

Client-End Connection: Opening a WSS

Opening an SSS

Parent Topic: PAP and NetWare

Opening a File Descriptor for ATP

Since ATP is not a connection-oriented protocol, the concept of "opening an ATP session" does not apply. Instead, requests and responses are sent to and from a particular **file descriptor**. Each file descriptor corresponds to one socket. To open a file descriptor, call **ATAtpOpen**. A CLIB file descriptor and DDP socket is allocated for your use. The file descriptor is now ready for sending and receiving requests and responses. You probably want to register this socket with NBP as a service provider.

Parent Topic: ATP and NetWare

Opening a File Descriptor for DDP

ATDdpOpen no longer allocates a socket. It opens a file descriptor for reading and writing but does not bind a socket to that file descriptor.

In addition, **ATDdpOpen** now returns the file descriptor as a parameter (fd), not as the return code of the function.

Parent Topic: DDP and NetWare

Opening an SSS

On the server end, connections are listened for and the process is more involved. To prepare to receive connections, you first create a listening socket and establish how many connections you are willing to accept. After receiving a connection request, you create an SSS for it.

Opening a Session Listening Socket (SLS): Like most protocols, PAP establishes a listener socket to receive connection requests. This socket is known as the **Session Listening Socket**, or SLS. To create an SLS, call **ATPapOpen** to get a file descriptor, then call **ATPapSLInit** to make that file descriptor into an SLS. Next call **ATPapGet**, waiting for the PAP_EVENT_SLS_INIT event to indicate success or failure of the initialization. If it succeeded, call **ATPapSetStatus** to set the values of the

two automatic statuses: one that is generated in response to a status inquiry and one that is generated in response to a connection request. Now that the SLS has been initialized and is ready to accept connections, you probably should call NBP to register this service on the net.

When you are ready to accept connections, or **jobs**, call **ATPapGetNextJob** and indicate how many jobs you will accept. LaserWriter printers only accept one job at a time but, given the resources on a NetWare server, you might want to accept 5, 10, or many more jobs at a time. PAP keeps a count of the number of jobs you have indicated you will accept and, for each PAP_EVENT_OPENSESS_REQ event that arrives, decrements this counter by one. Thus, to avoid having a connection request rejected because of insufficient get-next-job credit on a particular SLS, be sure to continue increasing the credit by calling **ATPapGetNextJob**. To revoke this credit, just call **ATPapGetNextJob** with a negative value as num_jobs.

Now you are ready to call **ATPapGet** and wait for the PAP_EVENT_OPENSESS_REQ events to come up. Open session requests need to be serviced as they come in. As mentioned before, you have the option of accepting or rejecting them. To reject, call **ATPapReject** with the address and ID that was passed up in the *pass->U.ConnReq.addr* and *.id* fields. Also, specify the status string that should be included in this rejection. To accept, on the other hand, you must make ready an SSS; **accepting a job means creating a new SSS**.

Creating an SSS: Call **ATPapOpen** to get a new file descriptor, then call **ATPapAccept** on the new file descriptor, with the address and ID fields mentioned above. This new file descriptor becomes the new SSS and is open for reading and writing in the data transfer phase.

As long as you want to continue accepting jobs, call **ATPapGetNextJob** and **ATPapGet**. Remember, as long as there is get-next-job credit available, you must keep checking with **ATPapGet** for PAP_EVENT_OPENSESS_REQ events. (PAP does not automatically check for you.) If you don't continually check, connection requests go unanswered and time out. After getting a PAP_EVENT_OPENSESS_REQ event, call either **ATPapAccept** or **ATPapReject**.

Parent Topic: Opening a Connection in PAP

PAP

PAP is a session-level protocol that enables communication between workstations and servers. It is a connection-oriented protocol that handles these functions:

Connection setup

Connection maintenance

Connection closure

Connection closure

Data transfer over the connection

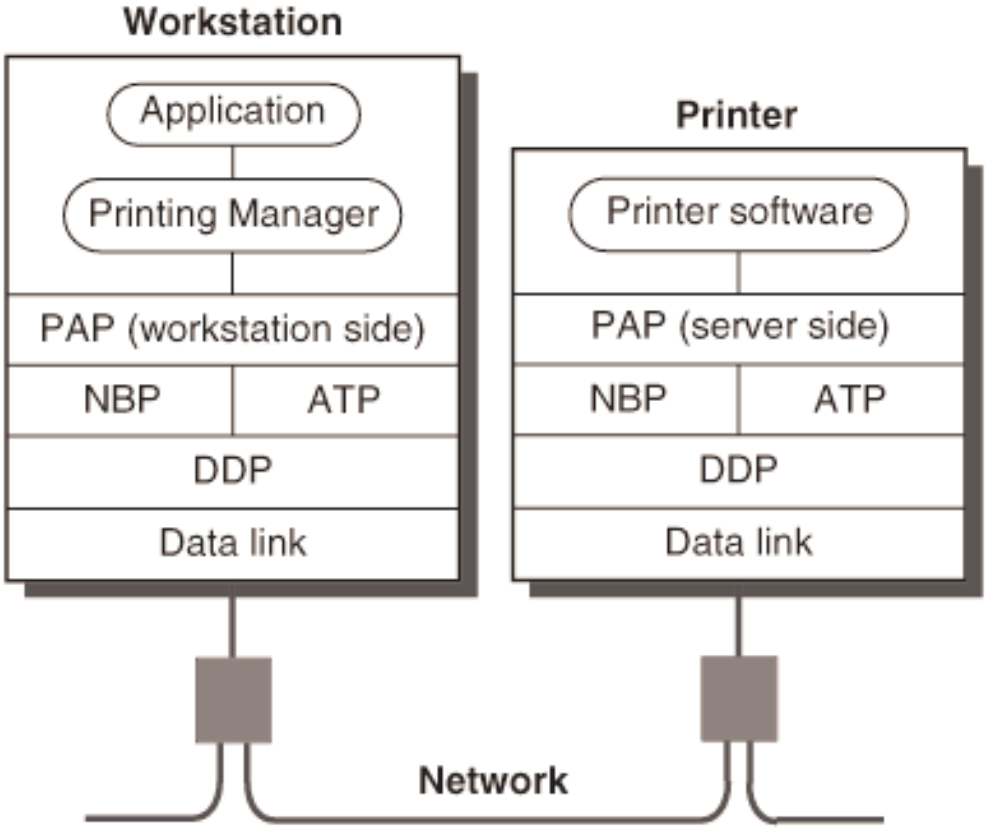
PAP allows multiple connections at both the workstation and server ends.

PAP envisions a server node as containing one or more processes that are accessible to workstations through PAP. In this chapter, these processes are referred to as **servers**. A server makes itself visible over the network by opening a **session listening socket** (**SLS**) on which it registers its name.

The use of the word **printer** in the name of this protocol is purely historical. Although the protocol was originally designed for the specific purpose of communication with print servers, it has no special features for printing and can be used by a wide variety of other kinds of servers.

The following figure illustrates the protocol architecture used for communication between a user's computer (workstation) and a print server on a network. PAP is a client of the Appletalk Transaction Protocol (ATP) and the Name Binding Protocol (NBP). Both of these protocols use the Datagram Delivery Protocol (DDP). PAP is an asymmetric protocol; the PAP code in the workstation is different from the PAP code in the printer.

Figure 6. PAP Protocol Architecture



> **NOTE:** Only the server-side functions are presented in this guide. Be sure to read the description of PAP in *Inside AppleTalk* for a complete introduction to PAP concepts and terms.

The commands and data sent through the PAP connection are printer-dependent. For example, for the LaserWriter* printer the dialog is in PostScript* format.

For a full description of the PAP functions, see AppleTalk: Functions.

Parent Topic: AppleTalk Overview

PAP and NetWare

PAP is a bidirectional, streaming byte protocol. Built for printers, it is very simple in its data transfer but includes a few features in the open sequence not seen in other protocols and an out-of-band means of gathering status. Novell offers both the client end (called "workstation side" in *Inside AppleTalk*) and server end of this protocol. After the open connection phase, the protocol becomes **symmetric**, that is, server sessions and workstation sessions are the same.

The following introduction to PAP discusses the open-connection dialogue first, then the data transfer phase, then the processes of closing a connection and shutting down a service. Although much of this information is covered in *Inside AppleTalk*, there are some important ways in which Novell's implementation diverges from the sample API discussed in *Inside AppleTalk*.

Related Topics

Opening a Connection in PAP

Transferring Data in PAP

Closing a Connection in PAP

Shutting Down a Service in PAP

About This PAP Interface

PAP State Tables

PAP Function List

Table auto. AppleTalk Services PAP Functions

Function	Purpose

ATPapAccept	Makes an SSS (server session socket) out of an unused file descriptor and links a connection request on the SLS (session listening socket) to this SSS.
ATPapClose	Destroys the file descriptor and frees all memory, closing any sessions open on this file descriptor.
ATPapConnect	A workstation-side (NLM requesting connection to a server) function that changes the specified new file descriptor into a WSS (workstation session socket).
ATPapDisconnect	Notifies PAP that you want to close the connection but want all data already sent with **ATPapWrite** to be delivered.
ATPapGet	Retrieves a message from the PAP protocol client.
ATPapGetNextJob	After initializing the server side, this function is called when the server side is ready to accept a job, to set the number of jobs the SLS (Server Listening Socket) is willing to accept.
ATPapGetStatus	A workstation function that finds out the status of any given remote PAP server endpoint.
ATPapOpen	Opens a PAP file descriptor.
ATPapReject	Called on the SLS (session listening socket) when a connection request comes in that you want to deny.
ATPapSetStatus	Sets the current status of the SLS after initialization.
ATPapSLInit	Makes the specified file descriptor into a server listening socket (SLS).
ATPapWrite	Sends data over the connection, which must have been established first.

PAP State Tables

Two types of PAP events are described in the following tables, the kind returned by PAP, which begin with PAP_EVENT, and the PAP library functions. The first table tells you which state your NLM is in when PAP events are received. The second table is a more dynamic, state transition table. It starts with a state and tells you how to transition to the next state, if appropriate.

Table auto. States and Corresponding PAP Events

Event	State
PAP_EVENT_CLOSECONN	Only while in data transfer (open) state on WSS or SSS

PAP_EVENT_GETSTAT_REPLY	Only when you have an outstanding **ATPapGetStatus** request. Since the get-status request and reply are out of band, this event could occur in any state if you have called **ATAtpGetStatus**
PAP_EVENT_SLS_INIT	Only after you have called **ATPapSLInit** on a file descriptor. No other event comes up before this one comes up.
PAP_EVENT_OPENSESS_REPLY	Only when you have an **ATPapConnect** outstanding on this file descriptor, thus, in the WSS-opening state
PAP_EVENT_DATA	Only when you are in the data transfer (open) state on a WSS or SSS
PAP_EVENT_CONN_DIED	Only when you are in the data transfer (open) state on a WSS or SSS
PAP_EVENT_OPENSESS_REQ	While in the SLS listening state
PAP_EVENT_DISCONN	While in the closing state. This is after calling **ATPapDisconnect** on a WSS or SSS.

Table auto. PAP State Transition Table

State/Event	**Next State**
Uninitialized	
ATPapSLInit	Opening SLS
ATPapConnect	Opening WSS
ATPapAccept	Open---Data Transfer. Must have an outstanding open connection request (**ATPapConnect**) on an SLS.
Listening on SLS The following events are received but do not cause a state transition; the state can only move to**ATPapClose**.	
ATPapSetStatus	
ATPapGetNextJob	
ATPapReject	
PAP_EVENT_OPENSESS_REQ	
Open, Data Transfer---SSS or WSS	
ATPapWrite	
ATPapDisconnect	Closing WSS or SSS
ATPapClose	
PAP_EVENT_DATA	
PAP_EVENT_CONN_DIED	Go to Closed state.

PAP_EVENT_CLOSECONN_REQ	Go to Closed state.
Closing---WSS or SSS	
PAP_EVENT_DISCONN	Go to Closed state.
Opening SLS	
PAP_EVENT_SLS_INIT	Go to Listening SLS state.
Opening WSS	
PAP_EVENT_OPENSESS_REPLY	Go to Open if no error, closed if error.
Closed	
Call **ATPapClose** to dispose of file descriptor.	

Parent Topic: PAP and NetWare

Receiving DDP Packets

To receive a DDP packet on an open socket, use **ATDdpRead**.

Parent Topic: DDP and NetWare

Referencing Names

On the other hand, the human-readable names of network entities, such as those returned by **ATNbpLookup**, are C strings and are referenced as an array. For more information on referencing names, see ATEntity_t.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

Referencing Zones

Human-readable AppleTalk zones, such as those returned by **ATZipGetZoneList** and **ATZipGetNBPZones**, are returned in the **packed** format described in *Inside AppleTalk*. This format is a list of ATNveStr_t structures. Each of these structures is a **P string**, which has a different format than the **C string** you may have expected. A P string is a Pascal-formatted string; it is length-preceded rather than being NULL-terminated as C strings are. The following figure depicts a list of zones as a series of P strings in packed format.

Figure 7. Packet Format of Zone List

Each P string consists of two parts:

1. A length byte
2. Data that is **length byte** long

Remember, these are lists, not arrays. Therefore, do not try to reference the zones in the list as you would an array. For more information on referencing zones, see ATNveStr_t.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

Registering Listeners

In order to receive packets on a socket, the client must register listeners. Use **ATDdpRegisterListener** to allocate a socket from which you can begin receiving and sending packets.

More than one listener can be opened per file descriptor, but then **ATNbpRegister** (fd) does not work.

Parent Topic: DDP and NetWare

Sending DDP Packets

You create a DDP packet for sending with the ATDdp_t structure. This structure contains fields for the DDP packet's extended header and data. The header consists of a checksum, the source and destination net,node,socket, and the protocol type. The fields in the ATDdp_t structure are explained in depth in the "AppleTalk Services" section in the *Structure Reference* book.

In keeping with the AppleTalk model, a DDP session allows you to send DDP packets anywhere, but you must also choose which socket to send a packet **from**. This is specified in the DDP packet by setting the src_socket

number. No checking is possible on this socket number, and no default is allowed.

> **IMPORTANT:** When you send a DDP packet, DDP overwrites the source network ID and source node ID of the address you specify with the correct net and node. It does not, however, overwrite the socket ID; you must specify the correct source socket number.

The ddp.h file contains a macro to access the length field of the DDP packet. DDP_LENGTH. DDP_LENGTH(ddp) returns the value in the length field of a DDP packet that you read or write. The ddp argument is a pointer to an ATDdp_t structure.

Sending DDP short or multicast packets is not allowed.

To send a DDP packet to an open socket, use **ATDdpWrite**.

Notice that the new function interface does not use STREAMS write. This is to allow the possibility of a transparent move away from STREAMS in the future, possibly the near future.

> **IMPORTANT:** Like other datagram protocols, DDP does not guarantee delivery. Likewise, when it is not able to deliver packets for some reason, DDP does not return errors (most of the time). This is true even when the error occurred in the server and, therefore, could be returned.

Parent Topic: DDP and NetWare

Sending Replies in ATP

Replies are sent using the **ATAtpSendRsp** function. The request that you are responding to has received in an **ATAtpGet** call. The most important fields in the ATAtpPass_t structure to fill when sending a reply are those listed below:

Table auto. Reply Fields in the ATAtpPass_t Structure

Purpose	Field
a. Extra four bytes that are provided by the ATP protocol in case the user needs to specify information. It is usually used for a connection ID, request type, sequence number, or other information of a similar kind.	
Matches the response to the request	*TransID*
Data you are sending in the response	*data* *data_len*
Four bytes of request data[a]	*usersdata*[0]

Parent Topic: ATP and NetWare

Sending Requests in ATP

Requests are sent using the **ATAtpSendReq** function. Set the information regarding the request in the ATAtpPass_t structure that you pass in. You send requests to an AppleTalk address (net, node, socket) that your application either determined with NBP or gleaned from a preceding transaction.

The fields in the ATAtpPass_t structure are explained in depth in the structure description. The most important fields to fill when sending a request are those listed below:

Table auto. Request Fields in the ATAtpPass_t Structure

Purpose	**Field**
a. Extra four bytes that are provided by the ATP protocol in case the user needs to specify information. It is usually used for a connection ID, request type, sequence number, or other information of a similar kind.	
Who you are sending the request to	*at_addr*
What data you are sending in the request	*data*, *data_len*
Four bytes of request data [a]	*usersdata*[0]
Whether you are request XO or ALO service. (XO is most common.)	*xo*

Parent Topic: ATP and NetWare

Shutting Down a Service in PAP

When you want to stop receiving jobs, there are two methods for shutting down a service, the first is abortive and the second is orderly. The first method is to close the file descriptor associated with the SLS with **ATPapClose**, which aborts all current jobs. Each SSS receives a PAP_EVENT_CLOSECONN_REQ event and **ATPapClose** must be called.

The second method is to de-register the SLS with NBP and wait until all SSS connections close. This probably is within a short period of time since each connection usually lasts for a single print job. After removing the entity from the network with NBP, deregister the SLS by calling **ATPapGetNextJob** with a negative value, revoking the ability of the SLS to accept jobs. It is a good idea to keep a count of all current SSSs so you can

make sure the count reaches 0 before actually closing the SLS with **ATPapClose**.

Parent Topic: PAP and NetWare

Synopsis of the Structures' Functions

The one structure that you must always use is ATAdspOpt_t. In conjunction with **t_optmgmt (Function)** and the t_optmgmt (Structure) and netbuf structures, it retrieves the version of the ADSP interface that is running on the server. Please note that the DDP interface version is retrieved by the **ATAppletalkVersion** function and may be different from the ADSP interface version.

Some of the structures described in this section are used to find out protocol-specific information like structure sizes (t_info).

DDP options can be set and retrieved when data is sent or received with the ATDdpOpt_t structure. The t_unitdata and netbuf structures also figure into the process of setting DDP options.

Last but by no means least, the ATInet_t structure is the three-part address of an AppleTalk internet entity (net,node,socket).

Related Topics

How to Determine the ADSP Interface Version

How to Determine Structure Sizes for AppleTalk

Parent Topic: AppleTalk TLI Structures

The Internal AppleTalk Network

The internal network has no physical components; it is entirely contained within APPLETLK.NLM.

The internal network is configured with an AppleTalk network number and zone name, just as any physical, external network. The internal network always supports two nodes: the protocol stack and the router. The stack is node #1 and the router is node #2 on the internal network. A service using the AppleTalk stack is logically part of the node on the internal network.

Parent Topic: Notes on the AppleTalk Interface, both TLI and Native

t_info (tiuser.h)

This structure is used by **t_open** and **t_getinfo** to return protocol-specific values to the application using the protocol. The values specify the size, and one specifies the type, of information handled by the protocol.

```
struct t_info {
   long    addr;
   long    options;
   long    tsdu;
   long    etsdu;
   long    connect;
   long    discon;
   long    servtype;
}; t_info;
```

addr	Maximum size of a transport address
options	Maximum bytes of protocol-specific options that may be passed between the transport user and transport provider
tsdu	Maximum message size that may be transmitted in either connection mode or connectionless mode
etsdu	Maximum expedited data message size that may be sent over a transport connection
connect	Maximum bytes of user data that may be passed between users during connection establishment
discon	Maximum bytes of user data that may be passed between users during the abortive release of a connection
servtype	Type of service supported by the transport provider (connection-mode or connectionless-mode service)

The following two tables list the information that t_info returns for DDP and ADSP.

Table auto. t_info Values for ADSP

Field	**Value**
addr	sizeof(ATInet_t)
options	sizeof(ATAdspOpt_t)
tsdu	-1 (no limit)
etsdu	570 bytes
connect	-2 (not supported)
discon	-2 (not supported)
servtype	T_COTS (connection-oriented service)

Table auto. t_info Values for DDP

Field	Value
addr	sizeof(ATInet_t)
options	sizeof(ATDdpOpt_t)
tsdu	586 bytes
etsdu	-2 (not supported)
connect	-2 (not supported)
discon	-2 (not supported)
servtype	T_CLTS (connectionless transport service)

Parent Topic: AppleTalk TLI Structures

TLI Functions: AppleTalk Notes

These notes cover issues specific to using TLI functions with the AppleTalk protocols. (For reference pages on the TLI functions, see TLI: Functions.) Notes are provided about the following:

AppleTalk Notes: t_accept (applies to ADSP only)

AppleTalk Notes: t_bind

AppleTalk Notes: t_close

AppleTalk Notes: t_connect (applies to ADSP only)

AppleTalk Notes: t_getinfo

AppleTalk Notes: t_listen (applies to ADSP only)

AppleTalk Notes: t_open

AppleTalk Notes: t_optmgmt

AppleTalk Notes: t_rcv (applies to ADSP only)

AppleTalk Notes: t_rcvconnect (applies to ADSP only)

AppleTalk Notes: t_rcvdis (applies to ADSP only)

AppleTalk Notes: t_rcvrel

AppleTalk Notes: t_rcvudata (applies to DDP only)

AppleTalk Notes: t_snd (applies to ADSP only)

AppleTalk Notes: t_snddis (applies to ADSP only)

AppleTalk Notes: t_sndrel

AppleTalk Notes: t_sndudata (applies to DDP only)

AppleTalk Notes: t_unbind

Parent Topic: TLI Implementation Notes for AppleTalk

TLI Implementation Notes for AppleTalk

> **IMPORTANT:** Because of the way TLI accesses NetWare® services, you must have a thorough understanding of the chapters on TLI in this manual. Treat this chapter as an appendix to those chapters.

This chapter discusses issues you should be aware of when using TLI functions to access DDP and ADSP.

The first section, Synopsis of the Structures' Functions, mentions all of the structures used by TLI and describes the ones that touch upon the following AppleTalk issues:

Determining which version of the AppleTalk ADSP or DDP interface is running on the local server with **t_optmgmt (Function)**

Retrieving protocol-specific structure sizes with the **t_open** and **t_getinfo** functions

Specifying DDP options, checksum and ddp_type. (The ADSP options, TransThresh and TransTimerInterval, have been disabled.)

The second section, ADSP Performance Notes, gives some advice on achieving higher performance using ADSP.

The third section, TLI Functions: AppleTalk Notes, discusses issues that arise from using TLI specifically with AppleTalk.

The fourth section, AppleTalk Services Restrictions on TLI, mentions limitations on certain TLI functions.

New sample code is provided. It shows how to use TLI functions, including the new version feature.

Related Topics

AppleTalk TLI Structures

ADSP Performance Notes

TLI Functions: AppleTalk Notes

t_optmgmt (adsp.h)

This structure is explained in How to Determine the ADSP Interface Version .

```
struct t_optmgmt {
   struct netbuf    opt;
   long             flags;
};
```

Parent Topic: AppleTalk TLI Structures

Transferring Data in ADSP

You can think of data transfer as occurring over a pipe---from the local application, through TLI, STREAMS, and ADSP protocols, over the physical wires, through any intervening router, to the remote application. When the amount of data you send or receive approaches the amount of data that can be stored in other sections of the pipe or sent at one time, you may find that throughput is limited. Therefore, you must be aware of the size of send and receive buffers, both local and remote, at various points in the data transfer process.

You must be especially aware of the sizes of these buffers:

- The send and receive buffers for ADSP at both ends of the connection
- The send and receive buffers for the applications at both ends of the connection

The local send buffer is the one you pass to **t_snd**; the local receive buffer is the one you pass to **t_rcv**. Each buffer should be as large as possible, at least 2 K in size. You cannot control the size of the send and receive buffers of ADSP directly, but this implementation uses send and receive buffers of 8192 bytes.

You must also take the maximum size of a STREAMS message into consideration; this size is determined by the version of STREAMS.NLM you are using and the parameters you specify on the load line.

Once you have taken all the buffer sizes into account, do your best to avoid sending or receiving small amounts of data at a time. The smaller the amount of data you send or receive, the more memory, packet-receive buffers and time per byte used. This situation holds especially true for large bandwidth-delay product connections, such as across WANs and satellite links.

Parent Topic: ADSP Performance Notes

Transferring Data in PAP

Now that you have a file descriptor (WSS or SSS) in the data transfer phase, you can call **ATPapWrite** or **ATPapGet** to read and write data. PAP has a flow control mechanism, so **ATPapWrite** can block while waiting for buffers to clear. If you don't want to block, use the CLIB **poll** function to poll the file descriptor for writeability.

PAP attempts to gather out-going data into the fewest number of transactions, so if you want data to be written immediately, use the flush parameter. In some printing protocols, it is necessary to not just get the data out now, but also to know that certain data ends a transaction. If this is desired, use the EOT (End Of Transaction) parameter. If you want to set the EOF bit in a particular transaction, set the EOF parameter.

> **IMPORTANT:** It can be useful to call **ATPapWrite** with NULL data and length and EOF set to generate a transaction with EOF set but no data.

To read data, call **ATPapGet**, expecting the PAP_EVENT_DATA message. Set the buffer to 4096 bytes. The data is copied into the buffer. Each PAP_EVENT_DATA message corresponds to exactly one data transaction, so no data in the message would reflect an implied transaction. To determine if the EOF flag was set on a particular transaction, examine the *pass->U.Data.eof* field.

Another event that could be returned in **ATPapGet** at this phase is PAP_EVENT_CONN_CLOSED. This indicates that the remote endpoint has closed the connection. In this case you must close the file descriptor with **ATPapClose**.

Yet another possible event is PAP_EVENT_CONN_DIED. This implies that the connection was invalidated due to a lack of packets received. Again, close the file descriptor with **ATPapClose**.

Parent Topic: PAP and NetWare

Using the T_MORE Bit

When using ADSP, an application often sends a certain amount of data before waiting for a reply or further communication (another query, in the case of a database server). This data can constitute a single ADSP message. However, the application need not send all the data in the message at the same time. It can accomplish the file transfer by sending large blocks of data, one after the other; each block of data is transferred with a call to **t_snd**.

Your application should set the T_MORE bit when sending a packet, unless the packet being sent completes the message. In the last packet, do not set the T_MORE bit. Use these guidelines:

Set the T_MORE bit when you are sure that application continues to send data.

Do not set the T_MORE bit when the application has completed its data transfer and is waiting to receive a reply or further communication from the remote connection end.

If your application fails to set the T_MORE bit in any packet except the last one, throughput may suffer. The cost depends upon the number of bytes you transfer in a packet. When you transfer less than one or two kilobytes of data and do not set T_MORE, throughput is adversely affected. The smaller the size of the data transferred, the worse the effect. As you increase the amount of data transferred, the effect disappears.

This behavior can be explained in the following manner. It takes nearly the same amount of time to process a packet with only a few bytes as it does to process a packet with the maximum number of bytes. The number of bytes processed per unit of time goes down for packets with fewer bytes, and the data transfer is very slow.

If you do not set the T_MORE flag during a **t_snd** call, ADSP sets the EOM bit for the packet containing the data going out on that **t_snd** call. Because an EOM indicator must logically be found at the end of an ADSP packet, the use of the T_MORE flag may influence the size of each packet. For example, if you send 100 bytes five times, and you do not set the T_MORE flag each time, ADSP sends the data in five packets, each containing 100 bytes of user data; the EOM indicator appears at the logical end of each packet. However, if you send 100 bytes five times using the T_MORE flag on the first four sends, ADSP may send the data as a single packet; in this case, the EOM indicator appears at the logical end of the packet. Keep in mind the performance benefits of using the T_MORE flag when using **t_snd**.

So, if your message is smaller than the maximum number of bytes ADSP can fit in a packet, performance suffer. Be aware, however, that even if your message is larger than a single packet, the last packet in the message probably is not full.

Advice to Backup and File-Transfer Application Developers: Set the T_MORE bit for each block of data you send, except the block of data that completes your bulk transfer. If you do not set the T_MORE bit within the bulk transfer, throughput suffers.

Advice to Database Developers: When returning multiple records to a query, set the T_MORE bit at the field or record boundaries. Do not set the T_MORE bit when you come to the end of the whole response to the query. Send data in blocks of at least one or two kilobytes.

Advice to Terminal-Emulation Developers: You probably do not want to set the T_MORE bit in these situations:

At the end of every line (for line-oriented echo)

After every character (for character-oriented echo

When you come to the end of a whole screen or update

Parent Topic: ADSP Performance Notes

XO or ALO?

When choosing between eXactly Once (XO) and At Least Once (ALO) transaction mode, remember that XO guarantees only one delivery of the data. In most circumstances, like file service, this is quite important. However, in some circumstances, such as network management, multiple requests are acceptable as long as the response succeeds. XO mode is by far the more common. (For a thorough discussion of XO and ALO transaction modes, see pages 9-5 through 9-9 of *Inside AppleTalk*, Second Edition.)

Parent Topic: ATP and NetWare

ZIP

ZIP provides two major services:

ZIP enables each router to share and maintain zone information for an AppleTalk internet.

ZIP enables each nonrouter node to obtain zone information maintained by the routers.

On an AppleTalk internet, a **zone** is simply a name for groups of nodes; for example, a file server used for office administration might be in the "Administration" zone.

Each router on an AppleTalk internet maintains network-number-to-zone-name mappings in the **Zone Information Table**. This table consists of one entry for each network in the AppleTalk internet, and defines a mapping between networks and zone names. The information in this table is constantly updated to ensure reliable mappings.

The ZIP utility functions available through APPLETLK.NLM enable each node on the network to obtain this information from a router's Zone Information Table:

The zone in which the current node resides

The list of zones on the local network

The list of zones on the AppleTalk internet

For a full description of the ZIP functions, see AppleTalk: Functions.

Parent Topic: AppleTalk Overview

ZIP and NetWare

The following describes the differences between versions 1.x and 2.0.

The **ATZipGetLocalZones** and **ATZipGetZoneList** have changed. Both functions have similar parameter changes. start is the number of the first zone to hand in, like it was in the old function. zones is a pointer to the buffer to fill in. This is in a packed format. *z_size* is passed in and out as the size of the buffer IN BYTES. last is set if the router told us it was the end of the list (new feature). The *router* field is used as follows.

If there are multiple requests, you must talk to the same router. Set the net, node, and socket to zero the first time you call the function, and it is returned with the router you are talking to. Don't reset the values, and you always get the same router.

Two new functions have been added, **ATZipGetNBPZones** and **ATZipZoneXlate**. **ATZipGetNBPZones** gives the list of zones that can be explicitly registered on. **ATZipZoneXlate** translates a packed zone list from the AppleTalk code page to the local code page.

ZIP is a client of ATP and receives its packets from there. These functions are the ZIP client, and thus send packets out over the net to get the answers to their questions. Because each function opens a file descriptor, sets up an ATP client, sends a request, and waits for a response, it can be an expensive proposition. Since most operations are a single ATP transaction to the nearest router, which is always on the local net, and they involve one reply packet, the functions should not take a long time. But they do involve network access, and multiple context switches, as do those in the previous interface (v1.x).

This interface is roughly equal to the one in "AppleTalk APIs for UNIX* System V* Release 4". It is a needed upgrade to v1.x, which was equal to the zip(3N) AUX 2.0 interfaces.

ZIP passes the zone list upward as it was received from a router, in whatever character set that might be. It probably is in MacASCII, but could be in Kanji. The Macintosh uses SHIFT-JIS as its encoding method, thus all lower ASCII are kept intact.

This interface includes a NetWare specific function that translates a zone list from the MacASCII code page into the code page being used by the local server.

Notice that all of the replies to functions change as the network

Notice that all of the replies to functions change as the network configuration changes. Since some of the functionality can require multiple nonatomic requests, configuration changes during this operation can have unpredictable results. It should be assumed that AppleTalk networks are fairly stable, except in the time that a node is learning its configuration (seconds after coming on line). ZIP does its best to recognize this case and either delay a request or error it out.

ZIP Function List

ATZipGetLocalZones

Obtains a complete list of all the zone names defined on the local network (the network the calling node is bound to).

ATZipGetMyZone

Finds the name of the zone the node is in.

ATZipGetNBPZones

Finds the zones that are valid for you to register explicitly an entity in.

ATZipGetZoneList

Obtains all the zones on the internet.

ATZipZoneXlate

Translates, in place, a zone list from the MacASCII code page into the code page being used by the local server.

AppleTalk: Functions

ATAppletalkVersion

Returns AppleTalk* Interface version information
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <appletlk.h>

int ATAppletalkVersion (
   char   *major,
   char   *minor,
   char   *revision);
```

Parameters

major

(IN/OUT) Points to the major version number you are requesting. The return confirms the presence of the presence of the major version on the server or returns zero (0).

minor

(IN/OUT) Points to the minor version number you are requesting. The return confirms the presence of the presence of the minor version on the server or returns zero (0). You can pass in his pointer as NULL.

revision

(IN/OUT) Points to the revision version number you are requesting. The return confirms the presence of the presence of the revision version on the server or returns zero (0). You can pass in his pointer as NULL.

Return Values

0	No match, or you passed in NULL for *major*
1	Match

Remarks

ATAppletalkVersion allows your NLM™ application to request a

particular level of interface service and returns whether or not the version of native AppleTalk interface you requested is running on the local server. If you are only interested in the major version number, you can pass NULL pointers for *minor* and *revision*.

After you pass in the version numbers of the interface that you want, this function returns 1 if that version is supported by the server, and 0 if it is not (or if you passed in NULL for *major*).

For example, the version described in this manual is 2.0. If you write your NLM to this version of the interface, call **ATAppletalkVersion** with *major* as 2 and *minor* as 0. If version 2.0 is not running on your server, the function returns 0 (FALSE).

NOTE: Please note that although functions usually return 0 to indicate success, in this function 0 denotes FALSE, and 1 is TRUE.

ATAspAttn

Sends an attention request to the remote WSS (workstation session socket)
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspAttn (
   void        *fd,
   void        *id,
   u_short      attn,
   ATRetry_t   *Retry);
```

Parameters

fd

(IN) Indicates the file descriptor.

id

(IN) Indicates the attention ID.

attn

(IN) Indicates a two-byte attention code. Must not be zero.

Retry

(IN) Indicates the retry specification for the attention request. For a discussion of the structure, see ATRetry_t.

Acknowledgment or error comes in **ATAspGet**.

Return Values

0	Success
EBADF	Invalid file descriptor
EINVAL	Illegal socket type or state, or zero attention code
ENOBUFS	Resource limitation
ECONNABORTED	Request was aborted by a close socket call

Remarks

The Attention ID is created by the client for use in demultiplexing responses. It is handed back up in **ATAspGet**. If only one request at a time is going to be outstanding, you could simply set the ID to 0. Note that attention requests are ALO (at-least-once) requests, so the problems with multiple requests possible in ALO transactions can happen with ASP Attention messages. (For a thorough discussion of this subject, see pages 9-5 through 9-9 of *Inside AppleTalk*, Second Edition.)

IMPORTANT: Remember the attention message is outside the usual state diagram for ASP. It is communication against the usual flow; the server initiates a request. The response could come at any time, potentially disrupting any event you thought would come in.

ATAspClose

Closes the session, freeing up any internal structures and discarding all undelivered packets, making all other calls complete

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspClose (
   int   *fd);
```

Parameters

fd

(IN) Points to a file descriptor.

Return Values

0	Success
EBADF	Someone else closed the file descriptor, you should stop trying
STREAMS error	Try again

Remarks

ATAspClose can close more than one session if the file descriptor is the session listening socket (SLS) *fd*.

Notice that **ATAspCloseSession** has not been implemented. Each file descriptor is a session, so when a close is desired, **ATAspClose** should be called on that file descriptor. Outstanding calls complete with an error (to be supplied).

See Also

ATAspOpen

ATAspCmdReply

Sends a command reply
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspCmdReply (
   int       fd,
   u_long   req_result,
   char    *reply,
   int      replylen);
```

Parameters

fd

(IN) Indicates the file descriptor.

req_result

(IN) Indicates the 4-byte command result.

reply

(IN) Points to a response buffer.

replylen

(IN) Indicates the length of the response buffer.

Return Values

0	Success

Remarks

Call **ATAspCmdReply** after receiving an ASP_EVENT_SSS_CMD_REQ event.

After receiving a command request event, the server responds with a 4-byte error code and a buffer size of QuantumSize (value pointed to by *quantum* in **ATAspGetParms** upon return).

There can be only one command request at a time, so no extra identifier is necessary.

ATAspCmdReply does not wait for network access; hence, it is quick.

ATAspGet

Reads an event from the ASP protocol
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspGet (
   int            fd,
   ATAspPass_t   *pass,
   char          *buf,
   int           *len);
```

Parameters

fd

(IN) Indicates the file descriptor.

pass

(OUT) Points to the ATAspPass_t structure. Information is passed back in that pointer. For a discussion of the structure, see ATAspPass_t.

buf

(OUT) Points to a buffer.

len

(IN/OUT) Points to the size of the buffer. The number of bytes actually read into the buffer is returned in that pointer.

Return Values

0	Success or received close request
EBADF	Invalid file descriptor
EBUFTOOSMALL	Try again with larger buffer
ECONNABORTED	This session is in the closed state
EINVAL	Bad parameter or bad state for this call

ENOBUFS	Not enough memory to send reply
ESHUTDOWN	SLS that spawned this session was closed
ENETRESET	Network has been reconfigured
ETIMEDOUT	No packets received for a long time, probably dead

Remarks

ATAspGet understands any type of ASP data arriving---request, response, or error---and returns the meaning of the data.

ATAspGet reads an event from the ASP protocol. This is the way that the protocol communicates with the client. Example events include ASP_SSS_CMD_REQ, which indicates that a command request has arrived on this SSS, and ASP_EVENT_SLS_INIT, which indicates that a session listening socket (SLS) has been initialized and is ready to go.

ATAspGet does not take long if the **poll** function is called on this file descriptor, and it has data. Otherwise, this function waits until data arrives.

If **ATAspGet** fails to receive the event properly, a nonzero value is returned. This is different from a protocol error, which is returned in the *ATAspPass_t.ret* field. Thus, this function could fail due to a lack of buffer space, which would be returned as the return value. In contrast, the protocol error EADDRNOTAVAIL, indicating an SLS initialization failure, would be found in the *ATAspPass_t.ret* field.

In most cases, the caller knows the state of ASP and be able to guess the size of the buffer. A buffer size of QuantumSize (value pointed to by *quantum* in **ATAspGetParms** upon return) is sufficient for any possible case.

The possible events are listed in the following table. This table lists the name of the event, the way the event should be interpreted, and the parameters filled in with information about the event in the ATAspPass_t structure. See ATAspPass_t. For your convenience, this structure is also listed below.

IMPORTANT: Do not rely on a number associated with an event; this number can change. Rather, select functions based on the "#define" statements themselves.

Event	Interpretation	Parameter
SLS		
ASP_EVENT_SLS_OPEN_REQ	Received a request for connection. Reply with	No errors *OpenReq.addr* ---address of

	ATAspSSSInit or **ATAspSLSReject**.	requestor *OpenReq.id*---ID used by the protocol. Hand it back in acceptance or rejection.
ASP_EVENT_SLS_STATUS_REQ	If you have asked to reply to each status request individually, this event shows that a status request has come in. Reply to it with **ATAspSendStat**.	No errors *StatReq.addr* ---Address of requestor. Pass it back with the reply.
ASP_EVENT_SLS_INIT	Returns that the SLS has been initialized on this descriptor, or returns an error as to why the initialization failed.	No data *ret*---Error (out of sockets, out of memory)
ASP_EVENT_SLS_SETSTAT	Returned if the set status command was in error	No data *ret*---EINVAL, ENOBUFS
ASP_EVENT_SLS_SENDSTAT	Sent up if the ATAspSendStat command failed.	*ret*---EINVAL, ENOBUFS
ASP_EVENT_SLS_CLOSE	Usually due to some kind of fatal error, the SLS connection has been terminated. Close the SLS.	No return, no data
SSS		
ASP_EVENT_SSS_CMD_REQ	Received when the SSS receives a command request on the open session. Reply with **ATAspCmdReply**.	No errors Nothing in pass structure Data is the command request, of ASP_MAXCMD maximum size
ASP_EVENT_SSS_WRITE_REQ	Received a write request. If Fastwrite is not in effect, reply with **ATAspWCont** or	No errors Nothing in pass structure Data is the write request data,

	ATAspWCont or **ATAspWReply**. If Fastwrite is in effect, this message contains the write request data but needs no **ATAspWCont** reply.	of CmdSize maximum size
ASP_EVENT_SSS_WRITE CONT_REPLY	Received a write continue reply. Reply with **ATAspWReply**.	No errors Data is the write data received, of QuantumSize maximum size
ASP_EVENT_SSS_ATTN_ REPLY	Received the acknowledgment of reply to an outstanding send_attention, or the Attention transaction timed out. No reply is necessary.	*ret*---0, ETIMEDOUT *Attn.id*---ID code that client passed in No data
ASP_EVENT_SSS_SND_A TTN	Was trying to send the Attention request, but an error got in the way; the Attention request never made it onto the wire. Try again, if you want.	*ret*---EINVAL, ENOBUFS *Attn.id* ---ID code that client passed in No data
ASP_EVENT_SSS_CLOSE	The SSS session has been closed.Call **ATAspClose**.	*ret*---Possible errors: ESHUTDOWN: SLS that spawned the SSS was closed ENETRESET: network has been reconfigured 0: Received a close request from remote endpoint ETIMEDOUT: No packets received for a long time, probably dead
ASP_EVENT_SSS_WRITE _REPLY	**ATAspWReply** was called, but the system could not send the reply.	*ret*---Possible errors: ECONNABORTED: This session is in the closed state EINVAL: Bad

		EINVAL: Bad parameter or bad state for this call ENOBUFS: Not enough memory to send reply
ASP_EVENT_SSS_WRITE CONT_REQ	**ATAspWCont** request was called but the system could not send the request.	*ret*---Possible errors: EINVAL: Bad parameter or bad state for this call ENOBUFS: Not enough memory to send request

```
typedef struct
ATAspPass {
    u_short   cmd;
    long      ret;
    union {
      struct Attn_s {              /* Used externally */
         short attn_code;         /* 13 bytes */
         ATRetry_t  retry;
         u_long       id;
      } Attn;
      struct StatReq_s {           /* Used externally */
         ATInet_t     addr;
      } StatReq;
      struct OpenReq_s {
         ATInet_t      addr;
         u_long        id;
         u_short       fastwrite;
      } OpenReq;
    } U;
} ATAspPass_t ;
```

See Also

ATAtpGet, **ATPapGet**

ATAspGetParms

Allows a client to find particular constants, the maximum size of an ASP command and the maximum size of an ASP response before a session is active

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspGetParms (
   int   fd,
   int  *maxcmd,
   int  *quantum);
```

Parameters

fd

(IN) Indicates the file descriptor.

maxcmd

(IN/OUT) Points to an integer. Maximum command size is returned.

quantum

(IN/OUT) Points to an integer. ASP quantum size is returned.

Return Values

0	Success
EBADF	Bad file descriptor

Remarks

In reality, all ASP implementations use a maximum command size of 578 bytes and an ASP quantum size of 4624 bytes. There is no negotiation between endpoints about the size to use. For maximum compatibility, call **ATAspGetParms** anyway.

ATAspGetParms can be called from a server or a workstation. It is

allowed on any ASP file descriptors.

ATAspGetParms is a blocking function, but is nevertheless quick.

ATAspGetWorkstation

Returns the AppleTalk address of the workstation to which this file descriptor is connected

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspGetWorkstation (
   void       fd,
   ATInet_t   *addr);
```

Parameters

fd

(IN) Indicates the file descriptor.

addr

(OUT) Points to an AppleTalk address in the pointer that you passed in.

Return Values

0	Success
EBADF	Invalid file descriptor
EINVAL	No session created yet through this file descriptor

Remarks

ATAspGetWorkstation is a blocking function, but is quick (no network access).

ATAspOpen

Opens a file descriptor for reading and writing, but does not bind a session to this file descriptor

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspOpen (
   int   *fd);
```

Parameters

fd

(OUT) Points to the file descriptor returned in the pointer that you passed in.

Return Values

0	Success
ENOBUFS	Insufficient buffer space

Remarks

ATAspOpen involves no network traffic.

See Also

ATAspClose

ATAspSendStat

Replies to a GetStatus request received through the **ATAspGet** function
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspSendStat (
   int        fd,
   char      *buf,
   int        len,
   ATInet_t   addr);
```

Parameters

fd

(IN) Indicates the file descriptor.

buf

(IN) Points to the status string. Maximum size of the status buffer is MAXCMDSIZE.

len

(IN) Indicates u_short, length of status string.

addr

(IN) Points to the remote AppleTalk address of the sender of the request.

Return Values

0	Success
EINVAL	Buffer too long, StatusID invalid

Remarks

ATAspSendStat is a quick function.

ATAspSetStat

Sets the status string on an SLS (session listening socket) when an **ATAspGetStatus** packet is received

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspSetStat (
   int    fd,
   char  *buf,
   int    len);
```

Parameters

fd

(IN) Indicates the file descriptor.

buf

(IN) Points to the status string.

len

(IN) Indicates the length of the status string.

Return Values

0	Success
ENOBUFS	Could not allocate buffer to copy into. Try again later or abandon the SLS.

Remarks

If a status is already set, **ATAspSetStat** resets the status.

IMPORTANT: If you, the client, want to reply to each individual get-status request packet, set the *stat_flag* parameter of **ATAspSLSInit** to the value of 1 when calling instead of this function.

If the buffer passed in is zero length | zero pointer, that is what the status string is.

Maximum size is MAXCMDSIZE.

ATAspSetStat is a quick function.

ATAspSLSInit

Makes the specified file descriptor into an SLS (session listening socket)
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspSLSInit (
   int   fd,
   int   stat_flag);
```

Parameters

fd

(IN) Points to the file descriptor.

stat_flag

(IN) Indicates to ASP either to handle all get status requests internally or to pass each one to your application:

0 = internal
1 = application

Return Values

0	Success

Remarks

ASP allocates two sockets, one for the incoming open request, and one for the sessions spawned from this SLS. After this function completes, that is, after the **ATAspGet** comes back, **ATNbpRegister** can be called.

If *stat_flag* is 0, the application must set the status. **ATAspSetStatus** must be called. Otherwise, status requests come to the application, and each one should be replied to individually, with **ATAspSendStat**.

Errors are returned in an **ATAspGet**, as are notification that the operation succeeded. Possible errors are "out of sockets" or "out of

memory."

ATAspSLSInit is a quick function.

ATAspSSSInit

Creates a new SSS (server session socket) when a request comes in
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspSSSInit (
   int         SSSfd,
   void       *sss_id,
   ATInet_t   *wss_addr,
   int         fastwrite);
```

Parameters

SSSfd

(IN) Indicates the file descriptor of the SSS.

sss_id

(IN) Indicates the OpenConn request identifier.

wss_addr

(IN) Indicates the network address of the workstation attempting to connect.

fastwrite

(IN) Indicates the Fastwrite option, set (1). Otherwise, clear (0).

Return Values

0	Success

Remarks

Use *U.OpenReq.id* (of the ATAspPass_t structure) passed up in the ASP_EVENT_SLS_OPEN_REQ event.

ATAspSSSInit is a quick function; it does not wait for any replies.

ATAspWCont

Sends a write continue request
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspWCont (
   int   fd,
   int   len);
```

Parameters

fd

(IN) Indicates the file descriptor.

len

(IN) Indicates an integer specifying the number of bytes your application accepts in the write continue reply.

Return Values

0	Success

Remarks

Call **ATAspWCont** after receiving a write request event, if the write request isn't in error and the Fastwrite option is off.

ATAspWCont specifies to the workstation the buffer size that the server accepts.

See Fastwrite for an explanation of how to avoid using this function.

ATAspWCont does not wait for network access.

ATAspWReply

Sends a write reply
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <asp.h>

int ATAspWReply (
   int      fd,
   u_long   result,
   char    *buf,
   int      len);
```

Parameters

fd

(IN) Indicates the file descriptor.

result

(IN) Indicates the 4-byte write result.

buf

(IN) Points to write reply buffer.

len

(IN) Indicates the length of write reply buffer.

Return Values

0	Success
EBADF	File descriptor is invalid

Remarks

Call **ATAspWReply** after receiving an ASP_EVENT_SSS_WRITE_REQ event to indicate success or failure of the write request.

Because of the inverted, two-transaction nature of an ASP write,

ATAspWReply can be called either after receiving the ASP_EVENT_SSS_WRITE_REQ, which would preclude the write continue transaction, or after the ASP_EVENT_WRITE_CONT_REPLY if the write continue is entered into. By convention, the first case is an error case and the result should be nonzero, but not necessarily always. For further discussion of this point, see ASP, including the flowchart.

There can be only one write continue request at a time, so no extra identifier is necessary.

ATAspWReply does not wait for network access.

ATAtpCancelRecvReq

Allows you to terminate a received XO request prematurely
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpCancelRecvReq (
   int    fd,
   void  *TransID);
```

Parameters

fd

(IN) Specifies the file descriptor associated with the socket sending the TReq packet. The file descriptor is returned by the function **ATAtpOpen**, which opens the socket.

TransID

(IN) Points to a void-sized pointer. The transaction ID (TID) of the **ATAtpSendReq** you are cancelling is returned in that pointer. In contrast to its counterpart in**ATAtpCancelSendReq**, this is the computer-generated TID, which is unique.

Return Values

0	Success
EBADF	File descriptor is invalid
EINVAL	There is no such transaction ID on this fd.

Remarks

IMPORTANT: Do not cancel an ALO request. The ALO Transaction ID space is not unique, so canceling an ALO request could end up canceling an XO request (but more likely would return a "transaction not found" error). There is no memory stored of ALO requests, so they don't need to be canceled.

If you receive an XO request and have no intention of replying, it is necessary to cancel the request to free system resources. The memory used to store the XO request is freed, along with all other buffers. If the remote endpoint retries, you receive the request indication again. No packets are sent out over the wire. The transaction ID here (TransID) is the ATP-generated one, which is unique.

Since **ATAtpCancelRecvReq** and **ATAtpCancelSendReq** functions are synchronous and the interface uses STREAMS, only one **ATAtpCancelRecvReq** or **ATAtpCancelSendReq** can be outstanding at a time. Thus, multiple threads accessing the same file descriptor could get hung.

Example

ATAtpCancelRecvReq

```
#include <atp.h>

main()
{
    int             my_fd, err;
    ATAtpPass_t     pass;
    /* Open the my_fd file descriptor */
    pass.data = malloc(ATP_DATA_SIZE * 8);
    if (!pass.data)
      return;
    pass.data_len = ATP_DATA_SIZE * 8;
    /* Call ATAtpGet to get an OpenConnection request */
    err = ATAtpGet(my_fd, pass);
    if (err || pass.ret)
      return;
    if (pass.event == ATP_EVENT_RCV_REQ)
    {
      /* Try to do some action, like read from disk... but it fails */
      if (pass.xo)
      ATAtpCancelRecvReq(my_fd, pass.TransID);
    }
}
```

ATAtpCancelSendReq

Allows a request transaction generated by the local endpoint to be terminated earlier than normal

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpCancelSendReq (
   int    fd,
   void  *TransID);
```

Parameters

fd

(IN) Specifies the file descriptor associated with the socket that sent the TReq packet.

TransID

(IN) Points to a void-sized pointer. The transaction ID (TID) of the **ATAtpSendReq** that you are cancelling is returned in that pointer.

Return Values

0	Success
EBADF	File descriptor is invalid
EINVAL	There is no such transaction ID on this fd.

Remarks

Normally, a request transaction is terminated when the reply comes in or the retry structure says to do so.**ATAtpCancelSendReq** allows a request transaction generated by the local endpoint to be terminated earlier.

NOTE: Because the transaction ID here is created by the API client, it is not necessarily unique. So it is possible for unintentional cancellations to occur since this function cancels all requests with the specified transaction ID.

ATAtpCancelSendReq is necessary with ALO because ALO requests stay outstanding until the correct reply comes back.

Note that an **ATAtpClose** automatically cancels all transactions, so under normal operation, this function need not be used. Also, because the ATP transaction database is not indexed on the void pointer transaction ID, **ATAtpCancelSendReq** can be relatively slow.

Note also that, because **ATAtpCancelSendReq**n and **ATAtpCancelRecvReq** are synchronous, and the interface uses STREAMS, only one **ATAtpCancelSendReq** or **ATAtpCancelRecvReq** can be outstanding at a time. Thus, multiple threads accessing the same file descriptor could get hung.

Example

ATAtpCancelSendReq

```
#include <atp.h>

main()
{
    int             my_fd, err;
    ATAtpPass_t     pass;
    ATInet_t        where_to_send_request;

    /* Open the my_fd file descriptor */
    /* Call ATNbpLookup to set where_to_send_request */
    pass.at_addr = where_to_send_request;
    pass.data = 0;
    pass.data_len = 0;
    pass.xo = 1;
    pass.xo_relt = ATP_XO_DEF_REL_TIME;
    pass.retry.interval = 0;          /* Gets default retry settings. *
    pass.bitmap = 0x01;
    pass.packetize = 0;
    pass.userdata[0] = 0x42;
    pass.TransID = 0x42726942;

    /* Send out the request */
    err = ATAtpSendReq(my_fd, pass);
    if (err) return;

    /* Decided you don't care about the answer. Cancel that request. */
    ATAtpCancelSendReq(my_fd, 0x42726942);
}
```

ATAtpClose

Closes a previously opened socket with which this file descriptor is associated, freeing up any internal structures and discarding all undelivered requests and responses, making all other calls complete

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpClose (
   int   fd);
```

Parameters

fd

(IN) Specifies the file descriptor that was returned by **ATAtpOpen** when the socket was opened.

Return Values

0	Success
ENOENT	You specified an invalid entity (in this case, a file descriptor)
STREAMS error	Try again.

Remarks

ATAtpClose frees the allocated DDP socket.

ATAtpClose closes a previously opened socket with which this file descriptor is associated, freeing up any internal structures and discarding all undelivered requests and responses, making all other calls complete. It frees the allocated DDP socket.

IMPORTANT: **ATAtpClose** sends no closing notification to any "connected" protocol clients (because there are no connections in ATP).

See Also

ATAtpOpen

Example

ATAtpClose

```
#include <atp.h>

main()
{
int my_fd, sock, err;

/*
    Open a socket.
*/
    sock = 0;       /* Get a dynamic socket. */
    if( err = ATAtpOpen(&my_fd,&sock) )
    {
      printf("ATAtpOpen failed: err = %x\n",err);
      exit(__LINE__);
    }
.
.
. Do some work.
.
.
/* Gracefully close the connection.  */
    if(err = ATAtpClose(my_fd))
    {
      printf("ATAtpClose failed: err = %x\n",err);
      exit(__LINE__);
    }
}
```

ATAtpGet

Retrieves the next event from the protocol client
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpGet (
   int            fd,
   ATAtpPass_t   *pass);
```

Parameters

fd

(IN) Specifies the file descriptor to retrieve the event from. The file descriptor is returned by the function **ATAtpOpen**, which opens the socket.

pass

(OUT) Points to an ATAtpPass_t structure, which points to the data being transferred and the specifics of what arrived. See ATPapPass_t.

Return Values

0	Success
EBADF	You specified an invalid file descriptor.
ENOBUFS	Insufficient buffer space is available.
ETIMEDOUT	Your request timed out.
ERANGE	Data remains to be read the second time.

Remarks

ATAtpGet retrieves the next event from the protocol client. Unlike its predecessors (**atp_getreq**), this function retrieves all events, not just a request or a response, allowing multiple requests and responses to be outstanding at a time.

WARNING: Note that the *data* and *data_len* fields must be initialized in the ATAtpPass_t structure, at least to zero, or the machine will crash.

The *data* and *data_len* parameters have remained in the ATAtpPass_t structure instead of being moved to the "get" functions as they were in ASP and PAP. (In some ways, the ATAspPass_t and ATPapPass_t structures are like parameters to their respective get functions.)

ATAtpGet does not take long if **poll** is called on this file descriptor, or it has data. Otherwise, it waits until data arrives.

Because **ATAtpGet** can decipher multiple kinds of data, it provides the advantage of retrieving any kind of data---request or response or error---without needing to know, in advance, what kind of data is waiting.

The currently defined event types are listed in the following table. This table lists the name of the event, the way the event should be interpreted, and the parameters that are filled in with information about the event in the ATAtpPass_t structure. See ATAtpPass_t and the listing below.

IMPORTANT: Do not rely on a number associated with an event; this number can change. Rather, select your functions based on the "#defines" themselves.

Event	Interpretation	Parameter
All		*ret*---Error code (ENOBUFS, ERANGE) *at_addr*---Address of the remote endpoint referred to in this transaction *TransID*---associates between request and response. For transactions started by the network, it passes the same TransID you received. For transactions generated by you, the response has whatever TransID you specified in the request. *xo*---Refers to whether the transaction is exactly-once or at-least- once.
ATP_EVENT_RCV_REQ	Received a request	*xo_relt*---Set to the *xo_relt* field in the incoming request. This is informational only; the ATP protocol sets the timer and all. *userdata*[0]---Contains the ATP *userdata* field of the request

		packet_len[0]---Contains the length of the request data *bitmap*---Points to the bitmap received in the request. Use this to determine the maximum size of response you can send. *data* and *data_len*---Note that these must be initialized in the structure, at least to zero,**or the machine will crash**. Maximum is ATP_DATA_SIZE
ATP_EVENT_RCV_RSP	Received a response	*bitmap*---Maps which packets were received in the response *userdata*[0...7]---The *userdata* array contains the *userdata* field from each packet associated with the corresponding *bitmap.* *packet_len*[0...7]---Contains the length of data received in each packet *data* and *data_len*---Note that these must be initialized in the structure, at least to zero,**or the machine will crash.** The maximum data is a *rcv_resp* and be limited to ATP_DATA_SIZE * (bits set in its requesting bitmap). So, if bitmap is set to 1, max data is ATP_DATA_SIZE, bitmap 0x03 = 2 bits set, thus ATP_DATA_SIZE * 2.
ATP_EVENT_SND_REQ	Request sent did not complete due to an error.	*ret*---Possible errors: ETIMEDOUT, EINVAL, ENOBUFS
ATP_EVENT_SND_RSP	Response sent did not complete due to an error	*ret*---Possible errors: ETIMEDOUT, EINVAL, ENOBUFS

```
typedef struct ATAtpPass {
    u_short      event;
    int          ret;
    ATInet_t     at_addr;
    void         *TransID;
    u_char       xo;
    u_char       xo_relt;
    u_char       bitmap;
```

```
    u_char       packetize;
    ATRetry_t     retry;
    void          *data;
    u_short       data_len;
    u_long        userdata[8];
    u_short       packet_len[8];
} ATAtpPass_t;
```

See Also

ATAspGet, ATAtpSendReq, ATAtpSendRsp, ATPapGet

Example

ATAtpGet

```
#include <atp.h>

main()
{
ATAtpPass_t rcv_pass;
int err;
.
.
. Do some work.
.
.
/*
    Allocate memory for the receive buffer.
*/
    rcv_pass.data = (void *)malloc(ATP_DATA_SIZE);
    rcv_pass.data_len = ATP_DATA_SIZE *8;
    printf("Waiting for request...\n\n");
    do
    {
      if(err = ATAtpGet(my_fd,&rcv_pass) )
      {
         printf("ATAtpGet failed: err = %x\n",err);
         ATAtpClose(my_fd);
         exit(__LINE__);
      }
/*
    Check for an error.
*/
    if(rcv_pass.ret)
    printf("rcv_pass.ret returned error (%x)\n",rcv_pass.ret);
}
```

ATAtpOpen

Opens a socket that listens for packets from the file descriptor that is returned by **ATAtpOpen**

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpOpen (
   int         *fd,
   ATSocket    *socket);
```

Parameters

fd

(OUT) Points to the file descriptor.

socket

(IN/OUT) Points to an ATSocket (defined in appletlk.h).

Return Values

0	Success
ENOBUFS	Could not allocate per socket structure; try again.
EADDRNOTAVAIL	No sockets are available.
EADDRINUSE	Another protocol client is using the socket you specified.
EINVAL	You specified a socket in the dynamic range.
ENOENT	You specified an invalid entity (in this case, a socket).
STREAMS error	Try again.

NOTE: If you specified an invalid socket, **ATAtpOpen can return either** EBADF or ENOENT.

Remarks

ATAtpOpen opens a socket that listens for packets from the file descriptor that it returns. You can either pass a socket number to ATP (**static** socket assignment) or you can pass 0 or a NULL pointer and the socket number is returned to you (**dynamic** socket assignment). To close the socket, call **ATAtpClose**.

To open a statically assigned socket, *socket* must be a pointer to a number within the static range of 1 through 127. (See **ATDdpRegisterListener** for more about static and dynamic sockets.)

To open a dynamically assigned socket, *socket* must either point to socket number zero or be a NULL pointer:

socket Input	Result
socket = 0	*socket* returns the socket that has been opened
socket = 0	*socket* does not return information about which socket has been opened

When opening this socket, you are, in effect, opening a "transaction listening socket." Unlike DDP, the ATP interface accepts only one listening socket per file descriptor.

See Also

ATAtpClose

Example

ATAtpOpen

```
#include <atp.h>

main()
{
    int my_fd, sock, err;
    /*
      Open a socket.
    */
    sock = 0;       /* Get a dynamic socket. */
    if( err = ATAtpOpen(&my_fd,&sock) )
    {
      printf("ATAtpOpen failed: err = %x\n",err);
      exit(__LINE__);
    }
.
.
```

```
. Do some work.
}
```

ATAtpSendReq

Sends a request to a remote endpoint
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpSendReq (
   int            fd,
   ATAtpPass_t   *pass);
```

Parameters

fd

(IN) Specifies the file descriptor associated with the socket sending the TReq packet. The file descriptor is returned by **ATAtpOpen**, which opens the socket.

pass

(IN) Points to a pass structure, which points to the data being transferred and tells how and where to send the request. For a discussion of the structure, including packetizing and retry options, see ATAtpPass_t.

Return Values

0	Success
EBADF	You specified an invalid file descriptor.
EINVAL	The packet is larger than the ATP maximum of ATP_DATA_SIZE bytes. Or you specified a negative number of retries. Or you specified an invalid destination address.
ENOBUFS	Insufficient buffer space is available.
ETIMEDOUT	Timed out (returned in *ATAtpGet*)

Remarks

ATAtpSendReq's answers return in **ATAtpGet**, including return values.

In **ATAtpSendReq**, a transaction ID is available to allow the client to match a request with the response. It can be any number. The Transaction ID sent over the network is a different number, generated internally by ATP. The transaction ID generated by the requestor is returned in **ATAtpGet** when the response appears. Also, if you want to cancel the request, you must present this transaction ID.

This transaction ID need not be unique, only unique enough for your needs. If you have only one request outstanding at a time, for example, no transaction ID is necessary.

See Also

ATAtpCancelSendReq, **ATAtpGet**

Example

ATAtpSendReq

```
#include <atp.h>

main ()
{
    ATAtpPass_t send_pass;
    int err, my_fd;
.
.
. Get socket, file descriptor, AppleTalk address. See the
.                     sample program "ATPCLNT.C".
.
.
/*
    Fill in the pass data structure.
*/
/*
   *buf_p is a global that contains the desired
   AppleTalk address
*/
    send_pass.at_addr.net = buf_p[0].enu_addr.net;
    send_pass.at_addr.node = buf_p[0].enu_addr.node;
    send_pass.at_addr.socket = buf_p[0].enu_addr.socket;
    send_pass.bitmap = 1;
    send_pass.retry.interval = 2000;
    send_pass.retry.retries = 2;
    send_pass.retry.backoff = 2;
    send_pass.ret = 0;
    send_pass.data = buffer;
    send_pass.data_len = strlen(buffer)+1;
```

```
        send_pass.userdata[0] = 0;
        send_pass.xo= 0;
        send_pass.xo_relt = ATP_XO_DEF_REL_TIME;
        send_pass.packetize = 0;
        do
        {
          if(err = ATAtpSendReq(my_fd,&send_pass))
          {
             printf("ATAtpSendReq failed: err = %x\n",err);
             ATAtpClose(my_fd);
             exit(__LINE__);
          }
        }
}
```

ATAtpSendRsp

Sends a response to a remote requesting endpoint
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <atp.h>

int ATAtpSendRsp (
   int            fd,
   ATAtpPass_t   *pass);
```

Parameters

fd

(IN) Specifies the file descriptor associated with the socket sending the TResp packet. The file descriptor is returned by the function **ATAtpOpen**, which opens the socket.

pass

(IN) Points to a pass structure, which points to the data being transferred. For a discussion of this structure, including its packetizing and retry options, see ATAtpPass_t.

Return Values

0	Success
EBADF	Invalid file descriptor
EINVAL	The packet is larger than the ATP maximum of ATP_DATA_SIZE.

The following two error codes can be returned in the **ATAtpGet** function:

ENOBUFS---insufficient buffer space

ENOENT---When the function sends an XO response, ATP cannot find the transaction record (the *tid* parameter is invalid), or the responder did not send a response within 30 seconds.

Remarks

ATAtpSendRsp returns as soon as ATP queues the response packets for transmission to the requester.

IMPORTANT: The Transaction ID passed in the **ATAtpGet** function that got the request must be passed back in the response.

Call **AtpSendRsp** to respond to a request retrieved with **ATAtpGet**. All errors are be returned in **ATAtpGet**, and identified by a Transaction ID.

ATAtpSendRsp is a quick function.

Receiving Replies

In the ATAtpPass_t structure, the *bitmap* field is set to show the exact mapping of reply packets received. The *userdata* array is set to the userdata field of each received response packet, or, in the case of a request, is *userdata*[0]. The *packet_len* array is set to the number of bytes of ATP data received in each packet. (Usually this information is not needed.) The *data* buffer, supplied by the application, is a concatenation of all data received.

Sending Replies

You can send replies in two ways---ignoring packetization or controlling packetization. Although the vast majority of replies ignore packetization, both methods are included in the following discussion.

To ignore packetization, set the *pass->packetize* variable to 0. Set the *data* buffer to the data you want to send in the transaction. Since you won't know which *userdata* fields are set, set only *userdata*[0]. ATP fills the packets for *userdata*[1]-[7] to the maximum size (MAX_ATP_DATA) and ignores *bitmap* field.

The packetization control fields have been included because some clients can require more control; notably PAP requires that only 512 bytes of ATP data be sent in every reply. In the general design of ATP, the application shouldn't have responsibility for the exact control of packets (how many packets, and so on) but should instead be controlling transactions.

In the ATAtpPass_t structure, the *bitmap* field maps the arrangement of *pass->userdata* packets, in right to left order. For example, a *pass->bitmap* hexadecimal value of 85 (binary 10000101) specifies *pass->userdata* packets in in the first, third, and eighth elements (*pass->userdata*[0], *pass->userdata*[2], and *pass->userdata*[7]).

Packetization is controlled in the ATAtpPass_t structure in the following way. Set the *pass->packetize* to 1. Set the *pass->bitmap* field to the exact mapping of response packets that you want to send. Set the *pass->userdata*[ ] fields for the packets you want to send. Set the lengths of these packets in the *pass->packet_len* field. A length of 0 means that there is no ATP data, but that there is a response packet and a *pass->userdata*[ ]

entry. Set the *pass->data* and *pass->data_len* fields to the total amount of data you want to send. Make sure the sum of the *pass->packet_len* array is equal to the total size of the data, that is, all the *pass->data_len* field.

For example, if you need to send three reply packets, numbers 0, 1, and 2, with *userdata* fields of 3465, 6223, 9867 and data of 0, 100, and 200 bytes, respectively. Set the *pass->bitmap* field to the hexadecimal value 07 (binary 00000111). Set *pass->userdata*[0] to 3465, *pass->userdata*[1] to 6223, and *pass->userdata*[2] to 9867. The *pass->data* buffer should point to the 300 bytes to be sent. The *pass->data_len* field should be 300. The *pass->data_len*[0] field should be 0, *pass->data_len*[1] should be 100, and *pass->data_len*[2] should be 200.

In requests, *bitmap* specifies responses allowed, and *userdata*[0] is the *userdata* in the request. If you are receiving requests, these values are passed through **ATAtpGet**; *bitmap* is already set, and *userdata*[0] has the *userdata* in the request packet.

NOTE: It is not possible to send a zero-buffer response. If you set *pass->packetize* to 1 and *pass->bitmap* to 0, no error would be generated, and one packet would be sent. However, the contents of this buffer would depend entirely on the rest of the ATAtpPass_t structure. Whatever was in *pass->userdata*[0] (very possibly garbage) would be the ATP header's user data field. No ATP data would be in the packet, regardless of settings in the *pass->data_len* and *pass->data* fields; in this case, these fields would be ignored.

IMPORTANT: Byte ordering is a crucial issue with DDP packets. See AppleTalk Byte Ordering.

See Also

ATAtpGet, **ATAtpCancelSendReq**, **ATAtpSendReq**, **ATAtpCancelRecvReq**

Example

ATAtpSendRsp

```
#include <atp.h>

main()
{
    ATAtpPass_t resp_pass;
.
.
. Open a socket, do some work.
.
.
/*
    Fill in the pass structure.
```

```
*/
    resp_pass.at_addr = rcv_pass.at_addr;
    resp_pass.bitmap = 1;  /* For documenation only--1 packet. */
    resp_pass.packetize = 0;
    resp_pass.retry.interval = 2000;
    resp_pass.retry.retries = 2;
    resp_pass.retry.backoff = 2;
    resp_pass.ret =0;
    resp_pass.data = resp_buf;
    resp_pass.data_len = strlen(resp_buf)+1;
    resp_pass.userdata[0] = 0;
    resp_pass.xo = 0;
    resp_pass.xo_relt = ATP_XO_DEF_REL_TIME;
    resp_pass.TransID = rcv_pass.TransID;
    if(err= ATAtpSendRsp(my_fd,&resp_pass) )
    {
      printf("ATAtpSendRsp failed: err = %x\n",err);
      ATAtpClose(my_fd);
      exit(__LINE__);
    }
}
```

ATDdpClose

Closes a socket opened by **ATDdpOpen**
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpClose (
   int   fd);
```

Parameters

fd

(IN) Specifies the file descriptor associated with the opened socket.

Return Values

0	Success
EBADF	Someone else closed the file descriptor; you should stop trying.
STREAMS error	Try again

Remarks

ATDdpClose closes the file descriptor, freeing up any internal structures and throwing away all undelivered packets, making all other calls complete.

Notice that **ATDdpClose** causes no network traffic.

IMPORTANT: ATDdpClose removes all listeners on this file descriptor and all NBP registrations.

See Also

ATDdpOpen

Example

ATDdpClose

```
#include <ddp.h>

main()
{
     int my_fd;
     /*
       Open a socket.
     */
     my_fd = 0;
     if (err = ATDdpOpen(&my_fd))
     {
       printf("ATDdpOpen failed: err= %x\n",err);
       exit(__LINE__);
     }
     /*
       Close the DDP socket.
     */
     if(err = ATDdpClose(my_fd))
     {
       printf("ATDdpClose failed: err = %x\n",err);
       exit(__LINE__);
     }
}
```

ATDdpDeregisterListener

Removes a listener on the specified file descriptor
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpDeregisterListener (
   int         fd,
   ATSocket    sock);
```

Parameters

fd

(IN) Specifies a file descriptor.

soc

(IN) Specifies the AppleTalk socket number to de-register.

Return Values

0	Success
EBADF	Someone else closed the file descriptor and you should stop trying.
ENOENT	The socket to be de-registered is not registered.
STREAMS errors	Try again.

Remarks

This file descriptor stops receiving all DDP packets destined to that socket.

The de-registration functionality also occurs when the DDP socket is closed.

IMPORTANT: Be aware that if **ATDdpDeregisterListener** is called

with a socket of 0, all listeners are closed on this file descriptor.

See Also

ATDdpOpen

Example

ATDdpDeregisterListener

```
#include <ddp.h>

main()
{
    int my_fd;
    ATInet_t newsock;
    /*
      Open a socket.
    */
    my_fd = 0;
    if (err = ATDdpOpen(&my_fd))
    {
      printf("ATDdpOpen failed: err = %x\n",err);
      exit(__LINE__);
    }
    newsock.socket = 0;
    if(err = ATDdpRegisterListener(my_fd,&newsock))
    {
      printf("ATDdpRegisterListener failed: err = %x\n",err);
      ATDdpClose(my_fd);
      exit(__LINE__);
    }
}
```

ATDdpNetinfo

Gets information about this interface
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpNetinfo (
   int         fd,
   ATInet_t   *node,
   ATInet_t   *router,
   int        *flags);
```

Parameters

fd

(IN) Specifies a file descriptor.

node

(OUT) Points to the ATInet_t structure. The AppleTalk addresses for your net and node are passed back in that structure.

router

(OUT) Points to the ATInet_t structure and the AppleTalk address for A_ROUTER is passed back in that structure.

flags

(OUT) Points in which a long word of IF flags returns, telling whether the network that the stack is bound to is extended or nonextended:

1 = nonextended
0 = extended
2 = unbound

Return Values

0	Success
EBADF	Someone else closed the file descriptor and you should stop trying.
STREAMS	Try again.

errors	

Remarks

For a complete definition of an **extended** network, see *Inside AppleTalk, Second Edition*. Basically, an extended network allows you to have zone lists and network cable ranges. The ZIP procedures are also different, but the NetWare AppleTalk interface takes care of them for you.

If you don't happen to have an *fd* handy, you can still find out information about the network by passing in negative 1 (-1) as *fd*. The function creates a new file descriptor, get the network information, and then disposes of the file descriptor.

Any of the pointers (**node*, **router*, **flags*) can be passed in as NULL if that particular type of information is not requested. For example, if you don't want to ask about the router, pass in 0.

Be aware that A_ROUTER changes constantly.

No registered listeners are necessary to call this function. No network traffic occurs.

ATDdpNetinfo succeeds when no node is bound.

For PAP, ASP, and ATP clients, **ATDdpNetinfo** succeeds on those kinds of file descriptors. The socket is set to a nonzero value in the cases of PAP and ASP, but if there is a case of multiple listeners in DDP or ATP, the value is set to 0 (or to the listener if there is only one listener).

Example

ATDdpNetinfo

```
#include <ddp.h>

main()
{
    ATInet_t info;              /* The server's AppleTalk internet addre
    ATInet_t router_info;       /* The server's adjacent AppleTalk
      router address. */
    int flags;
    /*
      Get configuration information.
    */
   flags = 0;
   if(err = ATDdpNetinfo(my_fd,&info,&router_info, &flags) )
   {
      printf("ATDdpNetinfo failed: err = %x\n", err);
      ATDdpClose(my_fd);
```

```
            exit(__LINE__);
        }
    }
```

ATDdpOpen

Opens a file descriptor on the local host for reading and writing but does not bind an AppleTalk socket to this file descriptor

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpOpen (
   int   *fd);
```

Parameters

fd

(OUT) Points to a file descriptor. Returns the file descriptor opened here, to be used with **ATDdpRead** and **ATDdpWrite**.

Return Values

0	Success

Remarks

You should be aware that **ATDdpOpen** succeeds even if no node is bound yet, because when APPLETLK.NLM is loaded, it is not yet attached (**bound**) to a network. It can't communicate, since the stack node is not bound to a card. However, you can still open DDP, open sockets, and do various other things.

To bind a socket, call **ATDdpRegisterListener**. The file descriptor is (currently) a full STREAMS file descriptor, and thus responds to the **poll** function.

Note that **ATDdpOpen** causes no network traffic and is a blocking function.

To close a socket opened with this function, call **ATDdpClose**.

See Also

ATDdpClose

Example

ATDdpOpen

```
#include <ddp.h>

main()
{
     int my_fd;
     /*
       Open a socket.
     */
     my_fd = 0;
     if (err = ATDdpOpen(&my_fd))
     {
       printf("ATDdpOpen failed: err= %x\n",err);
       exit(__LINE__);
     }
}
```

ATDdpRead

Reads a packet
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpRead (
   int        fd,
   ATDdp_t   *ddp,
   int       *len);
```

Parameters

fd

(IN) Indicates the file descriptor.

ddp

(OUT) Points to the data buffer to copy the packet into. See ATDdp_t.

len

(IN/OUT) Points to int, passed in with buffer length, passed out with number of bytes read.

Return Values

0	Success
EMSGSIZE	The packet you read is too large to fit in the buffer you provided.
ENETDOWN	The network you attempted to send a datagram on is down.
ENOBUFS	Insufficient buffer space is available.
ENOTCON	The socket you specified has not been opened.

Remarks

IMPORTANT: If you supply a buffer smaller than the actual

incoming packet, **ATDdpRead** does not let you know that there was more data to be read. Instead, you get the rest of the data the next time you do a read. For this reason be sure to set the buffer to the maximum datagram size, DDP_DATAGRAM_SIZE.

A DDP file descriptor must register listeners before it receives any packets. Call **ATDdpRegisterListener**.

Packet headers are always DDP long format. They are expanded if received as short.

The DDP header is set to whatever was received (with the possible exception of the *checksum* field, which might be cleared after the sum has been checked.)

The **poll** function can be used to make this function take less time. Otherwise the function waits until the next packet arrives.

See Also

ATDdpWrite

Example

ATDdpRead

```
#include <ddp.h>

main()
{
    ATDdp_t recv_pkt;   /* The server's received packet structure. */
    int recv_len;
    /*
      The read function will block until data is received. The server
      will then send a response back to the requesting node.
    */
    recv_len = DDP_DATA_SIZE;
    if(err = ATDdpRead(my_fd,&recv_pkt,&recv_len))
    {
      printf("ATDdpRead failed: err = %x\n",err);
      ATDdpClose(my_fd);
      exit(__LINE__);
    }
}
```

ATDdpRegisterListener

Allows a DDP client to begin listening on a particular socket
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpRegisterListener (
   int          fd,
   ATInet_t   *addr);
```

Parameters

fd

(IN) Specifies a file descriptor.

addr

(IN/OUT) Points to AppleTalk address and returns the address you were allocated.

Return Values

0	Success
EBADF	Someone else closed the file descriptor and you should stop trying.
ENOENT	The socket requested is not available.
EADDRINUSE	Another protocol client is using the socket you specified.
EADDRNOTAVAIL	No sockets are available.

Remarks

The *addr* parameter returns the entire DDP address that the listener is on. This can change as routers are attached and removed from the network and as services are bound and unbound. If there is no node bound, the net, node is 0,0.

AppleTalk sockets fall into two categories: statically assigned sockets and dynamically assigned sockets. Statically assigned sockets are denoted by numbers 1 through 127; dynamically assigned sockets are denoted by numbers 128 through 254. Apple* Computer, Inc. reserves the use of sockets 1 through 63; you can use sockets 64 through 127 experimentally, **but not in released products**.

For **ATDdpRegisterListener** to open a statically assigned socket, the *addr.socket* field must be the socket number in the static range that you want to use.

For **ATDdpRegisterListener** to open a dynamically assigned socket, *addr.socket* must be 0, or *addr* must be a NULL pointer. If *addr* is non-null, the function returns the dynamic socket number in *addr.socket* and, as a courtesy, *addr.net* and *addr.node* are set to the node's current net and node numbers. If you specify a socket number in the dynamic range, the function generates the error EINVAL.

Note that multiple listeners can be registered on the same file descriptor.

ATDdpRegisterListener allows a DDP client to allocate a dynamic socket. However, make sure you know which one has been allocated because, although it would be possible to send out packets on a socket that you are not a listener for, that would be in very bad taste.

This interface does not support opening socket 0xff.

ATDdpRegisterListener succeeds when no node is bound; however, no packets are received. Periodically, a message is posted to the console saying that there are listeners registered but no node bound. If a node does become bound, it immediately becomes functional and the listener starts receiving packets. If the node is unbound while a listener is registered, the listener stops receiving packets but won't be notified in any other way.

Novell® has traditionally not allowed the static allocation of sockets in the dynamic range (128 through 254) but did, in a previous version of these APIs, allow allocation of dynamic sockets in the static range (1 through 127) when necessary, for example when there weren't any free sockets in the dynamic range. Now, neither of these types of allocation is allowed.

See Also

ATDdpOpen

Example

ATDdpRegisterListener

```
#include <ddp.h>
```

```
#include <ddp.h>

main()
{
    int my_fd,err;
    ATInet_t newsock;
    /*
      Open a socket.
    */
    my_fd = 0;
    if (err = ATDdpOpen(&my_fd))
    {
      printf("ATDdpOpen failed: err = %x\n",err);
      exit(__LINE__);
    }
    newsock.socket = 0;
    if(err = ATDdpRegisterListener(my_fd,&newsock))
    {
      printf("ATDdpRegisterListener failed: err = %x\n",err);
      ATDdpClose(my_fd);
      exit(__LINE__);
    }
}
```

ATDdpWrite

Sends a packet
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <ddp.h>

int ATDdpWrite (
   int        fd,
   ATDdp_t   *ddp,
   long       len);
```

Parameters

fd

(IN) Specifies a file descriptor to send packet on.

ddp

(IN) Points to packet data, including header, as data. See ATDdp_t.

len

(IN) Indicates an unsigned four-byte for packet length.

Return Values.

0	Success
EMGSIZE	The packet you sent is too large or too small.
ENETDOWN	The network you attempted to send a datagram on is down.
ENOBUFS	Insufficient buffer space is available.
ENOTCON	The socket you specified has not been opened.
EINVAL	Bad length; illegal type.

Remarks

In the ATDdp_t structure, the source net/node is overwritten by the

stack, but not the source socket number---which **must** be filled in by the API client---and destination address. Hop count and ddp length need not be filled in. DDP type is of your own choosing. The destination net, node, and socket are where to send the packet to.

To send a broadcast packet, send the packet to network 0, socket ff for a cable-wide broadcast. For a network-specific broadcast, send the packet to network net, socket ff. There is no way to do a cable broadcast for a cable you aren't on (AppleTalk limitation). Currently this interface doesn't support sending multicasts.

If the *checksum* field is nonzero, DDP calculates the checksum.

Packet headers must be DDP long format.

IMPORTANT: **ATDdpWrite** should not block ("sending" packets are never queued), but the packet can be discarded if no sending packet buffers are available (and no error is returned).

Also a note about implementation: If the packet is destined to the same node it was sent from, queuing and context switches occur, which can back up STREAMS flow control.

IMPORTANT: Remember, if the packet can't be delivered, no error is returned.

See Also

ATDdpRead

Example

ATDdpWrite

```
#include<ddp.h>

main()
{
    ATDdp_t send_pkt;   /* The server's send packet structure. */
    /*
      Prepare data to send back to the client. Fill in the send_pkt dat
      structure.
      NOTE: The source net received in recv_pkt.src_net is already in
      network order.
    */
    send_pkt.dst_net     =  recv_pkt.src_net;
    send_pkt.dst_node    =  recv_pkt.src_node;
    send_pkt.dst_socket  =  recv_pkt.src_socket; /* Source socket. */
    send_pkt.type        =  73;                  /* The server's protoc
    send_pkt.src_socket  =  info.socket;
    send_pkt.checksum    =  0;                   /* Don't calculate the
    strcpy(send_pkt.data,myMsg);                 /* Message to the clie
```

```
        send_len = + strlen(myMsg) + 1;
        /*
          Now, everything is in order, so send the data to the client.
        */
        if( err = ATDdpWrite(my_fd,&send_pkt,send_len) )
        {
          printf("ATDdpWrite failed: err = %x\n",err);
          ATDdpClose(my_fd);
          exit(__LINE__);
        }
    }
```

ATNbpConfirm

Verifies that the network address of the specified NBP entity is still valid
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpConfirm (
   ATEntity_t   *entity,
   ATInet_t     *dest,
   ATRetry_t    *retry);
```

Parameters

entity

(IN) Points to the ATEntity_t structure that contains the three-part name (object, type, and zone) of the NBP entity whose address you want to confirm.

Each member of the ATEntity_t structure is an ATNveStr_t structure, or a P string (length-preceded string) whose maximum size is 32 bytes (NBP_NVE_STR_SIZE). See ATNveStr_t.

You cannot include wildcards in the entity name. **ATNbpConfirm** determines whether the entity name you have specified is valid.

dest

(IN) Points to the ATInet_t structure that contains the entity's AppleTalk internet address (net,node,socket). See ATInet_t.

If the net and node are still valid but the socket has changed, the function returns EFAULT. In that case, call **ATNbpDirectedLookup** to find out the current socket number.

retry

(IN) Determines how long the function keeps trying before it times out.

You have the choice of passing a pointer to an ATRetry_t structure that you have filled in with specific retry settings, or passing in NULL (0) for the system defaults. For suggestions on making this decision, see ATRetry_t.

Remember that, on return, the string containing the object, type, or zone name is **not** NULL-terminated.

Return Values

0	Mapping is still valid.
EINVAL	You specified an invalid parameter.
ENOENT	You specified an invalid entity.
ETIMEDOUT	Function timed out.
EFAULT	The entity is registered at the same net,node but uses a different socket. Call **ATNbpDirectedLookup** to find out the current socket number.

Remarks

ATNbpConfirm verifies that the network address of the specified NBP entity is still valid by sending an entity-to-address mapping (*entity*-to-*dest*) to the address you specify. If the mapping is still valid, **ATNbpConfirm** returns 0. If it is not valid, **ATNbpConfirm** returns an error.

Call **ATNbpConfirm** when you have previously performed an **ATNbpLookup** and you want to make sure that mapping is still valid.

Although you could always confirm an address by calling **ATNbpLookup** again, **ATNbpConfirm** is highly recommended instead because it generates considerably less network traffic, especially on complex internets. Even when each zone is associated with a single network, **ATNbpConfirm** generates less network traffic than **ATNbpLookup**, although the difference is less dramatic.

NOTE: It is possible for a network service to have changed sockets while retaining the same net, node. In this case, **ATNbpConfirm** returns the EFAULT error. Call **ATNbpDirectedLookup** to get the current socket.

Note, too, that **ATNbpConfirm** blocks until it gets a response, an error, or times out. Most of the time, it either returns immediately or waits for the entire retry period. For this reason, NBP functions tend to run a greater risk of being cut off in the middle of their process because someone unloads the NLM while these functions are blocking.

See Also

ATNbpLookup, **ATNbpParseEntity**

Example

ATNbpConfirm

```
#include    <nbp.h>

main()
{
    int           err, max, more;
    ATEntity_t    e;
    ATInet_t      a;
    ATNbpTuple    t;
    /*
      Retrieve stored AppleTalk address that you have for this service,
      stash it in a.
    */
    /* Retrieve stored entity name for this service. */
    err = ATNbpConfirm(&e, &a, 0);
    if (err == ETIMEDOUT)
    {
      /* Guess that stashed value is bad, try to lookup for service */
      max = 1;
      more = 1;
      err = ATNbpLookup(&e, &t, &max, 0, &more);
      if (!err)
      {
         a = t.enu_addr;
      }
    }
    if (err == 0) {
      /* Got a great address! Try to connect! */
    }
    return;
}
```

ATNbpDirectedLookup

Directs a lookup query to a particular node
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpDirectedLookup (
   ATEntity_t     *entity,
   ATInet_t       *addr,
   ATNbptuple_t   *buf,
   int            *max,
   ATRetry_t      *retry,
   int            *more);
```

Remarks

Like **ATNbpLookup**, **ATNbpDirectedLookup** maps an entity's name and its internet address with two important differences:

ATNbpDirectedLookup dispenses with the overhead of searching the entire distributed database on an internet; it inquires only about services on a particular node (*addr*).

Whereas **ATNbpConfirm** gives a yes/no-type answer, this function returns a list of services available on the node (*addr*).

Otherwise, **ATNbpDirectedLookup** is identical to **ATNbpLookup**.

NOTE: **ATNbpDirectedLookup** differs from **ATNbpConfirm** in two ways. The socket is not specified and wildcards are allowed.

ATNbpLookup

Searches of the names directory and returns a mapping of the specified entity's name to its internet addresses

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpLookup (
   ATEntity_t      *entity,
   ATNbptuple_t    *buf,
   int             *max,
   ATRetry_t       *retry,
   int             *more);
```

Parameters

entity

(IN) Points to the ATEntity_t structure that has the complete name (object, type, zone) of the entity you are inquiring about. See ATEntity_t, which includes a description of the wildcards you can use in an entity's name.

buf

(IN) Points to an array of ATNbpTuple_t structures; the function fills in the structures pointed to by *buf* with the name-address tuples it returns. See ATNbptuple_t.

max

(IN/OUT) Indicates the size of the buffer (pass in from 1 to *n* as the maximum number of tuples the buffer holds.

There is no maximum number of tuples that could be generated. For example, there could be a large number of NBP entities registered for each end node in a zone. A typical number is 1-4 per Macintosh* computer: one for InterPoll/responder, one for a mail package, one for personal AppleShare*, and one for another entity. So if a zone has 1,000 Macintoshes, 1,000 to 4,000 tuples would typically be received on a wildcard lookup.

max returns the total number of tuples received by the buffer as a result of calling this function.

retry

(IN) Determines how long the function keeps trying before it times out.

You have the choice of passing a pointer to an ATRetry_t structure that you have filled in with specific retry settings, or passing in NULL (0), which gives you the system defaults. For guidelines on making this choice, see ATRetry_t.

Notice that since there are multiple retries, there are multiple responses from a given node. NBP filters these multiple responses, and present the client with only one response for each entity. Matching is done as described in *Inside AppleTalk* (case matching against MacASCII is done).

more

(IN/OUT) Lets you control whether the buffer returns its contents as soon as it fills up or waits until it overflows by one tuple (or until the function times out):

0---Return buffer's contents as soon as it fills up ---Wait until buffer overflows to return its contents *0 (NULL pointer)---Use system default (currently, wait until buffer overflows or function times out).

If you pass in 1 or a NULL pointer and the buffer never overflows, 6 seconds or so elapse before the buffer returns its contents. On return, *more* tells you whether the buffer overflowed or not:

0---Buffer returned the exact number of tuples, or less than what was requested in the *max* parameter 1---Buffer overflowed

> **NOTE:** Remember that the size of the buffer is expressed in tuples, not bytes.

Return Values

0	Success
EINVAL	You specified an invalid value (bad wildcards).
ENOBUFS	Insufficient buffer space is available. Try again. (Don't believe the parameters passed back.)
ENOENT	You specified an invalid entity.
ETIMEDOUT	No entity found within the time and number of retries specified. Also, 0 length in list.

Remarks

ATNbpLookup makes a search of the **names directory**, a database of

network entities that is distributed throughout the internet, and returns a mapping of the specified entity's name to its internet address(es). This mapping is known as an NBP **name-address tuple**.

Special characters (wildcards) are allowed, as described in ATEntity_t (also in *Inside AppleTalk*, p. 7-4). The **name** part of the tuple is human-readable. **ATNbpLookup** is often used with wildcards to display a list of entities to the user. The user can then choose the entity desired.

For your application to access an entity, it needs to find out the entity's internet address. The **ATNbpLookup** function searches throughout the internet and returns the internet address of the entity or entities that you specified. (For more information about the names directory, see **ATNbpRegister** or **ATNbpRegByAddr**.)

IMPORTANT: The *more* parameter gives you the option of forcing the buffer's contents to return before the customary waiting period has elapsed. Lookup Functions usually wait about six seconds to make sure all the possible name-address pairs have been received. See the discussion of the *more* parameter for details.

If the buffer does overflow, try again with either a larger buffer or do a more specific search (remove wildcards).

If you know how many tuples are returned, you can avoid waiting the 6 seconds or so that **ATNbpLookup** waits before returning the contents of the buffer. For example, if you are sure your lookup returns only one tuple (you have used no wildcards), pass in1 as *max*, which sets the buffer size to only 1 tuple, and pass in 0 as *more*, which causes the buffer to return its contents as soon as it fills up. Consequently, as soon as a single tuple is received, the buffer returns its contents, significantly cutting the amount of time involved in lookup.

See Also

ATNbpConfirm, **ATNbpDirectedLookup**, **ATNbpParseEntity**

Example

ATNbpLookup

```
#include <nbp.h>

main()
{
    ATEntity_t LookUpEntity;
    retry.interval = 1000;      /* Timeout Value. */
    retry.retries = 2;          /* Number of retries.*/
    retry.backoff = 2;          /* Gradually increases in time between r
    max_tuples = 1;
    more = NULL;
    /* Note: buf_p is a global. */
```

```
        nbp_tuples_found = ATNbpLookup(&LookUpEntity,buf_p,&max_tuples,
                                       &retry,&more);
    }
```

ATNbpMakeEntity

Creates a complete three-part NBP entity
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpMakeEntity (
   ATEntity_t   *entity,
   char         *object,
   char         *type,
   char         *zone);
```

Parameters

entity

(IN/OUT) Points to the ATEntity_t structure that is filled in by the function with the specified object, type, and zone. See ATEntity_t. **ATNbpMakeEntity** determines whether the entity name you specify is valid before returning a value.

object

(IN) Points to a NULL-terminated string of up to 32 characters that specifies the entity's name.

type

(IN) Points to a NULL-terminated string of up to 32 characters that specifies the entity's type.

zone

(IN) Points to a NULL-terminated string of up to 32 characters that specifies the zone in which the entity resides. To indicate the local zone, you can specify star (*).

NOTE: On return, the ATEntity_t's strings containing the object, type, or zone name are not NULL-terminated!

Return Values

0	Success

EINVAL	Incorrectly formatted strings

Remarks

The **ATNbpMakeEntity** function creates a complete three-part NBP entity name by copying the object, type, and zone you specify into the ATentity_t structure pointed to by *entity*. Use **ATNbpMakeEntity** to create an NBP entity name for a node or service that you want to register on the internet (using **ATNbpRegister** or **ATNbpRegByAddr**).

NOTE: You pass in object, type, and zone as NULL-terminated C strings but they become P strings (length-preceded strings) in the ATEntity_t structure. See the explanation in ATEntity_t.

See Also

ATNbpConfirm, **ATNbpLookup**, **ATNbpMakeEntityXlate**, **ATNbpParseEntity**, **ATNbpRegister**, **ATNbpRemove**

Example

ATNbpMakeEntity

```
#include <nbp.h>

main()
{
ATEntity_t *Made_Entity;
char my_object[]="THE_OBJECT_NAME";
char my_type[]="THE_TYPE";
char my_zone[]='THE_ZONE";
.
.
. Do some work.
.
.
    if( err = ATNbpEntity(Made_Entity,my_object,my_type,my_zone) )
    {
      printf("ATNbpEntity failed: err = %x\n",err);
      exit(__LINE__);
    }
}
```

ATNbpMakeEntityXlate

Same as **ATNbpMakeEntity**, but also translates the incoming object, type, and zone strings from the local code page to the MacASCII code page

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpMakeEntityXlate (
   ATEntity_t   *entity,
   char         *object,
   char         *type,
   char         *zone);
```

Remarks

A NetWare specific version of **ATNbpMakeEntity**, **ATNbpMakeEntityXlate** has the same uses but also translates the incoming object, type, and zone strings from the local code page to the MacASCII code page. You would call **ATNbpMakeEntityXlate** to translate an entity's name from server code page to MacASCII code page before calling **ATNbpLookup**. The mechanics of this function are identical to **ATNbpMakeEntity**.

See Also

CstrIBMCPToMac, **CstrMacToIBMCP**, **PstrIBMCPToMac**, **PstrMacToIBMCP**

ATNbpParseEntity

Converts a string from the "object:type@zone" form to the three-part NBP entity form

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpParseEntity (
   ATEntity_t   *entity,
   char         *str);
```

Parameters

entity

(OUT) Points to the ATEntity_t structure that **ATNbpParseEntity** fills in with the entity name it resolves from the specified string. See ATEntity_t.

str

(IN) Points to the NULL-terminated string that you want the function to resolve into an entity name.

NOTE: On return, the string containing the object, type, or zone name is not NULL-terminated.

Return Values

0	Success
-1	Unsuccessful
EINVAL	Incorrectly formatted string, string too long

Remarks

ATNbpParseEntity converts a string from the "object:type@zone" form to the three- part NBP entity form. (For details on this form see ATEntity_t). Call **ATNbpParseEntity** to construct an NBP entity name that you want to find on the internet later (for example, with **ATNbpLookup**).

ATNbpParseEntity is useful when an NBP tuple is stored in a file because some user interfaces are based on single-string names, such as "johnd@wc.novell.com." The AppleTalk interface is based on three-string names, and this function converts single-string names to three-string.

The string is NULL-terminated and can have any of the following forms:

object
object:type
object:type@zone

Each component of str can contain up to 32 characters. If you do not specify a zone, **ATNbpParseEntity** substitutes a star (*), indicating "local zone." If you do not specify a type, **ATNbpParseEntity** substitutes an equal sign (=), indicating "any type."

It is assumed the string is in C string format (NULL-terminated).

The parsing routine looks for the first colon (:), then the first ampersand (@). Multiple colons and multiple ampersands are allowed, but the rule is first colons, then ampersands.

Like all other NBP functions, it is assumed that the string you pass in is in MacASCII. If you don't have a MacASCII string, use one of the conversion functions first.

See Also

ATNbpLookup, **ATNbpMakeEntity**

Example

ATNbpParseEntity

```
#include <nbp.h>

main()
{
    char nbp_name[] = "MY_OBJECT.MY_TYPE@MY_ZONE";
    ATEntity_t thisEntity;
    if (err = ATNbpParseEntity(&thisEntity, nbp_name))
    {
      printf("ATNbpParseEntity failed: err = %x\n",err);
      return( err ); /* Clean up and exit is done in main. */
    }
}
```

ATNbpRegByAddr

Registers a service by its AppleTalk socket number alone, not by its file descriptor as **ATNbpRegister** does

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpRegByAddr (
   ATEntity_t  *entity,
   ATInet_t    *addr,
   ATRetry_t   *retry);
```

Parameters

entity

(IN) Points to the at_entity_t structure containing the name you want to register. See ATEntity_t. The name must be unique in the zone. **ATNbpRegister** determines whether the entity name you specify is valid before returning a value.

addr

(IN) Specifies the AppleTalk socket where you register the service (cannot be zero).

retry

(IN) Determines how long **ATNbpRegByAddr** tries to find duplicates on the net before it times out. You have the choice of passing a pointer to an ATRetry_t structure that you have filled in with specific retry settings, or passing in NULL, which gives you the system defaults. For suggestions on making this choice, see ATRetry_t.

Return Values

0	Success
EADDRNOT AVAIL	The socket is already registered to another service.
EINVAL	You specified a bad socket number to register on.
ENOENT	You specified an invalid entity (in this case, a socket).

ETIMEDOUT	**ATNbpRegByAddr** did not return a value within the time interval and the number of retries you specified.
ESOMETHING	You specified a name that is already registered to another node.
ENOBUFS	Out of memory, try again.

NOTE: Note that if the socket number is invalid, **ATNbpRegByAddr** can return either EBADF or ENOENT. Also, any number of STREAMS errors could be returned if something went wrong in STREAMS.

Remarks

ATNbpRegByAddr registers a service by its AppleTalk socket number alone, not by its file descriptor as **ATNbpRegister** does. Otherwise it is identical to **ATNbpRegister**, including support for explicit zone registration when an AppleTalk internal network is present (not to be confused with the IPX internal network). In explicit registration, the NBP client specifies exactly which zone to register in. The choice is from the zones on the internal network's list.

Call **ATNbpRegByAddr** if you don't have a file descriptor for the service you want to register. Other nodes on the internet are then able to access the service by socket.

When you want to remove the entity from the internet, you should either remove its registration with **ATNbpRemoveByAddr** or wait for the socket to close. Even without a file descriptor, a socket close causes the destruction of all NBP registrations.

IMPORTANT: The AppleTalk socket cannot be zero (0).

See **ATNbpRegister** for information about registering a service.

See Also

ATNbpMakeEntity, **ATNbpParseEntity**, **ATNbpRegister**, **ATNbpRemoveByAddr**

Example

ATNbpRegByAddr

```
#include<nbp.h>

main()
{
int err;
ATRetry_t retry;
```

```
char nbp_name[]="MY_NAME.MY_TYPE@*";
ATEntity_t thisEntity;
ATInet_t a;
    /*
      Set up the structure that holds the timeout and retries.
    */
    retry.interval = 1000;   /* Timeout Value. */
    retry.retries  = 8;      /* Number of retries.*/
    retry.backoff = 2;       /* Gradually increases in time between ret
    if (err = ATNbpParseEntity(&thisEntity, nbp_name))
    {
      printf("ATNbpParseEntity failed: err = %x\n",err);
      return( STREAM_ERROR );         /* Clean up and exit is done in m
    }
    a.socket = 100;
    if (err = ATNbpRegisterByAddr( &thisEntity, &a, (ATRetry_t *)&retry
    {
      printf("ATNbpRegister failed: err = %x\n",err);
      return(STREAM_ERROR);           /* Clean up and exit is done in m
    }
}
```

ATNbpRegister

Registers a service by file descriptor so that it becomes visible on the internet
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpRegister (
   ATEntity_t  *entity,
   int          fd,
   ATRetry_t   *retry);
```

Parameters

entity

(IN/OUT) Points to the ATEntity_t structure that contains the name you want to register. See ATEntity_t. The name must be unique in the zone. **ATNbpRegister** determines whether the entity name you specify is valid before returning a value.

fd

(IN) Specifies the entity's file descriptor.

retry

(IN) Determines how long **ATNbpRegByAddr** tries to find duplicates on the net before it times out. You have the choice of passing a pointer to an ATRetry_t structure that you have filled in with specific retry settings, or passing in NULL, which gives you the system defaults. For suggestions on making this choice, see ATRetry_t.

Return Values

0	Success
EADDRNOT AVAIL	The name-address tuple already exists.
EBADF	You specified an invalid file descriptor.
EINVAL	You specified a bad socket number to register on.
ENOENT	You specified an invalid entity (in this case, a file descriptor).

ETIMEDOUT	The function did not return a value within the time interval and the number of retries you specified.
ESOMETHING	You specified a name that is already registered to another node.
ENOBUFS	Out of memory, try again.

Note that if the file descriptor is invalid, **ATNbpRegister** can return either EBADF or ENOENT. It could also return a STREAMS error if something went wrong in STREAMS.

Remarks

ATNbpRegister registers a service by file descriptor so that it becomes visible on the internet. Going beyond the interface suggested in *Inside AppleTalk*, this function and **ATNbpRegByAddr** allow explicit zone registration **when an Appletalk internal network is present** (not to be confused with the IPX internal network). (**ATNbpRegByAddr** registers by socket alone, not complete file descriptor.) In explicit registration, the NBP client specifies exactly which zone to register in. The choice is from the zones on the internal network's list.

ATNbpRegister adds a name-to-address mapping, known as an NBP **name-address tuple**, to the **names table** on the local node. This causes NBP to add a corresponding mapping to the distributed database known as the **names directory**. The names directory is the union of the individual names tables in the nodes on the internet. The database does not require different portions to be duplicated; it can be distributed among all nodes containing named **network-visible entities** (NVEs). An entity becomes an NVE by entering its name and address into the local names table with **ATNbpRegister** or **ATNbpRegByAddr**.

When a node comes up on the internet, its names table is empty. Therefore, when restarted, each NVE must re-register its name(s) in the names table.

Registration is usually done on listeners of some sort, but can also be done on connections. After creating a listener or connection with another protocol (DDP, ATP, ADSP, ASP, PAP), call **ATNbpRegister** and pass the file descriptor you obtained from the other protocol. See the individual API description to find out at what point it is valid to call **ATNbpRegister** on the file descriptor.

A close on this file descriptor removes the registration, so an explicit call to **ATNbpRemove** is not necessary.

Explicit Registration

Contrary to suggestions in *Inside AppleTalk*, Novell's interface supports explicit zone registration, but only **when an AppleTalk (not IPX) internal network is available**.

When registering an entity with **ATNbpRegister** or **ATNbpRegByAddr**, you can either enter a star (*) as the zone to register on, meaning the zone for this node, or you can specify an exact zone. In the case of a star (*), the registration moves from zone to zone as the zone changes for the node. This is not so for an explicit registration; therefore, Novell has restricted explicit registration to the case where there is an AppleTalk internal network available; as long as there is an internal network, the zone list is stable for as long as AppleTalk is loaded.

The error EFAULT is returned for an invalid explicit zone (not EINVAL).

To promote better user interface design and to allow you to check if an explicit zone is valid in the current configuration (present on the internal network's list), Novell has created a ZIP function, **ATZipGetNBPZones**. This function returns a list of the currently valid explicit zones.

See Also

ATNbpMakeEntity, **ATNbpMakeEntityXlate**, **ATNbpParseEntity**, **ATNbpRegByAddr**, **ATNbpRemove**

Example

ATNbpRegister

```
#include<nbp.h>

main()
{
int err;
ATRetry_t retry;
char nbp_name[]="MY_NAME.MY_TYPE@*";
ATEntity_t thisEntity;
    /*
      Set up the structure that holds the timeout and retries.
    */
    retry.interval = 1000; /* Timeout Value. */
    retry.retries  = 8;    /* Number of retries.*/
    retry.backoff = 2;     /* Gradually increases in time between retri
    if (err = ATNbpParseEntity(&thisEntity, nbp_name))
    {
      printf("ATNbpParseEntity failed: err = %x\n",err);
      return( STREAM_ERROR ); /* Clean up and exit is done in main. */
    }
    if (err = ATNbpRegister( &thisEntity, fd, (ATRetry_t *)&retry))
    {
      printf("ATNbpRegister failed: err = %x\n",err);
      return(STREAM_ERROR);/* Clean up and exit is done in main. */
    }
}
```

ATNbpRemove

Removes an NBP registration from the local names table
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpRemove (
   ATEntity_t   *entity,
   int           fd);
```

Parameters

entity

(IN) Specifies the entity argument: points to an ATEntity_t structure containing the name of the entity you want to remove. See ATEntity_t.

fd

(IN) Specifies the file descriptor of the entity you want to remove.

Return Values

0	Success
EBADF	You specified an invalid file descriptor.
EINVAL	You specified an invalid parameter.
ENOENT	The name you specified was not found.

NOTE: If the file descriptor is invalid, **ATNbpRemove** can return either EBADF or ENOENT.

Remarks

NOTE: If the registration was made on a particular file descriptor (with **ATNbpRegister**) the close of that file descriptor triggers an automatic de-registration.

ATNbpRemove removes an NBP registration from the local names table. Use this function when you have the file descriptor you registered on. Typically registrations made with **ATNbpRegister** are removed with **ATNbpRemove**.

If the pointer to an NBP entity is NULL, the function removes all registrations on the specified file descriptor. If the file descriptor is NULL, it removes the NBP entity without regard to the file descriptor it is on.

For a description of the local names table, see **ATNbpRegister** or **ATNbpRegByAddr**.

ATNbpRemove blocks until the registration is deleted from the local names table.

See Also

ATNbpMakeEntity, **ATNbpMakeEntityXlate**, **ATNbpParseEntity**, **ATNbpRegister**

Example

ATNbpRemove

```
#include <nbp.h>

main()
{
ATEntity_t     thisEntity;
int            my_fd;
.
.
. Register the entity. See ATNbpRegister on page 127.
.
.
    /*
      Remove the name registered with NBP.
    */
    if( err = ATNbpRemove(&thisEntity,my_fd))
    {
      printf("ATNbpRemove failed: err = %x\n",err);
      exit(__LINE__);
    }
}
```

ATNbpRemoveByAddr

Removes an NBP registration from the local names table
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <nbp.h>

int ATNbpRemoveByAddr (
   ATEntity_t   *entity,
   ATSocket     *socket);
```

Parameters

entity

(IN) Points to an ATEntity_t structure containing the name of the entity you want to remove. See ATEntity_t.

socket

(IN) Indicates the socket that the entity was registered on.

Return Values

0	Success
EINVAL	You specified an invalid parameter.
ENOENT	The name you specified was not found.

Remarks

ATNbpRemoveByAddr removes an NBP registration from the local names table. Use this function when you do not have the file descriptor you registered with, or you registered by socket only. Typically registrations made with **ATNbpRegByAddr** are removed with **ATNbpRemoveByAddr**.

If the pointer to an NBP entity is NULL, the function removes all registrations on the specified socket.

For a description of the local names table, see **ATNbpRegister** or **ATNbpRegByAddr**.

ATNbpRemoveByAddr blocks until the registration is deleted from the local names table.

See Also

ATNbpMakeEntity, **ATNbpMakeEntityXlate**, **ATNbpParseEntity**, **ATNbpRegByAddr**, **ATNbpRegister**, **ATNbpRemove**

Example

ATNbpRemoveByAddr

```
#include <nbp.h>

main()
{
ATEntity_t thisEntity;
.
.
. Register the entity. See ATNbpRegister on page 127.
.
.
    /*
      Remove the name registered with NBP.
    */
    if( err = ATNbpRemoveByAddr(&thisEntity,100))
    {
      printf("ATNbpRemoveByAddr failed: err = %x\n",err);
      exit(__LINE__);
    }
}
```

ATPapAccept

Makes an SSS (server session socket) out of an unused file descriptor and links a connection request on the SLS (session listening socket) to this SSS

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapAccept (
   int         fd,
   void       *connect_id,
   ATInet_t   *addr,
   char       *status,
   u_short     status_len);
```

Parameters

fd

(IN) Specifies the file descriptor.

connect_id

(IN) Points to the ID of connection passed up in OPENCONN_REQ event.

addr

(IN) Points to the address of the connection, passed up in OPENCONN_REQ event.

status

(IN) Points to status buffer.

status_len

(IN) Specifies the length of the status buffer.

Return Values

0	Success
EINVAL	Invalid file descriptor
Other	STREAMS errors

Remarks

ATPapAccept makes an SSS (server session socket) out of an unused file descriptor and links a connection request on the SLS (session listening socket) to this SSS. It opens the connection, data starts flowing. **ATPapRead** and **ATPapWrite** are valid after calling **ATPapAccept**.

ATPapAccept can be called at the state when a connection has come in after making an **ATPapGetNextJob** on an SLS Stream. When the connection request occurs, a PAP_EVENT_OPENCONN_REQ event occurs on the SLS. If the client wants to accept the job, it creates a new PAP file descriptor with **ATPapOpen**, then calls **ATPapAccept** with the ID and address passed up in the event.

When PAP sends the open connection reply packet to the remote endpoint requesting the open, a status string is included. This status string is the one passed in the **ATPapAccept**. The status is taken as an arbitrary string of bytes, so if the printer expects a P string (length-preceded string), the client of this function must pass in a P string.

If you wanted to refuse the connection, you would call **ATPapReject**.

ATPapClose

Destroys the file descriptor and frees all memory, closing any sessions open on this file descriptor

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapClose (
   int   fd);
```

Parameters

fd

(IN) Specifies the file descriptor to close.

Return Values

0	Success
STREAMS errors	EBADF (invalid file descriptor) or a similar type of message

Remarks

All functions blocking on this file descriptor are freed, with an error to the effect (EBADF or ECLOSED or a similar error).

IMPORTANT: If the file descriptor is an SLS (session listening socket), all SSSs (server session sockets) spawned from this SLS are invalidated.

ATPapConnect

A workstation-side (NLM requesting connection to a server) function that changes the specified new file descriptor into a WSS (workstation session socket)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapConnect (
   int         fd,
   ATInet_t   *addr,
   u_short     waittime_waited,
   int         retry_time);
```

Parameters

fd

(IN) Specifies a file descriptor.

addr

(IN) Points to the AppleTalk address to attach to.

waittime_waited

(IN) Indicates the *waittime*, in seconds (u_short).

retry_time

(IN) Specifies the time to attempt connection, in seconds.

Return Values

0	Success
STREAMS errors	In a bad state for a connect, for example, the socket is already an SLS

Remarks

A workstation-side (NLM requesting connection to a server) function that

changes the specified new file descriptor into a WSS (workstation session socket). If the connection request succeeds, notification occurs through an **ATPapGet**.

The *waittime* is the period of time that your request has been waiting for a connection. Typically, you set this parameter to zero on the first try and add the number of seconds you have waited so far, that is, the value of *retry_time*, on each successive try.

The *waittime* value is passed to the server so that it can enforce fairness among those trying to connect; it allocates connections first to those who have been waiting the longest. Although the *Inside AppleTalk* PAP interface automatically sets a *waittime* value, this PAP interface gives that power to you, trusting that you keep the needs of other users in mind so that the delicate balance of resources on the network is preserved.

NOTE: **ATPapConnect** currently sends out connection requests every two seconds, as specified in the PAP specification. However, it updates the waittime every other request.

ATPapDisconnect

When you are in the data transport phase, notifies PAP that you want to close the connection but want all data already sent with **ATPapWrite** to be delivered.

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapDisconnect (
   int   fd);
```

Parameters

fd

(IN) Specifies the file descriptor.

Return Values

0	Success
STREAMS errors	

Remarks

When you are in the data transport phase, **ATPapDisconnect** notifies PAP that you want to close the connection but want all data already sent with **ATPapWrite** to be delivered.

In previous interfaces, the only way to close a PAP connection was to call **ATPapClose**, which would discard any pending data. **ATPapDisconnect** allows orderly close by waiting for the next PAP_EVENT_DATA transaction to arrive after all data has been delivered.

ATPapDisconnect begins to close the session, but waits until all data currently being transferred is sent before reporting that it is safe to close. After calling this function, no data is allowed to be written.

IMPORTANT: Note that calling **ATPapDisconnect** allows closing of the connection while still answering all "get status" functions that might be outstanding.

ATPapGet

Retrieves a message from the PAP protocol client
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapGet (
   int            fd,
   ATPapPass_t    *pass,
   char           *buf,
   int            *len);
```

Parameters

fd

(IN) Specifies a file descriptor, describing connection.

pass

(OUT) Points to a ATPapPass_t structure, which contains various non-data pieces of information about the event. See ATPapPass_t.

buf

(OUT) Points to a buffer. Data is read into this buffer on return.

len

(IN/OUT) Points to an integer, which you must set to the allocated size of the buffer *buf*. It is filled with the size of the data on return.

Return Values

0	Success
EMSGSIZE	There is data left to be read the second time
STREAMS errors	Same as STREAMS's GetMsg, at the moment

Remarks

Retrieves a message from the PAP protocol client. Common messages are PAP_EVENT_DATA, which describes that data has arrived on this connection, and PAP_EVENT_GETSTAT_REPLY, which lets you know that an SLS (server listening socket) has been initialized but is not ready to accept data.

The following tables explain PAP state transition and how to work with the ATPapPass_t structure: States and Corresponding PAP Events and PAP State Transition Table, and the table listing PAP events and their corresponding ATPapPass_t parameters in the description of **ATPapGet**.

ATPapGet waits until there is an event available. You can avoid a long wait by using **poll**.

If the buffer passed in is smaller than the amount of data that needs to be read, the next time **ATPapGet** is called the caller receives the next piece of the data but with the same ATPapPass_t structure and you get an ERANGE error code, indicating that there is data left to be read.

IMPORTANT: Be sure to check the *ATPapPass_t.ret* for errors that occur in the protocol.

If no event is ready, **ATPapGet** blocks. **ATPapGet** can be made nonblocking by using the **poll** function on the file descriptor, and only actually reading when data is known to be there.

The maximum data size that is ever be sent up is 4096 bytes.

The ERANGE error signals that data remains to be read. Call **ATPapGet** again.

The ATPapPass_t structure (listed below) contains a command field, which contains the event type, an error field, which describes errors (0 if none), then a large union. The values of the union which are valid are derived from the event type. In most cases they should be self evident.

After the structure, the currently defined event types are listed inthe following table. This table lists the name of the event, how it should be interpreted, and the parameters that are filled in with information about the event in the ATPapPass_t structure.

IMPORTANT: Do not rely on a number associated with an event; this number can change. Rather, select your functions based on the "#defines" themselves.

```
typedef struct ATPapPass_s {
    u_short   cmd;
    long      ret;
    union {
      struct ConnReq_s {
         ATInet_t   addr;
         void       *id;
      } ConnReq;
      struct Data_s {
```

```
            u_char     eof;
         } Data;
         struct ConnReply_s {
            u_short    result;
         } ConnReply;
         struct GetStat_s {
            ATInet_t   addr;
            ATRetry_t  retry;
         void       *id;
         } GetStat;
      } U;
} ATPapPass_t ;
```

Event	Interpretation	Parameter
PAP_EVENT_CLOSECONN_REQ	Received request to close connection, sent response, nothing to do but close.	No data comes up.
PAP_EVENT_GETSTAT_REPLY	Received when the response to a previous **ATPapGetStatus** request comes in.	If *pass->ret*, an error occurred. Don't assume that the request completed. *U.GetStat.addr* ---Address that the request was made of. *U.GetStat.id*---ID the requestor sent down.
PAP_EVENT_SLS_INIT	SLS has been initiated and is now ready to accept everything.	Could return allocation errors and no socket errors in *pass->ret*. No data.
PAP_EVENT_OPENSESS_REPLY	Received as a companion to **ATPapConnect**. Says that the WSS connection has been established or failed (memory allocation failure, socket allocation failure).	*pass->ret* returns errors that occurred locally. Data: *U.ConnReply.result* is the result code from the remote PAP server; 0 is success. **ATPapGet** data is filled with the status message from the remote end---255 bytes maximum.
PAP_EVENT_DATA	Received data from the remote endpoint	*pass->ret* returns local errors. *U.Data.eof* returns if an EOF was sent with this

		transaction. **ATPapGet** data is filled with data sent. Max PAP_DATA_SIZE*8. Data could be of length 0, which would indicate a 0 length transaction. Transactions are never coalesced.
PAP_EVENT_CONN_DIED	Received when an SSS or WSS dies. Could be for a number of reasons. Examples: tickle timeout; SLS is closed, closing all SSSs. Protocol stack becomes unbound or reconfigured from the network.	The error fields are 0, no data.
PAP_EVENT_OPENSESS_REQ	Received on an SLS when OpenSess comes in. Indicates that a request for Connection made it through the fairness arbitration and get-next-job screening. Accept or reject it.	Data--- *U.ConnReq.addr* is the address that the request came in on. use it for deciding to reject or accept the request. *U.ConnReq.id* is the ID of the request. Hand it back when rejecting or accepting the request.
PAP_EVENT_DISCONN	A PAP disconnect was completed. **ATPapDisconnect** was called, and now all data has been delivered and the close transaction has been completed, all gracefully. Please close the file descriptor.	No data.

See Also

ATAspGet, **ATAtpGet**

ATPapGetNextJob

Sets the number of jobs the SLS (Server Listening Socket) is willing to accept
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapGetNextJob (
   int   fd,
   int   num_jobs);
```

Parameters

fd

(IN) Specifies the file descriptor.

num_jobs

(IN) Specifies the number of jobs the server side is ready to accept.

Return Values

0	Success
EBADSTATE	**ATPapGetNextJob** must be on an SLS
STREAMS errors	

Remarks

After initializing the server side, **ATPapGetNextJob** is called when the server side is ready to accept a job, to set the number of jobs the SLS (Server Listening Socket) is willing to accept.

ATPapGetNextJob works with the internal counter that PAP uses to track the number of connections (jobs) an SLS is ready to accept at any given time. You can set this parameter to any number. The counter is incremented by this function by the value of *num_jobs* and decremented whenever PAP sends out a PAP_EVENT_OPENSESS_REQ event.

After you have received *num_jobs* number of

PAP_EVENT_OPENSESS_REQ events, the "get-next-job credit" on the SLS expires and must be renewed, if desired. When the counter is at zero, all client open-connection requests are rejected; therefore, to allow your server to receive jobs, call **ATPapGetNextJob**.

If you want to revoke granted get-next-job credit, call **ATPapGetNextJob** with a negative *num_jobs* parameter.

Note that **ATPapGetNextJob** is additive, so that calling **ATPapGetNextJob** twice with a parameter of 1 is equivalent to calling it once with a parameter of 2.

NOTE: To obtain a performance gain, you can avoid the 2-second arbitration period by keeping the counter filled (1 or more). In this way, when the open connection packet is received from the workstation, the PAP_EVENT_OPENSESS_REQ event is passed up to the application immediately. Since a print job can last as little as one second, this represents a significant gain.

ATPapGetStatus

Finds out the status of any given remote PAP server endpoint (workstation function)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapGetStatus (
   int          fd,
   ATInet_t    *addr,
   ATRetry_t   *retry,
   void        *id);
```

Parameters

fd

(IN) Points to file descriptor.

addr

(IN) Points to an AppleTalk address.

retry

(IN) Points to ATRetry_t structure.

id

(IN) Points to get Status ID.

Return Values

0	Success
EBADF	Invalid file descriptor

Remarks

A workstation function that finds out the status of any given remote PAP server endpoint. You can call **ATPapGetStatus** on any PAP file descriptor at any time and it can be made on a blocked file descriptor

without blocking.

Notification occurs as an **ATPapGet** on this file descriptor. When the **ATAspGet** completes, the ID and address are passed back so the upward client can de-multiplex.

The meaning in the status string is arbitrary. Usually it is a human-readable P string (Pascal-format: length-preceded rather than NULL-terminated) that indicates the status of the printer or spooler.

ATPapOpen

Opens a PAP file descriptor
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapOpen (
   int   fd);
```

Parameters

fd

(OUT) Points to the file descriptor to fill in.

Return Values

0	Success
STREAMS errors only	ENOBUFS or other resource limitation problems

Remarks

ATPapOpen opens a PAP file descriptor and, hence, a Stream, but it does not go any further than that. It does not make possible any actual network activity; other operations must be performed on the file descriptor before it can be used. Once open, it is possible to send GetStatus requests and create an SLS (session listening socket), WSS (workstation session socket), or SSS (server session socket).

ATPapReject

Denies a connection request on the SLS (session listening socket)
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapReject (
   int         fd,
   void       *connect_id,
   ATInet_t   *addr,
   char       *status,
   u_short     status_len);
```

Parameters

fd

(IN) Specifies the file descriptor.

connect_id

(IN) Points to the ID of connection passed up in OPENCONN_REQ event.

addr

(IN) Points to the address of the connection, passed up in OPENCONN_REQ event.

status

(IN) Points to the status buffer.

status_len

(IN) Specifies the length of the status buffer.

Return Values

0	Success
EINVAL	Invalid file descriptor
STREAMS errors	

Remarks

NOTE: Because of the get-next-job mechanism of indicating readiness, the need to call **ATPapReject** is rare.

When PAP sends the reply packet to the remote endpoint requesting an open, a status string is included. This status string is the one pointed to by the *status* parameter. The status is taken as an arbitrary string of bytes, so if the printer expects a P string (length-preceded string), the client of this API must pass in a P string.

ATPapSetStatus

Sets the current status of the SLS
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapSetStatus (
   int        fd,
   u_short    flag,
   u_short    status_len,
   char      *status);
```

Parameters

fd

(IN) Specifies a file descriptor, which determines connection.

flag

(IN) Specifies the flag to indicate which statuses to set. Must be nonzero.

status_len

(IN) Specifies the length of the buffer.

status

(IN) Points to the buffer with the status.

Return Values

0	Success
EINVAL	Buffer too long, too short, or similar problem
STREAMS errors	PAP stream not an SLS

Remarks

After initialization, **ATPapSetStatus** must be called to set the current

status of the SLS.

ATPapSetStatus is effective on an SLS (Session Listening Socket) file descriptor only, so you need to have an SLS before calling it.

The status is defined by an upper layer. It is often a human-readable P string (Pascal-format: length-preceded instead of NULL-terminated). At one point in *Inside AppleTalk*, the status string is mandated as a P string, but Novell allows the flexibility of other interpretations. (See **ATPapAccept** for a discussion of statuses in general.)

There are two statuses. They can be set independently. The first is the status returned if someone sends a "GetStatus" message. The second is the status returned when someone sends an open connection packet but the open-connection fails due to no outstanding get-next-job credits (blocked state) or the connection losing the arbitration cycle. In order to select which of these two statuses you want to set, AND together the appropriate flags (defined in the pap.h file) and set the *flag* parameter.

WARNING: In order to set the status string to the NULL, you could send a NULL pointer, or the length of zero, in the function. However, this is not recommended. There should always be a status because a reply is sent but if it has no bytes in the data, it could have disastrous results because some applications assume there is some data in the status string.

Regarding internationalization, the string that the client currently passes to PAP is exactly what is sent over the network. It is the client's problem to manage internationalization of strings.

ATPapSLInit

Makes the specified file descriptor into a server listening socket (SLS)
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapSLInit (
   int   fd);
```

Parameters

fd

(IN) Specifies the file descriptor to set up as SLS.

Return Values

0	Success
STREAMS errors	PAP stream not in the correct state to become an SLS

Remarks

IMPORTANT: When creating an SLS and WSS, PAP clients are not allowed to choose a specific socket to open on.

ATPapSLInit fails if the file descriptor is already open as a listener, as a client, or as a server. Should be a file descriptor that, up to this time, has not been used.

Wait for a PAP_EVENT_SLS_INIT event to come up in **ATPapGet** on the SLS saying that all resources---for example, sockets---have been allocated. Check the *pass->ret* field for errors. After receiving this event, you must call **ATPapSetStatus** to set the current status of this connection. After that, the **ATPapGetNextJob**, **ATPapGetStatus**, **ATPapAccept**, and **ATPapReject** functions are valid.

ATPapWrite

Sends data over an established connection
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <pap.h>

int ATPapWrite (
   int      fd,
   char    *data,
   int     *data_len,
   u_char   flush,
   u_char   eof,
   u_char   eot);
```

Parameters

fd

(IN) Specifies the file descriptor, describing connection.

data

(IN) Points to data to be written. This should be 0 if no data is to be sent.

data_len

(IN/OUT) Points to size of data. Actual amount written is returned.

flush

(IN) Indicates the flush indicator. Set to TRUE to allow this data to be sent before a complete 4096 bytes of data can be collected, when you have send credit.

eof

(IN) Indicates the end of file indicator---set to TRUE if you want this data to be followed by an end-of-file indicator.

eot

(IN) Indicates the end-of-transaction indicator---set when you want this data to end a transaction.

Return Values

0	Success
ENOTCONN	No connection open on this file descriptor
EBADSTATE	For example, a write on an SLS
STREAMS errors	For example, the connection closed unexpectedly

Remarks

IMPORTANT: Notice that the current session cannot have current **send credit**. Thus, the data cannot be sent out immediately. **ATPapWrite** takes time when too much data has been written without being sent. To avoid this wait, poll for writeability with the **poll** function.

The length of data is not particularly limited. **ATPapWrite** buffers the data to be delivered until a full buffer is reached. This buffer size is dynamic, depending on the flow quantum of the connection, but is generally either about 2048 or 4096 bytes. To allow the connection to send the data when send credit is available even though the buffer might not be full, set the flush indicator parameter to TRUE.

The performance penalty for setting the *flush* flag is low, but the penalty for setting the *eof* bit can be high. Setting the *flush* bit doesn't stop the coalescing of data into larger transactions, but setting the *eof* and *eot* bits does. Therefore, use them only when you need to mark special boundaries. The current Macintosh printing applications should not pose a problem of this nature since they are defined in an efficient way.)

ATZipGetLocalZones

Obtains a complete list of all the zone names defined on the local network (the network the calling node is bound to)

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <zip.h>

int ATZipGetLocalZones (
   int          *start,
   ATNveStr_t   *zones,
   int          *z_size,
   int          *last,
   ATInet_t     *router);
```

Parameters

start

(IN/OUT) Indicates the index of current zone requested and last zone returned.

zones

(IN) Points to the buffer to be filled with zone names on return. The zones are stored as a list (not an array) of zone names, each of which is contained in an ATNveStr_t structure; zones points to the first ATNveStr_t structure. See ATNveStr_t.

If there is no router on the local network, *zones* returns a star (*). A router can return one zone with length 0 if it is in the initialization process and is not yet fully configured.

z_size

(IN/OUT) Points to the size of the zones buffer, **in bytes**. This buffer should be at least the size of the maximum number of responses, which is ATP_DATA_SIZE (defined in atp.h). If this buffer isn't at least that size, you'll get back the STREAMS error ERANGE. Call the function again with a larger buffer.

On return, *z_size* is the size of the buffer that has been filled in.

last

(OUT) Returns 0 if all the zones on the list have not yet been returned, returns 1 when the last zone on the list has been returned.

router

(IN/OUT) Points to the DDP address of the router the inquiry is sent to. This can either be a default router chosen by AppleTalk or a particular router that you trust.

To send the inquiry to a default router, on the first call, pass in 0,0. *router* returns the actual address of the router that received the request.

The string containing the zone name is NOT NULL-terminated.

In multi-call inquiries, you must make sure that each call goes to the same router. To do so, on each call after the first one, set*router* to the address that was returned on the first call (instead of 0,0).

Return Values

0	Success
ENOBUFS	Insufficient buffer space is available
ETIMEDOUT	No response
EBADSIZE	ATNveStr_t buffer too small for incoming data, try again
ERANGE	Zones buffer too small; try again with larger buffer

Remarks

When you call **ATZipGetLocalZones**, ZIP sends a **ATZipGetLocalZones** packet to a router on the local network. The router replies with list of all the zone names on the local network in one or more packets.

If the router happens to be in the local node, no packets are sent out over the network.

The complete list of zones might not fit into your buffer, so you might need to call this function more than once to get a complete list. With each call, you get more zones from the list. The *start* parameter is used to keep track of the number of zones returned. When you have gotten all the zones on the list, the function sets the *last* flag to 1; otherwise, the function sets the *last* flag to zero (0), indicating that there are more zones to be returned. As long as the *last* flag is 0, continue to re-call the function, keeping track of zones with the *start* parameter.

For example, on the first call, set *start* to 1. If *start* returns 3 and the last flag is 0, on the second call add 1 plus 3 to set *start* at 4. If *start* returns 7 and the last flag is still 0, on the third call, add 4+7 to set *start* at 11. If *start* returns 15 and the *last* flag is now set to 1, you are done.

See Also

ATZipGetZoneList

Example

ATZipGetLocalZones

This example and the one for **ATZipGetZoneList** are the same.

```
#include <zip.h>
#include <nbp.h>
/*
    The following table is for uppercasing MacASCII characters. Instead
    toupper(), use this character table to uppercase the index into an
    unsigned chars!! (This table can also be found in Inside AppleTalk
,
    Appendix D.)
*/
u_char norm[] = {
    0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x
    0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x
    0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2a,0x2b,0x2c,0x
    0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x3b,0x3c,0x
    0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x
    0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x5b,0x5c,0x
    0x60,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x
    0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x7b,0x7c,0x
    0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0xcb,0x89,0x80,0xcc,0x81,0x
    0x90,0x91,0x92,0x93,0x94,0x95,0x84,0x97,0x98,0x99,0x85,0xcd,0x9c,0x
    0xa0,0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xab,0xac,0x
    0xb0,0xb1,0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xbb,0xbc,0x
    0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xcb,0xcc,0x
    0xd0,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xdb,0xdc,0x
    0xe0,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xeb,0xec,0x
    0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0x
};
main()
/*
    LocateServer() finds the file server.
    This function asks the user for the zone in which the server is
    registered, validates that zone, and tries to locate the server
    in that zone.
*/
int LocateServer()
{
   int start,oldstart;/* These two variables are the starting index and
      the oldstart index. These are used for multiple
      responses in the ATZipGetZoneList call.
      */
   int tmp=0; /* This is the temp count for how many zones were returne
```

```
      ATZipGetZoneList.
      */
    char zones[ATP_DATA_SIZE]; /* List of zone names will be placed her
      NOTE: THIS IS NOT AN ARRAY!!!!
      */
    int z_size;               /* This is the size of the zones in the b
    int last; /* This will be set to zero when all the zones have been
    ATInet_t router; /* This is the AppleTalk internet address of the r
    int err;
    char ZoneToFind[33];      /* The legal limit for a zone is 32 chara
      so I will add one byte on the end to allow
      for the '\0'
      */
    int FoundZone =0;    /*  Variable to let me know if I ever found th
    char *zone_addr;     /*  Local buffer to copy the zone name(s) retu
      ATZipGetZoneList zones parameter (described above).
      I will use this buffer to compare the user-
      specified zone against the zone(s) returned in
      the ATZipGetZoneList call.
      */
    int i;                 /* Your friendly counter. */
    ATEntity_t LookUpEntity; /* Once the zone is found, I will need to
      an NBP entity. This will hold that entity.
      */
    ATRetry_t retry;         /* The retry structure for NBP. */
    int nbp_tuples_found=0;  /* Tuples returned in ATNBPLookUp(). */
    int more;
    int max_tuples;
    /*
      This will be the entire lookup string. For example,
      in this case it will be
      =:ATP_SERVER.NATIVE@<zone_name>
      zone_name is still unknown. Make sure to save enough
      room for the zone name, which is a max of 32 characters.
      In this case, I am
      using 20 bytes for the known string, 32 for the possible
      zone name, and 1 just in case.
    */
   char LookUp_string[53]="=:ATP_SERVER.NATIVE@";
/*
    Find out the zone in which the user wants me to find the server.
*/
    printf("Please enter what zone the server has registered. >> ");
    if( gets(ZoneToFind) == NULL)
    {
      perror("gets failed");
      return(STREAM_ERROR);
    }
/*
    Initialize the router to 0.0, allowing the request to go to any rou
*/
    router.net =0;
```

```
    router.node = 0;
/*
    I will stay in this loop as long as there are zones to be found or
    if I find the zone that the user specified. Once I find the zone th
    specified then I will do a lookup for the server in that zone.
*/
    start = oldstart = 1;
    last = 1;
    printf("Checking for the zone %s, Hang On....\n",ZoneToFind);
    do
    {
      z_size=ATP_DATA_SIZE;
      if( err = ATZipGetZoneList(&start,(ATNvestr_t *)zones,
      &z_size,&last,&router) )
      {
         printf("ATZipGetLocalZoneList failed: err = %x\n",err);
         return(err);
      }
      zone_addr = &zones;
      for(i=0;i<start;i++)
      {
      /*
         Now, check the zone against the zone the user typed in.
      */
         if( String_Match(ZoneToFind,zone_addr) )
         {
            FoundZone = 1;
            strncat(LookUp_string,zone_addr+1,*zone_addr);
            break;
         }
         /*
            Increment the index for the next zone name. I need to incre
            1 (for the length) + zone name length.
         */
         zone_addr += 1 +  *zone_addr;
      }
      if(FoundZone)
      break;
/*
    Set start equal to the previous start plus the zones that were just
    returned.
*/
      tmp = start;
      start+=oldstart;
      oldstart+=tmp;
    }while ( !last );
/*
    A router may not have been found.  If there is no router on the net
    ATZipGetZoneList will return an "*" as the zone name and set the
    last flag, causing me to drop out of the loop.  I need to check the
    zones field to see if the length is 1 and the next byte is an "*".
*/
```

```
    if( zones[0] == 1 && zones[1] == '*' )
    {
      /* Let the console know there is no router. */
      printf("No router found, using an '*' for the zone.\n");

      /* Append the "*" to the end of my lookup string.  */
      strncat(LookUp_string,"*",1);
    }
    else
      if( !FoundZone )
         return(NO_ZONE_FOUND);
    if(err = ATNbpParseEntity(&LookUpEntity,LookUp_string) )
    {
      printf("ATNbpParseEntity failed: err= %x\n",err);
      return(STREAM_ERROR);
    }
    retry.interval = 1000;
    retry.retries = 2;
    retry.backoff = 2;
    max_tuples = MAX;
    more = NULL;
    /* Note: buf_p is a global. */
    nbp_tuples_found = ATNbpLookup(&LookUpEntity,buf_p,&max_tuples,
      &retry,&more);
    switch(nbp_tuples_found)
    {
      case ETIMEDOUT:  return(NO_SERVER_FOUND);
      case ENOBUFS:    return(STREAM_ERROR); /* This sample code should
      get this error.*/
      case EINVAL:     printf("Bad WildCard specified\n");
      return(STREAM_ERROR);
      default: return(nbp_tuples_found);
    }
}/* End of LocateServer. */

/*
    String_Match checks str1 (C string) and str2 (P string) against the
    MacASCII table defined as a global. All NBP and ZIP name searches
    should be done case-insensitive (Inside AppleTalk p.8-4).  This cal
    will return TRUE (1) if this is a match or FALSE(0) if there is not
    match.
*/
int String_Match(char *str1,char *str2)
{
    int i;
/*
    Compare the two strings.  Remember str1 is a C string and str2 is a
    string.
*/
    if( strlen(str1) != str2[0] )
      return ( FALSE );
    for ( i=0 ; i < strlen(str1) ; i++ )
```

```
        {
          if (norm[ (u_char) str1[i] ] != norm[ (u_char) str2[i+1] ])
          return ( FALSE );
        }
        return ( TRUE );
    }
```

ATZipGetMyZone

Finds the name of the zone a node is in
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <zip.h>

int ATZipGetMyZone (
   ATNveStr_t   *zone);
```

Parameters

zone

(OUT) Points to the ATNveStr_t structure filled with the zone name upon return. See ATNveStr_t.

> **NOTE:** The string containing the zone name is **not** NULL-terminated.

Return Values

0	Success
ENETDOWN	Stack is loaded but not bound to any network
EINVAL	Invalid parameter
ENOBUFS	Insufficient buffer space
ETIMEDOUT	No router or router busy
STREAMS error	Bad file descriptor

Remarks

The main use of **ATZipGetMyZone** is to show the user what the current zone is. This zone is known as "my zone" and all NBP registrations occur here.

Quite often **ATZipGetMyZone** does not send a request out over the

network. Sometimes it does.

ATZipGetMyZone is a blocking function.

The buffer should be a full ATNveStr_t.

If AFP is loaded but the AppleTalk stack is not bound to a network, zone returns a star (*) and the function returns the error ENETDOWN. You might want to use ENETDOWN to display a warning, such as "Caution: AFP is loaded but the AppleTalk protocol stack is unbound."

See Also

ATZipGetLocalZones, **ATZipGetZoneList**

Example

ATZipGetMyZone

```
#include <zip.h>

main()
{
ATNvestr_t myZone;
int err;

/*
    Let the console know what zone the server is registered in.
*/
   if(err = ATZipGetMyZone(&myZone) )
   {
      printf("ATZipGetMyZone failed: err = %x\n",err);
      ATDdpClose(my_fd);
      exit(__LINE__);
   }
   myZone.str[myZone.len] = 0; /* NULL terminate the string. */
   printf("The server is registered in the zone: %s\n",myZone.str);
}
```

ATZipGetNBPZones

Finds the valid zones for you to explicitly register an entity in
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <zip.h>

int ATZipGetNBPZones (
   ATNveStr_t  *zones,
   int         *z_size,
   int          more);
```

Parameters

zones

(OUT) Points to a buffer that holds the list of valid zones. The zone names are returned in packed format, that is, as a packed array of P strings.

The *zones* argument is a pointer to a list of valid explicit zone names. Each zone name is contained in an ATNveStr_t structure; *zones* is a pointer to the first ATNveStr_t structure holding a zone name. See ATNveStr_t.

z_size

(IN/OUT) Points to the size **in bytes** of the zones buffer. It returns the size of the buffer that was actually filled in, typically between 10 and 60 bytes.

Novell recommends that you set this buffer to ATP_DATA_SIZE, the maximum size of responses as defined in atp.h. As long as you set *z_size* to at least ATP_DATA_SIZE, your buffer rarely is too small; therefore, you won't have to call **ATZipGetNBPZones** more than once.

more

(OUT) Indicates if your buffer was too small to hold all the available NBP zones. As explained above (under the *z_size* parameter), this parameter is not likely to be set.

more returns the following two values:

0---All NBP-valid zones fit into buffer

1---Buffer too small; try again with larger buffer

NOTE: The string containing the zone name is not NULL-terminated.

Return Values

0	Success
ENOBUFS	Insufficient buffer space available
EINVAL	Bad buffer
ENETDOWN	The stack wasn't up, so no zone list could be found.
ETIMEDOUT	No response
EBADSIZE	ATNveStr_t buffer too small, try again with larger buffer
STREAMS error	Bad file descriptor

Remarks

ATZipGetNBPZones finds the zones that are valid for you to register explicitly an entity in. In explicit registration, the NBP client specifies exactly which zone to register in. The choice is from the zones on the AppleTalk internal network's list (not to be confused with the IPX internal network).

Contrary to the interface suggested in *Inside AppleTalk,* Novell now allows explicit zone registration where there is an AppleTalk internal network available. This function allows you to check if an explicit zone is valid in the current configuration. (For a more detailed description of explicit zone registration, see **ATNbpRegister**.)

ATZipGetNBPZones is a blocking, but very short, function. No network access is needed.

Example

ATZipGetNBPZones

```
#include    <zip.h>

main()
{
    char    *buf, *bp;
    int     err, num_zones=0, more, z_size = 512, lz = 512;
```

```
        do
        {
          buf = malloc(lz);
          if (!buf)
          return;
          z_size = lz;
          err = ATZipGetNBPZones((ATNveStr_t *) buf, &z_size, &more);
          if (err)
          return;
          if (more)
          lz *= 2;
        } while (more);
        bp = buf;
        while (bp < z_size + buf)
        {
          /* Display bp, a PString, so the user can choose from a list of z
          bp += *bp + 1;
          num_zones++;
        }
}
```

ATZipGetZoneList

Obtains all the zones on the internet
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 4.x
Platform: NLM
Service: AppleTalk

Syntax

```
#include <zip.h>

int ATZipGetZoneList (
   int          *start,
   ATNveStr_t   *zones,
   int          *z_size,
   int          *last,
   ATInet_t     *router);
```

Remarks

With **ATZipGetZoneList**, you get **all the zones on the internet**, not just those on the local network. The mechanics of **ATZipGetZoneList** are identical to those of **ATZipGetLocalZones**.

NOTE: A huge amount of network traffic would be generated by attempting to search all the zones of an internetwork at once. Therefore, the most common use of **ATZipGetZoneList** is for presenting a list of zones to users from which they can choose one to be searched. This is also true for **ATZipGetLocalZones**, although it generates less network traffic. Also note that when presenting a scrolling list, the initial selected zone should be set to the return of **ATGetMyZone**.

See Also

ATZipGetLocalZones, **ATZipGetMyZone**

Example

See the example for **ATZipGetLocalZones**.

ATZipZoneXlate

Translates, in place, a zone list from the MacASCII code page into the code page being used by the local server

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <zip.h>

void ATZipZoneXlate (
   ATNveStr_t  *zones,
   int          z_size);
```

Parameters

zones

(IN/OUT) Points to a list of valid NBP zone names. The zone names are returned in packed format, that is, as a packed array of P strings.

Each zone name is contained in an ATNveStr_t structure; zones points to the first ATNveStr_t structure holding a zone name. See ATNveStr_t .

z_size

(IN) Specifies the size of the zones buffer, **in bytes**.

NOTE: Note that the string containing the zone name is not NULL-terminated.

Return Values

ATZipZoneXlate always succeeds.

Remarks

ATZipZoneXlate is a NetWare-specific function that translates, in place, a zone list from the MacASCII code page into the code page being used by the local server. All the other functions return zones in the MacASCII code page; **ATZipZoneXlate** translates them into the code page being used by the local server.

ATZipZoneXlate is a nonblocking, immediate function.

ATZipZoneXlate is the only function that the client passes zones in to (instead of having a list of zones returned).

See Also

CstrIBMCPToMac, **CstrMacToIBMCP**, **PstrIBMCPToMac**, **PstrMacToIBMCP**

Example

ATZipZoneXlate

```
#include    <zip.h>

main()
{
    char   *buf, *bp;
    int    err, num_zones=0, more, z_size = 512, lz = 512;
    do
    {
      buf = malloc(lz);
      if (!buf)
      return;
      z_size = lz;
      err = ATZipGetNBPZones((ATNveStr_t *) buf, &z_size, &more);
      if (err)
      return;
      if (more)
      lz *= 2;
    } while (more);
    ATZipZoneXlate((ATNveStr_t *) buf, z_size);
    bp = buf;
    while (bp < z_size + buf)
    {
      /* Display bp, a PString, so the user can choose from a list of z
      bp += *bp + 1;
      num_zones++;
    }
}
```

CstrIBMCPToMac

Translates C-string formatted strings (NULL-terminated strings) from the code page in use on the local server to the MacASCII code page

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <appletlk.h>

char *CstrIBMCPToMac (
  char   *src,
  char   *dst);
```

Parameters

src

(IN) Points to the buffer (supplied by the caller) that holds the strings to be translated.

dst

(OUT) Points to the buffer (supplied by the caller) that holds the translated strings.

Return Values

If successful, **CstrIBMCPToMac** returns the *dst* pointer, and if not, it returns NULL (0).

Remarks

CstrIBMCPToMac is a non-blocking, immediate call.

See Also

CstrMacToIBMCP, **PstrIBMCPToMac**, **PstrMacToIBMCP**

CstrMacToIBMCP

Translates a C-string formatted file (NULL-terminated strings) from the MacASCII code page to the code page in use on the local server

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <appletlk.h>

char *CstrMacToIBMCP (
  char   *src,
  char   *dst);
```

Parameters

src

(IN) Points to the buffer (supplied by the caller) that holds the file to be translated.

dst

(OUT) Points to the buffer (supplied by the caller) that holds the translated file.

Return Values

If successful, **CstrMacToIBMCP** returns the *dst* pointer, and if not, it returns NULL (0).

Remarks

CstrMacToIBMCP is a non-blocking, immediate call.

See Also

CstrIBMCPToMac, **PstrIBMCPToMac**, **PstrMacToIBMCP**

PstrIBMCPToMac

Translates P-string formatted strings (length-preceded strings) from the code page in use on the local server to the MacASCII code page

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <appletlk.h>

char *PstrIBMCPToMac (
   char   *src,
   char   *dst);
```

Parameters

src

(IN) Points to the buffer (supplied by the caller) that holds the strings to be translated.

dst

(OUT) Points to the buffer (supplied by the caller) that holds the translated strings.

Return Values

If successful, **PstrIBMCPToMac** returns the *dst* pointer, and if not, it returns NULL (0).

Remarks

PstrIBMCPToMac is a nonblocking, immediate call.

See Also

CstrIBMCPToMac, **CstrMacToIBMCP**, **PstrMacToIBMCP**

PstrMacToIBMCP

Translates P-string formatted strings (length-preceded strings) from the MacASCII code page to the code page in use on the local server

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 4.x

Platform: NLM

Service: AppleTalk

Syntax

```
#include <appletlk.h>

char *PstrMacToIBMPC (
   char   *src,
   char   *dst);
```

Synopsis

src

(IN) Points to the buffer (supplied by the caller) that holds the strings to be translated.

dst

(OUT) Points to the buffer (supplied by the caller) that holds the translated strings.

Return Values

If successful, **PstrMacToIBMPC** returns the *dst* pointer, and if not, it returns NULL (0).

Remarks

PstrMacToIBMPC is a nonblocking, immediate call.

See Also

CstrIBMCPToMac, **PstrIBMCPToMac**, **CstrMacToIBMCP**

Communication Service Group

AppleTalk: Structures

ATAdspOpt_t

Contains option information for ADSP
Service: AppleTalk
Defined In: adsp.h

Structure

```
typedef struct ATAdspOpt {
   char             Major;
   char             Minor;
   char             Revision;
   unsigned char    reserved1;
   unsigned short   TransThresh;          /* Obsolete */
   unsigned         TransTimerIntrvl;     /* Obsolete */
   unsigned char    reserved2[118];
} ATAdspOpt_t;
```

Fields

Major

Contains the AppleTalk interface major version number.

Minor

Contains the AppleTalk interface minor version number.

Revision

Contains the AppleTalk interface revision version number.

The other fields are reserved or obsolete.

Remarks

The major, minor, and revision version numbers are defined as shown in the following figure.

Figure 8. Version Numbers

The major number of the ADSP interface is "revved" (increased in value) when a release contains one or more new functions, the minor number is revved when options have changed, and the revision number is revved when the size of an options buffer has changed.

IMPORTANT: You must retrieve the defaults before changing any values.

See AppleTalk TLI Structures for more information.

ATAspPass_t

Contains information about an ATP transaction
Service: AppleTalk
Defined In: asp.h

Structure

```
typedef struct ATAspPass {
   u_short   cmd;
   long      ret;
   union {
      struct Attn_s {                /* Used externally */
         short       attn_code;    /* 13 bytes */
         ATRetry_t   retry;
         u_long      id;
      } Attn;
      struct StatReq_s {             /* Used externally */
         ATInet_t    addr;
      } StatReq;
      struct OpenReq_s {
         ATInet_t   addr;
         u_long     id;
         u_short    fastwrite;
      } OpenReq;
   } U;
} ATAspPass_t;
```

Fields

cmd

ret

attn_code

retry

id (Attn_s)

addr (StatReq)

addr (OpenReq)

id (OpenReq)

fastwrite

Remarks

This structure is a message with information about a transaction that

This structure is a message with information about a transaction that came in after calling **ATAtpGet**. It contains a union with various substructures. Look in the appropriate substructure for the information about a certain transaction. For example, if an ASP_EVENT_SLS_STATUS_REQ comes in, look in the U.StatReq substructure. The substructures are named after the types of events to make it easy for you to find the right one.

ATAtpPass_t

Contains information about an ATP transaction
Service: AppleTalk
Defined In: asp.h

Structure

```
typedef struct ATAtpPass {
   u_short     event;
   int         ret;
   ATInet_t    at_addr;
   void        *TransID;
   u_char      xo;
   u_char      xo_relt;
   u_char      bitmap;
   u_char      packetize;
   ATRetry_t   retry;
   void        *data;
   u_short     data_len;
   u_long      userdata[8];
   u_short     packet_len[8];
} ATAtpPass_t;
```

Fields

event

ret

at_addr

Describes where a request or response came from or is going to.

TransID

Associates requests and responses. The same transaction ID that was passed in the **ATAtpGet** function that retrieved the request must be passed back in the response.

xo

Describes whether a request is XO (exactly once---the request is delivered only one time, thuse protecting against damage that could result from a duplicate transaction) or ALO (at least once---the request is repeated until a response is received by the requester or until a maximum retry count is reached, thus ensuring that the transaction is executed at least one time).

xo_relt

Describes the state of the *xo_relt* bits in a request. (This is the transaction release timer. For a description, see *Inside AppleTalk*.)

bitmap

The *bitmap* field is set to show the exact reply packets received.

packetize

Turns on or off the ability to specify the exact amount of ATP data per packet in an ATP response. In general, ATP's application client should not have to exert exact control over packets, such as manage how many packets and so forth. It should instead manage transactions. Nevertheless, some clients require more control, such as PAP, which requires that only 512 bytes of ATP data be sent in every reply. for these clients, the *packetize* field is included.

retry

See the description of the ATRetry_t structure.

data

The *data* buffer, supplied by the API client, is a concatenation of all data received.

data_len

Contains the length of *data*.

userdata

The *userdata* array is set to the userdata of each received response, or in the case of a request, *userdata* [0].

packet_len

The *packet_len* array is set to the number of bytes of ATP data received in each packet. (Usually this information is not needed.)

Remarks

Byte order in the *userdata* field can be confusing. Most protocols want to use this field on a byte-by-byte basis, but the field is defined as a long data type. This is in host order, so setting the *userdata* to the value 15 results in the 4 bytes "00 00 00 15" being sent over the network. To set each individual byte of this field, use the following C code:

```
userdata = hihibyte << 24 + hilobyte << 16 + lohibylte <<
   8 + lolobyte;
```

This results in the following byte ordering on the wire:

```
hihibyte hilobyte lohibyte lolobyte
```

For more information about byte ordering, see AppleTalk Byte Ordering.

ATDdp_t

Defines a DDP packet
Service: AppleTalk
Defined In: ddp.h

Structure

```
typedef struct {
   u_short   length_hi2:2,
             hopcount:4,
             unused:2,
             length_lo8:8;
   u_short   checksum;
   ATNet     dst_net;
   ATNet     src_net;
   ATNode    dst_node;
   ATNode    src_node;
   ATSocket  dst_socket;
   ATSocket  src_socket;
   u_char    type;
   char      data[DDP_DATA_SIZE];
} ATDdp_t;
```

Fields

length_hi2, hopcount, unused, length_lo8

checksum
Contains the packet checksum.

dst_net
Contains the destination network number.

src_net
Contains the source network number.

dst_node
Contains the destination node ID.

src_node
Contains the source node ID.

dst_socket
Contains the destination socket number.

src_socket
Contains the source socket number.

type

Contains the protocol type.

data

Contains the packet data.

Remarks

Use this structure to create a DDP packet for sending. ATDdp_t contains fields for the DDP packet extended header and data, and is defined in ddp.h. The ATDdp_t structure depends upon these type definitions, as defined in appletlk.h:

```
typedef unsigned short   at_net;
typedef unsigned char    at_node;
typedef unsigned char    at_socket;
```

By default, DDP_DATA_SIZE is 586, the maximum number of bytes allowed in the *data* field of a DDP packet.

If you want DDP to calculate a checksum, specify a nonzero value in the *checksum* field. If you do not want DDP to calculate a checksum, specify 0 for the *checksum* field.

When you send a DDP packet, DDP overwrites part of the source internet address you specify. It overwrites the network ID and the node ID with the correct network and node IDs.

Because of bit and byte-swapping problems, the DDP length can be difficult to retrieve. Therefore, Novell offers, in the ddp.h file, the DDP_LENGTH and SET_DDP_LENGTH macros. DDP_LENGTH takes one parameter, a pointer to an ATDdp_t structure. It returns the DDP length (the first value in the DDP packet, the number of bytes from the beginning of the DDP length field to the end of the packet). SET_DDP_LENGTH takes two parameters, a pointer to an ATDdp_t structure and the length that you want to set the DDP length field to. DDP_LENGTH (*ddp*) returns the value in the length field of a DDP packet that you read or write. The *ddp* argument is a pointer to an ATDdp_t structure.

ATDdpOpt_t

Contains DDP options
Service: AppleTalk
Defined In: ddp.h

Structure

```
typedef struct ATDdpOpt {
   u_short     checksum;       /* optional */
   u_char      ddp_type;       /* required */
   char        major;          /* a read-only option */
   char        minor;          /* a read-only option */
   char        revision;       /* a read-only option */
} ATDdpOpt_t;
```

Fields

checksum

Enables the sending client to indicate whether DDP should calculate a checksum for a packet it sends out. The DDP header contains a checksum field, in which the value of the checksum appears.

ddp_type

The DDP header contains a DDP type field, an unsigned byte field that identifies the protocol contained in the data section of the datagram.

major

Contains the AppleTalk interface major version number.

minor

Contains the AppleTalk interface minor version number.

revision

Contains the AppleTalk interface revision version number.

Remarks

You can set five fields in this structure, the DDP options checksum and data type and the AppleTalk version number.

The major, minor, and revision version numbers are defined as shown in the following figure.

The major number of the ADSP interface is "revved" (increased in value) when a release contains one or more new functions, the minor number is revved when options have changed, and the revision number is revved when the size of an options buffer has changed.

The t_unitdata and t_uderr structures have an *opt* field, which lets you access protocol options. *opt* points to a netbuf structure, which contains the **buf* field, which in turn points to the ATDdpOpt_t structure.

A nonzero value in the *checksum* field of the ATDdpOpt_t structure causes DDP to generate a checksum when a transport client sends a datagram. If the *checksum* field of a DDP datagram is nonzero when the datagram arrives at its destination, DDP verifies the checksum. If the checksum is valid, DDP passes the packet on to the next protocol layer. If the checksum is in error, DDP discards the packet.

The checksum option also enables a receiving client to verify whether the sender has used a checksum. Therefore, if a transport client receives data with **t_rcvudata**, and the *checksum* field of ATDdpOpt_t is nonzero, it is considered "polite" to send subsequent datagrams using a checksum.

If the *checksum* field of ATDdpOpt_t is 0, DDP does not calculate a checksum. The default value is 0.

The checksum option controls the use of checksums only on send, and does not automatically affect the use of checksums on other transport endpoints. The use of checksums is not universal, because it does add some overhead to the handling of a packet. However, we recommend the use of checksums in large internets where packets may become corrupted when crossing multiple repeaters or routers.

NOTE: Even if you do not specify the use of checksums in this way, they may be turned on for the AppleTalk stack as a whole. This is true for DDP and ADSP.

Apple Computer, Inc. has established the following universally known values for the *data_type* field of the DDP packet:

0	Invalid (do not use)
1	Routing Table Maintenance Protocol (RTMP) response or data

	packet
2	Name Binding Protocol (NBP) packet
3	AppleTalk Transaction Protocol (ATP) packet
4	AppleTalk Echo Protocol (AEP) packet
5	RTMP request packet
6	Zone Information Protocol (ZIP) packet
7	AppleTalk Data Stream Protocol (ADSP) packet

Values 8 through 255 are valid but not universally known. Apple Computer, Inc. has reserved values 1 through 15 for its own use.

When a transport client receives a datagram with **t_rcvudata**, the *ddp_type* field of ATDdpOpt_t is set to the DDP type field in the DDP header.

When a client sends a datagram with **t_sndudata**, the DDP type field of the DDP header is set to the value specified by the *ddp_type* field of ATDdpOpt_t.

The default value is 128.

ATEntity_t

Contains an NBP entity's human-readable name (object, type, and zone)
Service: AppleTalk
Defined In: nbp.h

Structure

```
typedef struct ATEntity {
   at_nvestr_t    object;
   at_nvestr_t    type;
   at_nvestr_t    zone;
} ATEntity_t;
```

Fields

object

Contains a zone-unique string that identifies the NBP entity, such as "Marketing Print Server".

type

Contains the type of object, such as "print server".

zone

Contains an AppleTalk zone, such as "Enet".

Remarks

This structure is made up of the three strings that constitute an NBP entity's human-readable name: object, type, and zone. Each name is a maximum of 32 characters long, and each string is an ATNveStr_t structure (P string, defined above).

You can include one wildcard per string (field). The choices are listed in the following table.

Table auto. Wildcard Options

Field	Wildcard	Matches
a. Not really a wildcard. The asterisk represents "this zone."		
object	Equal sign (=)	All possible characters
object	Approximately equal sign(Ý)	Zero or more characters (Phase II only)
type	Equal sign (=)	All possible characters
type	Approximately equal sign (Ý)	Zero or more characters (Phase II nodes aren't required to respond to

		this)
zone	Asterisk (*)[a]	Default value (local zone)

The equal sign (=) is defined in nbp.h as NBP_ORD_WILDCARD. The approximately equal sign (Ý) is defined in nbp.h as NBP_SPL_WILDCARD with a value of hexadecimal C5.

ATNbpLookup determines whether the entity name you specify is valid.

NOTE: As the table above indicates, you cannot specify a wildcard character to indicate all possible zones. Therefore, if you do not know the name of the zone in which the entity resides, you must use **ATZipGetLocalZones** or **ATZipGetZoneList** to obtain a list of zone names. **ATZipGetLocalZones** returns the names of all the zones on the local network. **ATZipGetZoneList** returns the names of all the zones on the internet. Once you have obtained the zone names, you can attempt a lookup in one or more possible zones.

For information on using **ATZipGetLocalZones** and **ATZipGetZoneList** with **ATNbpLookup**, see the sample programs that came with the SDK.

ATInet_t

Contains a three-part AppleTalk internet address (net, node, socket)
Service: AppleTalk
Defined In: appletalk.h

Structure

```
typedef u_short   ATNet;
typedef u_char    ATNode;
typedef u_char    ATSocket;
typedef struct ATInet {
   ATNet      net;
   ATNode     node;
   ATSocket   socket;
} ATInet_t;
```

Fields

net

Contains a network (cable) number

node

Contains a node number

socket

Contains a socket number

IMPORTANT: Remember that this structure is in Intel byte order.

ATNbptuple_t

Contains the human-readable name of an NBP entity and its AppleTalk address

Service: AppleTalk

Defined In: nbp.h

Structure

```
typedef struct ATNbptuple {
   ATInet_t   enu_addr;
   union {
      struct {
         u_char       enumerator;
         ATEntity_t   entity;
      } en_se;
      struct {
         u_char   enumerator;
         u_char   name[NBP_TUPLE_SIZE];
      } en_sn;
   } en_u;
} ATNbptuple_t;
```

Fields

enu_addr

Contains the Internet address of the specified entity; an ATInet_t structure

enumerator (in en_se)

Contains the enumerator

entity

Contains the name of the specified entity; an ATEntity_t structure

enumerator (in en_sn)

Contains the enumerator

name

Remarks

An NBP **name-address tuple** matches the human-readable name of an NBP entity with its complete AppleTalk address so that services can be found on the network. This structure also has an enumerator to help applications distinguish between the multiple names that can be associated with a particular socket.

The *enu_entity* field is actually a macro that has been defined to provide

easier access to the structure.

The **at_nbptuple_t** pointer (*) in functions is used as a normal array, not as the packed format that you find in the ZIP interface. In general, you would access a series of NBP entities as shown by the following pseudocode:

```
ATEntity_t *e;     /* Entities returned from lookup */
for (i = 0 ; i < num_entities ; i++)
{
   /* Do what you need to with the entity, e. */
   e++;
}
```

ATNveStr_t

Describes a service in human-readable form
Service: AppleTalk
Defined In: nbp.h

Structure

```
typedef struct ATNveStr_t {
   char    len;
   char    str [NBP_NVE_STR_SIZE];
} ATNveStr_t;
```

Fields

len

Contains the size in bytes of the NBP object, type, or zone's human-readable name.

str

Contains an NBP object, type, or zone's human-readable name.

Remarks

This structure is a 33-byte P string (length-preceded string) used to describe the human-readable form of a network service. (NVE stands for **network visible entity**.) The first byte of the string gives the actual length of the human-readable name, which is usually less than the allotted 32 bytes (NBP_NVE_STR_SIZE).

IMPORTANT: NBP entities are returned in array format (not in the "packed" format of zones). Accordingly, you would reference NBP entities as the following pseudocode shows:

```
ATEntity_t *e;    /* Entities returned from lookup */
for (i = 0 ; i <  num_entities ; i++)
{
   /* Do what you need to with the entity, e. */
   e++;
}
```

NOTE: On the other hand, zones are returned in the packed format described in Inside AppleTalk. **Do not try to reference this structure as an array (even though it would be syntactically valid).** The packing method has P strings in a list (not an array): length, length-bytes of zone data. Accordingly, you would reference zones as the following pseudocode shows:

```
ATNvestr_t *zones;         /* Zone buffer retrieved */
```

```
ATNvestr_t *zones;          /* Zone buffer retrieved */
ATNvestr_t *now_zone, *end_zone;
end_zone = ((char *)zones) + z_size;
now_zone = zones;
while (now_zone < end_zone)
{
  /* Do what you need to with now_zone. */
   now_zone = (char *)now_zone +
      now_zone->len + 1;
}
```

ATPapPass_t

Contains information about a PAP transaction
Service: AppleTalk
Defined In: pap.h

Structure

```
typedef struct ATPapPass_s {
   u_short   cmd;
   long      ret;
   union {
      struct ConnReq_s {
         ATInet_t    addr;
         void        *id;
      } ConnReq;
      struct Data_s {
         u_char       eof;
      } Data;
      struct ConnReply_s {
         u_short      result;
      } ConnReply;
      struct GetStat_s {
         ATInet_t     addr;
         ATRetry_t    retry;
         void         *id;
      } GetStat;
   } U;
} ATPapPass_t;
```

Fields

cmd

ret

addr (in ConnReq_s)

id (in ConnReq_s)

eof

result

addr (in GetStat)

retry

id (in GetStat)

Remarks

Remarks

This structure contains a union with various substructures. Look in the appropriate substructure for the information about a certain transaction. For example, if a connection reply comes in, look in theConnReply substructure. The substructures are named after types of events to make it easy to find the right one.

ATRetry_t

Defines the retry behavior of a function
Service: AppleTalk
Defined In: appletalk.h

Structure

```
typedef struct ATRetry {
   u_long  interval;
   short   retries;
   u_char  backoff;
} ATRetry_t;
```

Fields

interval

Contains the retry interval, in 1/1000s of a second

retries

Contains the maximum number of retries; -1 specifies an infinite number of retries (such as for a regularly scheduled event)

backoff

Indicates whether or not *interval* increases with successive retries. Must be between 0 and 4:

0 or 1 = no increase
4 = large increase

Remarks

This structure defines the retry behavior of a function. It sets the number of times the function retries, the interval between successive retries, whether the interval increases or not, and whether the increase is gradual (linear) or radical (exponential) in nature.

The NBP, ATP, and ZIP functions that use retries give you a choice between passing in NULL (0) to get the system default retry settings or passing in a pointer to an ATRetry_t structure you have set. Because the default settings are workable for most situations, Novell recommends that you use NULL in the early stages of development and later, when you are able to improve upon the defaults, setting an ATRetry_t structure.

IMPORTANT: Notice that interval is measured in **milliseconds**, not seconds, and is type u_long, not short.

Using the *backoff* field, you can opt to gradually increase the interval

between retries, thereby increasing the total time before the function times out. You have four backoff settings to choose from:

0 or 1---no increase in intervals

2---slight increase

3---moderate increase

4---large increase

You have to experiment to see which setting works best for your application. The exact interval in milliseconds that each of these settings produce is actually a system function, so when you select a *backoff* setting, you are making a value judgment about the general rate at which your retries should "back off," not an exact calculation of the amount of time each interval is.

Backoffs could be constant, linear, or exponential in type. **Constant backoff** is actually a contradictory phrase; because the interval remains constant, there is really no backoff involved. This is the type that a backoff of 0 or 1 produces. However, backoffs 2 through 4 might be linear or exponential, depending on how Novell has set them.

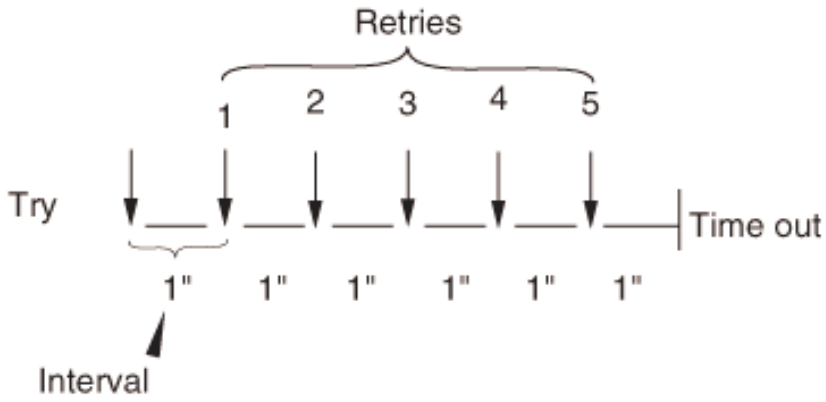
Constant Backoff

In a constant backoff, the interval remains the same between all retries. The following example depicts a constant backoff with the following settings.

interval: 1000 (1")

retries: 5

backoff: 1



The interval of one second (1") between retries remains constant,

resulting in an aggregate retry of 6 seconds (the last second occurs between the final retry and the point of timing out).

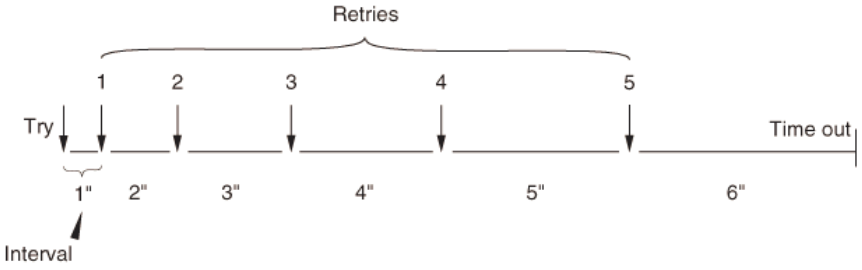
Linear Backoff

A linear backoff is one that increases the time between retries by the same amount each time. The following example shows a linear backoff where *backoff* 2 implies a system default of one second (1"):

interval: 1000

retries: 5

backoff: 2



The interval increases by one second after each retry, adding up to an aggregate retry of 21 seconds (1+2+3+4+5+6=21).

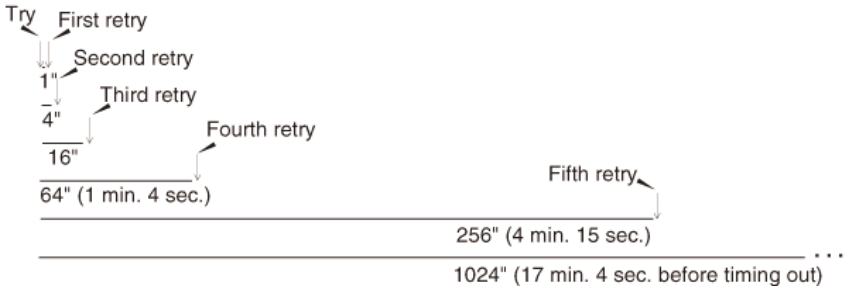
Exponential Backoff

An exponential backoff is one that multiplies the time between retries by a set factor, a system default. The following example shows an exponential backoff with the same settings but where *backoff* 2 implies a system default of four seconds (4"):

interval: 1000

retries: 5

backoff: 2



The page is not wide enough to show the true proportions of the last two exponential increases. In total, the intervals add up to 1365 seconds, (1+4+16+64+256+1024=1365 seconds), or an aggregate retry of 22 minutes and 45 seconds.

Asynchronous I/O

Asynchronous I/O: Guides

Asynchronous I/O: Task Guide

Asynchronous I/O: Concept Guide

Asynchronous I/O: Tasks

Acquiring an Asynchronous I/O Port

Call **AIOAcquirePort** to gain control of a port. You must specify the appropriate hardware type and board and port numbers in the function. The application then owns the port until it calls **AIOReleasePort** to give up control of it. During the time the application owns the port, no other application can affect the port in any way.

Asynchronous I/O uses a unique handle value, returned to the application by **AIOAcquirePort**, to identify the port and prevent unauthorized access to it. This handle value has no meaning to applications and should not be altered.

Call **AIOGetFirstPortInfo** and **AIOGetNextPortInfo** to scan the pool of ports known to Asynchronous I/O. Enough information about each port is returned to enable you to determine which ports are suitable for use.

Typical NetWare 3.1x applications call **AIOAcquirePort**. **AIOAcquirePortWithRTag** is provided only for backward compatibility with applications written without the NetWare API.

Setting Function Parameters for Return Values

For all functions, the return value is the completion status, which indicates the success or failure of a request. A return of 0 indicates the successful completion of a request. The return of a negative status (such as -1) indicates an error or failure.

When a function returns a value to an application, the function parameters include pointers that give the address of variables or structures. These pointers are used to return the resulting values.

In many cases, you can set these pointer parameters for return values to NULL (0), indicating that the value is not to be returned for this function. You can use this to request the return of only part of the information that a function is capable of returning.

When no information can be returned in a structure (that is, when a driver-specific structure is not implemented by a particular driver) the *returnLength* field in the structure is set to 0.

Asynchronous I/O: Concepts

Application-to-Asynchronous I/O Functions

Asynchronous I/O functions permit all necessary manipulations of an asynchronous port. These functions provide to applications the character-oriented, buffered I/O that Asynchronous I/O drivers offer. Application-to-Asynchronous I/O services include the following:

Read and Write Data Transfer Functions

Status Reporting Functions

Control Functions

NOTE: This release of Asynchronous I/O does not provide event detection or notification services to applications. Thus, successful use of asynchronous ports through Asynchronous I/O usually requires periodic polling of ports for the current status of external signals, as well as receive and transmit buffer statuses.

Asynchronous I/O Functions

Table auto. Asynchronous I/O Functions

Function	Purpose
AIOAcquirePort	Requests exclusive ownership of a port
AIOAcquirePortWithRTag	Requests exclusive ownership or a port
AIOConfigurePort	Sets parameters that affect the format of data transmission
AIOFlushBuffers	Discards data in the receive buffer or transmit buffer
AIOGetDriverList	Returns information about drivers registered with Asynchronous I/O
AIOGetExternalStatus	Returns information about the state of hardware signals from external equipment
AIOGetFirstPortInfo	Returns information about ports

	known to Asynchronous I/O without acquiring them
AIOGetNextPortInfo	Used after **AIOGetFirstPortInfo** to obtain information about the next port known to Asynchronous I/O
AIOGetPortCapability	Returns capability information for a previously acquired port
AIOGetPortConfiguration	Returns configuration information for an acquired port
AIOGetPortStatistics	Returns statistical information for a port
AIOGetPortStatus	Returns the complete state of the port
AIOGetReadBufferSize	Returns the total size of all buffers that a port uses to store received data
AIOGetWriteBufferSize	Returns the total size of all buffers that a port uses to store data to be transmitted
AIOReadData	Copies data from a port's receive buffer to the NLM buffer
AIOReadStatus	Returns status information about data reception
AIOReleasePort	Relinquishes ownership of a port
AIOSetControlData	Passes a request and data structure to an Asynchronous I/O driver
AIOSetExternalControl	Controls the state of external signals
AIOSetFlowControl	Changes flow control modes independently of other configuration parameters given for **AIOConfigurePort**
AIOSetFlowControlCharacters	Sets data values used with software flow control
AIOSetReadBufferSize	Sets the size of a port's receive buffers
AIOSetWriteBufferSize	Sets the size of a port's transmit buffers
AIOWriteData	Copies data from an NLM buffer to a transmit buffer
AIOWriteStatus	Returns information about data transmission

Buffer Size

Two Asynchronous I/O functions allow applications to set the size of receive and transmit buffers, as desired: **AIOSetReadBufferSize** and **AIOSetWriteBufferSize**. You could call these functions, for instance, to alter the size of a receive buffer depending on the baud rate used.

Asynchronous I/O sets the size of receive and transmit buffers to the default when a port is acquired. The default size for receive and transmit buffers varies among drivers. For example, the default size for both receive and transmit buffers for the AIOCOMX driver is 1,024 bytes.

> **NOTE:** Many asynchronous boards and associated drivers have no provision for dynamic buffer sizing. If a port does not support dynamic buffer sizing, the Asynchronous I/O buffer size functions return an AIO_FUNC_NOT_SUPPORTED error status. Asynchronous I/O applications should be prepared for buffer sizing not to be supported by some ports.

Control Functions

Control functions allow applications to configure hardware to send and receive data using various speeds, formatting, and flow control methods. (Flow control allows applications to turn source transmission off if the recipient's receive buffer fills.) The control functions are as follows:

AIOConfigurePort

AIOFlushBuffers

AIOSetControlData

AIOSetExternalControl

AIOSetFlowControl

AIOSetFlowControlCharacters

Deadman Timer

AIOSetExternalControl has a deadman timer feature that requests that the port's driver independently monitor the length of time between AIO_SET_DEADMAN_TIMER requests by the application. If the time interval exceeds the given length, the driver assumes the application or server has abended or otherwise failed and turns off the RS-232 signals DTR and RTS. Using the deadman timer, drivers can automatically detect failure of the port's user and signal the attached hardware that the connection with the application has failed.

The deadman timer is initially disabled when a port has just been acquired.

Applications can enable it by calling **AIOSetExternalControl**. The deadman timer is also disabled when the port is released.

Driver Information Functions

Some Asynchronous I/O functions allow applications to receive driver-specific values or data structures. Additionally, applications can generate driver-specific commands.

In many cases where Asynchronous I/O-defined data structures are passed, an application can also make calls to driver-specific data structures. This permits knowledgeable applications to manipulate drivers that employ information beyond that specified by the Asynchronous I/O definition.

The format of these structures is not defined beyond the required use of the *returnLength* field, which must be used as described earlier in this chapter.

> **NOTE:** This document does not include the information or descriptions required for an application to interact with a driver in a driver-specific manner. Similarly, the aio.h file does not include the definitions of constants and structures for driver-specific data.
>
> Rather, each Asynchronous I/O driver developer supplies, if applicable, an include file and associated documentation describing the use of any features unique to that particular driver.

The following functions allow driver-specific structures:

AIOGetFirstPortInfo

AIOGetNextPortInfo

AIOGetPortCapability

AIOGetPortConfiguration

AIOGetPortStatistics

AIOSetControlData

External Status Bit Mask Functions

Each function that returns the current external status values bit mask also returns an additional bit mask. The additional bit mask indicates those external status signals that have changed state since the last time a function returned external status values. This allows applications to better determine when they should act upon external status values.

For each external status signal, a bit is set to 1 when any transition occurs for that signal, whether the signal changes from off to on or on to off. A set bit

remains set even if the signal returns to its original value. The changed external status bit mask is not reset to 0 until after an application reads the bit mask (using either **AIOGetExternalStatus** or **AIOGetPortStatus**).

The following functions return external status bit masks:

AIOGetExternalStatus

AIOGetFirstPortInfo

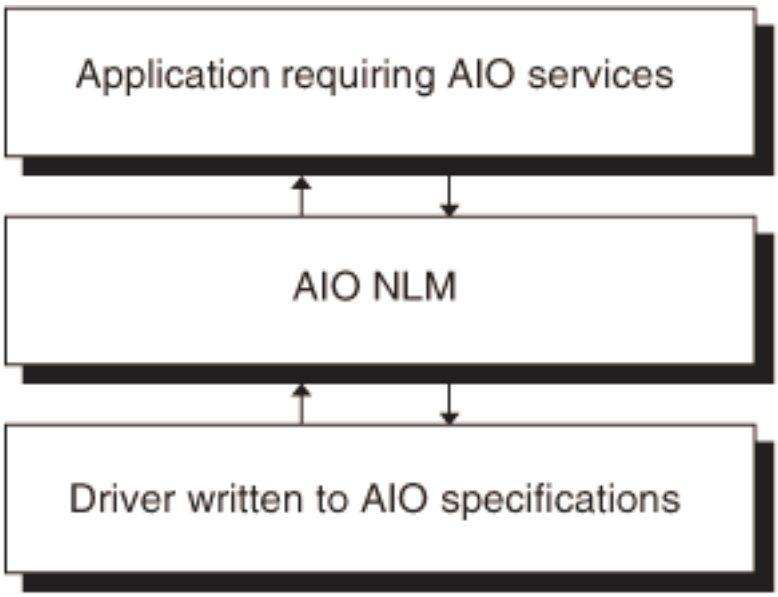
AIOGetNextPortInfo

AIOGetPortStatus

Introduction to Asynchronous I/O

Asynchronous I/O is a library of functions that run as an NLM™ application. The Asynchronous I/O NLM presents a single point of control through which applications can access all asynchronous communications services in the NetWare® 3.1 and above environment. The following figure illustrates this use of the Asynchronous I/O NLM by applications and drivers.

Figure 9. Asynchronous I/O Application Programming Interface



Applications can execute calls to functions in the Asynchronous I/O library to request services. Most functions identify a specific port to be serviced; the

requested services are then either performed immediately or rejected for later retry. Other functions return information without referring to specific ports; for example, a function might indicate whether any ports in the pool of ports known to Asynchronous I/O are available.

Individual Asynchronous I/O drivers register with Asynchronous I/O and identify the asynchronous ports provided by each driver. Asynchronous I/O adds these ports to the pool of ports that it provides. Applications can either request ports from the whole pool of ports or request a specific port, board, or driver hardware type.

Port Owner Name

The **AIOGetFirstPortInfo** and **AIOGetNextPortInfo** functions allow an application to request the NLM name of a port owner if the port has been acquired. This allows applications to display the name of the owning NLM when issuing messages about the failure to acquire a port.

Read and Write Data Transfer Functions

The basic read and write data transfer functions are as follows:

AIOAcquirePort

AIOAcquirePortWithRTag

AIOGetReadBufferSize

AIOGetWriteBufferSize

AIOReadData

AIOReleasePort

AIOSetReadBufferSize

AIOSetWriteBufferSize

AIOWriteData

Return Length Field

Asynchronous I/O uses a standardized header for passing data structures between Asynchronous I/O and the application. The header contains a *returnLength* field that allows an application or Asynchronous I/O to determine the exact size of the data structure being passed.

The *returnLength* field is needed for two reasons. First, it is required when

the data that can be passed within a structure varies in size; this applies to many Asynchronous I/O structures. Second, this field is needed because structures themselves may vary in size among versions of Asynchronous I/O. You can use an application with earlier versions of Asynchronous I/O, because Asynchronous I/O does not overflow the application structure or overwrite application data areas.

Applications must set the *returnLength* field before a call to an Asynchronous I/O function that returns an actual byte length; Asynchronous I/O does not use more than the specified length. Asynchronous I/O changes the *returnLength* field upon return from the function to reflect the byte length of the data returned in the structure. The actual length returned is never greater than the original length that the application set.

Before using the structure as an argument in a call to Asynchronous I/O, an application must initialize the *returnLength* field of the structure to the byte length of the structure, as follows:

```
AIOPORTCONFIG   portConfig;
portConfig.returnLength = sizeof(portConfig);
ccode = AIOGetPortConfiguration(porthandle,
      &portConfig, NULL);
```

This allows Asynchronous I/O to determine how much data can be returned in the application-provided structure. In addition, the application can determine how much information was returned.

The following structures contain a *returnLength* field:

AIOPORTCAPABILITIES

AIODVRCAPABILITIES

AIOPORTCONFIG

AIODVRCONFIG

AIOPORTSTATISTICS

AIODVRSTATISTICS

AIOPORTINFO

AIODRIVERLIST

Status Reporting Functions

Status reporting functions allow applications to track the progress of data transfers and changes in external conditions. The status reporting functions are as follows:

AIOGetDriverList

AIOGetExternalStatus

AIOGetFirstPortInfo

AIOGetNextPortInfo

AIOGetPortCapability

AIOGetPortConfiguration

AIOGetPortStatistics

AIOGetPortStatus

AIOReadStatus

AIOWriteStatus

Version Fields

Some of the Asynchronous I/O structures contain major and minor version fields permitting identification of structure formats. Applications can check the major and minor versions to determine whether structure contents are different from those expected, and whether the structure is so different that any use of its data would be unsafe.

The version fields are defined as follows:

The minor version number reflects minor changes made to structures. Minor changes might occur if structure fields are replaced or no longer supported, or if new fields are appended to the end of the structure. Even when the minor version is different from that expected, most references to structure fields remain valid.

The major version number reflects major reorganizations of a structure (especially if a preexisting field's offset is changed).

The Asynchronous I/O structures that include major and minor version numbers also contain a *notSupportedMask* field. The *notSupportedMask* field allows applications to determine whether individual fields are valid across minor version changes. This 4-byte bit mask uses one bit position per structure field to indicate which structure fields are not supported. When a previously existing field is superseded or no longer supported, the bit position in this mask is set to 1.

The aio.h file contains #define statements for each field in the structures. The naming convention takes the form:

```
AIO_<tableName>_NS_<fieldName>
```

For example, the *receiveBytes* field of the AIOPORTSTATISTICS structure has a bit mask named AIO_STATS_NS_RECEIVEBYTES. Refer to the aio.h file for specific bit mask names and values.

Applications can use the *notSupportedMask* field, in combination with the minor version, to determine when a field data value is valid. Bit positions for fields not yet defined are set to 1 to help maintain application compatibility with previous versions of Asynchronous I/O.

The structures that use minor and major version numbers are:

AIOPORTCAPABILITIES

AIOPORTCONFIG

AIOPORTINFO

AIOPORTSTATISTICS

Wildcard Functions

Some Asynchronous I/O functions accept independent wildcard values as parameters. This wildcard capability allows any mix of wildcard and nonwildcard values. For example, it is possible to specify the following:

```
AIOAcquirePort( -1, -1, 0);
```

The first -1 specifies "any driver"; the second -1 specifies "any board"; and the 0 specifies "first port." Thus, this function asks for the first port on any board of any driver.

These Asynchronous I/O functions accept wildcard values:

AIOAcquirePort

AIOGetFirstPortInfo

NOTE: The **AIOGetFirstPortInfo** and **AIOGetNextPortInfo** functions are used in a manner similar to **DOSFindFirstFile** and **DOSFindNextFile**. The "first" functions set up search parameters and return the first matching port. The "next" functions use the original search information to return subsequent matching ports, one at a time.

Asynchronous I/O: Functions

AIOAcquirePort

Allows an application to request exclusive ownership of a port
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOAcquirePort (
   int  *hardwareType,
   int  *boardNumber,
   int  *portNumber,
   int  *portHandle);
```

Parameters

hardwareType

(IN/OUT) Points to the hardware type value for port selection. Can be equal to -1; if so, the actual selected value is returned.

boardNumber

(IN/OUT) Points to the board number within the hardware type. Can be equal to -1; if so, the actual board number value is returned.

portNumber

(IN/OUT) Points to the number of the port within the board. Can be equal to -1; if so, the actual port number value is returned.

portHandle

(OUT) Points to the returned handle value of the acquired port.

Return Values

0	AIO_SUCCESS	Successful. The port handle value is returned.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-6	AIO_PORT_NOT_AVAILA	The requested port is not

-6	AIO_PORT_NOT_AVAILABLE	The requested port is not available. If you use a wildcard request, no free ports are available.
-10	AIO_TYPE_NUMBER_INVALID	No hardware driver of the requested type is registered with Asynchronous I/O, yet.
-11	AIO_BOARD_NUMBER_INVALID	No board of the hardware type requested has been registered with Asynchronous I/O, yet.
-12	AIO_PORT_NUMBER_INVALID	The requested port is an unknown port to Asynchronous I/O.

Remarks

You can use specific parameters to select particular ports or use wildcard parameters to select any free port from a class of ports that is hardware specific.

If you provide nonnegative values for the first three parameters, a specific Asynchronous I/O port is selected. That is, Asynchronous I/O selects a specific hardware driver, a specific board controlled by that driver, and a specific port on that board. If another application has already acquired this port, the request is rejected with status AIO_PORT_NOT_AVAILABLE.

If one of the first three parameters is not valid, the request is rejected as follows:

If a specific *hardwareType* is given and that type is not loaded, AIO_TYPE_NUMBER_INVALID is returned.

If a wildcard *hardwareType* is given and no types are loaded, AIO_TYPE_NUMBER_INVALID is returned.

If a specific *boardNumber* is given and that board is not found, AIO_BOARD_NUMBER_INVALID is returned.

If a wildcard *boardNumber* is given and no boards are registered, AIO_BOARD_NUMBER_INVALID is returned.

If a specific *portNumber* is given and that port is not found, AIO_PORT_NUMBER_INVALID is returned.

If a wildcard *portNumber* is given and no boards are registered, AIO_PORT_NUMBER_INVALID is returned.

You can also use a value of -1 for any of the first three parameters to request nonspecific selection of available free ports. Asynchronous I/O locates an available port and acquires it for the application, returning the actual hardware type, board number, and port number values for the

selected port. If no ports are available, Asynchronous I/O returns the AIO_PORT_NOT_AVAILABLE status.

You can also use the wildcard value to define a range of ports that Asynchronous I/O must check for an available port. Thus, a wildcard value for the hardware type means Asynchronous I/O can check for ports across all hardware drivers. Using a wildcard value for the board number instructs Asynchronous I/O to check for available ports across all boards controlled by the selected driver. Similarly, use of the wildcard value for the port number parameter allows selection of any available port found on the selected boards and hardware types.

You can use this feature to limit the range of ports to those provided by one particular hardware type. To do so, specify the board and port number parameters using the wildcard value of -1 and use the desired, unique hardware type value. If no available ports are found, try the next most desirable hardware type in the next request.

Whenever you use the wildcard value for any of the addressing parameters and the request successfully selects an available port, the wildcard value is replaced with the address value for the selected port.

Once a port is successfully acquired, the application can verify configuration parameters by calling the **AIOGetPortConfiguration** function. Typically, all configuration parameters are reset to driver-defined default values. For example, the default size for transmit and receive buffers for the AIOCOMX driver is 1,024 bytes. The specific settings depend on the driver and hardware for the port and should be verified as correct for the application's desired use.

Most ports have default values of 2,400 for bit rate, 8 bits for character length, 1 bit for stop bits, no parity bit generation, and no flow control modes active. The software flow control characters are reset to the ASCII XON and XOFF character values. Signals that would be set using **AIOSetExternalControl** are off; the RS232 break signal is off, too. No data exists in either receive or transmit buffers.

See Also

AIOGetPortConfiguration, **AIOReleasePort**

AIOAcquirePortWithRTag

Allows an application to request exclusive ownership of a port
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOAcquirePortWithRTag (
   int   *hardwareType,
   int   *boardNumber,
   int   *portNumber,
   int   *portHandle,
   LONG   RTag);
```

Parameters

hardwareType

(IN/OUT) Points to the hardware type value for port selection. Can be equal to -1; if so, the actual selected value is returned.

boardNumber

(IN/OUT) Points to the board number within the hardware type. Can be equal to -1; if so, the actual board number value is returned.

portNumber

(IN/OUT) Points to the number of the port within the board. Can be equal to -1; if so, the actual port number value is returned.

portHandle

(OUT) Points to the returned handle value of the acquired port.

RTag

(IN) Specifies the allocated asynchronous port resource tag value using the signature ASYNCIOSignature, defined in the AIO.H file.

Return Values

0	AIO_SUCCESS	Successful. The port handle value is returned.

-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-6	AIO_PORT_NOT_AVAILABLE	The requested port is not available. If you use a wildcard request, no free ports are available.
-10	AIO_TYPE_NUMBER_INVALID	No hardware driver of the requested type is registered with Asynchronous I/O, yet.
-11	AIO_BOARD_NUMBER_INVALID	No board of the hardware type requested has been registered with Asynchronous I/O, yet.
-12	AIO_PORT_NUMBER_INVALID	The requested port is an unknown port to Asynchronous I/O.
-21	AIO_RTAG_INVALID	The requested resource tag is not a valid Asynchronous I/O resource tag.

Remarks

This function is provided only for those NLM™ applications that do not use the NetWare® API. Typical NetWare 3.1 and above applications, which use the NetWare API, use **AIOAcquirePort**.

See Also

AIOAcquirePort, **AIOReleasePort**

AIOConfigurePort

Sets the configuration parameters affecting the format of data transmission
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOConfigurePort (
   int    portHandle,
   BYTE   bitRate,
   BYTE   dataBits,
   BYTE   stopBits,
   BYTE   parityMode,
   BYTE   flowCtrlMode);
```

Parameters

portHandle

(IN) Specifies the handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

bitRate

(IN) Specifies the index for selecting the data rate in bits per second (bps). A value of 255 (0xFF) can be used if no change to the previous value is desired.

dataBits

(IN) Specifies the index for selecting the number of data bits per character. A value of 255 (0xFF) can be used if no change to the previous value is desired.

stopBits

(IN) Specifies the index for selecting the number of stop bits per character. A value of 255 (0xFF) can be used if no change to the previous value is desired.

parityMode

(IN) Specifies the index for selecting the type of generated parity.

flowCtrlMode

(IN) Specifies the flow control-mode settings value. This value

contains one or more bit masks, which can be combined using the bitwise inclusive OR operator (|).

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-7	AIO_QUALIFIED_SUCCESS	Accepted, but some parameters were substituted with different values.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

The **AIOConfigurePort** function sets configuration parameters such as data rate, data length in bits, number of stop bits, and type of generated parity. Additionally, you can specify the enabled data flow control modes here.

The index values for the *bitRate* parameter are defined as follows:

Bit Rate Index	Actual Bit Rate (bps)
AIO_BAUD_50	50
AIO_BAUD_75	75
AIO_BAUD_110	110
AIO_BAUD_134p5	134.5
AIO_BAUD_150	150
AIO_BAUD_300	300
AIO_BAUD_600	600
AIO_BAUD_1200	1200
AIO_BAUD_1800	1800
AIO_BAUD_2000	2000
AIO_BAUD_2400	2400
AIO_BAUD_3600	3600
AIO_BAUD_4800	4800
AIO_BAUD_7200	7200

AIO_BAUD_9600	9600
AIO_BAUD_19200	19200
AIO_BAUD_38400	38400
AIO_BAUD_57600	57600
AIO_BAUD_115200	115200

The index values for the *dataBits* parameter are defined as follows:

Data Bits Index	Actual Data Bits
AIO_DATA_BITS_5	5
AIO_DATA_BITS_6	6
AIO_DATA_BITS_7	7
AIO_DATA_BITS_8	8

The index values for the *stopBits* parameter are defined as follows:

Stop Bits Index	Actual Stop Bits
AIO_STOP_BITS_1	1
AIO_STOP_BITS_1p5	1.5
AIO_STOP_BITS_2	2

The index values for the *parityMode* parameter are defined as follows:

Parity Index	Parity Generated
AIO_PARITY_NONE	None
AIO_PARITY_ODD	Odd parity
AIO_PARITY_EVEN	Even parity
AIO_PARITY_MARK	Mark parity
AIO_PARITY_SPACE	Space parity

The *flowCtrlMode* parameter can contain one or more bit masks, which can be combined using the bitwise inclusive OR operator (|). The following values are defined:

AIO_SOFTWARE_FLOW_CONTROL_OFF
AIO_SOFTWARE_FLOW_CONTROL_ON
AIO_HARDWARE_FLOW_CONTROL_OFF
AIO_HARDWARE_FLOW_CONTROL_ON

NOTE: Not all asynchronous hardware supports all the parameters and values outlined previously. Some drivers might choose to substitute the requested values for values the hardware does support. In this case, the status AIO_QUALIFIED_SUCCESS is returned. You can use **AIOGetPortConfiguration** to read the actual configuration values set for the hardware.

For example, the AIOCOMX driver does not support 1.5 stop bits per character, except when a character is made up of 5 data bits only. Therefore, when a character is set to 6, 7, or 8 data bits and 1.5 stop bits per character are specified, the AIOCOMX driver uses 2 stop bits.

When hardware flow control is enabled, the Clear To Send (CTS) signal controls when Asynchronous I/O can transmit data. While the signal is on, Asynchronous I/O transmits data; when the external attached device turns the signal off, Asynchronous I/O stops transmitting until the signal is turned on again.

Similarly, Asynchronous I/O uses the Request To Send (RTS) signal to indicate when the external attached device is permitted to pass data into the attached port's receive buffer. When the receive buffer is full, Asynchronous I/O turns off the RTS signal; when more room becomes available in the receive buffer, the signal is turned on again.

NOTE: The application must have enabled the RTS signal using **AIOSetExternalControl** for hardware flow control to be able to raise and lower the RTS signal. The application can always override the enabling of RTS by explicitly lowering RTS itself.

An application can repeat this request as many times as needed. However, note that updating the parameters that control data transmission formats could easily disrupt any ongoing data transmissions.

See Also

AIOGetPortConfiguration, **AIOSetFlowControl**

AIOFlushBuffers

Discards data resident in the receive buffer (if not yet read) or transmit buffer (if not yet output from the port)

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOFlushBuffers (
   int    portHandle,
   WORD   flushType);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

flushType

(IN) Specifies from which buffers to discard data.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function flushes the receive buffer, the transmit buffer, or both. The

flushType parameter indicates which buffers to flush. This parameter can contain one or more bit masks, which can be combined using the bitwise inclusive OR operator (|). The following *flushType* mask values are defined:

AIO_FLUSH_WRITE_BUFFER
AIO_FLUSH_READ_BUFFER

When AIO_FLUSH_READ_BUFFER is specified and any flow control method is active, then certain actions can take place in addition to flushing the receive buffer.

The driver sends a transmit XON character to permit remote transmission if all of the following conditions are met:

Software flow control is enabled.

A transmit XOFF was previously sent to the remote end because the receive buffer reached or exceeded the threshold (set by the driver).

The Request To Send (RTS) signal is turned on if all of the following conditions are met:

Hardware flow control is enabled.

RTS was turned off because the receive buffer reached or exceeded the threshold (set by the driver).

RTS is still enabled by the application (using **AIOSetExternalControl**).

See Also

AIOSetExternalControl, **AIOSetFlowControl**

AIOGetDriverList

Returns information on drivers registered with Asynchronous I/O
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetDriverList (
   int             lastHardwareType,
   AIODRIVERLIST   *DriverList)
```

Parameters

lastHardwareType

(IN) Specifies the number of the last hardware type returned by **AIOGetDriverList** for a repeated call. This argument should be equal to zero (0) on the first call to this function.

pDriverList

(IN/OUT) Points to a user-allocated buffer into which the driver list information is stored.

Return Values

0	AIO_SUCCESS	Successful.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-10	AIO_TYPE_NUMBER_INVALID	No driver with this hardware type is registered with Asynchronous I/O.

Remarks

This function fills a user buffer with information about drivers registered

with Asynchronous I/O. This information includes the hardware type value used with **AIOAcquirePort** and **AIOGetFirstPortInfo**; the total number of ports accessible through this driver; and the ASCII name of the driver.

You can determine the number of entries Asynchronous I/O returns by examining the *returnLength* field. Subtract from the *returnLength* field the byte length of the field (2), and divide the result by the byte length of the AIODRIVERLISTENTRY structure (see the "Example" below). This yields the number of entries returned. The number of entries can be zero if the preceding call returned the remainder of the driver information entries.

Because the amount of information for all drivers registered with Asynchronous I/O might be too big to fit in the buffer provided by the application, this function can be repeated as many times as required to return all the information. For a repeated call, the *lastHardwareType* parameter value should be the hardware type value of the last driver for which information was returned.

On the first call to **AIOGetDriverList**, this value should be zero, indicating that no prior drivers were examined and that Asynchronous I/O should return data starting with the first driver registered with Asynchronous I/O. Subsequent calls use the *hardwareType* in the last filled driver entry returned from the previous call.

The **AIOGetDriverList** function should be repeated any time the number of entries returned equals the maximum number that can fit within the length of the user-provided buffer.

See Also

AIOAcquirePort, **AIOGetFirstPortInfo**, **AIOGetNextPortInfo**

AIOGetExternalStatus

Returns information regarding the state of hardware signals from external equipment

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetExternalStatus (
   int   portHandle,
   LONG  *extStatus,
   LONG  *chgdExtStatus);
```

Parameters

portHandle

(IN) Specifies the handle identifying the port to be addressed. The **AIOAcquirePort** function returns this handle. If the handle value is not valid, the request is rejected.

extStatus

(OUT) Points to the bit mask to receive the current external status values bit mask. (Optionally, this can be NULL.)

chgdExtStatus

(OUT) Points to the bit mask to receive the changed external status values bit mask. (Optionally, this can be NULL.)

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function returns information regarding the state of hardware signals from external equipment. For most serial equipment these signals include Data Set Ready (DSR), Clear To Send (CTS), and Data Carrier Detect (DCD). The information format can be specific to the particular driver employed.

This function returns two bit masks, the first of which indicates the current external signal values. The second indicates whether any signal values have changed since the last request for external status.

The bit value assignments for signals are the same in both bit masks and are defined as follows:

Name	Meaning	Values
AIO_EXTSTA_RI	Ring Indicator signal	0x00000001
AIO_EXTSTA_DCD	DCD signal	0x00000008
AIO_EXTSTA_DSR	DSR signal	0x00000010
AIO_EXTSTA_CTS	CTS signal	0x00000020
AIO_EXTSTA_BREAK	Break signal	0x00000080

Whenever the external status values are requested, the changed status mask is reset to zero. This allows an application to determine which signals have changed state since the last time the external status values were requested.

NOTE: AIOGetPortStatus also resets the changed external status values.

An individual bit is set in the changed mask when that signal changes value, whether from on to off or off to on. A set bit remains set even when the signal makes several transitions.

The status bit mask values previously outlined are those defined for RS232-type asynchronous devices. Other devices might return status values that emulate these RS232 inputs. Certain drivers might define additional status values specific to the particular devices handled. These status bits are defined using the higher-order bits in the LONG values.

The "break" status value is not an RS232 hardware signal input. Rather, it

indicates whether the port has received a "break" data signal. The current status bit mask is set only while actually receiving the break signal. Once the break signal turns off, the current status bit becomes zero. Thus, applications should test both the current and changed status masks.

See Also

AIOGetPortStatus

AIOGetFirstPortInfo

Examines information for ports known to Asynchronous I/O without having to acquire them to do so

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetFirstPortInfo (
   int                   hardwareType,
   int                   boardNumber,
   int                   portNumber,
   AIOPORTSEARCH        *portSearch,
   AIOPORTINFO          *portInfo,
   AIOPORTCAPABILITIES  *capabilities,
   AIODVRCAPABILITIES   *dvrCapabilities,
   char                 *NLMModuleName);
```

Parameters

hardwareType

(IN) Specifies the hardware type address value for port selection. Can be equal to -1.

boardNumber

(IN) Specifies the board number (within the hardware type) for port selection. Can be equal to -1.

portNumber

(IN) Specifies the port number (within the board number) for port selection. Can be equal to -1.

portSearch

(IN/OUT) Points to a user-allocated buffer used by this function and **AIOGetNextPortInfo**. Do not modify this pointer between calls to **AIOGetNextPortInfo**. Define the structure using the typedef name AIOPORTSEARCH. This pointer cannot be NULL.

portInfo

(OUT) Points to a user-allocated buffer to receive the returned port information structure value.

capabilities

(OUT) Points to a user-allocated buffer to receive the returned port capabilities structure values (optionally, this can be NULL). The *returnLength* field of the AIOPORTCAPABILITIES structure must be initialized prior to the call.

dvrcapabilities

(IN/OUT) Points to a user-allocated buffer to receive the returned driver-specific port capability structure values. (Optionally, this can be NULL.) The *returnLength* field of the AIODVRCAPABILITIES structure must be initialized prior to the call.

NLMModuleName

(OUT) Points to a 128-byte character array. (Optionally, this can be NULL.) If the returned port is currently acquired by an NLM, the module name of the owning NLM is copied into this character array. The name is NULL-terminated.

Return Values

0	AIO_SUCCESS	Successful.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-8	AIO_NO_MORE_PORTS	No ports exist with the given address values.
-10	AIO_TYPE_NUMBER_INVALID	No driver with this hardware type is registered with Asynchronous I/O.
-11	AIO_BOARD_NUMBER_INVALID	No board with this number exists for the selected hardware type.
-12	AIO_PORT_NUMBER_INVALID	The requested port is not known to Asynchronous I/O.

Remarks

This function can be used to select information for one port or to select a subset of ports for which information is to be returned. This information includes the current availability of the port and its capabilities. When wildcard address values are used, subsequent calls to **AIOGetNextPortInfo** can be used to return more ports in the selected subset of known ports.

When the first three parameters are specified with nonnegative values, a single port is addressed. That is, the*hardwareType* parameter specifies a

particular hardware driver, the *boardNumber* parameter specifies a particular board controlled by that driver, and the *portNumber* parameter specifies a particular port on that board.

Asynchronous I/O drivers can control multiple boards, with each board containing multiple ports. When an Asynchronous I/O driver is loaded, it provides Asynchronous I/O with information about the boards and ports controlled by that driver. The type and number parameter values are limited to those known by Asynchronous I/O, which were registered with Asynchronous I/O by each of the Asynchronous I/O drivers when they were loaded.

Wildcard values can be used in any combination with nonnegative values to define a range of ports for which Asynchronous I/O is to report information. Thus, a wildcard value for the *hardwareType* parameter means Asynchronous I/O can check for ports across all hardware drivers. A wildcard value for the *boardNumber* parameter means Asynchronous I/O checks for available ports across all boards controlled by the selected drivers. Similarly, a wildcard value for the *portNumber* parameter means Asynchronous I/O checks all available ports found on the specified boards and hardware types.

One obvious use of this feature is to limit the range of ports to those provided by one particular hardware type. In this case, the board and port number parameters are specified using the wildcard value of -1, and the hardware type is the desired, unique hardware type value.

Whenever the wildcard value is used for any of the addressing parameters, the application should check the returned data structure parameters for the actual hardware type, board number, and port number of the port selected.

If one of the first three parameters is not valid, the request is rejected as follows:

If a specific *hardwareType* is given and that type is not loaded, AIO_TYPE_NUMBER_INVALID is returned.

If a wildcard *hardwareType* is given and no types are loaded, AIO_TYPE_NUMBER_INVALID is returned.

If a specific *boardNumber* is given and that board is not found, AIO_BOARD_NUMBER_INVALID is returned.

If a wildcard *boardNumber* is given and no boards are registered, AIO_BOARD_NUMBER_INVALID is returned.

If a specific *portNumber* is given and that port is not found, AIO_PORT_NUMBER_INVALID is returned.

If a wildcard *portNumber* is given and no boards are registered, AIO_PORT_NUMBER_INVALID is returned.

The actual format of the driver-specific data structure is not known to

Asynchronous I/O, and the driver developer must supply the description and definition of the structure. If an application does not need this information, the *dvrcapabilities* parameter should be set to NULL. If you want to get driver-specific information, set the return buffer to the appropriate size and set its length in the *returnLength* field of the structure. In no case does a driver return more than the number of bytes indicated by the buffer length. If there is less driver data to be returned, the *returnLength* field is set to the actual number of bytes of data returned.

The AIOPORTINFO structure contains a *chgdExtStatus* value. Unlike **AIOGetPortStatus** and **AIOGetExternalStatus**, **AIOGetFirstPortInfo** does not reset this value.

See Also

AIOGetNextPortInfo, **AIOGetPortCapability**

AIOGetNextPortInfo

Used after **AIOGetFirstPortInfo** to examine the next port of a subset of ports known to Asynchronous I/O

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetNextPortInfo (
   AIOPORTSEARCH         *portSearch,
   AIOPORTINFO           *portInfo,
   AIOPORTCAPABILITIES   *capabilities,
   AIODVRCAPABILITIES    *dvrCapabilities,
   char                  *NLMModuleName);
```

Parameters

portSearch

(IN/OUT) Points to a user-allocated buffer used by this function and **AIOGetFirstPortInfo**. Do not modify this pointer between calls to **AIOGetNextPortInfo**. Define the structure using the typedef name AIOPORTSEARCH.

portInfo

(OUT) Points to a user-allocated buffer to receive the returned port information structure value.

capabilities

(OUT) Points to a user-allocated buffer to receive the returned port capabilities structure values. (Optionally, this can be NULL.) The *returnLength* field of the AIOPORTCAPABILITIES structure must be initialized prior to the call.

dvrcapabilities

(IN/OUT) Points to a user-allocated buffer to receive the returned driver-specific port capability structure values. (Optionally, this can be NULL.) The *returnLength* field of the AIODVRCAPABILITIES structure must be initialized prior to the call.

NLMModuleName

(OUT) Points to a 128-byte character array. If the returned port is

currently acquired by an NLM, the module name of the owning NLM is copied into this character array. The name is NULL-terminated.

Return Values

0	AIO_SUCCESS	Successful.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-8	AIO_NO_MORE_PORTS	No further ports exist with the given address values.

Remarks

This function continues searching for the next port for which to return information based on the search pattern set by **AIOGetFirstPortInfo**. The AIOPORTSEARCH structure must have been set up using **AIOGetFirstPortInfo** and must remain unchanged between calls.

See Also

AIOGetFirstPortInfo, **AIOGetPortCapability**

AIOGetPortCapability

Queries for capability information about a specific, previously acquired port
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetPortCapability (
   int                    portHandle,
   AIOPORTCAPABILITIES   *capabilities,
   AIODVRCAPABILITIES    *dvrCapabilities);
```

Parameters

portHandle

(IN) Specifies the handle identifying the port to be addressed. The **AIOAcquirePort** function returns this handle. If the handle value is not valid, the request is rejected.

capabilities

(IN/OUT) Points to a user-allocated buffer to receive the returned port capability structure values. (Optionally, this can be NULL.)

dvrCapabilities

(IN/OUT) Points to a user-allocated buffer to receive the returned driver-specific port capability structure values. (Optionally, this can be NULL.)

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function allows an application to query capability information about a port. Capabilities, such as baud rate and flow control methods, might not be present in some Asynchronous I/O drivers.

Note that the capabilities data returned in the AIOPORTCAPABILITIES structure characterizes only the capabilities of the driver and hardware installed in the server. This does not indicate what the capabilities of the asynchronous hardware attached to the port might be. For example, assume a that port's driver and hardware are able to transfer data at 57.6 kilobits per second (Kbps), but a Hayes 2400 modem is attached. The data transfer is limited to the capabilities of the external device.

The actual format of the driver-specific data structure is not known to Asynchronous I/O, and the driver developer must supply the description and definition of the structure. If an application does not need this information, the *dvrCapabilities* parameter should be set to NULL. If the driver-specific information is desired, the return buffer should be of an appropriate size, and its length set in the *returnLength* field of the structure. In no case does a driver return more than the number of bytes indicated by the buffer length. If there is less driver data to be returned, the *returnLength* field is set to the actual number of bytes of data returned.

If the minimum and maximum read or write buffer sizes are equal, the **AIOSetReadBufferSize** or **AIOSetWriteBufferSize** functions are not supported.

See Also

AIOConfigurePort, **AIOGetPortConfiguration**, **AIOGetReadBufferSize** , **AIOGetWriteBufferSize**, **AIOSetExternalControl**

AIOGetPortConfiguration

Requests the current configuration information from an acquired port
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetPortConfiguration (
   int              portHandle,
   AIOPORTCONFIG   *configuration,
   AIODVRCONFIG    *dvrconfiguration);
```

Parameters

portHandle

(IN) Specifies the handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

configuration

(OUT) Points to a user-allocated port configuration structure. (Optionally, this value can be NULL.) The *returnLength* field of the AIOPORTCONFIG structure must be initialized before the call.

dvrconfiguration

(IN/OUT) Points to a user-allocated buffer to receive the returned driver-specific port configuration structure values. (Optionally, this value can be NULL.) The *returnLength* field of the AIOPORTCONFIG structure must be initialized before the call.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.

-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

The configuration values currently in use by the port are returned. These values are the cumulative result of any requests to the port that change how data is transmitted or received (such as **AIOConfigurePort** or **AIOSetExternalControl**).

You can use this function to check whether all application-specified configuration values given in the **AIOConfigurePort** function were valid for a particular port. As discussed in the description of that function, requested configuration values that are not supported by a driver can be modified to supported values. The values returned in this structure reflect those values actually used by the driver.

The actual format of the driver-specific data structure is not known to Asynchronous I/O, and the driver developer supplies the description and definition of the structure. If an application does not need this information, the *dvrconfiguration* parameter should be set to NULL.

If the driver-specific information is desired, the return buffer should be of an appropriate size, and its length should be set in the *returnLength* field of the AIODVRCONFIG structure. In no case does a driver return more than the number of bytes indicated by the buffer length. If there is less driver data to be returned, the *returnLength* field is set to the actual number of bytes of data returned.

See Also

AIOConfigurePort, **AIOSetExternalControl**, **AIOSetFlowControl**, **AIOSetFlowControlCharacters**, **AIOSetReadBufferSize**, **AIOSetWriteBufferSize**

Example

AIOGetPortConfiguration

```
#include <aio.h>

int   portHandle;      /* Port handle returned by AIOAcquirePort */
int   ccode;
AIOPORTCONFIG   portConfig;

/*
   Fetch the active port configuration values,
   Is break on?
*/
```

```
portConfig.returnLength = sizeof(portConfig);
ccode = AIOGetPortConfiguration( portHandle, &portConfig, NULL ):
if( ccode )
   /* Error handling routine goes here */
if( portConfig.breakMode == AIO_SET_BREAK_ON )
{
   /* Turn off break with AIOSetExternalControl */
}
```

AIOGetPortStatistics

Returns statistical information for a port, such as counts of received and transmitted data bytes and counts of errors encountered

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetPortStatistics (
   int                  portHandle,
   AIOPORTSTATISTICS   *statistics,
   AIODVRSTATISTICS    *dvrstatistics);
```

Parameters

portHandle

(IN) Specifies the handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

statistics

(IN/OUT) Points to the structure AIOPORTSTATISTICS. (Optionally, this can be NULL.) The returnLength field of this structure must be initialized prior to the call.

dvrstatistics

(IN/OUT) Points to a user-allocated buffer to receive the returned driver-specific port statistics structure values. (Optionally, this can be NULL.) The returnLength field of the AIODVRSTATISTICS structure must be initialized prior to the call.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling

		application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function begins collecting statistics when the Asynchronous I/O driver loads the board and ports. It continues collecting counts throughout the lifetime of the driver's usage and ceases only when the server operator unloads the Asynchronous I/O driver NLM.

An application that wants to derive the counters' delta values from an application's use of a port should, after acquiring the port, request the initial statistics values. From then on, the differences between those initial values and the current values are the statistics for the current usage.

The actual format of the driver-specific data structure is not known to Asynchronous I/O, and the driver developer supplies the description and definition of this structure. If an application does not need this information, the *dvrstatistics* parameter should be set to NULL.

If the driver-specific information is desired, the return buffer should be of an appropriate size, and its length should be set in the *returnLength* field of the structure. In no case does a driver return more than the number of bytes indicated by the buffer length. If there is less driver data to be returned, the returnLength field is set to the actual number of bytes of data returned.

AIOGetPortStatus

Returns in one request the complete state of the port
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetPortStatus (
   int    portHandle,
   LONG  *writeCount,
   WORD  *writeState,
   LONG  *readCount,
   WORD  *readState,
   LONG  *extStatus,
   LONG  *chgdExtStatus);
```

Parameters

portHandle

(IN) Specifies the handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

writeCount

(OUT) Points to the number of bytes yet to be transmitted from the port. (Optionally, this can be NULL.) The count can vary from zero to the length of the transmit buffer.

writeState

(OUT) Points to the current output stream state value. (Optionally, this can be NULL.)

readCount

(OUT) Points to the number of bytes present in the port's receive buffer. (Optionally, this can be NULL.) The count can vary from zero to the length of the receive buffer.

readState

(OUT) Points to the current receive stream state value. (Optionally, this can be NULL.)

extStatus

(OUT) Points to a bit mask to receive the current external status bit mask value. (Optionally, this can be NULL.)

chgdExtStatus

(OUT) Points to a bit mask to receive the external status changed values bit mask. (Optionally, this can be NULL.)

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function returns complete state information for a specified port, combining information from **AIOWriteStatus**, **AIOReadStatus**, and **AIOGetExternalStatus**.

The *writeCount* parameter is number of data bytes yet to be transmitted. Before you use the **AIOReleasePort** function, it is best to verify that data queued for transmission has actually been sent.

The *writeState* parameter is the transmission state (for example, whether transmission is stopped due to flow control). The*writeState* value for a port allows checking for transmission in progress, transmission complete, and transmission stopped due to software flow control. Applications might consider these state values when scheduling output to a port.

The following values are defined for the *writeState* parameter:

Name	Meaning
AIO_TRANSMIT_IDLE	Transmit buffer is empty
AIO_TRANSMIT_ACTIVE	Transmission is in progress
AIO_TRANSMIT_XOFFED	Transmission is stopped due to software flow control

The *readCont* parameter is the number of data bytes currently held in the port's receive buffer. This count allows an application to determine whether data is present in the port's receive buffer. If the returned count is nonzero, **AIOReadData** can then copy data to an application buffer.

The *readState* parameter is the state of receive buffering (that is, whether input buffers have filled, making overrun possible). When the receive buffer is completely filled, any additional data that arrives might be lost. If the application can read some or all of the data before additional data arrives, no data is lost. At high data rates, however, you should try to avoid this risk.

The following values are defined for the *readState* parameter:

Name	Meaning
AIO_RECEIVE_ACTIVE	Reception is in progress
AIO_RECEIVE_FULL	Receive buffer is full

The *extStatus* and *chgdExtStatus* parameters are bit masks that reflect the state of hardware signals from external equipment. For most serial equipment these signals include Data Set Ready (DSR), Clear To Send (CTS), and Data Carrier Detect (DCD). The information format might be specific to the particular driver employed.

The *extStatus* bit mask indicates the current external signal values. The *chgdExtStatus* bit mask indicates whether any signal values have changed since the last request for external status. An individual bit is set in the changed mask when that signal changes value, whether from on to off or off to on. A set bit remains set even when the signal makes several transitions.

The bit masks for the *extStatus* and *chgdExtStatus* parameters are the same and are defined as follows:

Name	Meaning
AIO_EXTSTA_RI	RingIndicator signal
AIO_EXTSTA_DCD	DCD signal
AIO_EXTSTA_DSR	DSR signal
AIO_EXTSTA_CTS	CTS signal
AIO_EXTSTA_BREAK	Break detected

NOTE: When external status values are requested, the changed status mask is reset to zero. This allows an application to determine which signals have changed state since the last time external status values were requested.

The status bit mask values previously outlined are those defined for RS232-type asynchronous devices. Other devices might return status values that emulate these RS232 inputs. Certain drivers might define additional status values specific to the particular devices handled. These status bits are defined using the higher-order bits in the LONG values.

The "break" status value is not an RS232 hardware signal input. Rather, it indicates whether the port has received a "break" data signal. The current status bit mask is set only while actually receiving the break signal. Once the break signal turns off, the current status bit becomes zero. Thus, applications should test both the current and changed status masks.

See Also

AIOGetExternalStatus, **AIOReadStatus**, **AIOWriteStatus**

AIOGetReadBufferSize

Returns the total size of all buffers that a port uses to store received data
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetReadBufferSize (
   int    portHandle,
   LONG  *readSize);
```

Parameters

portHandle

(IN) Specifies a handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

readSize

(OUT) Points to the total size of the receive buffers.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

You can use this function to implement data transmission pacing. You can avoid pauses due to flow control by sizing the amount of data sent by

the remote device to match the receive buffer size.

The default receive buffer size is set when a port is acquired. For the AIOCOMX driver, the default receive buffer size is 1,024 bytes.

See Also

AIOGetWriteBufferSize

AIOGetWriteBufferSize

Returns the total size of all buffers that a port uses to store data to be transmitted

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOGetWriteBufferSize (
   int    portHandle,
   LONG  *writeSize);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

writeSize

(OUT) Points to the total size of the transmit buffers.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

You can use this function to implement data transmission pacing. You

might want to take the transmit buffer size into consideration when scheduling output to the remote device.

The default transmit buffer size is set when a port is acquired. For the AIOCOMX driver, the default transmit buffer size is 1,024 bytes.

See Also

AIOGetReadBufferSize

AIOReadData

Copies data from a port's receive data buffer to the application buffer
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOReadData (
   int    portHandle,
   char  *buffer,
   LONG   lengthOfBuffer,
   LONG  *numberOfBytesRead);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

buffer

(IN) Points to the application's data buffer.

lengthOfBuffer

(IN) Specifies the maximum number of data bytes to be copied from the port's receive buffer.

numberOfBytesRead

(OUT) Points to the number of data bytes actually copied from the port. This value can be zero.

Return Values

0	AIO_SUCCESS	Successful. Returns the *numberOfBytesRead* value.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling

		control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

The copy count returned in the *numberOfBytesRead* parameter can be less than the *lengthOfBuffer* value when fewer bytes than the *lengthOfBuffer* remain to be copied from the receive buffer. The returned count value in the *numberOfBytesRead* parameter is zero when no data exists in the port's receive buffer.

Always check the *numberOfBytesRead* value, even when the return code is AIO_SUCCESS. The only way to determine the actual number of bytes read is to check the return value.

If the maximum number of bytes requested has been returned (*lengthOfBuffer == numberOfBytesRead*), you should repeat the function call, because there might be more data remaining in the receive buffer.

The default receive buffer size is driver specific. For example, the default receive buffer size for the AIOCOMX driver is 1,024 bytes.

See Also

AIOGetReadBufferSize, **AIOReadStatus**, **AIOWriteData**

AIOReadStatus

Returns information regarding the status of data reception
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOReadStatus (
   int    portHandle,
   LONG  *count,
   WORD  *state);
```

Parameters

portHandle

(IN) Specifies a handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

count

(OUT) Points to the number of bytes present in the port's receive buffer. The count can vary from zero to the length of the receive buffer. The pointer can be NULL.

state

(OUT) Points to the current receive stream state value. The pointer can be NULL.

Return Values

0	AIO_SUCCESS	Successful. Count and state values are returned.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GO	Requested port's board has been deregistered

	NE	by the driver.

Remarks

This function returns information including the count of data bytes received, the count of data bytes currently held in receive buffers, and the condition of receive buffering (that is, whether input buffers have filled, making overrun possible). This function also allows an application to determine whether data is present in the port's receive buffer. If the returned count is nonzero, **AIOReadData** can then copy data to an application buffer.

The following state values are defined:

AIO_RECEIVE_ACTIVE
AIO_RECEIVE_FULL

The receive state value for a port allows checking for the possibility of receive data overrun. When the receive buffer is completely filled, any additional data that arrives might be discarded. If the application can read some or all of the data before further data arrives, no data is lost. At high data rates, however, you should try to avoid this risk.

See Also

AIOGetPortStatus, **AIOWriteStatus**

AIOReleasePort

Allows an application to relinquish ownership of a port, freeing it for use by another application

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOReleasePort (
   int   portHandle);
```

Parameters

portHandle

(IN) Specifies a handle that identifies the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

Return Values

0	AIO_SUCCES	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application. The port is released anyway.

Remarks

Releasing a port resets the current configuration and external signals to driver-specified default values. It also flushes the receive and transmit buffers. Any buffered data (including data yet to be transmitted) is lost.

You can release a port using this function even when the driver, board, or port is malfunctioning. A nonzero return value indicates that an error occurred. When AIO_FAILURE is returned, the port is released anyway.

Communication Service Group

See Also

AIOAcquirePort

AIOSetControlData

Allows an application to pass a request and data structure to an Asynchronous I/O driver

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetControlData (
   int               portHandle,
   int               requestType,
   AIOCONTROLDATA  *requestData);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

requestType

(IN) Specifies the action to perform or the particular data structure being passed to the driver. An individual request can use only one of the possible *requestType* values.

> **NOTE:** At this time, there are no values defined for this parameter.

requestData

(IN/OUT) Points to a user-allocated buffer. This buffer can be used to pass user data to a driver, to receive returned driver data, or both. The *returnLength* field of the AIOCONTROLDATA structure must be initialized before the call.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred

-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-3	AIO_FUNC_NOT_SUPPORTED	The driver does not support this function.
-5	AIO_INVALID_PARAMETER	Bad *requestType* passed in.
-15	AIO_BAD_REQUEST_TYPE	Request type is not valid or is illegal.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function provides a general mechanism for altering various control information for a communications port. This information might include protocol-specific parameters or information required by the type of equipment connected (such as X.21 parameters). This function is an extension of **AIOSetExternalControl**, which is used for requests with single data values. This function allows for requests that require multiple or complex data values.

The *requestType* parameter specifies which state or mode is updated, and the *requestData* parameter supplies the new value for the state or mode. The *requestType* values have no current definitions.

The *returnLength* field is set to the byte length of the supplied return buffer. It is returned as the length filled by the function, which is always less than or equal to the original length. This is the only structure field Asynchronous I/O defines.

See Also

AIOConfigurePort, **AIOSetExternalControl**

AIOSetExternalControl

Allows an application to control the states of external signals
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetExternalControl (
   int   portHandle,
   int   requestType,
   int   requestValue);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

requestType

(IN) Specifies which state or mode is being updated. An individual request can use only one of these *requestType* values.

requestValue

(IN) Specifies a new value for state or mode.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-3	AIO_FUNC_NOT_SUPPORTED	The driver does not support this function.
-5	AIO_INVALID_PARAMETER	Request value is not valid.

-15	AIO_BAD_REQUEST_TYPE	Request type is not valid or is illegal.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

This function provides a general mechanism for altering the states and modes of communications ports. These include the RS232 external control signals, the asynchronous "break" signal, the deadman timer, and flow control modes.

The *requestType* parameter specifies which state or mode is updated, and the *requestValue* parameter supplies the new value for the state or mode. The *requestType* values are defined as follows:

AIO_EXTERNAL_CONTROL	External control signals
AIO_BREAK_CONTROL	Asynchronous break signal
AIO_FLOW_CONTROL	Asynchronous flow control mode
AIO_FLOW_CONTROL_CHARACTERS	Asynchronous flow control characters
AIO_SET_DEADMAN_TIMER	Set deadman time interval

NOTE: You can use the AIO_EXTERNAL_CONTROL request to override control of the Request To Send (RTS) signal by hardware flow control. That is, if hardware flow control tries to reenable RTS, but the application has lowered the signal using this request, the signal remains off.

When used with the AIO_EXTERNAL_CONTROL request, the *requestValue* parameter turns the external control signals (DTR or RTS) either on or off. The request always affects both signals. A request to change the state of one signal must be accompanied by the desired state of the other signal.

AIO_EXTCTRL_DTR
AIO_EXTCTRL_RTS

When used with the AIO_BREAK_CONTROL request, the *requestValue* parameter turns the generation of the asynchronous break signal either on or off. Once turned on, the break signal remains on until turned off; the application is responsible for any required timing. The following *requestValue* values are defined for use with the AIO_BREAK_CONTROL request:

AIO_SET_BREAK_	Turn asynchronous break signal off

OFF	
AIO_SET_BREAK_ON	Turn asynchronous break signal on

When used with the AIO_FLOW_CONTROL request, the *requestValue* parameter can contain one or more bit masks that control which flow control modes are active. The software and hardware flow control modes are both set at the same time. That is, when setting the software flow control mode to on, the corresponding hardware flow control mode is set to off, and vice versa. The following *requestValue* bit mask values can be combined using the bitwise inclusive OR operator (|):

AIO_SOFTWARE_FLOW_CONTROL_ON|
AIO_HARDWARE_FLOW_CONTROL_OFF

AIO_SOFTWARE_FLOW_CONTROL_OFF|
AIO_HARDWARE_FLOW_CONTROL_ON

As with **AIOConfigurePort**, the RTS signal must have been enabled using AIO_EXTERNAL_CONTROL before hardware flow control can properly raise and lower the RTS signal as needed.

When used with the AIO_FLOW_CONTROL_CHARACTERS request, the *requestValue* parameter is a concatenation of four single BYTE values, in the order of MSB to LSB:

Bits	Position
31 to 24	transmitXon
23 to 16	transmitXoff
15 to 8	receiveXon
7 to 0	receiveXoff

These BYTE values are used when software flow control is enabled.

When used with the AIO_SET_DEADMAN_TIMER request, the *requestValue* parameter specifies whether the deadman timer feature is used. When the value is zero, the timer is disabled. When the value is nonzero, the deadman feature is enabled for the port and the value defines the time interval (in seconds) that is the greatest length of time allowed until the next AIO_SET_DEADMAN_TIMER request.

The driver decrements the timer value. The application can issue subsequent **AIOSetExternalControl** calls to reset the timer to a nonzero value and thus continue deadman monitoring, or the application can turn off the deadman timer feature by using a value of zero. If the time interval exceeds the given length, the driver assumes the application or server has failed (such as abended) and turns off the RS232 signals DTR and RTS.

NOTE: Call **AIOGetPortCapability** to determine whether the port supports the use of a deadman timer.

See Also

AIOConfigurePort, **AIOGetPortCapability**, **AIOSetFlowControl**, **AIOSetFlowControlCharacters**

AIOSetFlowControl

Allows the flow control modes for a port to be changed independently of the other configuration parameters that are given on an **AIOConfigurePort** function

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetFlowControl (
   int   portHandle,
   int   flowCtrlMode);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

flowCtrlMode

(IN) Specifies new flow control mode settings.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-5	AIO_INVALID_PARAMETER	Returned if not in the range of 0 - 3.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

Remarks

The software and hardware flow control modes are both set at the same time using the *flowCtrlMode* parameter, which can contain one or more bit masks. A request to change the state of one mode must be accompanied by the desired state of the other mode.

The following *flowCtrlMode* bit mask values can be combined using the bitwise inclusive OR operator (|).

AIO_SOFTWARE_FLOW_CONTROL_ON|
AIO_HARDWARE_FLOW_CONTROL_OFF

AIO_SOFTWARE_FLOW_CONTROL_OFF|
AIO_HARDWARE_FLOW_CONTROL_ON

When an application enables flow control mode using **AIOSetFlowControl**, certain actions can take place immediately if the receive buffer is currently at or over the receive threshold (set by the driver):

If software flow control is enabled, the transmit XOFF character is sent to the remote end.

If hardware flow control is enabled and Request To Send (RTS) is currently enabled by the application, RTS is disabled.

When an application disables flow control using **AIOSetFlowControl**, certain actions can take place immediately, depending on prior flow control actions, if any.

The driver sends an XON character to permit remote transmission if all of the following conditions are met:

Software flow control changes from enabled to disabled.

An XOFF was previously sent to the remote end because the receive buffer reached or exceeded the threshold (set by the driver).

The RTS signal is turned on if all of the following conditions are met:

Hardware flow control changes from enabled to disabled.

RTS was turned off because the receive buffer reached or exceeded the threshold (set by the driver).

RTS is still enabled by the application (using **AIOSetExternalControl**).

The driver restarts write data transmission if one of the following conditions are met:

Write data transmission was paused because the driver received an XOFF and software flow control changes from enabled to disabled.

The Clear To Send (CTS) signal is turned off and hardware flow control changes from enabled to disabled.

See Also

AIOConfigurePort, **AIOSetExternalControl**

AIOSetFlowControlCharacters

Establishes the data values used with software flow control
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetFlowControlCharacters (
   int    portHandle,
   BYTE   transmitXon,
   BYTE   transmitXoff,
   BYTE   receiveXon,
   BYTE   receiveXoff);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

transmitXon

(IN) Specifies a new enable transmission character for sending to the remote end.

transmitXoff

(IN) Specifies a new disable transmission character for sending to the remote end.

receiveXon

(IN) Specifies a new enable transmission character for receiving from the remote end.

receiveXoff

(IN) Specifies a new disable transmission character for receiving from the remote end.

Return Values

0	AIO_SUCCESS	Successful.

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

When a port is released, Asynchronous I/O resets the software flow control characters to their default values: ASCII XON (0x11) and XOFF (0x13) for both the transmit and receive directions.

See Also

AIOConfigurePort, **AIOSetExternalControl**, **AIOSetFlowControl**

AIOSetReadBufferSize

Dynamically sets the size of a port's buffers to be used for receiving data
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetReadBufferSize (
   int    portHandle,
   LONG   bufferSize);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

bufferSize

(IN) Specifies the new size (in bytes) for the receive buffers used with the port.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-3	AIO_FUNC_NOT_SUPPORTED	The selected driver does not support this function.
-14	AIO_DATA_PRESENT	Since data exists in the receive buffer, the request was not performed.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

If the port's driver supports it, this function resets the byte length of the receive buffer for the specified port. You can use this function to adjust the buffer size according to the data rates used with the port.

The default receive buffer size is set when a port is acquired. For the AIOCOMX driver, the default receive buffer size is 1,024 bytes.

If you call this function while data is present in the existing receive data buffer, the function returns AIO_DATA_PRESENT. The data remains in the buffer, and the buffer size is not changed. The application must either read the data using **AIOReadData**, or discard the data using **AIOFlushBuffers** before again calling **AIOSetReadBufferSize**.

See Also

AIOGetReadBufferSize, **AIOSetWriteBufferSize**

AIOSetWriteBufferSize

Dynamically sets the size of a port's buffers to be used for transmitting data
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOSetWriteBufferSize (
   int    portHandle,
   LONG   bufferSize);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

bufferSize

(IN) Specifies the new size (in bytes) for the transmit buffers used with the port.

Return Values

0	AIO_SUCCESS	Successful.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-3	AIO_FUNC_NOT_SUPPORTED	The selected driver does not support this function.
-14	AIO_DATA_PRESENT	Since data exists in the transmit buffer, the request was not performed.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver

Remarks

If the port's driver supports it, this function resets the byte length of the transmit buffer for the specified port. An application might use this to adjust the buffer size according to the data rates used with the port.

The default transmit buffer size is set when a port is acquired. For the AIOCOMX driver, the default transmit buffer size is 1,024 bytes.

If you call this function while data is present in the existing transmit data buffer, the function returns AIO_DATA_PRESENT. The data remains in the existing buffer, and the buffer size is not changed. The application must either wait until all data is transmitted, or discard the data using **AIOFlushBuffers** before again calling **AIOSetWriteBufferSize**.

See Also

AIOGetWriteBufferSize, **AIOSetReadBufferSize**

AIOWriteData

Copies data from an application buffer into the transmit data buffer for the port

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 3.1, 3.11, 3.12, 4.x

Platform: NLM

SMP Aware: No

Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOWriteData (
   int    portHandle,
   char  *buffer,
   LONG   lengthOfBuffer,
   LONG  *numberOfBytesWritten);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

buffer

(IN) Points to the application's data buffer.

lengthOfBuffer

(IN) Specifies the number of data bytes to be copied to the port's transmit buffer.

numberOfBytesWritten

(OUT) Points to the number of data bytes actually copied to the port's transmit buffer. This value can be zero.

Return Values

0	AIO_SUCCESS	Successful; *numberOfBytesWritten* parameter value is returned.
-1	AIO_BAD_HANDLE	Port handle value is not valid.

-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GONE	Requested port's board has been deregistered by the driver.

Remarks

Once you call this function, data transmission begins immediately if it is not already in progress, unless flow control has suspended output. Compare the number of bytes written to the *lengthOfBuffer* parameter after a successful return to see if all bytes of the buffer were actually written.

You can repeat this function to fill the transmit buffer; as long as sufficient room for the data exists in the transmit buffer, data is copied. As data is transmitted, more room is made available in the buffer, allowing more requests. For the AIOCOMX driver, the default transmit buffer size is 1,024 bytes.

Data transmission can be temporarily suspended when either software or hardware flow control is enabled.

See Also

AIOGetWriteBufferSize, **AIOFlushBuffers**, **AIOSetFlowControl**, **AIOSetWriteBufferSize**, **AIOWriteStatus**

AIOWriteStatus

Returns information regarding the status of data transmission
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.1, 3.11, 3.12, 4.x
Platform: NLM
SMP Aware: No
Service: Asynchronous I/O

Syntax

```
#include <aio.h>

int AIOWriteStatus (
   int    portHandle,
   LONG  *count,
   WORD  *state);
```

Parameters

portHandle

(IN) Specifies a handle identifying the port to be addressed. **AIOAcquirePort** returns this handle. If the handle value is not valid, the request is rejected.

count

(OUT) Points to the number of bytes yet to be transmitted from the port's transmit buffer. (Optionally, this can be NULL.) The count can vary from zero to the length of the transmit buffer.

state

(OUT) Points to the current output stream state value. (Optionally, this can be NULL.)

Return Values

0	AIO_SUCCESS	Successful, count and state values are returned.
-1	AIO_BAD_HANDLE	Port handle value is not valid.
-2	AIO_FAILURE	Unsuccessful. An error occurred beyond the control of Asynchronous I/O or the calling application.
-20	AIO_PORT_GON	Requested port's board has been

	E	deregistered by the driver.

Remarks

This function returns the count of data bytes yet to be transmitted and the transmission state (for example, whether output is stopped due to flow control). In addition, this function allows an application to determine when all data copied to a port for output has been transmitted. Before you call **AIOReleasePort**, it is best to verify that data queued for transmission has actually been sent.

The following state values are defined:

AIO_TRANSMIT_IDLE	Transmit buffer empty
AIO_TRANSMIT_ACTIVE	Transmission in progress
AIO_TRANSMIT_XOFFED	Stopped due to software flow control

The output state value for a port allows checking for transmission in progress, transmission complete, and transmission stopped due to software flow control. Applications might take these state values into consideration when scheduling output to a port.

See Also

AIOGetPortStatus, **AIOReadStatus**

Communication Service Group

Asynchronous I/O: Structures

AIOCONTROLDATA

Contains developer-defined Asynchronous I/O control data
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   byteData[2];
} AIOCONTROLDATA;
```

Fields

returnLength

Specifies the byte length of the supplied return buffer.

byteData

Specifies the user-defined contents.

Remarks

The *returnLength* field is returned as the length filled by the AIOCONTROLDATA structure, which is always less than or equal to the original length.

AIODRIVERLIST

Contains a list of Asynchronous I/O drivers
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD                 returnLength;
   AIODRIVERLISTENTRY   driver[1];
} AIODRIVERLIST;
```

Fields

returnLength

Specifies the byte length of the supplied structure.

driver

Specifies an array of driver entries (found by examining the *returnLength* field).

Remarks

The AIODRIVERLIST structure is used by the **AIOGetDriverList** function.

The *returnLength* field is returned as the length filled by the **AIOGetDriverList** function (always less than or equal to the original length). The *returnLength* field must be initialized before calling the **AIOGetDriverList** function.

AIODRIVERLISTENTRY

Contains information about a driver
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   int    hardwareType;
   int    ports;
   char   name[128];
} AIODRIVERLISTENTRY;
```

Fields

hardwareType

Specifies the hardware type of the Asynchronous I/O driver.

ports

Specifies the number of ports accessed.

name

Specifies the name of the driver (from 1 to 127 characters long and NULL-terminated).

Remarks

The AIODRIVERLISTENTRY structure is used by the **AIOGetDriverList** function.

AIODVRCAPABILITIES

Contains information about the capabilities of a driver
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   byteData[2];
} AIODVRCAPABILITIES;
```

Fields

returnLength

Specifies the byte length of the supplied return buffer.

byteData

Specifies the user-defined data.

Remarks

The AIODVRCAPABILITIES structure is used by the **AIOGetFirstPortInfo**, the **AIOGetNextPortInfo**, and the **AIOGetPortCapability** functions.

The *returnLength* field is returned as the length filled by the function, which is always less than or equal to the original length.

The *returnLength* field is the only structure field Asynchronous I/O defines.

AIODVRCONFIG

Contains driver configuration information
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
      WORD   returnLength;
      BYTE   byteData[2]
} AIODVRCONFIG;
```

Fields

returnLength

Specifies the byte length of the supplied return buffer.

byteData

Specifies the user-defined data.

Remarks

The AIODVRCONFIG structure is used by the **AIOGetPortConfiguration** function.

The *returnLength* field is returned as the length filled by the function, which is always less than or equal to the original length.

The *returnLength* field is the only structure field Asynchronous I/O defines.

AIODVRSTATISTICS

Contains driver statistical information
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   byteData[2];
} AIODVRSTATISTICS;
```

Fields

returnLength

Specifies the byte length of the supplied return buffer.

byteData

Specifies the user-definrd data.

Remarks

The AIODVRSTATISTICS structure is used by the **AIOGetStatistics** function.

The *returnLength* field is returned as the length filled by the function, which is always less than or equal to the original length.

AIOPORTCAPABILITIES

Contains information about the capabilities of a port
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   majorVersion;
   BYTE   minorVersion;
   LONG   notSupportedMask;
   BYTE   minBitRate;
   BYTE   maxBitRate;
   BYTE   minDataBits;
   BYTE   maxDataBits;
   BYTE   minStopBits;
   BYTE   maxStopBits;
   BYTE   minParityMode;
   BYTE   maxParityMode;
   BYTE   minFlowCtrlMode;
   BYTE   maxFlowCtrlMode;
   LONG   miscCapabilities;
   LONG   minReadBufferSize;
   LONG   maxReadBufferSize;
   LONG   minWriteBufferSize;
   LONG   maxWriteBufferSize;
   WORD   minDeadmanTime;
   WORD   maxDeadmanTime;
} AIOPORTCAPABILITIES;
```

Fields

returnLength

Specifies the byte length of the supplied structure.

majorVersion

Specifies the major version number of the AIOPORTCAPABILITIES structure (number 1).

minorVersion

Specifies the minor version number of the AIOPORTCAPABILITIES structure (number 0).

notSupportedMask

Specifies a bit mask (beginning with the most significant bit representing the *minBitRate* field) indicating which fields are currently supported.

minBitRate

Specifies the minimum index for the supported bit rates.

maxBitRate

Specifies the maximum index for the supported bit rates.

minDataBits

Specifies the minimum index for the supported data bits per character.

maxDataBits

Specifies the maximum index for the supported data bits per character.

minStopBits

Specifies the minimum index for the supported stop bits per character.

maxStopBits

Specifies the maximum index for the supported stop bits per character.

minParityMode

Specifies the minimum index for the supported parity mode generation.

maxParityMode

Specifies the maximum index for the supported parity mode generation.

minFlowCtrlMode

Specifies the minimum index for the supported flow control modes.

maxFlowCtrlMode

Specifies the maximum index for the supported flow control modes.

minReadBufferSize

Specifies the minimum receive buffer size.

maxReadBufferSize

Specifies the maximum receive buffer size.

minWriteBufferSize

Specifies the minimum transmit buffer size.

maxWriteBufferSize

Specifies the maximum transmit buffer size.

minDeadmanTime

Specifies the minimum value (in seconds) for the deadman timer interval.

maxDeadmanTime

Specifies the maximum value (in seconds) for the deadman timer interval.

Remarks

The AIOPORTCAPABILITIES structure is used by the **AIOGetFirstPortInfo**, the **AIOGetNextPortInfo**, and the **AIOGetPortCapability** functions.

The *returnLength* field is returned as the length filled, which is always less than or equal to the original length.

The index values for the *minBitRate* and *maxBitRate* fields follow:

AIO_BAUD_50
AIO_BAUD_75
AIO_BAUD_110
AIO_BAUD_150
AIO_BAUD_300
AIO_BAUD_600
AIO_BAUD_1200
AIO_BAUD_1345
AIO_BAUD_1800
AIO_BAUD_2000
AIO_BAUD_2400
AIO_BAUD_3600
AIO_BAUD_4800
AIO_BAUD_7200
AIO_BAUD_9600
AIO_BAUD_19200
AIO_BAUD_38400
AIO_BAUD_57600
AIO_BAUD_115200

The index values for the *minDataBits* and *maxDataBits* fields are defined as follows:

AIO_DATA_BITS_5
AIO_DATA_BITS_6
AIO_DATA_BITS_7
AIO_DATA_BITS_8

The index values for the *minStopBits* and *maxStopBits* fields are defined as follows:

AIO_STOP_BITS_1
AIO_STOP_BITS_1p5
AIO_STOP_BITS_2

The index values for the *minParityMode* and *maxParityMode* fields are defined as follows:

Parity Index	Parity Generated
AIO_PARITY_NONE	None
AIO_PARITY_ODD	Odd parity
AIO_PARITY_EVEN	Even parity
AIO_PARITY_MARK	Mark parity
AIO_PARITY_SPACE	Space parity

The *minFlowCtrlMode* and *maxFlowCtrlMode* fields can contain one or more bit masks (which can be combined using the bitwise inclusive OR operator) and are defined as follows:

AIO_SOFTWARE_FLOW_CONTROL_OF
AIO_SOFTWARE_FLOW_CONTROL_ON
AIO_HARDWARE_FLOW_CONTROL_OFF
AIO_HARDWARE_FLOW_CONTROL_ON

If the *minReadBufferSize* and *maxReadBufferSize* fields are equal, **AIOSetReadBufferSize** is not supported.

If the *minWriteBufferSize* and *maxWriteBufferSize* fields are equal, **AIOSetWriteBufferSize** is not supported.

Values for the *miscCapabilities* field follow:

0x00000002 AIO_CAP_OUTPUT_BREAK
0x00000004 AIO_CAP_FLOWCTRLCHARS
0x00000008 AIO_CAP_PROGRAMMABLE
0x00000010 AIO_CAP_INPUT
0x00000020 AIO_CAP_OUTPUT

AIOPORTCONFIG

Contains information about port configuration
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   majorVersion;
   BYTE   minorVersion;
   LONG   notSupportedMask;
   int    hardwareType;
   int    boardNumber;
   int    portNumber;
   BYTE   bitRate;
   BYTE   dataBits;
   BYTE   stopBits;
   BYTE   parityMode;
   BYTE   flowCtrlMode;
   BYTE   breakMode;
   LONG   readSize;
   LONG   writeSize;
   BYTE   transmitXon;
   BYTE   transmitXoff;
   BYTE   receiveXon;
   BYTE   receiveXoff;
   WORD   externalControl;
} AIOPORTCONFIG;
```

Fields

returnLength

Specifies the byte length of the supplied structure.

majorVersion

Specifies the major version number of the AIOPORTCONFIG structure (number 1).

minorVersion

Specifies the minor version number of the AIOPORTCONFIG structure (number 0).

notSupportedMask

Specifies a bit mask (beginning with the most significant bit representing the *bitRate* field) that indicates which fields are currently supported.

hardwareType

Contains the hardware type value that identifies the ports attached to one unique type of hardware driver.

boardNumber

Contains the number value (zero-based) that indicates which board of the selected hardware type is connected to the port.

portNumber

Contains the number value (zero-based) that indicates a particular port on the board.

bitRate

This field is set by the **AIOConfigurePort** function and can be modified by the Asynchronous I/O driver. See AIOPORTCAPABILITIES for more information on this field.

dataBits

This field is set by the **AIOConfigurePort** function and can be modified by the Asynchronous I/O driver. See AIOPORTCAPABILITIES for more information on this field.

stopBits

This field is set by the **AIOConfigurePort** function and can be modified by the Asynchronous I/O driver. See AIOPORTCAPABILITIES for more information on this field.

parityMode

This field is set by the **AIOConfigurePort** function and can be modified by the Asynchronous I/O driver. See AIOPORTCAPABILITIES for more information on this field.

flowCtrlMode

This field is set by the **AIOConfigurePort** function and can be modified by the Asynchronous I/O driver. See AIOPORTCAPABILITIES for more information on this field.

breakMode

Set by **AIOSetExternalControl**. There are two possible values for this field, as follows:

AIO_BREAK_MODE_ON

AIO_BREAK_MODE_OFF

readSize

Set by **AIOSetReadBufferSize**. It is a hexadecimal value reflecting the size in number of bytes.

writeSize

Set by **AIOSetWriteBufferSize**. It is a hexadecimal value reflecting the size in number of bytes.

transmitXon

Contains a hexadecimal value set by **AIOSetFlowControlCharacters**.

The default value is ASCII XON (0x11).

transmitXoff

Contains a hexadecimal value set by **AIOSetFlowControlCharacters**. The default value is ASCII XOFF (0x13).

receiveXon

Contains a hexadecimal value set by **AIOSetFlowControlCharacters**. The default value is ASCII XON (0x11).

receiveXoff

Contains a hexadecimal value set by **AIOSetFlowControlCharacters**. The default value is ASCII XOFF (0x13).

externalControl

Set by **AIOSetExternalControl** using the AIO_EXTERNAL_CONTROL request. This field can contain one or more bit masks which indicate the external control signals to turn on or off. These bit masks can be combined using the bitwise inclusive OR operator (|). The following values are defined for use with the AIO_EXTERNAL_CONTROL request:

AIO_EXTCTRL_DTR_DISABLE
AIO_EXTCTRL_DTR_ENABLE
AIO_EXTCTRL_RTS_DISABLE
AIO_EXTCTRL_RTS_ENABLE

Remarks

This structure is used by **AIOGetPortConfiguration**.

The *returnLength* function is returned as the length filled by the function, which is always less than or equal to the original length.

If the*notSupportedMask* bit is zero, the field is supported.

If the *notSupportedMask* bit is 1, the field is not supported. Currently, this mask is 0x0000FFFF.

AIOPORTINFO

Contains information about a port
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   majorVersion;
   BYTE   minorVersion;
   LONG   notSupportedMask;
   int    hardwareType;
   int    boardNumber;
   int    portNumber;
   WORD   availability;
   LONG   externalStatus;
   LONG   chgdExtStatus;
} AIOPORTINFO;
```

Fields

returnLength

Set to the byte length of the supplied structure. It is returned as the length filled by the function, which is always less than or equal to the original length. The *returnLength* field must be initialized prior to the call.

majorVersion

Contains the major version number of the AIOPORTINFO structure. Currently, this number is 1.

minorVersion

Contains the minor version number of the AIOPORTINFO structure. Currently, this number is 0.

notSupportedMask

Contains a bit mask (beginning with the most significant bit representing hardwareType) indicating which fields are currently supported. If the bit is zero, the field is supported. If the bit is 1, the field is not supported. Currently, this mask is 0x03FFFFFF.

hardwareType

Contains the hardware type value that identifies the ports attached to one particular type of hardware board (device).

boardNumber

Contains a number value (zero-based) that indicates which board of

the hardware type is connected to the port.

portNumber

Contains a number value (zero-based) that indicates a particular port on the board.

availability

Contains a status value that indicates port availability. The following status values are defined:

AIO_AVAILABLE_FOR_ACQUIRE
AIO_ALREADY_ACQUIRED
AIO_UNAVAILABLE

externalStatus

Indicates the current external status value bit mask for the selected port (see Remarks).

chgdExtStatus

Contains the changed status values bit mask (see Remarks).

Remarks

This structure is used by **AIOGetFirstPortInfo** and **AIOGetNextPortInfo** .

The values for the bit mask used by *externalStatus* and *chgdExtStatus* are listed in the following table:

Table auto. Port Status Values

Name	Meaning
AIO_EXTSTA_RI	Ring Indicator signal
AIO_EXTSTA_DCD	Data Carrier Detect signal
AIO_EXTSTA_DSR	Data Set Ready signal
AIO_EXTSTA_CTS	Clear To Send signal
AIO_EXTSTA_BREAK	Break signal

When the external status values are requested, the changed status mask is reset to zero. This allows an application to determine which signals have changed state since the last time the external status values were requested.

AIOPORTSEARCH

Contains port search information used by **AIOGetFirstPortInfo** and **AIOGetNextPortInfo**

Service: Asynchronous I/O

Defined In: aio.h

Structure

```
typedef struct {
   int   typeMask;
   int   boardMask;
   int   portMask;
   int   reserved[6];
} AIOPORTSEARCH;
```

Fields

typeMask

boardMask

portMask

reserved

AIOPORTSTATISTICS

Contains information about ports
Service: Asynchronous I/O
Defined In: aio.h

Structure

```
typedef struct {
   WORD   returnLength;
   BYTE   majorVersion;
   BYTE   minorVersion;
   LONG   notSupportedMask;
   LONG   receiveBytes;
   LONG   transmitBytes;
   LONG   parityErrors;
   LONG   framingErrors;
   LONG   overrunSoftware;
   LONG   overrunHardware;
} AIOPORTSTATISTICS;
```

Fields

returnLength

Contains the byte length of the supplied structure. It is returned as the length filled by the function, which is always less than or equal to the original length.

majorVersion

Contains the major version number of the AIOPORTSTATISTICS structure. Currently, this number is 1.

minorVersion

Contains the minor version number of the AIOPORTSTATISTICS structure. Currently, this number is 0.

notSupportedMask

Contains a bit mask (beginning with the most significant bit representing *receiveBytes*) that indicates which fields are currently supported. If the bit is zero, the field is supported. If the bit is 1, the field is not supported. Currently, this mask is 0x03FFFFFF.

receiveBytes

Contains the count of all data characters received through the selected port since the port was first registered.

transmitBytes

Contains the count of all data characters transmitted from the selected port since the port was first registered.

parityErrors

Contains a count of the number of times a character was received with a parity error. The character received is placed in the receive buffer.

framingErrors

Contains a count of the number of times a character was received with a framing error. The character received is placed in the receive buffer.

overrunSoftware

Contains a count that reflects only the number of occurrences of a software overrun error. The counts do not accurately show how many receive characters might have been lost due to overrun conditions, but rather how many times such a condition occurred. Since several characters might have been lost during each overrun condition, applications cannot derive the count of lost characters from these fields.

overrunHardware

Contains a count that reflects only the number of occurrences of a hardware overrun error. The counts do not accurately show how many receive characters might have been lost due to overrun conditions, but rather how many times such a condition occurred. Since several characters might have been lost during each overrun condition, applications cannot derive the count of lost characters from these fields.

Remarks

This structure is used by **AIOGetStatistics**.

AIOCOMX Communication Driver

This section describes the use of the AIOMCOMX.NLM, an asynchronous I/O driver for the NetWare server's COM ports.

Parent Topic: NetWare SDK Getting Started

AIOCOMX General Information

The NetWare® SDK includes AIOCOMX.NLM, an asynchronous I/O driver for the NetWare server's COM ports. AIOCOMX can service up to four COM ports (COM1, COM2, COM3, and COM4).

After loading AIOCOMX, you can use the AIO functions (provided by AIO.NLM) to send and receive data through the COM ports. You can also use the AIO functions to tell AIOCOMX to modify the configurations of the COM ports, such as transfer rate, parity, and data size.

An overview of using AIO is provided in Asynchronous I/O: Concepts. The AIO functions are described in Asynchronous I/O: Functions.

Parent Topic: AIOCOMX Communication Driver

AIOCOMX Load Syntax

To view the options that can be set when loading AIOCOMX, issue the following command:

```
LOAD AIOCOMX ?
```

This command displays a help screen but does not actually load the driver.

LOAD syntax for the AIOCOMX driver is as follows:

```
LOAD AIOCOMX [PORT=W] [INT=X] [NAME=Y] [NODE=Z] [FORCE]
[NOFIFO] [RXT=A] [TXQ=B]
```

The parameters are optional.

PORT specifies the I/O port address (in hexadecimal) of the COM port you want to install. It specifies the start of the I/O port address only; the COM port uses addresses from W to W+7 for its operations.

INT specifies the interrupt number (2 through 15). Each COM port requires a nonshared interrupt, regardless of the PC bus (ISA, EISA, MCA) on which the hardware resides.

In the case of MCA, a single non-shared interrupt is used for both transmit and receive. For example, rx int = 4 and tx int = 4.

NAME specifies the name to call the board. If the name is omitted, the default name is based upon the hardware configuration of the port as shown in Table 2. This name is used unless it is in use by another of the driver's registered boards. Uncommon hardware configurations produce a NULL string name.

NODE specifies a number to give the board. If the node number is omitted, the default board number is used in the order shown in the table entitled Default Settings for AIOCOMX, below. This default number is used unless it is in use by another of the driver's registered boards.

If you change your COM port settings from the default values listed to those in the table entitled Default Settings for AIOCOMX, a default board number is calculated based on the port address and interrupt number of the COM port. You can override this default board number with the NODE parameter.

FORCE tells AIOCOMX not to verify that the hardware address actually exists. (For example, you can force it to accept an address for COM3, even if the server doesn't have a COM3 port.) This can lead to very undesirable consequences if a non-UART type device resides at the specified port I/O address.

If AIOCOMX fails to load because device verification fails to confirm that a 8250/16450/16550 device is physically present, using the FORCE

keyword can be dangerous because it instructs AIOCOMX to ignore what might be a valid reason for aborting the load.

The intent of the FORCE keyword is to accommodate a potential device that does not meet the UART identification tests. This permits those UARTs to be used if they are present.

NOFIFO specifies to override from the FIFO 16550 mode to the 8250/16450 non-FIFO mode (described in UARTs).

RXT specifies a 16550 receiver trigger (described in UARTs). This trigger can be 1, 4, 8, or 14. The default setting is 4.

TXQ specifies a 16550 transmit queue (described in UARTs). This queue can be 1, 4, 8, 12, or 16. The default setting is 16.

NOTE: If you do not load AIO.NLM before you load AIOCOMX, the loader will automatically load AIO.NLM while loading AIOCOMX.

Parent Topic: AIOCOMX Communication Driver

Loading AIOCOMX Multiple Times

You must load the AIOCOMX driver once for each COM port it will service. AIOCOMX will give each COM port a unique board number. If your server's hardware is configured as shown in the table entitled Default Settings for AIOCOMX, the first port that AIOCOMX installs is port 0 on board 0, the second is port 0 on board 1, and so on.

The AIOCOMX driver is reentrant, so although you load once for each COM port it services, the NetWare OS maintains a single driver image in memory.

Parent Topic: AIOCOMX Communication Driver

AIOCOMX Default Board Settings

If you do not supply parameters when installing AIOCOMX, it attempts to automatically install the server's COM ports with default values that are listed in the following table.

Table auto. Default Settings for AIOCOMX

Load	I/O Port Address	Int	Default Name	Default Board #
1	0x3F8	4	COM1	0
2	0x2F8	3	COM2	1
3	0x3E8	4	COM3	2

4	0x2E8	3	COM4	3

Using the default values, the COM ports will be installed in the following manner if the server's hardware matches the settings in the table entitled Default Settings for AIOCOMX:

On the first load, port address = 0x3F8, int = 4, name = "COM1" and board # = 0.

On the second load, port address = 0x2F8, int = 3, name = "COM2" and board # = 1.

On the third load, port address = 0x3E8, int = 4, name = "COM3" and board # = 2, but since interrupt 4 is already in use, AIOCOMX prompts the user for an interrupt number.

On the fourth load, port address = 0x2E8, int = 3, name = "COM4" and board # = 3, but since interrupt 3 is already in use, AIOCOMX prompts the user for a new interrupt number.

In all cases, user-provided values override the default values.

Parent Topic: AIOCOMX Communication Driver

Related Topics:

Loading AIOCOMX Multiple Times

AIOCOMX Default Internal Configuration

When AIOCOMX loads, it sets its internal configuration information to the default values shown in the table below. These values are the same for each COM port that AIOCOMX installs.

Table auto. AIOCOMX Configuration Fields

Field	Value
Bit rate (bps)	2400
Data bits	8
Stop bits	1
Parity mode	off
Flow control mode	none
Break mode	off
Read buffer size	1024 bytes

Write buffer size	1024 bytes
Transmit XON	0x11
Transmit XOFF	0x13
Receive XON	0x11
Receive XOFF	0x13

Parent Topic: AIOCOMX Communication Driver

Changing the AIOCOMX Default Configuration

The AIO functions provide an interface that you can use to change the configuration of the AIOCOMX driver. For example, **AIOConfigurePort** allows you to change the bit rate, data bits, stop bits, parity mode, and flow control-mode settings. See Asynchronous I/O: Functions for a listing of functions that can be used to configure AIO drivers.

Parent Topic: AIOCOMX Communication Driver

UARTs

The Universal Asynchronous Receiver/Transmitter (UART) is responsible for receiving data as bytes and sending it out of the COM ports one bit at a time. It also receives data one bit at a time and turns the bits into bytes of data.

Three types of UARTs (8250, 16450, and 16550) are used with the COM ports of PCs. The speed at which data can be processed through the port depends upon the UART that is used.

The 8250 and 16450 UARTs cannot hold more than one byte of data. As soon as a byte is received, it must be removed from the register before another byte arrives, or the incoming data will overwrite the previous byte. If interrupts have been enabled for these UARTs, they will generate an interrupt each time they receive a byte and each time they send a byte.

The 16550 UART, on the other hand, has a 16-byte transmit buffer and a 16-byte receive buffer. These buffers are processed in a FIFO manner. An advantage to the 16550 UART is that you can program when it will send an interrupt.

For example, when AIOCOMX detects a 16550 UART, it sets the receiver

trigger to 4, which means the UART will send an interrupt when it has received 4 bytes. In the meantime, the receive buffer can hold an additional 12 bytes of data, which can be received before the interrupt is processed.

The default interrupt setting for the transmit queue is 16.

The 16550 UART also has a 16-byte transmit queue that buffers data until it holds a specified number of bytes. AIOCOMX sets this number to 16 by default.

If you do not want to use the 16-byte buffers, you can set the receiver trigger and transmit queue to 1. Doing this will force the 16550 UART to duplicate the behavior of the 8250 and 16450 UARTs.

> **NOTE:** It is only possible to configure the FIFO parameters for AIOCOMX at load time. See the RXT and TXQ parameters in AIOCOMX Load Syntax.

Parent Topic: AIOCOMX Communication Driver

Recommended AIOCOMX Bit Rates

When AIOCOMX is loaded, the rated transfer rate should be set to 2400 or 19200 bps, depending on the type of the server's UARTs.

If the server is equipped with 8250 or 16450 UARTs, the COM ports do not have any buffering, and the character in the chip's buffer must be removed before another character arrives. Therefore, an interrupt occurs when each character arrives, and the server's CPU must process the code that retrieves the data. In this case, 2400 is the recommended maximum setting.

If the server is equipped with a 16550 UART, the COM ports can buffer up to 16 characters in a FIFO queue, as mentioned in the previous section. This increases the speed at which the data can be processed, without losing any data. Therefore, the maximum setting is 19200.

In either case, the usable transfer rate also depends upon the remote device you are communicating with. If your server can use the 19200 transfer rate and the remote device can only use the 2400 transfer rate, you will need to limit your server's transfer rate to 2400 also.

By default, AIOCOMX sets the transfer rate to 2400, no matter what type of UART is used.

Parent Topic: AIOCOMX Communication Driver

The AIOCOMX Read and Write Buffers

AIOCOMX creates a read buffer and a write buffer for each of the COM

ports it services. As AIOCOMX driver receives data through the COM port, it places the data in the buffer associated with that port. When AIOCOMX detects data in a port's write buffer, it removes the data from the buffer and sends the data out the COM port.

Your interface to these buffers is through **AIOReadData** and **AIOWriteData** . **AIOReadData** removes data from the read buffer, and **AIOWriteData** places data in the write buffer.

AIOCOMX has a default buffer size of 1,024 bytes for the read buffer and 1,024 bytes for its write buffer. Your NLM applications can change the size of these buffers with **AIOSetReadBufferSize** and **AIOSetWriteBufferSize**. The upper limit for each of these buffers is 64 KB, but keep in mind that a large buffer is an unnecessary drain from system memory.

Parent Topic: AIOCOMX Communication Driver

Related Topics:

Considerations for AIOCOMX Read and Write Buffer Sizes

Considerations for AIOCOMX Read and Write Buffer Sizes

Although AIOCOMX allows you to resize its read and write buffers, buffer resizing is not a requirement for AIO drivers to be certified. The requirement is that all drivers must support 1,024-byte buffers. If your application is going to interface with drivers other than AIOCOMX, you must ensure that your application works with 1,024 byte buffers.

The size of the driver's read and write buffers determines how your program sends and receives its data. For example, one of the parameters for **AIOWriteData** is how many bytes to write into the buffer. If the available space in the write buffer is less than the amount specified to send, **AIOWriteData** does not write any data to the buffer because the AIOCOMX driver is designed to take all or nothing.

If the AIOCOMX's default buffer size is used, more than 1,024 bytes cannot be sent at a time; therefore, if something large is sent such as a PC's screen (without attributes) that is 2,000 characters, split the screen into two fragments containing 1,000 bytes each. (If the 2,000-byte buffer is sent at one time, **AIOWriteData** will always fail, since 2,000 is always larger than 1,024.)

Another factor to consider is that since AIO is asynchronous communication, the data you place in the write buffer might not all be sent by the time you are ready to place more data in the buffer. For example, if you break your screen fragments into 1,000 bytes, you must call **AIOWriteData** twice to send the screen to a remote machine. If you place the first fragment into the write buffer with **AIOWriteData**, and then

the first fragment into the write buffer with **AIOWriteData**, and then immediately use **AIOWriteData** with the next fragment, the second call will fail if more than 24 bytes of the first fragment remain in the write buffer (have not been sent by AIOCOMX). In this case, you might loop until the space is available and then send the second fragment.

Parent Topic: The AIOCOMX Read and Write Buffers

Unloading AIOCOMX

To remove AIOCOMX from memory, issue the following command:

```
UNLOAD AIOCOMX
```

While you must enter multiple LOAD commands for each COM port you want to use with the AIOCOMX driver, a single UNLOAD command removes the entire driver from memory.

Parent Topic: AIOCOMX Communication Driver

BSD Socket

Communication Service Group

BSD Socket: Guides

BSD Socket: General Guide

Porting BSD Applications to the NetWare OS

Socket Domains and Protocols

Socket Handles

Concurrent Access to Sockets

Error Number and Error Handling

Limitations of the select Function

Asynchronous I/O in BSD

Out-of-Band Data in BSD

Socket Options

The ioctl Function

BSD Interface Options

Internet Library Functions

Additional Links

BSD Socket: Functions

BSD Socket: Structures

BSD Socket: Concepts

Accepting a BSD Connection

Remote clients can connect to the server after the server issues a successful **listen** call. The **accept** function returns the new socket from the backlog queue of the parent socket (the socket issuing the **accept** function). The **accept** function has the following form:

```
s2 = accept(s, addr, addrlen)
```

The **accept** function from the server creates a new socket handle with the same properties as *s*, the parent socket, and assigns a new socket file handle *s2*. This handle must be used for all I/O on this particular connection. A value other than -1 indicates acceptance of the call and represents the new socket handle. A value of -1 indicates failure on the parent socket.

This file handle was first created with the **socket** function. It was bound to an address with **bind** and prepared to accept incoming connections with **listen**.

The server's listening socket continues to queue connection requests from remote clients until the queue backlog is reached or the socket is closed; therefore, multiple **accept** calls can be performed on one listening socket. Each call to **accept** removes a socket from the queue and allows it to admit new connection requests into the queue.

The addr argument points to a sockaddr_in or sockaddr_ipx structure that receives the Internet address and port used by the client program upon successful completion of **accept**.

The *addrlen* argument is a pointer to a variable that initially contains the length of *addr* in bytes. After an **accept** function concludes, it contains the length of the resulting Internet address.

In the TCSKSERV.C program, ds is the socket handle created by **accept**, and *s* is a socket created by **socket**. In this program, client_sockaddr points to the sockaddr_in structure that contains the Internet address and port used by the sample client program TCSKCLNT.C upon successful completion of **accept**.

The input length of *addr* is pointed to by *addrlen*. On return from **accept**, it contains the length of the resulting Internet address.

Parent Topic: Socket Functions from the Server Program

Asynchronous I/O in BSD

BSD sockets provide a style of I/O that signals a process whenever a socket is ready for an I/O operation. This facility is not provided in NetWare because it cannot be emulated precisely. As with **select**, the purpose of this feature can be accomplished more naturally by creating a thread to wait on each I/O stream that can block.

Parent Topic: Porting BSD Applications to the NetWare OS

Binding to a Network Address

The **bind** function on a socket file handle associates a network address with a socket. The function has the following form:

```
status = bind (s, addr, addrlen)
```

The function returns a status code status. A value of 0 indicates that the binding was successful. A value of -1 indicates that the binding was unsuccessful. When a function fails, the variable errno contains a specific error code.

The socket file handle is *s*. This is the value previously returned by **socket**.

For TCP/IP and UDP/IP sockets, addr points to the sockaddr_in structure containing a network address in the PF_INET domain. If the structure specifies an IP address or port number that is zero, the BSD Socket API assigns the local host IP address or port number as required. Either (or both) can be specified as zero.

For TCP/IPX sockets, addr points to the sockaddr_ipx structure containing a network address in the PF_IPX domain for TCP sockets. If the structure specifies an IPX address or port number that is zero, the BSD Socket API assigns the local host IPX address or port number as required. Either (or both) can be specified as zero.

Typically, a client program uses zeros for both fields. This ensures that the client program is unique for each session of the network application running with the same server. Normally, a server specifies only the port.

The addrlen argument is the length in bytes of the sockaddr_in structure pointed to by *addr*.

In the TCSKCLNT.C program, *s* is the socket file handle whose value was returned by the **socket** function.

Parent Topic: Socket Functions from the Client Program

BSD Error Handling

Socket functions return a value of -1 if an operation fails. The variable errno, which indicates the specific cause of the failure, is actually a macro that refers to a per-thread error value. Programs can reference errno only on the thread that encountered the error.

The include file ERRNO.H contains a listing of error codes and the symbolic names associated with them.

A program can print the error message text using **perror** in the following format:

```
perror (string);
```

The message prints the error in the following form:

```
string : message
```

The string variable is an ASCII string specified in **perror** identifying the source of the error. Usually, *string* is the address of a command or library function. The message variable is the message text associated with the error code.

When the TCPIP.NLM is being unloaded, further socket operations on existing sockets are in the nonblocking mode and fail. They return an ENETDOWN error. When this happens, the socket clients should exit as soon as possible. **In particular, a socket client should never loop to perform socket operations without checking for this error return. A failure to check for the error return causes the execution thread to loop infinitely without relinquishing control**.

A socket client should abort whenever an unknown error is encountered. This will help detect compatibility problems if error codes are changed or added in future releases.

Blocking and Nonblocking I/O Modes

By default, a BSD Socket function blocks further execution of the program until the transport system completes the requested operation. **Some functions have the potential to block indefinitely; for example, read blocks until network data becomes available for the socket**.

If the nonblocking I/O mode is established, the BSD Socket API determines whether the requested function can be completed without delay; if so, it performs the operation and returns successfully. If BSD determines a delay would occur, an EWOULDBLOCK error is returned in errno, indicating the transport is currently not ready to complete the operation.

The nonblocking mode is enabled on a per-socket basis using **ioctl** with the FIONBIO command.

Parent Topic: BSD Socket Overview

BSD Interface Options

BSD provides a number of socket **ioctl** operations that work on datalink interface devices instead of the socket itself. NetWare sockets do not provide this access. Interfaces are manipulated by loadtime arguments, bind arguments, and the SNMP agent.

Parent Topic: Porting BSD Applications to the NetWare OS

BSD Socket Overview

This chapter explains how network applications use the BSD Socket API to communicate between a client and a server. It also describes the Internet Protocol (IP) family (as defined by the BSD Socket API) and discusses the use of the BSD Socket API for TCP/IP and TCP/IPX in NetWare® 4.x. If you are already familiar with BSD socket programming, you might want to skip this chapter and proceed to BSD Socket: Functions for detailed information on the BSD Socket functions.

> **CAUTION: The BSD Socket API emulates closely the 4.3BSD Socket Programming Interface. There are, however, important semantic differences between these two interfaces that may affect an application porting from BSD UNIX. Porting BSD Applications to the NetWare OS describes these differences in detail.**

Many network applications involve a client program running on a user's workstation; the client program communicates with a server program running on a shared server system, such as the NetWare operating system. The server program runs continuously, while the user invokes the client program only when it needs the server program's services.

Generally, a transport communication protocol operates separately from client and server programs and is closely tied to the operating system. An application program uses the transport protocol by calling its API. A transport protocol can support more than one API (even in the same system). Multiple applications can use the transport protocol concurrently.

Related Topics

BSD Socket Overview

Using BSD Socket Functions

Socket Types

Socket Functions from the Client Program

Socket Functions from the Server Program

Protocol Families and BSD

Internet Protocol Family

BSD Error Handling

Closing a BSD Connection

Depending on the design of your application, either the client or the server may close the connection first. For example, when the client closes, the server detects it as a write operation that returns a value of zero. The server should then close the socket.

Parent Topic: Socket Functions from the Server Program

Closing a Socket

When the communication ends, the application must close the socket. The **close** function has the following form:

```
close(s)
```

where *s* is the socket being closed.

When a program ends, all associated sockets are closed. If a program needs to reuse a socket before the program ends, it must first close the socket and then reopen it.

Parent Topic: Socket Functions from the Client Program

Concurrent Access to Sockets

Just as multiple processes can share a socket in BSD UNIX, multiple threads can share a socket in NetWare. As in BSD, if they read or write concurrently, they make a hash of the I/O stream. However, it is quite practical for one thread to read a socket and for another thread to write the same socket.

An important distinction from UNIX I/O treatment is that the first close request on a socket actually closes the socket, because I/O handles are not bound to threads. When an NLM™ application is terminated, the NLM C Library closes all sockets it created. It is not possible to pass a socket from one NLM to another. Therefore, UNIX code that forks a child process should be modified so that it does not rely on duplication of the socket.

If one thread closes a socket while another thread is waiting for an operation to complete on that socket, the latter operation is aborted with an error. All

pending operations on a socket are aborted before the completion of the close call.

If a thread attempts to access a socket that has been closed, it receives the EBADF error. However, you should not rely on this behavior. Instead, avoid methods that would lead to attempted operations on sockets that have been closed (using **close** as opposed to being closed by the peer socket).

Parent Topic: Porting BSD Applications to the NetWare OS

Creating a Socket

When the client program initiates a socket request, it creates a socket on the local host. Socket requests are made with **socket**. This function performs two tasks:

It creates and initializes socket resources in the TCP/IP or TCP/IPX NLM™ application.

It opens and associates a NetWare file handle with the allocated socket resources.

The association of internal socket resources with NetWare file handles grants sockets the same operating system services given to other devices, including the automatic closure of a device when a program terminates abnormally.

The socket function has the following form:

```
s = socket (domain, type, protocol)
```

The function returns the socket file handle s. A value other than -1 indicates that the socket has been successfully opened. A value of -1 indicates an error.

The domain is the protocol family. The protocol family must be PF_INET or PF_IPX.

The type specifies the type of socket (TCP or UDP) to be used. The currently defined values for type are either SOCK_STREAM or SOCK_DGRAM.

The protocol is the particular protocol to be used with this socket. Normally, only a single protocol exists to support a particular socket type within a given protocol family. Typically, the value of this parameter is set to zero.

When loading the sample client program TCSKCLNT.C, specify the IP address of the node where the server resides. Once the file handle s has been created and the socket function returns successfully, the program opens a connection to the server port 2000.

In the TCSKCLNT.C program, the domain is PF_INET, the type of socket is

SOCK_STREAM, and the value of the protocol is zero.

Parent Topic: Socket Functions from the Client Program

Error Number and Error Handling

When an OS function fails, BSD and other UNIX derivatives return an error code in the global variable errno. This is not feasible in NetWare because multiple threads can share the same data space, creating confusion as to which thread is associated with an errno value. Therefore, NetWare implements a per-thread errno that is accessed with the errno macro. For NetWare socket clients, the proper errno value is available only to the thread that instigated the error.

In the NetWare environment, the underlying socket protocol NLM (TCPIP.NLM or TCPIPX.NLM) can be unloaded while a socket application program is running. When this happens, further socket operations on existing sockets are in nonblocking mode and fail. They return an ENETDOWN error. The socket client should then exit as soon as possible. In particular, the socket client should never loop to perform socket operations without checking for error returns. A failure to check for the ENETDOWN error causes the execution thread to loop indefinitely without relinquishing control.

Parent Topic: Porting BSD Applications to the NetWare OS

Establishing a Connection with BSD Sockets

The **connect** function on a socket file handle forwards a request to the TCP/IP transport system for a connection to be established between the client and the server. This function has the following form:

```
status = connect (s, addr, addrlen)
```

When the program calls **connect**, the protocol module attempts to open the specified connection. When a connection is successfully established, a status value of 0 is returned. If a connection cannot be established, the function returns a value of -1 and sets errno to ETIMEDOUT or ECONNREFUSED.

The argument *s* is the socket file handle representing the socket on the local host. This is the value previously returned by **socket**.

For TCP/IP, the *addr* argument is a pointer to a sockaddr_in structure containing the server's network address in the PF_INET domain. The client program must specify the IP address of the host running the server and the port on which the server is listening. If the IP address is 0, the host defaults to the local address.

For TCP/IPX, the *addr* argument is a pointer to a sockaddr_ipx structure

For TCP/IPX, the *addr* argument is a pointer to a sockaddr_ipx structure containing the server's network address in the PF_IPX domain. The client program must specify the IPX address of the host running the server and the port on which the server is listening. If the IPX address is 0, the host defaults to the local address.

The *addrlen* argument is the length in bytes of the sockaddr_in or sockaddr_ipx structure pointed to by *addr*.

Once a connection is established, the sample program TCSKCLNT.C sends 1,024 bytes of data and reads it back from the server before closing the connection.

Parent Topic: Socket Functions from the Client Program

Internet Library Functions

Support functions, such as **gethostbyname**, are used differently in NetWare. See Internet Network Library for more information.

Parent Topic: Porting BSD Applications to the NetWare OS

Internet Protocol Family

The Internet Protocol (IP) family includes the following protocols:

Internet Control Message Protocol (ICMP)

Transmission Control Protocol (TCP)

User Datagram Protocol (UDP)

Internet Protocol (IP)

Address Resolution Protocol (ARP)

TCP supports the STREAM socket type (SOCK_STREAM), while UDP supports the datagram socket type (SOCK_DGRAM). IP exists below both TCP and UDP. It provides end-to-end data delivery over a series of networks having (potentially) different media. ICMP and ARP are not directly accessible to the user; they are accessible only to the NetWare OS and are used internally between peer protocol stacks for management and control.

See the following topics for further information:

TCP and BSD

UDP and BSD

Parent Topic: BSD Socket Overview

Limitations of the select Function

The BSD function **select** works on a variety of file types, including pipes and ttys. In contrast, the NetWare function **select** works only on sockets. This should not affect typical server-oriented applications. If an application needs to wait simultaneously on sockets and nonsocket files, it should create separate threads for this purpose. This solution is more consistent with the design of NetWare, which (unlike UNIX) provides multiple threads in the same data space.

Socket handles can take any 32-bit value except 0 or -1. This defeats the BSD implementation of the fd_set structure, which is a bit array of socket index values. Instead, NetWare sockets define fd_set as an array of 32-bit file handles. If an application uses only macros (FD_ZERO, FD_SET, FD_ISSET, and FD_CLR) to manipulate the fd_set structure and does not work on more than FD_SETSIZE sockets, it is not affected. FD_SETSIZE is currently 16.

Parent Topic: Porting BSD Applications to the NetWare OS

Out-of-Band Data in BSD

BSD out-of-band (OOB) data delivery might rely on delivering a SIGURG signal to the client process. Again, signals cannot be precisely emulated, so this mechanism is not used in NetWare. To detect incoming OOB data, the client should use **select** and nonblocking I/O, which work the same as in BSD. Before reading from the socket, the client should deselect on it, using both read and except FD sets. If **select** returns with the except FD set for the socket, the client should proceed as in BSD, after receiving SIGURG. Otherwise, the methods for reading OOB data are the same as in BSD.

Parent Topic: Porting BSD Applications to the NetWare OS

Performing Network I/O through a Socket

When a connection is established, data is read and written (using **read** and **write**, respectively) as though a direct full-duplex data path existed between the two processes. Both reading and writing occur on the same logical data stream.

The **read** function has the following form:

```
cc = read(s, bufaddr, buflen)
```

The *s* argument is the socket file handle from which the data is read, *bufaddr* is the address of the buffer to receive the data, and *buflen* is the length of that buffer.

The **write** function has the following form:

```
cc = write(s, bufaddr, buflen)
```

The socket file handle *s* is the socket to which the data is written, *bufaddr* is the address of the buffer containing the data, and *buflen* is the length of the data.

Upon return, *cc* contains the number of characters actually read from or written to the logical data stream, or -1 if an error was detected. The value of *cc* may be less than what was requested. If the TCSKCLNT.C program does not return the full amount of characters actually read, the program repeats the read operation as needed.

Your application program is responsible for synchronizing its communication so that reading and writing do not cause a deadlock. The TCP/IP transport system guarantees reliable delivery of data. Thus, a simple command and response exchange is easy to create.

An error can occur during reading or writing if the remote system is inaccessible for reasons such as a cable failure, intermediate gateway crash, remote system malfunction, or remote program completion.

In addition to **read** and **write**, the BSD Socket API provides a number of other functions for performing network I/O, including these: **readv**, **recv**, **recvfrom**, **recvmsg**, **writev**, **send**, **sendto**, and **sendmsg**.

Parent Topic: Socket Functions from the Client Program

Porting BSD Applications to the NetWare OS

This appendix describes differences between the 4.3BSD Socket API and the NetWare® 3.x and 4.x socket programming interface, which emulates 4.3BSD closely. If you port an application from the BSD UNIX* environment, review your code fore methods and assumptions that would be affected.

See the following topics:

Socket Domains and Protocols

Socket Handles

Concurrent Access to Sockets

Error Number and Error Handling

Limitations of the select Function

Protocol Families and BSD

Every network protocol is associated with a specific protocol family. A **protocol family** normally consists of several protocols, one per socket type, but the protocol family is not required to support all socket types. Multiple protocols within the same protocol family can support the same socket abstraction.

To use a specific protocol, you request the appropriate protocol family and type when you create a socket. Each protocol normally accepts only one address format, usually determined by the addressing structure of the protocol family and network architecture. Certain semantics of the basic socket abstractions are protocol specific. Each protocol supports the basic model for its particular socket type and can also provide nonstandard facilities or extensions.

The BSD Socket API supports only the IP and IPX families, which has the following identifier:

```
#define PF_INET 2       /* internetwork: UDP, TCP, etc. */
#define PF_IPX 23       /* internetwork: TCP, etc. */
```

Parent Topic: BSD Socket Overview

Receiving a BSD Connection Request

The **listen** function on a socket file handle lets the socket receive incoming connections. The server must call this function before calling **accept**.

The **listen** function has the following form:

```
status = listen(s, backlog)
```

A status value of 0 indicates that the function is successful. A value of -1 indicates that the function is unsuccessful.

The socket file handle is s. This is the value previously returned by the

The socket file handle is s. This is the value previously returned by the socket function.

The backlog is an integer that specifies how many incoming connections can be in a queue before being processed by calls to **accept**. Each call to **accept** removes one connection from the backlog queue. The value of backlog must be between 0 and 5, with the actual queue limit being backlog+1.

In the server program TCSKSERV.C, the server listens at the port 2000 for incoming connection requests. When a request arrives, the server spins off a separate thread to accept the connection, and continues listening on the original thread for more requests.

The **listen** function in this program returns the status code *rc*. The socket file handle returned by **socket** is *s*. The backlog argument is 5.

Parent Topic: Socket Functions from the Server Program

Socket Domains and Protocols

NetWare sockets implement the PF_INET and PF_IPX domains, but do not provide PF_UNIX, and so on. The PF_INET domain offers SOCK_STREAM (IPPROTO_TCP) and SOCK_DGRAM (IPPROTO_UDP), and SOC_RAW (IPPROTO_ICMP) types. the PF_IPX domain offers only SOCK_STREAM (IPXPROTO_TCP).

Parent Topic: Porting BSD Applications to the NetWare OS

Socket Functions from the Client Program

Before you can send data from one host to another, you must create a socket on the local host. Once you create a socket, a client program typically performs the following BSD Socket functions:

bind (optional)

connect

read and **write** (for data transfer)

close

Subsequent sections discuss each of the above functions, as well as the socket function (which you use to create a socket). These discussions refer to the sample program TCSKCLNT.C. This sample client program applies to connection-oriented applications that use SOCK_STREAM sockets for a reliable, sequenced, two-way data transmission. Before running this client program, make sure that the server program TCSKSERV.C has already been loaded. Then, load the client program TCSKCLNT.C by specifying the IP

address (in dotted notation) of the node where the server resides.

Related Topics

Creating a Socket

Binding to a Network Address

Establishing a Connection with BSD Sockets

Performing Network I/O through a Socket

Closing a Socket

Parent Topic: BSD Socket Overview

Socket Functions from the Server Program

The server typically calls the following functions:

socket

bind

listen

accept

read and **write** (for data transfer)

close

This section illustrates each of these operations through a sample server program TCSKSERV.C. This sample program applies to connection-oriented applications using SOCK_STREAM sockets for a reliable, sequenced, two-way data transmission. It demonstrates how a typical server accepts a connection request while listening for more incoming requests at the same time.

A server program uses the functions **socket** and **bind** in the same manner as a client program. For instance, a socket request from the server creates a socket. However, with **bind**, the server specifies a well-known port number, one that the client uses with **connect**. Each server offering a specific service (for example, a file transfer server application) should be assigned a unique port where only servers of that type listen.

Related Topics

Receiving a BSD Connection Request

Accepting a BSD Connection

Closing a BSD Connection

Parent Topic: BSD Socket Overview

Socket Handles

The BSD function **socket** returns an integer that is an index into a per-process file table. The NetWare function **socket** returns an integer that is actually the address of a file control block. This distinction is not important to a client program unless the program is dependent on the range or value of a socket handle. All 32 bits are significant, and a valid handle can be a positive or negative value (other than -1).

Parent Topic: Porting BSD Applications to the NetWare OS

Socket Options

Most socket options in the BSD Socket API (for example, **getsockopt** and **setsockopt**) are faithfully emulated in NetWare. Exceptions are as follows:

SO_DEBUG---Has no effect. In BSD, it unleashes a torrent of perpacket TCP state information messages to the console.

SO_DONTROUTE---Has no effect. In BSD, it constrains outgoing packets to systems that can be reached without crossing a router.

SO_BROADCAST---Has no effect. All clients can send broadcast packets. In BSD, it enables a client to send broadcast packets.

Several other SO_XXX options in the BSD header file are not implemented in BSD TCP, as follows:

SO_SNDLOWAT
SO_RCVLOWAT
SO_SNDTIMEO
SO_RCVTIMEO

Parent Topic: Porting BSD Applications to the NetWare OS

Socket Types

All network programs use either Stream or datagram sockets:

Stream sockets provide sequenced, reliable, two-way, connection-based messages with an out-of-band mechanism (TCP urgent data). They are used with the TCP protocol.

Datagram sockets provide connectionless, unreliable messages of a

limited length for user datagrams. They are used with the UDP protocol.

Parent Topic: BSD Socket Overview

TCP and BSD

TCP provides reliable, two-way data transmission. It is a byte-stream protocol supporting STREAM (SOCK_STREAM) sockets. The socket address of each TCP socket is a unique identifier that is formed from both the host's IP address and its TCP port address.

Sockets using the TCP protocol are either active or passive. Active sockets initiate connections to passive sockets. By default, TCP sockets are created active; to create a passive socket, you must call **listen**. Only passive sockets can use **accept** to accept incoming connections. Only active sockets or passive sockets that have not received a connection request can use **connect** to initiate connections.

When specifying the local address in the **bind** parameter addr, give the IP address as INADDR_ANY (which is a value of 0). The protocol code then assigns the local address. You can still specify the TCP port in the *sin_port* field of the **bind** parameter addr. If this field is 0, the system assigns a port number.

Once a connection has been established, the connection is uniquely identified by four components:

IP address of the local host

Port number of the local socket

IP address of the remote host

Port number of the remote socket

To create a socket using TCP, use the following header files and structure declarations, and functions:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket_id , return_code;
struct sockaddr_in  my_socket = {AF_INET, 0,  0 };

socket_id  =  socket (PF_INET,  SOCK_STREAM, 0);
return_code =  bind (socket_id, &my_socket, sizeof(my_socket);
```

The TCP implementation supports a feature popularized by the BSD implementation known as **keep-alive packets**. Keep-alive packets check whether a peer TCP is still functioning. They are enabled by setting the **setsockopt** function's SO_KEEPALIVE option. Keep-alive packets

periodically poll the remote machine if the connection has been idle.

Parent Topic: Internet Protocol Family

The ioctl Function

NetWare TCP/IP does not support SIOCSPGRP and SIOCGPGRP. These are used in BSD to manipulate the process group for **select** and asynchronous I/O signals.

Parent Topic: Porting BSD Applications to the NetWare OS

UDP and BSD

UDP is a simple, unreliable datagram protocol that supports the SOCK_DGRAM abstraction for the IP family. UDP sockets are connectionless and normally use **send** and **recv**. You can use **connect** to set a destination for future packets, thereby allowing use of **read** and **write**.

UDP address formats are identical to those used for TCP. UDP provides a port identifier in addition to the normal Internet address for the host. UDP port space is distinct from TCP port space; therefore, a UDP port number can have the same value as a TCP port number. A UDP port cannot be connected to a TCP port.

To create a socket using UDP, you use the following header files, structure definitions, and functions:

```
#include <sys/socket.h> /* include file socket */
#include <netinet/in.h> /* include file */

int socket_id;
struct sockaddr_in my_socket = { AF_INET,  0, 0 };

socket _id  =    socket ( PF_INET, SOCK_DGRAM, 0 );
```

Parent Topic: Internet Protocol Family

Using BSD Socket Functions

The BSD Socket API is modeled after the UNIX* file system interface. Instead of opening a disk file, your application creates a socket and uses its I/O handle to read, write, and close a socket like a disk file. Because of the complex nature of network communications, your application must invoke several operations that are not required by disk files in the case of the UNIX

file system interface. Instead of a filename, each socket that an application creates has a unique network address. To send data, your application must supply the address of the destination socket.

The following summarizes how applications use the BSD Socket API to communicate with remote programs:

Sockets send data using the functions **write**, **writev**, **send**, **sendto**, and **sendmsg**.

Sockets receive data using the functions **read**, **readv**, **recv**, **recvfrom**, and **recvmsg**.

Only connected sockets can use the functions **write**, **writev**, **send**, **read**, **readv**, and **recv**.

Below is a list of the typical steps a network application program completes when using Stream sockets. Each step specifies the functions used to complete the step:

1. Create a socket and associate it with a specific protocol (TCP or UDP).

   Client Program: **socket**

   Server Program: **socket**

2. Associate the socket with the IP or IPX address of the local host and the port number of the local socket.

   Client Program: **bind**

   Server Program: **bind**

3. Associate the socket and the local port with the IP or IPX address of the remote host and the port number of the remote socket.

   Client Program: **connect**

   Server Program: **listen**, **accept**

4. Transfer data.

   Client Program: **read**, **write**, **sendto**, **rcvfrom**, or other data transfer functions

   Server Program: **read**, **write**, **sendto**, **rcvfrom**, or other data transfer functions

5. Terminate both endpoint associations (both local and remote hosts, as well as sockets) and free the socket.

   Client Program: **close**

   Server Program: **close**

The following figure illustrates the BSD programming model.

Figure 10. Berkeley Sockets Programming Model

Parent Topic: BSD Socket Overview

BSD Socket: Functions

accept

Accepts a connection from a remote host.
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int accept (
   int                s,
   struct sockaddr  *addr,
   int              *addrlen);
```

Parameters

s

(IN) Identifies a socket file handle that has been created with the **socket** function.

addr

(OUT) Specifies an address to be assigned to an unbound socket.

addrlen

(IN/OUT) Indicates the amount of space pointed to by *addr*.

Return Values

The **accept** function returns an integer handle for the accepted socket if successful. Otherwise, it returns a value of -1 to indicate an error. On failure, **accept** returns one of the following errors in *errno*:

EBADF	The socket file handle is invalid.
ENOTSOCK	The socket file handle refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

EINVAL	The socket is not in a listening state.

Remarks

accept is used with connection-based socket types, currently SOCK_STREAM.

The argument *s* is a socket file handle that has been created with the socket function and bound to an address with the **bind** function. The **listen** function prepares the socket to accept incoming connections; then, the **accept** function accepts the connection. The **accept** function extracts the first connection from the queue pending connections, creates a new socket with the same properties of *s*, and allocates a new file handle for the socket. In the case of no pending connections, the following apply:

If the socket is not marked as nonblocking, the **accept** function blocks the caller until a connection is present.

If the socket is marked nonblocking and there are no pending connections, the **accept** function returns an error.

The accepted socket handle cannot be used to accept more connections. The original socket *s* remains open and continues to listen for connections.

The argument *addr* is a result argument that receives the address of the connecting socket, as known to the communications layer. The exact form of the *addr* argument is determined by the domain in which communication occurs. The sockaddr_in structure in the header file NETINET/IN.H defines the form of the structure *addr* for TCP over IP sockets. The sockaddr_ipx structure in the header file NETIPX/IPX.H defines the form of the structure *addr* for TCP over IPX sockets. The length of the socket structure is fixed at 16 bytes.

The *addrlen* argument is a value-result argument; it must initially contain the amount of space pointed to by *addr*. On return, it contains the actual length (in bytes) of the address returned.

It is possible to select a socket for **accept** the way you would select it for **read**.

See Also

bind, **close**, **select**, **socket**

bind

Associates local address information with a socket
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int bind (
   int                s,
   struct sockaddr  *addr,
   int                addrlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket to which address information is to be assigned.

addr

(IN) Specifies the address to be assigned to an unbound socket.

addrlen

(IN) Indicates the amount of space pointed to by *addr*.

Return Values

If the binding is successful, bind returns a value of 0. Otherwise, bind returns a value of -1 and stores a more specific error code in *errno*. On failure, this function returns one of the following errors in *errno*:

EBADF	The argument s is not a valid socket filehandle.
ENOTSOCK	The argument s is a file, not a socket handle.
ENOPROTOOPT	The option is unknown at the level indicated.
EADDRNOTAVAIL	The specified address is not available from the local computer.

EADDRINUSE	The specified address is already in use.
EINVAL	The socket is already bound to an address, or an invalid address length is specified.
EACCES	The requested address is protected, and the current client lacks permission to access it.
EFAULT	The argument *addr* is not specified.

Remarks

The **bind** function assigns an argument *addr* to an unbound socket. When the **socket** function creates a socket, the socket exists in an *addr* space (address family) but has no address assigned.

The **bind** function requests that *addr* be assigned to the socket. Once the argument *addr* is bound to a socket, another socket can establish communication by referring to that address.

For a TCP or UDP socket, the argument *addr* points to a sockaddr_in structure, defined in NETINET/IN.H. When calling the **bind** function, the client should initialize the sockaddr_in structure's members as follows:

sin_family should be AF_INET.

sin_port can be a 16-bit local port value in the TCP or UDP domain, in network data order. NETINET/IN.H defines well-known values. If it is 0, the protocol assigns an unused, nonprivileged value between 1024 and 5000. Values 1 through 1023 and 5001 through 65535 are reserved for well-known server ports.

sin_addr can usually be set to 0, allowing the protocol to determine the appropriate local IP address, depending on the route. The client can set a local address explicitly (in network data order) if the address belongs to the node.

gethostid can be used to obtain the default local IP address.

sin_zero must be set to eight bytes of 0.

For a TCP over IPX socket, the argument *addr* points to a sockaddr_ipx structure, defined in NETIPX/IPX.H. When calling the **bind** function, the client should initialize the sockaddr_ipx structure's members as follows:

sipx_family should be AF_IPX.

sipx_addr.x_net contains a four-character array that holds the network number.

sipx_addr.x_host contains a six-character array that holds the node

number.

sipx_addr.x_socket contains a two-character array that holds the socket number.

sipx_zero must be initialized to zero (0).

See Also

getsockname, **listen**, **socket**

connect

Initiates a connection on a socket
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: ANSI
Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int connect (
   int                s,
   struct sockaddr  *addr,
   int                addrlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket on which a connection is to be initiated.

addr

(IN) Specifies the address to be assigned to an unbound socket.

addrlen

(IN) Indicates the amount of space pointed to by *addr*.

Return Values

If the connection or binding succeeds, **connect** returns a value of 0. Otherwise, it returns a value of -1, and *errno* indicates a more specific error. On failure, **connect** returns one of the following errors in *errno*:

EBADF	The argument s is not a valid socket filehandle.
ENOTSOCK	The argument s is a file, not a socket handle.
ENOPROTOOPT	The option is unknown at the level indicated.
EADDRNOTAVAIL	The specified address is not available from the local computer.

EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	Connection timed out without establishing a connection.
ECONNREFUSED	The attempt to connect was forcefully rejected.
ENETUNREACH	The network cannot be reached from this host.
EADDRINUSE	The address is already in use.
EINPROGRESS	The socket is nonblocking, and the connection cannot be completed immediately.
EALREADY	The socket is nonblocking, and a previous connection attempt has not yet been completed.
EINVAL	The argument *namelen* is invalid, or the socket has been reset and is no longer valid.

Remarks

The argument *s* is a socket. If the socket is SOCK_DGRAM, the **connect** function specifies the peer (for example, a socket endpoint for either a client or a server) with which the socket is to be associated. Only the address of that socket endpoint receives or sends datagrams. If the socket is SOCK_STREAM, the **connect** function attempts to make a connection to another socket. The other socket is specified by *addr*, which for TCP and UDP over IP sockets is a sockaddr_in structure, as defined in NETINET/IN.H. For TCP over IPX sockets, the structure is sockaddr_ipx, as defined in NETIPX/IPX.H. The length of this structure is given in *addrlen*.

Generally, SOCK_STREAM sockets can successfully connect to a peer only once; SOCK_DGRAM sockets can use **connect** multiple times to change their association. Datagram sockets end the association by connecting to an invalid address, such as a NULL address.

If a SOCK_STREAM socket is in a blocking mode, the **connect** function blocks until a connection is established or the attempt fails. If the socket is set for nonblocking I/O, the connect function returns an error code, its value depending on the current state of the connection process. Depending on the conditions, the error code could be one of the following:

EINPROGRESS	This is in response to the initial connect request.

EALREADY	This is in response to subsequent requests before a connection is established.
EISCONN	This occurs after a connection is established.

By selecting the socket for writing, you can also select it for completing the connection.

See Also

accept, **getsockname**, **select**, **socket**

getpeername

Returns the address of a connected peer
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int getpeername (
   int                s,
   struct sockaddr   *addr,
   int               *addrlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket connected to the peer for which the *addr* argument is to be returned.

addr

(OUT) Indicates the address of the peer connected to the socket specified by *s*.

addrlen

(IN/OUT) Indicates the size of the address pointed to by *addr*.

Return Values

getpeername returns a value of 0 if the function succeeds and a value of -1 if it fails. On failure, **getpeername** returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid socket file handle.
ENOTSOCK	The argument *s* is a file, not a socket.
ENOTCONN	The socket is not connected.
ENOBUFS	Insufficient resources were available in the system to perform the operation.

Remarks

The **getpeername** function returns the *addr* argument of the peer (which is a socket endpoint of either a client or a server) connected to sockets. For TCP and UDP over IP sockets, the argument *addr* refers to the sockaddr_in structure, defined in NETINET/IN.H. For TCP over IPX sockets, the argument *addr* refers to the sockaddr_ipx structure, defined in NETIPX/IPX.H.

The *addrlen* argument should be initialized to the size of this structure. On return, the argument contains the actual size of the address returned (in bytes). The address is truncated if the provided buffer is too small.

See Also

accept, **bind**, **getsockname**, **socket**

getsockname

Returns the current address for the specified socket
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int getsockname (
   int                s,
   struct sockaddr   *addr,
   int               *addrlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket for which the *addr* argument is to be returned.

addr

(OUT) Indicates the address of the socket specified by *s*.

addrlen

(IN/OUT) Indicates the size of the address pointed to by *addr*.

Return Values

getsockname returns a value of 0 it succeeds and a value of -1 if it fails. On failure, **getsockname** returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid socket file handle.
ENOTSOCK	The argument *s* is a file, not a socket.
ENOBUFS	Insufficient resources were available in the system to perform the operation.

Remarks

For TCP and UDP over IP sockets,*addr* refers to the sockaddr_in structure, defined in NETINET/IN.H. For TCP over IPX sockets,*addr* refers to the sockaddr_ipx structure, defined in NETIPX/IPX.H. The *addrlen* argument should be initialized to indicate the size of this structure. On return, the argument contains the actual size of the address returned (in bytes).

See Also

bind, **socket**

getsockopt

Returns current parameters for socket operation
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int getsockopt (
   int    s,
   int    level,
   int    optname,
   char  *optval,
   int   *optlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket for which parameters are to be returned.

level

(IN) Specifies the level at which the option resides.

optname

(IN) Identifies the option for the request.

optval

(OUT) Points to an option value.

optlen

(IN/OUT) Points to a value describing the option's length.

Return Values

If **getsockopt** succeeds, it returns a value of 0. If **getsockopt** fails, it returns a value of -1, and *errno* indicates the error.

On failure, the **getsockopt** function return one of the following errors in *errno*:

EBADF	The argument *s* is not a valid socket file handle.
EINVAL	The value of argument optlen is inappropriate.
ENOTSOCK	The argument *s* is a file, not a socket.
ENOPROTOOPT	The option is unknown at the level indicated.

Remarks

The **getsockopt** functions manipulates options associated with a socket. These options are defined as arguments for these functions. Options exist at either the socket level or the underlying protocol level.

The *optname* argument identifies the option for the request. The include file SYS/ SOCKET.H contains definitions for socket-level options.

The argument *optval* points to an option value, while *optlen* is a pointer to a value describing the length of the option. These arguments identify a buffer in which the values for the requested option(s) are to be returned.

The *optlen* argument is a value-result argument, initially containing the size of the buffer pointed to by *optval* and modified on return to indicate the actual size of the value returned.

For all cases except SO_LINGER, *optval* refers to an integer object.

The following options are recognized at the socket level. Except as noted, each can be examined with **getsockopt** and set with **setsockopt**.

SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses. This option is typically enabled only for TCP sockets that connect and use a fixed local port value, as in the File Transport Protocol (FTP).
SO_KEEPALIVE	Enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken, and requests on the socket fail, returning the ETIMEDOUT error code.
SO_OOBINLINE	Requests that out-of-band data be placed in the normal data input queue as received; it is then accessible with **recv** or **read** functions without the MSG_OOB flag.
SO_SNDBUF	Specifies the limit placed on output data buffering. The limit can be increased for high-volume connections or decreased to limit the possible backlog of buffered data. The system places an absolute limit

	of 64 Kb on this value.
SO_RCVBUF	Is the counterpart to SO_SNDBUF for incoming data. It determines the size of TCP's receive window. For best efficiency, sender and receiver should have similar send and recv buffer limits.
SO_LINGER	Controls the action taken when unsent data is queued on a SOCK_STREAM socket and a **close** call is performed. The *optval* argument can point to a struct linger argument, defined in SYS/SOCKET.H, which specifies the desired state of the option and the linger interval. If SO_LINGER is enabled and the linger time is nonzero, the **close** call blocks indefinitely, or until the protocol is able to deliver the data. If the linger time is zero, the protocol aborts the connection when **close** is invoked, possibly losing data. If SO_LINGER is disabled and a **close** is issued, the protocol assumes responsibility for delivering the data and permits the client to continue without blocking. This is the default behavior.
SO_TYPE	Returns the type of the socket, such as SOCK_STREAM.
SO_ERROR	Returns any pending error code on the socket and clears the error status. You can use it to check errors on connected datagram sockets.

See Also

getprotoent, **setsockopt**, **socket**

listen

Prepares a socket to accept a connection
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: ANSI
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int listen (
   int   s,
   int   backlog);
```

Parameters

s

(IN) Identifies the socket file handle of a socket that is to be prepared to accept a connection.

backlog

(IN) Defines the maximum length for the queue of pending connections.

Return Values

A return value of 0 indicates success, and a value of -1 indicates an error. On failure, **listen** returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid socket file handle.
ENOTSOCK	The argument *s* is a file, not a socket file handle.
ENOPROTOOPT	The option is unknown at the level indicated.
EOPNOTSUPP	The socket does not support this function.

Remarks

The **socket** function first creates a socket *s*. The **listen** function prepares a socket to accept incoming connections and specifies a queue limit for incoming connections. Then the **accept** function accepts the connection.

The **listen** function lies only to sockets of type SOCK_STREAM.

The *backlog* argument defines the maximum length for the queue of pending connections. The effective queue limit is *backlog* + 1, with a silent limitation of 5 for the *backlog*. If a connect request arrives when the queue is full, the client receives an error with ECONNREFUSED.

See Also

accept, **connect**, **socket**

readv

Reads data from a socket and stores it in multiple user buffers as specified
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: ANSI
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>
#include <sys/types.h>
#include <sys/uio.h>

int readv (
   int            s,
   struct iovec  *iov,
   int            iovcnt);
```

Parameters

s

(IN) Identifies the socket file handle of a socket from which data is to be read.

iov

(OUT) Points to the iovec structure, which specifies the base address and length of an area in memory where data can be placed.

iovcnt

(IN) Indicates the buffers that the members of the *iov* array specify.

Return Values

If successful, the **readv** function returns either the EOF (zero) or the number of bytes actually read. Otherwise, it returns a value of -1 and set *errno* to indicate the error. On failure, the **readv** function returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid file or socket handle open for reading.
EWOULDBLOCK	The file was marked for nonblocking I/O, and *data* was not ready to be read.

EINVAL	One of the *iov_len* values in the *iov* array was negative, or the sum of the *iov_len* values in the *iov* array overflowed a 32bit integer.

Remarks

The **readv** function attempts to read nbytes of data from a SOCK_STREAM socket described by *s* into the buffer pointed to by *buf*. It scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt* - 1].

For **readv**, the iovec structure is defined in UIO.H as follows:

```
struct iovec {
    char   *iov_base;
    int    iov_len;
} iovec;
```

Each iovec entry specifies the base address and length of an area in memory where data can be placed. The **readv** function always fills an area completely before proceeding to the next.

Upon successful completion, **readv** returns the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if there are that many bytes left before the end-of-file. If the returned value is 0, the peer has closed the connection and no more data is received.

To read out-of-band data from a TCP socket by calling the **read** function, the **setsockopt** option (SO_OOBINLINE) must be enabled. The client can select the exception condition to detect the arrival of a new urgent boundary (just before the out-of-band byte); the client can select the **ioctl** operation (SIOCATMARK) to determine whether the read request has read up to the boundary yet. The system guarantees that a single read request does not cross the urgent boundary. When SIOCATMARK is detected, the next read request returns the out-of-band byte plus as much data past this byte as can fit in the buffer.

If SO_OOBINLINE is not enabled, the client must call the **recv** function to read the out-of-band byte. As mentioned previously, a read request reads up to the urgent boundary, but the next request reads data past the urgent boundary without reading the out-of-band byte, since it was not stored with the SO_OOBINLINE option.

SIOCATMARK reveals whether the out-of-band byte is readable by either method.

The **readv** function can also read messages from a SOCK_DGRAM socket, but **recv** and its derivatives are better suited for this job. Semantics are identical with **recv** except that the *flags* argument is not available.

Communication Service Group

See Also

recv, recvfrom, recvmsg, select, socket

recv, recvfrom, recvmsg

recv reads data from a socket and stores it in a buffer, with options for peeking and out-of-band data; **recvfrom** also obtains source address information; **recvmsg** is the same as **recvfrom** but works on a list of user buffers

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x

Platform: NLM

Standard: BSD

Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int recv (
   int    s,
   char  *buf,
   int    len,
   int    flags);

int recvfrom (
     int               s,
     char             *buf,
     int               len,
     int               flags,
     struct sockaddr  *from,
     int              *fromlen);

int recvmsg (
   int             s,
   struct msghdr  *msg,
   int             flags);
```

Parameters

recv

s

(IN) Identifies the socket file handle of a socket from which data is to be read.

buf

(OUT) Identifies a buffer in which data is to be stored.

len

(IN) Indicates the length of the buffer specified in *buf*.

flags

(IN) Is formed by using one of two values, as described in the "Remarks" section.

rcvfrom

s

(IN) Identifies the socket file handle of a socket from which data is to be read.

buf

(OUT) Identifies a buffer in which data is to be stored.

len

(IN) Indicates the length of the buffer specified in *buf*.

flags

(IN) Is formed by using one of two values, as described in the "Remarks" section.

from

(OUT) Points to the structure containing the source address of the message.

fromlen

(IN/OUT) Indicates the length of the buffer associated with *from*.

recvmsg

s

(IN) Identifies the socket file handle of a socket from which data is to be read.

msg

(IN/OUT) Specifies message address information and buffer address information.

flags

(IN) Formed by using one of two values, as described in the "Remarks" section.

Return Values

If successful, these functions return either EOF or the number of bytes received; they return a value of -1 if an error occurred. On failure, the **recv**, **recvfrom**, and **recvmsg** functions return one of the following errors in *errno*:

EBADF	The argument s is an invalid socket file handle.
ENOTSOCK	The argument s is a file, not a socket handle.
EWOULDBLOCK	The socket is marked nonblocking, and the receive operation would block.
EINVAL	The MSG_OOB flag is invalid when used with the SO_OOBINLINE option, or the out-of-band data is not present.
ENOTCONN	The socket is not in a connected state.
EOPNOTSUPP	The *flags* values are not supported.

Remarks

Typically, clients call the **recv**, **recvfrom**, and **recvmsg** functions to receive messages from a SOCK_DGRAM socket. They return a single datagram.

Clients normally call **recv** only on a connected socket (see **connect**), while **recvfrom** and **recvmsg** can be used to receive data on a socket whether or not it is in a connected state.

On a SOCK_STREAM socket, clients can call **recv**, **recvfrom**, and **recvmsg**, although the **read** or **readv** functions are nearly as capable in this case.

recv

If data is not available at the socket, the **recv** function blocks, unless the socket is nonblocking (see **ioctl**), in which case the function returns a value of -1 with *errno* set to EWOULDBLOCK.

The client can call **select** to determine when more data arrives.

The *flags* argument to a **recv** function is formed by using one or more of the following values:

```
#define   MSG_OOB   0x1   /* process out-of-band data */
#define   MSG_PEEK  0x2   /* peek at incoming message */
```

MSG_OOB is used only with TCP sockets. Out-of-band (OOB) data is a misnomer that actually refers to the byte of data following a TCP urgent boundary. After the socket client has discovered new out-of-band data (using **select** for exceptions) and has read normal data up to the out-of-band byte (using **ioctl** SIOCATMARK to recognize the boundary), it reads one byte of out-of-band data by setting the MSG_OOB flag.

MSG_PEEK permits the socket client to look at a message or STREAM data without removing it from the socket.

recvfrom

If the from argument to a **recvfrom** function is nonzero, the source

address of the message is filled. The *fromlen* argument is a value-result argument initialized to the size of the buffer associated with *from*. On return, the actual size of the source address is stored in *fromlen*.

For a UDP socket, if a message is too long to fit in the supplied buffer, **rcvfrom** truncates the excess bytes.

recvmsg

The **recvmsg** function uses a msghdr structure to minimize the number of directly supplied arguments. This structure has the following form, as defined in SYS/ SOCKET.H:

```
struct msghdr {
   char           *msg_name;       /* optional address  */
   int            msg_addrlen;     /* size of address  */
   struct iovec   *msg_iov;        /* scatter/gather array  */
   int            msg_iovlen;      /* # elements in msg_iov */
   char           *msg_accrights;   /* unused, set to 0   */
   int            msg_accrightslen; /* unused, set to 0  */
} msghdr;
```

Here, *msg_name* returns the source address of a message; *msg_name* can be given as a NULL pointer if no addresses are desired or required.

The *msg_iov* and *msg_iovlen* fields describe the scatter/gather locations, which constitute a buffer that is dispersed in different locations. The *msg_accrights* and *msg_accrightslen* fields do not apply to TCP or UDP sockets. They can be set to 0 in a request and are undefined in a reply.

See Also

getsockname, **readv**, **select**, **send, sendmsg, sendto**, **socket**

select

Performs multiplexed socket status queries
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>
#include <sys/types.h>
#include <sys/time.h>

int select (
   int              width,
   fd_set          *readfds,
   fd_set          *writefds,
   fd_set          *exceptfds,
   struct timeval  *timeout);
```

Parameters

width

(IN) Indicates the total number of socket handles to be examined.

readfds

(IN/OUT) Passes address information for the socket handle set to be examined.

writefds

(IN/OUT) Passes address information for the socket handle set to be examined.

exceptfds

(IN/OUT) Passes address information for the socket handle set to be examined.

timeout

(IN) Points to the timeval structure, which specifies a maximum interval to wait for the completion of **select**.

Return Values

Returns the number of ready handles contained in the handle sets, or it

returns -1 if an error occurs. If the time limit expires, returns a value of 0. If **select** returns with an error, the handle sets are not modified.

On failure, returns one of the following errors in *errno*:

EBADF	One of the handle sets specified an invalid handle.
EINVAL	The specified time limit is invalid. One of its components is negative or too large.
ENOTSOCK	One of the handle sets specified a nonsocket handle.

Remarks

The **select** function examines the socket handle sets whose addresses are passed in the *readfds*, *writefds*, and *exceptfds* parameters to see if any of their handles are ready for reading or writing, or have an exceptional condition pending. The socket handles from 0 through *width*-1 in the handle sets are examined. On return, the function replaces the given socket handle sets with subsets consisting of those handles that are ready for the operation.

The handle sets are arrays of size FD_SETSIZE containing socket handles. Each unused element must be set to 0. The following macros are provided for manipulating such handle sets:

FD_ZERO(&fhset)---Initializes a handle set fhset to the null set

FD_SET(fh, &fhset)---Includes a particular handle fh in fhset

FD_CLR(fh, &fhset)---Removes fh from fhset

FD_ISSET(fh, &fhset)---Is nonzero if fh is a member of fhset, zero otherwise

Here, fh specifies a handle to be manipulated, and fhset specifies a handle set.

If the client attempts to specify more file handles in FD_SET than the handles specified in FD_SETSIZE, the operation fails, silently.

If the *timeout* pointer is nonzero, it points to the timeval structure which specifies a maximum interval to wait for the completion of **select**. If the *timeout* parameter is a zero pointer, **select** blocks indefinitely. To poll, the *timeout* parameter can point to a zero-valued timeval structure. If handles are not of interest, the *readfds*, *writefds*, and *exceptfds* parameters can be zero pointers.

Normally, **select** should be used with nonblocking sockets. Even when **select** indicates that a socket handle is ready for an operation, it is possible that **select** might return EWOULDBLOCK when attempted. For

example, when two or more processes share a handle, **select** also indicates readiness for reading or writing when a socket fails or no longer performs the indicated operation. This nuance assures that a client wakes up and notices the new socket state.

select works only with socket handles and should not be used for other file types.

The *exceptfs* parameter is meaningful only for TCP sockets. When ready, the TCP sockets imply the existence of out-of-band urgent data on the socket.

select can also be used to check for completion of accept and connect operations on nonblocking sockets if the conditions correspond to read and write readiness respectively.

See Also

accept, **connect**, **readv**, **recv, recvfrom, recvmsg, send, sendmsg, sendto**, **writev**

send, sendmsg, sendto

send writes data from a user buffer to a socket, with the option for out-of-band data; **sendto** also takes destination address information; **sendmsg** is the same as **sendto** but works on a list of user buffers.

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.x, 4.x

Platform: NLM

Standard: BSD

Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int send (
   int    s,
   char  *msg,
   int    len,
   int    flags);

int sendmsg (
  int              s,
  struct msghdr   *msg,
  int              flags);

int sendto (
  int               s,
  char             *msg,
  int               len,
  int               flags,
  struct sockaddr  *to,
  int               tolen);
```

Parameters

send

s

(IN) Identifies the socket file handle of a socket to which the data is to be written.

msg

(IN) Specifies the message to be sent.

len

(IN) Indicates the length of the message.

flags

(IN) Is formed by including the MSG_OOB flag as described in the "Remarks" section.

sendmsg

s

(IN) Identifies the socket file handle of a socket to which the data is to be written.

msg

(IN) Specifies message address information and buffer address information.

flags

(IN) Formed by including the MSG_OOB flag, as described in the "Remarks" section.

sendto

s

(IN) Identifies the socket file handle of a socket to which the data is to be written.

msg

(IN) Specifies the message to be sent.

len

(IN) Indicates the length of the message.

flags

(IN) Can include the MSG_OOB flag, as described in the "Remarks" section.

to

(IN) Points to the sockaddr structure containing the destination address.

tolen

(IN) Indicates the size of the destination address.

Return Values

These functions return the number of characters sent, or -1 if an error occurred. On failure, they return one of the following errors in*errno*:

EBADF	An invalid handle was specified.
ENOTSOCK	The argument *s* is a file, not a socket handle.

EMSGSIZE	The socket requires that a message be sent atomically, and the size of the message to be sent made it impossible.
EWOULDBLOCK	The socket is marked nonblocking, and the requested operation would block it.
ENOBUFS	The system could not allocate an internal buffer. The operation might succeed if retried when buffers become available.

Remarks

The **send**, **sendto**, and **sendmsg** functions are normally used to transmit a message (such as a packet or datagram) from a SOCK_DGRAM socket to a similar socket. The **send** function can be used only when the socket is in a connected state, while **sendto** and **sendmsg** functions can be used any time.

The **send**, **sendto**, and **sendmsg** functions can also be used on SOCK_STREAM sockets, although **write** and **writev** are almost as capable and more commonly used. When used on a blocking SOCK_STREAM socket, these requests block until all of the client's data can be sent or buffered by the socket. When used on a nonblocking socket, these requests send or buffer the maximum amount of data that can be handled without blocking and return the amount that was taken. If no data is taken, they return a value of -1, indicating an EWOULDBLOCK error.

send

An indication of failure to deliver is not implicit in a SOCK_DGRAM **send** function. A return value of -1 indicates a locally detected error.

If the local socket lacks buffer space to hold the message to be transmitted, the **send** function normally blocks unless the socket has been placed in nonblocking I/O mode. The **select** function can be used to determine when it is possible to send more data.

The flags argument can include the following flag:

```
#define MSG_OOB 0x1   /* process out-of-band data */
```

The flag MSG_OOB is set to send out-of-band data on sockets that support out-of-band transmission. Only TCP sockets support out-of-band transmission, which is accomplished by placing a new urgent boundary just before the last byte of the sent message.

sendmsg

The **sendmsg** function uses a msghdr structure to minimize the number of directly supplied arguments. This structure has the following form, as defined in SYS/ SOCKET.H:

```
struct msghdr {
   char            *msg_name;        /* optional address  */
   int             msg_addrlen;      /* size of address  */
   struct iovec    *msg_iov;         /* scatter/gather array  */
   int             msg_iovlen;       /* # elements in msg_iov */
   char            *msg_accrights;   /* unused, set to 0   */
   int             msg_accrightslen; /* unused, set to 0  */
} msghdr;
```

Here, *msg_name* returns the source address of a message; *msg_name* can be given as a NULL pointer if no addresses are desired or required.

The *msg_iov* and *msg_iovlen* fields describe the scatter/gather locations, which constitute a buffer that is dispersed in different locations. The *msg_accrights* and *msg_accrightslen* fields do not apply to TCP or UDP sockets. They can be set to 0 in a request and are undefined in a reply.

sendto

The destination address is given by *to*, with *tolen* specifying its size. For UDP over IP sockets, the address is a sockaddr_in structure, defined in NETINET/IN.H. For TCP over IPX sockets, the address is a sockaddr_ipx structure, defined in NETIPX/IPX.H. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned in *errno*, and the message is not transmitted.

See Also

getsockname, **recv, recvfrom, recvmsg**, **select**, **socket**, **writev**

setsockopt

Establishes parameters for socket operation
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int setsockopt (
   int    s,
   int    level,
   int    optname,
   char  *optval,
   int    optlen);
```

Parameters

s

(IN) Identifies the socket file handle of a socket for which parameters are to be established.

level

(IN) Specifies the level at which the option resides.

optname

(IN) Identifies the option for the request.

optval

(IN) Points to an option value.

optlen

(IN) Indicates the length of the option.

Return Values

If **setsockopt** succeeds, it returns a value of 0. If it fails, it returns a value of -1, and *errno* indicates the error.

On failure, the **setsockopt** function returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid socket file handle.
EINVAL	The value of argument *optlen* is inappropriate.
ENOTSOCK	The argument *s* is a file, not a socket.
ENOPROTOOPT	The option is unknown at the level indicated.

Remarks

The **setsockopt** function manipulates options associated with a socket. These options are defined as arguments for these functions. Options exist at either the socket level or the underlying protocol level.

When setting socket options, you must specify the name of the option and the level at which it resides. To set options at the socket level, specify level as SOL_SOCKET. To set options at the protocol level, supply the number of the underlying protocol controlling the option. For example, for IP protocol families if you want TCP to interpret an option, set the level to IPPROTO_TCP, as defined in NETINET/IN.H. The protocol number for UDP is IPPROTO_UDP. For IPX protocol families if you want TCP to interpret an option, set the level to IPXPROTO_TCP, as defined in NETIPX/IPX.H. See **getprotoent**.

The *optname* argument identifies the option for the request. The include file SYS/ SOCKET.H contains definitions for socket-level options.

The argument *optval* points to an option value, while *optlen* is the length of the option.

For all cases except SO_LINGER, *optval* refers to an integer object. The *optval* argument points to a nonzero integer to enable a boolean option, or zero to disable the option.

The following options are recognized at the socket level. Except as noted, each can be examined with **getsockopt** and set with **setsockopt**.

SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses. This option is typically enabled only for TCP sockets that connect and use a fixed local port value, as in the File Transport Protocol (FTP).
SO_KEEPALIVE	Enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken, and requests on the socket fail, returning the ETIMEDOUT error code.
SO_OOBINLI	Requests that out-of-band data be placed in the

NE	normal data input queue as received; it is then accessible with **recv** or read calls without the MSG_OOB flag.
SO_SNDBUF	Specifies the limit placed on output data buffering. The limit can be increased for high-volume connections or decreased to limit the possible backlog of buffered data. The system places an absolute limit of 64 Kb on this value.
SO_RCVBUF	Is the counterpart to SO_SNDBUF for incoming data. It determines the size of TCP's receive window. For best efficiency, sender and receiver should have similar **send** and **recv** buffer limits.
SO_LINGER	Controls the action taken when unsent data is queued on a SOCK_STREAM socket and a close call is performed. The *optval* argument can point to a struct linger argument, defined in SYS/SOCKET.H, which specifies the desired state of the option and the linger interval. If SO_LINGER is enabled and the linger time is nonzero, the close call blocks indefinitely, or until the protocol is able to deliver the data. If the linger time is zero, the protocol aborts the connection when close is invoked, possibly losing data. If SO_LINGER is disabled and a close is issued, the protocol assumes responsibility for delivering the data and permits the client to continue without blocking. This is the default behavior.
SO_TYPE	Returns the type of the socket, such as SOCK_STREAM.
SO_ERROR	Returns any pending error code on the socket and clears the error status. You can use it to check errors on connected datagram sockets.

See Also

getprotoent, **socket**

shutdown

Ends all or part of a full-duplex connection
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int shutdown (
   int   s,
   int   how);
```

Parameters

s

(IN) Identifies the socket file handle of a socket on which the connection is to be ended.

how

(IN) Indicates the extent to which the connection is to be ended.

Return Values

If **shutdown** succeeds, it returns a value of 0. It returns -1 if it fails. On failure, **shutdown** returns one of the following errors in *errno*:

EBADF	The argument *s* is not a valid handle.
ENOTSOCK	The argument *s* is a file, not a socket handle.
ENOTCONN	The specified socket is not connected.

Remarks

The **shutdown** function ends all or part of a full-duplex connection on the socket associated with *s*. If *how* is 0 (receive shutdown), subsequent **recv** calls are not allowed. If *how* is 1 (send shutdown), further **send** calls are not allowed. If *how* is 2 (send and receive shutdown), subsequent **send** and **recv** calls are not allowed.

and **recv** calls are not allowed.

A send **shutdown** function on a TCP socket issues a graceful close. The socket can still be used to receive data. A receive **shutdown** call merely discards incoming data.

See Also

connect, **socket**

socket

Creates a socket endpoint for communication
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <sys/types.h>
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>

int socket (
   int   domain,
   int   type,
   int   protocol);
```

Parameters

domain

(IN) Specifies a domain where communication is to occur and selects the protocol family used.

type

(IN) Specifies the semantics of communication and implies a protocol.

protocol

(IN) Specifies a particular protocol to be used with the socket.

Return Values

If an error occurs, **socket** returns -1 and *errno* describes the error. Otherwise, the return value is a socket handle referencing the socket.

On failure, **socket** returns one of the following errors in *errno*:

EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.
EPROTOTYPE	The specified protocol type does not match its actual type.
ENOBUFS	The socket cannot be created until sufficient resources are freed.

Remarks

The **socket** function creates an endpoint for communication and returns a handle.

The *domain* argument specifies a domain where communication is to occur and selects the protocol family used. This version supports only the PF_INET family, which includes TCP and UDP. Both protocols use the AF_INET address family for the addresses supplied in later operations on the socket.

This version also supports the PF_IPX family, which includes TCP. This protocol uses the AF_IPX address family for the addresses supplied in later operations on the socket.

The *type* argument specifies the semantics of communication and implies a protocol. Within PF_INET, the SOCK_STREAM type uses TCP, and the SOCK_DGRAM type uses UDP, as follows:

TCP provides sequenced, reliable, two-way connection-based byte streams.

UDP supports datagrams (connectionless, unreliable messages shorter than 65,536 bytes).

The *protocol* argument specifies a particular protocol to be used with the socket. The PF_INET family supports one protocol per type, therefore it is sufficient to specify its value as 0. For completeness, the supported values for the PF_INET domain are defined in NETINET/IN.H. The protocol values for TCP and UDP are as follows:

IPPROTO_TCP	TCP
IPPROTO_UDP	UDP

The PF_IPX family supports only one protocol; therefore, it is sufficient to specify its value as 0. The supported values for the PF_IPX domain are defined in NETIPX/IPX.H. The protocol value for TCP is as follows:

IPXPROTO_TCP	TCP

A TCP socket must be in a connected state before it can receive or send any data. The **connect** function creates another socket. Once connected,

data can be transferred using the **read** and **write** functions or some variant of the **send** and **recv** functions. When a session is complete, a**close** function can be performed. Out-of-band urgent data can also be transmitted (as described in **send**) and received (as described in **recv**).

TCP ensures that data is not lost or duplicated. If the peer socket does not acknowledge a transmission attempt within a reasonable length of time, the connection is considered broken and the call fails, returning ETIMEDOUT in *errno*. TCP optionally keeps sockets active by forcing transmissions periodically in the absence of other activity.

SOCK_DGRAM sockets allow transmission of datagrams to correspondents named in **sendto** calls. The **recvfrom** function receives datagrams and returns the next datagram with its return address.

A **fcntl** function can be used to enable nonblocking I/O.

Socket-level options affect the operation of sockets. The file SYS/SOCKET.H defines these options. The **setsockopt** and **getsockopt** functions are used to set and get options, respectively.

See Also

accept, **bind**, **connect**, **getsockname**, **getsockopt**, **listen**, **recv, recvfrom, recvmsg, select**, **send, sendmsg, sendto**, **shutdown**, **writev**

writev

Writes data from a list of user buffers to a socket
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.x, 4.x
Platform: NLM
Standard: BSD
Service: BSD Socket

Syntax

```
#include <nlm/sys/socket.h>
#include <nlm/sys/bsdskt.h>
#include <sys/types.h>
#include <sys/uio.h>

int writev (
   int            s,
   struct iovec  *iov,
   int            iovcnt);
```

Parameters

s

(IN) Identifies the socket file handle of a socket to which data is to be written.

iov

(IN) Points to the iovec structure, which specifies a list of buffers from which data can be copied.

iovcnt

(IN) Indicates the number of buffers specified by the *iov* array.

Return Values

On failure, the **writev** function returns one of the following errors in*errno* :

EBADF	The argument s is not a valid socket file handle open for writing.
EMSGSIZE	On a SOCK_DGRAM socket, the size of the message would exceed the protocol's capability.
EPIPE	An attempt is made to write to a SOCK_STREAM socket that is not connected to a peer socket or has

	socket that is not connected to a peer socket or has been closed.
EWOULDBLOCK	The file was marked for nonblocking I/O, and no data could be written immediately.
EINVAL	One of the iov_len values in the iov array was negative, or the sum of the iov_len values in the iov array overflowed a 32-bit integer.
EDESTADDRREQ	The datagram socket has not associated itself with a destination address.

Remarks

The **writev** function attempts to write nbytes of data to a SOCK_STREAM socket described by the argument s from the buffer pointed to by *buf*. It gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1].

For **writev**, the iovec structure is defined as follows:

```
struct iovec
{
   char   *iov_base;
   int     iov_len;
} iovec;
```

Each iovec entry specifies the base address and length of an area in memory from which data can be copied.

When using nonblocking I/O on SOCK_STREAM sockets, **writev** might write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. Call **select** to determine when to try the operation again.

The **writev** function can also write data to SOCK_DGRAM sockets, but **send** and its derivatives are better suited for this job. Semantics are identical to a send request except the *flags* argument is not available.

See Also

select

Communication Service Group

BSD Socket: Structures

iovec

Service: BSD Socket
Defined In: socket.h

Structure

```
nlm/sys/socket.h
nlm/sys/bsdskt.h

struct iovec {
    char   *iov_base;
    int    iov_len;
} iovec;
```

Fields

iov_base

iov_len

msghdr

Contains message information
Service: BSD Socket
Defined In: socket.h

Structure

```
nlm/sys/socket.h
nlm/sys/bsdskt.h

struct msghdr {
   char           *msg_name;
   int            msg_addrlen;
   struct iovec   *msg_iov;
   int            msg_iovlen;
   char           *msg_accrights;
   int            msg_accrightslen;
} msghdr;
```

Fields

msg_name
Contains an optional address.

msg_addrlen
Contains the size of *msg_name*.

msg_iov
Contains the scatter/gather array.

msg_iovlen
Contains the number of elements in *msg_iov*.

msg_accrights
Unused; set to 0.

msg_accrightslen
Unused; set to 0.

sockaddr

Contains socket address information
Service: BSD Socket
Defined In: socket.h

Structure

```
nlm/sys/socket.h
nlm/sys/bsdskt.h
```

Fields

timeval

Contains BSD time information
Service: BSD Socket
Defined In: socket.h

Structure

```
nlm/sys/socket.h
nlm/sys/bsdskt.h
nlm/sys/timeval.h

struct {
   long   tv_sec;
   long   tv_usec;
} timeval;
```

Fields

tv_sec

Specifies the number of seconds for the timeout value.

tv_usec

Specifies the number of miscroseconds for the timeout value.

Diagnostic

Communication Service Group

Diagnostic: Guides

Diagnostic: General Guide

Diagnostic: Concepts

Configuration Reply Packet

Each network node not excluded from the broadcast of the Configuration Request Packet responds with a Configuration Reply Packet. By scanning IPX Configuration Reply Packets returned by network nodes, a diagnostic application can identify all nodes and their software components and build a map of the network.

The Configuration Reply Packet includes the following fields:

IPX header

Major version

Minor version

SPX diagnostic socket

Component count

Component type

component type identifies a software component residing at the responding node. The field is repeated in succession for each component present at the node. The following types are defined:

0 IPX/SPX
1 Bridge driver
2 Shell driver
3 Shell
4 VAP shell
5 External bridge
6 Internal bridge
7 Nondedicated NetWare® server
8 Star 68000 (IPX only)
9 DOS GNMA
10 OS/2 GNMA

For NetWare servers, bridges, and star 68000, each *component type* is followed by additional information describing the networks that the component communicates with. This information includes the following fields.

Network count

Network count

Local network type

Network address

Node address

local network type, *network address*, and *node address* are repeated for each local network the component communicates with. *local network type* can take the following values:

0 LAN board
1 Non-dedicated file server (virtual board)
2 Redirected remote line
3 Virtual SFT3 board
4 Idle (or Standby) virtual SFT3 board

ConfigurationResponseStruct is defined for interpreting Configuration Response Packets.

Parent Topic:

Diagnostic: General Guide

Related Topics:

Querying for Configuration Data

Configuration Request Packet

Configuration Request Packet

To broadcast a configuration request you build a Configuration Request Packet. Since IPX does not guarantee packet delivery, you may have to re-broadcast the request packet to unresponsive nodes. The packet includes the following fields:

IPX header

Number of exclusions

Exclusion list

IPX header is the IPX header filled in appropriately:

To query a specific network node, set *node address* in the IPX header to the node's address.

To query multiple nodes, set *node address* to 0xFFFFFFFFFFFF.

number of exclusions indicates the number of addresses in *exclusion list*. *exclusion list* should contain the addresses of any nodes you aren't interested in querying or have already queried. Don't include the node

address of a bridge in the exclusion list. If you don't want to exclude any nodes from the broadcast, set exclusion address count and exclusion address to zero.

IPXPacket and ExclusionPacketStructure can be used to create the Configuration Request Packet.

Parent Topic:

Diagnostic: General Guide

Related Topics:

Querying for Configuration Data

Configuration Reply Packet

Diagnostic Information Functions

These functions are used to return network diagnostic information.

Function	Header	Comment
GetAllKnownServers	nwdiag.h diag.h	Returns the server type and name of each server know to the specified bridge that the diagnostic application sends the request packet to.
GetBridgeDriverConfiguration	nwdiag.h diag.h	Returns the current configuration of the specified bridge driver.
GetBridgeDriverStatistics	nwdiag.h diag.h	Returns the entire driver diagnostic table of the specified bridge driver.
GetBridgeDriverStatus	nwdiag.h diag.h	Returns the status of all LAN boards installed in a bridge.
GetShellDriverConfiguration	nwdiag.h diag.h	Returns the current configuration of the specified workstation shell.
GetShellDriverStatistics	nwdiag.h diag.h	Returns the entire driver diagnostic table of the specified workstation shell.
GetOSVersionInfo	nwdiag.h diag.h	Returns the OS version of the target

		workstation.
GetShellAddress	nwdiag.h diag.h	Returns a workstation's 12-byte IPX network address.
GetShellStatistics	nwdiag.h diag.h	Returns the counters kept by the workstation shell.
GetServerAddressTable	nwdiag.h diag.h	Returns the entire connection ID table of the specified workstation.
GetServerNameTable	nwdiag.h diag.h	Returns the entire server name table as defined by the shell.
GetPrimaryServerNumbe r	nwdiag.h diag.h	Returns the primary server's number.
GetShellVersionInfo	nwdiag.h diag.h	Returns the workstation shell version number.
GetBridgeStatistics	nwdiag.h diag.h	Returns statistics pertaining to the specified bridge.
GetLocalTables	nwdiag.h diag.h	Returns the node address of each LAN board installed in the specified bridge.
GetSpecificNetworkInfo	nwdiag.h diag.h	Returns information concerning routing times and routes between the specified bridge and a specified network.
GetSpecificServerInfo	nwdiag.h diag.h	Returns information about the routing times and routes between the specified bridge and the specified server.

Parent Topic:

Diagnostic: General Guide

Diagnostic Introduction

Diagnostic provides information about the network's configuration and performance. Use this service to perform the following tasks:

Identify and query network nodes

Build a network map

Identify the software components on network nodes

Perform point-to-point performance tests

Query logical components on a network node

This information is written primarily from an assembly language programmer's view. That is, it describes how Diagnostic operates at the IPX™ and SPX™ level.

The C API library functions manage most diagnostic details for your application. This includes managing connections, building packets, and receiving results. The header nwdiag.h defines numerous structure for receiving diagnostic information.

Parent Topic:

Diagnostic: General Guide

Diagnostic Session Functions

These functions are used to manage a diagnostic session and obtain IPX and SPX performance data.

Function	Header	Comment
AbortSendingPackets	nwdiag.h diag.h	Instructs the sending node in a diagnostic point-to-point test to stop sending packets.
BeginDiagnostics	nwdiag.h diag.h	Performs the necessary initialization for the remainder of the diagnostic session.
EndDiagnostics	nwdiag.h diag.h	Terminates the connection to the target node and closes the socket.
FindComponentOffset	nwdiag.h diag.h	Searches through the component list returned either by an IPX configuration response packet or by **BeginDiagnostics**. The function returns the offset value of the

		component for which diagnostic information is wanted.
GetIPXSPXVersion	nwdiag.h diag.h	Returns the IPX and SPX version numbers of a network station.
GetIPXStatistics	nwdiag.h diag.h	Returns IPX performance statistics pertaining to a network station.
GetSPXStatistics	nwdiag.h diag.h	Returns SPX performance statistics pertaining to the network node.
StartCountingPkts	nwdiag.h diag.h	Prepares a node to participate in a point-to-point diagnostic test.
StartSendingPktsTimed	nwdiag.h diag.h	Initiates and controls a point-to-point diagnostic test.
ReturnReceivedPacketCount	nwdiag.h diag.h	Returns information about the destination node upon completing a point-to-point diagnostic test.

Parent Topic:

Diagnostic: General Guide

Querying for Configuration Data

Diagnostic relies on IPX to return configuration data about network nodes. With this information, you can build a map of the network's logical components.

Parent Topic:

Diagnostic: General Guide

Related Topics:

Configuration Request Packet

Configuration Reply Packet

Querying Specific Components

Diagnostic uses an SPX connection to return information about a node's specific software components. The following steps build a Diagnostic Request Packet.

1. The component list of the IPX Configuration Reply Packet is scanned for the target component.
2. The position of the component in the component list (first = 00h, second = 01h, etc.) is determined.
3. The destination SPX socket number is taken from the IPX Configuration Reply Packet.
4. The SPX packet header is prepared. The packet includes the following data:

   SPX header

   component number

   request type

   additional request data
5. The target component's position in *component number* is recorded, along with the request type and any additional data.
6. The SPX request packet is sent to the destination socket.

If successful, the results are returned as Diagnostic Reply Packets.

Parent Topic:

Diagnostic: General Guide

Communication Service Group

Diagnostic: Functions

AbortSendingPackets

Instructs the sending node in a diagnostic point-to-point test to stop sending packets

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Platform: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int AbortSendingPackets(
   WORD   connectionID,
   BYTE   componentNumber);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX™ Configuration Response packet (or **BeginDiagnostics** ' component list). To obtain this offset, call **FindComponentOffset**.

Remarks

AbortSendingPackets does not generate a reply packet.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

BeginDiagnostics

Performs the necessary initialization for the remainder of the diagnostics functions

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int BeginDiagnostics(
   BeginDiagnosticStruct  *destination,
   WORD                   *connectionID,
   BYTE                   *componentList);
```

Parameters

destination

(IN) Indicates the IPX address (that is, Network, Node, and Socket) destination packets are to be sent to.

connectionID

(OUT) Indicates the number assigned when the connection was established.

componentList

(OUT) Indicates the list (up to 540 bytes) of the components (bridge, file server, nondedicated IPX/SPX) on the target node. The first byte of the list is a count of the number of components listed. Each component on the list then takes one additional byte.

Return Values

0x00	Successful
0xFC	Could Not Establish Connection
0xFD	Could Not Begin Connection
0xFE	Could Not Open Socket
0xFF	General Failure

Remarks

After all diagnostic functions have been called, call **EndDiagnostics**.

EndDiagnostics

Terminates the connection to the target node and closes the socket
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>
int EndDiagnostics(
   WORD   connectionID);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

The connection is terminated by passing a connection ID.

See Also

BeginDiagnostics

FindComponentOffset

Searches through the component list returned either by an IPX Configuration Response packet or by **BeginDiagnostics**, and returns the offset value

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

BYTE FindComponentOffset(
   BYTE  *componentList,
   BYTE   componentID);
```

Parameters

componentList

(IN) Indicates the list (up to 540 bytes long) of the components (bridge, file server, and so on) on the target node.

componentID

(IN) Indicates the number associated with the software component to be queried by one of the Diagnostic Services functions.

Return Values

Offset Value	0 < offset value < 538
0xFF	General Failure

Remarks

The first byte of *componentList* is the number of components listed. Each component on the list takes one additional byte. The list is obtained from the Configuration Response packet or from **BeginDiagnostics**' *componentList*.

(0 < offset-value < 538) of the component for which diagnostics information is wanted.

componentID can be one of the following values:

0 IPX/SPX
1 Bridge Driver
2 Shell Driver
3 Shell
4 VAP Shell
5 External Bridge
6 Internal Bridge
7 Nondedicated File Server
8 Star 68000 (IPX only)
9 DOS GNMA
10 OS/2 GNMA

GetAllKnownNetworks

Returns the network address of each network known to the specified bridge
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetAllKnownNetworks(
   WORD                    connectionID,
   BYTE                    componentNumber,
   WORD                    nextNetworkOffset,
   AllResponseData         *response,
   AllKnownNetworksStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics** '*componentList*). To obtain this offset, call **FindComponentOffset**.

nextNetworkOffset

(IN) Indicates the offset number for subsequent calls to **GetAllKnownNetworks**. Should be set to 0 for the first call. If **GetAllKnownNetworks** returns a full set of 128 network addresses in reply to the first call, it should be set to 128 for the second call, and so on until the function returns a partial set of network addresses. A partial set of network addresses indicates the end of the known network address list.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to an AllKnownNetworksStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetAllKnownNetworks returns the network address of each network known to the bridge to which the diagnostic application sends the Request packet. It can return a maximum of 128 network addresses. An application can repeat **GetAllKnownNetworks** to return several sets of known network addresses.

GetAllKnownServers

Returns the server type and name of each server known to the specified bridge to which the diagnostic application sends the Request packet

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetAllKnownServers(
   WORD                    connectionID,
   BYTE                    componentNumber,
   WORD                    numberServersToSkip,
   AllResponseData        *response,
   AllKnownServersStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

numberServersToSkip

(IN) Indicates the number of servers not to count. It should be set to 0 for the first call. If **GetAllKnownServers** returns a full set of 10 server types and names in reply to the first call, *numberServersToSkip* should be set to 10 for the second call, and so on until the function returns a partial set of types and names. A partial set of types and names indicates the end of the list.

response

(OUT) Points to an *AllResponseData* structure containing the completion code and interval marker.

responseData

(OUT) Points to an AllKnownServersStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetAllKnownServers can return a maximum of 10 server types and names. An application can repeatedly call **GetAllKnownServers** to return several sets of known server types and names.

GetBridgeDriverConfiguration

Returns the current configuration of the specified bridge driver
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetBridgeDriverConfiguration(
   WORD                        connectionID,
   BYTE                        componentNumber,
   BYTE                        LANBoardNumber,
   AllResponseData            *response,
   DriverConfigurationStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

LANBoardNumber

(IN) Indicates the target LAN board (0, 1, 2, or 3).

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a DriverConfigurationStruct structure.

Return Values

0x00	Successful
0x01	Invalid Board Number
0xFF	General Failure

Remarks

GetBridgeDriverConfiguration returns the current configuration of the specified bridge driver (0, 1, 2, or 3) in the bridge to which the diagnostic application sends the Request packet. **GetBridgeDriverConfiguration** returns an error if the specified bridge driver does not exist.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetBridgeDriverStatistics

Returns the entire Driver Diagnostic Table of the specified bridge driver
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetBridgeDriverStatistics(
   WORD                    connectionID,
   BYTE                    componentNumber,
   BYTE                    LANBoardNumber,
   AllResponseData         *response,
   DriverStatisticsStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' *component list*). To obtain this offset, call **FindComponentOffset**.

LANBoardNumber

(IN) Indicates the target LAN board (0, 1, 2, or 3).

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a DriverStatisticsStruct structure.

Return Values

0x00	Successful
0x01	Invalid Board Number
0xFF	General Failure

Remarks

GetBridgeDriverStatistics returns the entire Driver Diagnostic Table of the specified bridge driver in the bridge to which the diagnostic application sends the Request packet. The table consists of a set of generic counters followed by a list of driver-dependent custom variables. Generic counters not meaningful for a particular driver are set to -1.

NOTE: Although **GetBridgeDriverStatistics** reports both actual drivers and virtual drivers, it only returns statistics for an actual board. For a virtual driver, it returns 0x01.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetBridgeDriverStatus

Returns the status of all LAN boards installed in a bridge
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetBridgeDriverStatus(
   WORD                        connectionID,
   BYTE                        componentNumber,
   AllResponseData            *response,
   BridgeDriverStatusStruct   *responseData)
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an *AllResponseData* structure containing the completion code and interval marker.

responseData

(OUT) Points to a BridgeDriverStatusStruct structure.

Return Values

0x00	Successful
0x01	Invalid Board Number
0xFF	General Failure

Remarks

GetBridgeDriverStatus returns the status of all LAN boards installed in

the bridge to which the diagnostic application sends the Request packet.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetBridgeStatistics

Returns statistics pertaining to the specified bridge
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetBridgeStatistics(
   WORD                    connectionID,
   BYTE                    componentNumber,
   AllResponseData         *response,
   BridgeStatisticsStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to anAllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a BridgeStatisticsStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetBridgeStatistics returns statistics pertaining to the bridge to which the diagnostic application sends the Request packet.

See Also

IPXSendPacket (DOS), IPXSendPacket (Win)

GetIPXSPXVersion

Returns the IPX andSPX™ version numbers of a network station
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>
int GetIPXSPXVersion(
   WORD              connectionID,
   BYTE              componentNumber,
   AllResponseData  *response,
   IPXSPXVersion    *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to an *IPXSPXVersion* structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetIPXSPXVersion returns the IPX and SPX version numbers of the network station to which the diagnostic application sends the Request packet. **GetIPXSPXVersion** does not return the version number of the target node's shell or operating system.

See Also

IPXSendPacket (DOS), IPXSendPacket (Win)

GetIPXStatistics

Returns IPX performance statistics pertaining to a network station
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetIPXStatistics(
   WORD                 connectionID,
   BYTE                 componentNumber,
   AllResponseData     *response,
   IPXStatisticsStruct *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to an IPXStatisticsStruct structure

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetIPXStatistics returns IPX performance statistics pertaining to the network station to which the diagnostic application sends the request packet.

See Also

IPXSendPacket (DOS), IPXSendPacket (Win)

GetLocalTables

Returns the node address of each LAN board installed in the specified bridge

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetLocalTables(
   WORD                connectionID,
   BYTE                componentNumber,
   AllResponseData    *response,
   LocalTablesStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a LocalTablesStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetLocalTables returns the node address of each LAN board installed in the bridge to which the diagnostic application sends the Request packet.

GetLocalTables also returns the network number corresponding to each LAN board. In the case of remote LAN-to-LAN networks, **GetLocalTables** also returns the node addresses of all virtual boards and their corresponding network numbers.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetOSVersionInfo

Return the OS version of the target workstation
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetOSVersionInfo(
   WORD              connectionID,
   BYTE              componentNumber,
   AllResponseData  *response,
   OSVersionStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the NetWare server connection ID.

componentNumber

(IN) Indicates the position of the target component within the Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to an OSVersionStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetOSVersionInfo returns information about the network station to which the diagnostic application sends the Request packet.

GetPrimaryServerNumber

Returns the primary server's number
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetPrimaryServerNumber(
   WORD                 connectionID,
   BYTE                 componentNumber,
   AllResponseData      *response,
   PrimaryServerStruct  *responseData);
```

Parameters

connectionID

Indicates the NetWare server connection ID.

componentNumber

(IN) Indicates the position of the target component within the Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a PrimaryServerStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

A diagnostic application sends this Request packet to a workstation. **GetPrimaryServerNum** returns a number (1-8) indicating the physical position of the primary server's address and name in Connection ID Table and Server Name Table respectively. *serverPositionNumber* is not

the server's Order Number returned by Return Connection ID Table.

GetServerAddressTable

Returns the entire Connection ID Table of the specified workstation
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetServerAddressTable(
   WORD                      connectionID,
   BYTE                      componentNumber,
   AllResponseData           *response,
   ServerAddressTableStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ServerAddressTableStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetServerAddressTable returns the entire Connection ID Table of the workstation to which the diagnostic application sends the Request packet. Connection ID Table consists of eight 32-byte entries. Each entry

identifies one file server.

GetServerNameTable

Returns the entire Server Name Table as currently defined by the shell
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetServerNameTable(
   WORD                    connectionID,
   BYTE                    componentNumber,
   AllResponseData        *response,
   ServerNameTableStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics** component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ServerNameTableStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetServerNameTable returns the entire Server Name Table of the workstation to which the diagnostic application sends the Request packet. Server Name 0 in the Server Name Table corresponds to entry 0

in the workstation's Connection ID Table.

GetShellAddress

Returns a workstation's 12-byte IPX internetwork address
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetShellAddress(
   WORD                 connectionID,
   BYTE                 componentNumber,
   AllResponseData     *response,
   ShellAddressStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ShellAddressStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetShellAddress returns the 12-byte IPX internetwork address the workstation's shell uses to receive packets from file servers.

GetShellDriverConfiguration

Returns the current configuration of the specified shell driver
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetShellDriverConfiguration(
   WORD                        connectionID,
   BYTE                        componentNumber,
   AllResponseData            *response,
   DriverConfigurationStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a *DriverConfigurationStruct* structure.

Return Values

0x00	Successful
0x01	Invalid Board Number
0xFF	General Failure

Remarks

GetShellDriverConfiguration returns the current configuration of the

shell driver in the workstation to which the diagnostic application sends the request packet.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetShellDriverStatistics

Returns the entire Driver Diagnostic Table of the specified shell driver
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetShellDriverStatistics(
   WORD                    connectionID,
   BYTE                    componentNumber,
   AllResponseData        *response,
   DriverStatisticsStruct *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a DriverStatisticsStruct structure.

Return Values

0x00	Successful
0x01	Invalid Board Number
0xFF	General Failure

Remarks

GetShellDriverStatistics returns the entire Driver Diagnostic Table of

the shell driver in the workstation to which the diagnostic application sends the Request packet. The table consists of a set of generic counters followed by a list of driver-dependent custom variables. Generic counters not meaningful for a particular driver are set to -1.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

GetShellStatistics

Returns the counters kept by the shell
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetShellStatistics(
   WORD                    connectionID,
   BYTE                    componentNumber,
   AllResponseData        *response,
   ShellStatisticsStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ShellStatisticsStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetShellStatistics returns the counters kept by the shell in the workstation to which the diagnostic application sends the request packet. These counters track the number of requests the shell has made to file

servers since the shell was activated. The counters also track the type of requests the shell has made and the errors the shell has handled.

GetShellVersionInfo

Returns the version number of the shell in the workstation to which the diagnostic application sends the Request packet

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetShellVersionInfo(
   WORD                 connectionID,
   BYTE                 componentNumber,
   AllResponseData     *response,
   ShellVersionStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics** component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ShellVersionStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

responseData indicates the major and minor version numbers of the target workstation's shell. For a shell version number of 1.0, 1 indicates the

major version number and 0 indicates the minor version number.

GetSpecificNetworkInfo

Returns information concerning routing times and routes between the specified bridge and a specified (destination) network

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetSpecificNetworkInfo(
   WORD                        connectionID,
   BYTE                        componentNumber,
   BYTE                       *networkAddress,
   AllResponseData            *response,
   SpecificNetworkInfoStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

networkAddress

(IN) Indicates the destination network.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a SpecificNetworkInfoStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetSpecificNetworkInfo returns information concerning routing times and routes between the bridge to which the diagnostic application sends the Request packet and one destination network one or more hops away from the bridge. If only one route exists between the bridge and the destination network, **GetSpecificNetworkInfo** returns the node address of the first interim bridge a packet meets enroute from the source bridge to the destination network. It also returns the routing time and number of hops pertaining to that route. If more than one route exists, **GetSpecificNetworkInfo** returns the node addresses of all interim bridges one hop away from the source bridge and the routing times and number of hops associated with all routes. The most efficient route appears at the top of the list in the Reply packet. The following fields refer to the other (less efficient) known routes to the specified server.

GetSpecificServerInfo

Returns information concerning routing times and routes between the specified bridge and the specified (destination) server

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetSpecificServerInfo(
   WORD                      connectionID,
   BYTE                      componentNumber,
   ServerInfoStruct          *server,
   AllResponseData           *response,
   SpecificServerInfoStruct  *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

server

(IN) Points to a ServerInfoStruct structure identifying the target server for which to get information.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a SpecificServerInfoStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetSpecificServerInfo returns information concerning routing times and routes between the bridge to which the diagnostic application sends the Request packet and one destination server one or more hops away from the bridge. If only one route exists between the bridge and the destination server, **GetSpecificServerInfo** returns the node address of the first interim bridge a packet meets enroute from the source bridge to the destination server. It also returns the routing time and number of hops pertaining to that route. If more than one route exists, **GetSpecificServerInfo** returns the node addresses of all interim bridges one hop away from the source bridge and the routing times and number of hops associated with all routes. The most efficient route appears at the top of the list in the Reply packet. The following fields refer to the other (less efficient) known routes to the specified server.

GetSPXStatistics

Returns SPX performance statistics pertaining to a network station
NetWare Servers: 2.2, 3.11, 4.0
Client Platforms: DOS, Win
Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int GetSPXStatistics(
   WORD                 connectionID,
   BYTE                 componentNumber,
   AllResponseData     *response,
   SPXStatisticsStruct *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a SPXStatisticsStruct structure

Return Values

0x00	Successful
0xFF	General Failure

Remarks

GetSPXStatistics returns SPX performance statistics pertaining to the network station to which the diagnostic application sends the Request packet.

See Also

IPXSendPacket (DOS), IPXSendPacket (Win)

ReturnReceivedPacketCount

Returns information about the destination node upon completion of a point-to-point diagnostic test

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int ReturnReceivedPacketCount(
   WORD                        connectionID,
   BYTE                        componentNumber,
   AllResponseData            *response,
   ReturnReceivedPacketStruct *responseData);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a ReturnReceivedPacketStruct structure.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

ReturnReceivedPacketCount sends a request packet to the destination node upon completion of a point-to-point diagnostic test. It returns the

number of point-to-point packets received by the destination node during the test.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**, **StartSendingPktsTimed**

StartCountingPkts

Prepares a node to participate as the destination node of a point-to-point diagnostic test

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int StartCountingPkts(
   WORD                          connectionID,
   BYTE                          componentNumber,
   AllResponseData              *response,
   StartCountingPacketsStruct   *responseData);
```

Return Values

0x00	Successful
0xFF0	General Failure

Remarks

StartCountingPkts sends a request packet resetting the counter used to track the number of point-to-point packets received by the destination node. **StartCountingPkts** also returns the socket number the application must use in the request packet of **StartSendingPktsTimed**.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**, **StartSendingPktsTimed**

StartSendingPktsTimed

Initiates and controls a point-to-point diagnostic test between the workstation to which the diagnostic application sends the Request packet and any network node

NetWare Servers: 2.2, 3.11, 4.0

Client Platforms: DOS, Win

Service: Diagnostic Services

Syntax

```
#include <nwipxspx.h>

int StartSendingPktsTimed(
   WORD                        connectionID,
   BYTE                        componentNumber,
   SendPacketsRequestStruct   *requestData,
   AllResponseData            *response,
   SendPacketsResponseStruct  *responseData,
   WORD                        ticks);
```

Parameters

connectionID

(IN) Indicates the number assigned when the connection was established.

componentNumber

(IN) Indicates the position of the target component within the IPX Configuration Response packet (or **BeginDiagnostics**' component list). To obtain this offset, call **FindComponentOffset**.

requestData

(IN) Points to a SendPacketsRequestStruct structure containing the number of packets to be sent and the timer tick interval.

response

(OUT) Points to an AllResponseData structure containing the completion code and interval marker.

responseData

(OUT) Points to a SendPacketsResponseStruct containing the number of Transmit Errors.

ticks

(IN) Indicates the maximum time limit (in units of 1/18 second) for the test.

Return Values

0x00	Successful
0xFF	General Failure

Remarks

Before calling **StartSendingPktsTimed**, the diagnostic application must call **StartCountingPkts** to another network node. After the diagnostic test is completed, the application must call **ReturnReceivedPacketCount**.

The *ticks* parameter is the maximum amount of time (in ticks) that **StartSendingPktsTimed** should wait before deciding that the test is not going to complete normally. For example, if your test is expected to take 36 ticks (2 seconds), you might set *ticks* to 72, telling **StartSendingPktsTimed** to quit sending packets and terminate if, for any reason, the test does complete in 4 seconds. If ticks is set to 0, **StartSendingPktsTimed** immediately returns with an error, terminating before any packets are sent.

See Also

IPXSendPacket (DOS), **IPXSendPacket (Win)**

Communication Service Group

Diagnostic: Structures

AddressTableStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructAddressTable {
    BYTE   serverUsed;
    BYTE   orderNumber;
    BYTE   serverNetwork[4];
    BYTE   serverNode[6];
    WORD   serverSocket;
    WORD   receivedTimeout;
    BYTE   immediateNode[6];
    BYTE   sequenceNumber;
    BYTE   connectionNumber;
    BYTE   connectionOk;
    WORD   maximumTimeout;
    BYTE   reserved[5];
} AddressTableStruct;
```

Fields

serverUsed

Specifies whether the corresponding 32-byte server entry is in use. 0x00 indicates the slot is empty. 0xFF indicates the entry is in use.

orderNumber

Specifies the order number (1-8) assigned to the corresponding server. The lowest order number indicates the server with the lowest 10-byte network/node address, assuming the first byte is the most significant. The second lowest order number indicates the second lowest address, and so on.

serverNetwork

Specifies the 4-byte network address of the associated server.

serverNode

Specifies the 6-byte node address of the network interface card installed in the associated server.

serverSocket

Specifies the file server socket number the target workstation uses to communicate with the associated server.

receivedTimeout

Specifies the value indicating the estimated round-trip time required for the target workstation to send a request packet to the file server

and receive acknowledgment.

immediateNode

Specifies the node address of the routing bridge the workstation uses to send packets to the file server.

sequenceNumber

Specifies the sequence number of the last packet the target workstation sent to the file server. Each time the workstation sends a packet to the file server, the workstation increases this counter.

connectionNumber

Specifies the connection number the workstation is using to communicate with the file server.

connectionOk

0x00 if the connection between the workstation and the server is bad.

maximumTimeout

Contains a value indicating the estimated maximum round-trip time required for the target workstation to send a request packet to the file server and receive a reply packet.

reserved

Is reserved for future use.

AllKnownNetworksStruct

Receives the addresses of known networks
Service: Diagnostic
Defined In: diag.h and nwdiag.h

AllKnownNetworksStruct

```
typedef struct StructAllKnownNetworks {
    WORD                  numberOfNetworkAddresses;
    NetworkAddressStruct  networkAddress[128];
} AllKnownNetworksStruct;
```

Fields

numberOfNetworkAddresses

Indicates how many network addresses (0 to 127) are returning in the current set.

networkAddress

Is an array of 128 structures of type NetworkAddressStruct.

Remarks

AllKnownNetworksStruct is used by **GetAllKnownNetworks**.

AllKnownServersStruct

Holds name and type information about a set of servers
Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructAllKnownServers {
    WORD              numberOfServers;
    ServerInfoStruct  serverInfoStruct[10];
} AllKnownServersStruct;
```

Fields

numberOfServers

Indicates how many server type and name combinations are contained in the current set. The maximum value is 10.

serverInfo

Is an array of 10 structures of type ServerInfoStruct.

Remarks

AllKnownServersStruct is used by **GetAllKnownServers**.

AllResponseData

Contains the completion code and interval marker of an operation
Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructAllResp {
    BYTE   completionCode;
    long   intervalMarker;
} AllResponseData;
```

Fields

completionCode

Contains the completion code of an operation.

intervalMarker

Contains the interval marker for an operation.

Remarks

AllResponseData is used by most of the Diagnostic Services functions.

BeginDiagnosticStruct

is a type definition for IPXAddress
Service: Diagnostic
Defined In: diag.h and nwdiag.h

Remarks

BeginDiagnosticStruct is used by **BeginDiagnostics**.

BridgeDriverStatusStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructBridgeDriverStatus {
    status   LANBoard[4];
} BridgeDriverStatusStruct;
```

Fields

status

Indicates the status of the corresponding LAN board.

Remarks

Each LANBoard returns one of the following values to indicate the status of the corresponding LAN board:

```
0x00    The board is alive and running
0x01    The board does not exist
0x02    The board is dead
```

BridgeStatisticsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructBridgeStatistics {
    WORD    tooManyHopsCount;
    WORD    unknownNetworkCount;
    WORD    noSpaceForServiceCount;
    WORD    noReceiveBuffersCount;
    WORD    notMyNetwork;
    long    netBIOSPropogateCount;
    long    totalPacketsServiced;
    long    totalPacketsRouted;
} BridgeStatisticsStruct;
```

Fields

tooManyHopsCount

Specifies the number of times (since the bridge was initialized) the bridge received packets on the fifteenth hop across an internetwork bridge. These packets are discarded.

unknownNetworkCount

Specifies the number of times (since the bridge was initialized) the bridge has received packets bound for an unknown network. These packets are discarded.

noSpaceForServiceCount

Specifies the number of times (since the bridge was initialized) the bridge has received internetwork packets it could not accommodate because the router did not have enough space in its DGroup area to copy the packets. These packets are lost.

noReceiveBuffersCount

Specifies the number of times (since the bridge was initialized) the bridge could not receive inbound packets because of inadequate buffer space. These packets are lost.

notMyNetwork

Specifies the number of incoming packets with a destination other than LAN A. (See **GetSpecificNetworkInfo** for information about identifying LAN boards.)

netBIOSPropogateCount

Specifies the number of times the bridge received NetBIOS broadcasts since it was initialized.

totalPacketsServiced

Specifies the total number of packets the bridge serviced since the bridge was initialized.

totalPacketsRouted

Specifies the total number of packets the router actually routed.

Remarks

BridgeStatisticsStruct is used by **GetBridgeStatistics**.

DriverConfigurationStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructDriverConf {
    BYTE   networkAddress[4];
    BYTE   nodeAddress[6];
    BYTE   LANMode;
    BYTE   nodeAddressType;
    WORD   maxDataSize;
    WORD   reserved1;
    BYTE   LANHardwareID;
    WORD   transportTime;
    BYTE   reserved2[11];
    BYTE   majorVersion;
    BYTE   minorVersion;
    BYTE   ethernetFlagBits;
    BYTE   selectedConfiguration;
    BYTE   LANDescription[80];
    WORD   IOAddress1;
    WORD   IODecodeRange1;
    WORD   IOAddress2;
    WORD   IODecodeRange2;
    BYTE   memoryAddress1[3];
    WORD   memoryDecodeRange1;
    BYTE   memoryAddress2[3];
    WORD   memoryDecodeRange2;
    BYTE   interruptIsUsed1;
    BYTE   interruptLine1;
    BYTE   interruptIsUsed2;
    BYTE   interruptLine2;
    BYTE   DMAIsUsed1;
    BYTE   DMALine1;
    BYTE   DMAIsUsed2;
    BYTE   DMALine2;
    BYTE   microChannelFlagBits;
    BYTE   reserved3;
    BYTE   textDescription[80];
} DriverConfigurationStruct;
```

Fields

networkAddress

4-byte network address of the LAN on which the driver communicates.

nodeAddress

6-byte node address of the LAN board corresponding to the target bridge driver. The node address uniquely identifies the driver and board on the network specified in the preceding field. If the node address is less than 6 bytes long (for example, 0x24E0), the address appears as follows: 00 00 00 00 24 E0

LANMode

1-byte field with the following bits defined:

Bit 0: 0=Place-holding dummy driver; 1=Real driver

Bit 1: Reserved; must be 0.

Bit 2: 0=Not 100% guaranteed driver; 1=100% guaranteed driver

Bit 3: Reserved; must be 0.

Bit 4: Reserved; must be 0.

Bit 5: Reserved; must be 0.

Bit 6: Reserved; must be 0.

Bit 7: 0=Driver uses DMA; no receive block stradles 64k physical address boundary 1=Driver does not use DMA

nodeAddressType indicates the following:

Who (the driver, the developer, or a configuration utility) records the node address in the driver to match the node address setting of the LAN board

How the node address is recorded in the driver

When the node address is recorded in the driver

The following values are defined:

0x00

The driver dynamically reads and records the node address by calling *DriverInitialize*

0x01

The developer hard codes the node address in the driver code Master Configuration Table

0x02

A configuration utility assigns the node address

maxDataSize

maximum size of a packet's data portion for the target driver. The data portion's maximum size is always 64 bytes less than the packet size advertised with the LAN board. (The packet header and control information require 64 bytes.) The following table shows the maximum size of data portion for four packets:

Largest Maximum Transmittable Packet	Data Size
576 bytes	512 bytes
1,088 bytes	1,024 bytes
2,112 bytes	2,048 bytes
4,160 bytes	4,096 bytes

reserved1

address information pertinent only to the driver.

LANHardwareID

whose value is hard-coded into the Master Configuration Table, uniquely identifies the LAN hardware. The OEM/Driver Support Group Manager at Novell® assigns this ID.

transportTime

value indicating the speed of the LAN associated with the target driver and board. Speed is measured by the amount of time it takes a 576-byte packet to travel from one node on the LAN to another node. Time is measured in units of 1/18 second and rounded to the next highest 1/18.

reserved2

reserved for future use. Currently, the last 2 bytes of this field identify the Ethernet type (if applicable).

majorVersion and *minorVersion*

identify the major and minor versions of the driver release; they do not identify the installed NetWare® version. Therefore, in a station running NetWare 2.1, the driver version may be 1.0, where 1 represents the major version and 0 represents the minor version.

selectedConfiguration

value indicating which hardware configuration in the Hardware Configuration Table the driver is using. The value ranges from 0 to n - 1, where n is the maximum number of configurations supported by the driver.

ethernetFlagBits

significant only for Ethernet drivers using the Ethernet protocol (not the IEEE 802.3 protocol). Xerox assigned a value of 0x8137 to Novell where 81 is the high-order byte and 37 is the low-order byte. Only drivers with identical Ethernet types can communicate. *ethernetFlagBits* is a 1-byte field with the following bits defined:

Bit 0: Etherlink

Bit 1: IEEE 802.3 protocol

Bit 2: 0=Driver can run in protocol mode; 1=Driver runs only in real mode on 286-based machines

Bit 3: Reserved; must be 0.

Bit 4: 0=Non-Ethernet or non-configurable board driver;
1=Configurable Ethernet board driver

Bit 5: Reserved; must be 0.

Bit 6: Reserved; must be 0.

Bit 7: Reserved; must be 0.

LANDescription

NULL-terminated string of not more than 69 bytes. The string lists the LAN hardware supported by the driver. The following is a short example:

```
NetWare RX-NET
```

IOAddress1

address of a block of I/O addresses to be decoded by the LAN board. Zeros returned in the second I/O address field indicate the driver is not using the field.

IODecodeRange1

number of paragraphs in its corresponding *memoryAddress* 's block.

memoryAddress

address of a block of memory address space to be decoded by the LAN board. The block is divided into one or more 16-byte paragraphs. This is a 3-byte address field. The first byte is the high-order byte. The remaining *uword* makes up the low-order portion of the address. Zeros returned in the second *memoryAddress* indicate the driver is not using the field.

memoryDecodeRange

following its corresponding *memoryAddress* indicates the number of ports to be decoded. Typically this value is 8, 16, or 32.

interruptIsUsed

whether the value in the following Interrupt Line field is valid. The following values can appear in *interruptIsUsed*:

0x00

No interrupt line defined

0xFF

Interrupt line defined for exclusive use

0xFE

Interrupt line defined for a particular LAN board but can be shared by others of the same type

interruptLine

value of the interrupt used by the LAN board. Zeros returned in the second *interruptLine* indicate the LAN board does not use the field.

DMAIsUsed

value in the following *DMALine* is valid. The following values can

appear in a *DMAIsUsed* :

0x00

No DMA line defined

0xFF

DMA line defined for exclusive use

0xFE

DMA line defined for a particular LAN board but may be shared by others of the same type

DMALine

value of the DMA line used by the LAN board. Zeros returned in the second *DMALine* indicate the LAN board does not use the field.

microChannelFlagBits

microchannel support for the configuration. The following bits are defined:

Bit 0: 0 or 1, defined as follows:

0=The configuration does not use microchannel.
1=The configuration uses microchannel but cannot be combined with other configurations that do not use microchannel.

Bit 1: If set, this configuration uses microchannel and can be combined with other configurations regardless of whether they use microchannel.

Bits 2-7: Undefined

nodeAddressType

indicates *reserved3* is for future use.

textDescription

NULL-terminated string of not more than 69 bytes. The string summarizes the configuration information contained in the preceding fields. The following is a short example of how a text description may appear in driver code:

```
I/O Base = 0x2E0, RAM at D000:0 for 0x800 bytes,IRQ 2, No DMA
```

Remarks

DriverConfigurationStruct is used by **GetShellDriverConfiguration**.

DriverStatisticsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructDriverStat {
    BYTE   driverVersion[2];
    BYTE   statisticsVersion[2];
    long   totalTxPacketCount;
    long   totalRxPacketCount;
    WORD   noECBAvailableCount;
    WORD   packetTxTooBigCount;
    WORD   packetTxTooSmallCount;
    WORD   packetRxOverflowCount;
    WORD   packetRxTooBigCount;
    WORD   packetRxTooSmallCount;
    WORD   packetTxMiscErrorCount
    WORD   packetRxMiscErrorCount;
    WORD   retryTxCount;
    WORD   checksumErrorCount;
    WORD   hardwareRxMismatchCount;
    WORD   numberOfCustomVariables;
    BYTE   variableData[495];
} DriverStatisticsStruct;
```

Fields

driverVersion

Specifies an array of LAN driver versions for the specified LAN board.

statisticsVersion

Specifies an array of major and minor version numbers a developer assigns and updates each time the developer modifies the Driver Diagnostic Table.

totalTxPacketCount

Specifies the number of packets the driver has successfully transmitted since the last driver reset or initialization.

totalRxPacketCount

Specifies the number of packets the driver has successfully received and passed into the system since the last reset or initialization.

noECBAvailableCount

Specifies the number of packets the driver has received (since the last reset or initialization) for which there was no listening ECB.

packetTxTooBigCount

Specifies the number of times (since the last reset or initialization) applications have asked the driver to send a packet over the maximum legal size.

packetTxTooSmallCount

Specifies the number of times (since the last reset or initialization) applications have asked the driver to send a packet under the minimum legal size.

packetRxOverflowCount

Specifies the number of times (since the last reset or initialization) the driver has received a packet larger than the buffer space allocated for the packet.

packetRxTooBigCount

Specifies the number of times (since the last reset or initialization) the driver has received a packet over the maximum legal size.

packetRxTooSmallCount

Specifies the number of times (since the last reset or initialization) the driver has received a packet under the minimum legal size.

packetTxMiscErrorCount

Specifies the number of miscellaneous errors preventing the driver from transmitting a packet (since the last reset or initialization).

packetRxMiscErrorCount

Specifies the number of miscellaneous errors preventing the driver from receiving a packet (since the last reset or initialization).

retryTxCount

Specifies the number of times (since the last reset or initialization) the driver resent a packet. For example, when the driver detects a collision, the driver resends a packet.

checksumErrorCount

Specifies the number of checksum errors occurring while receiving packets (since the last reset or initialization).

hardwareRxMismatchCount

Specifies the number of times (since the last reset or initialization) the hardware has received more or fewer bytes than expected.

numberOfCustomVariables

Specifies the number of custom variables following.

variableData

Each byte in *variableData* specifies information pertinent to the particular driver. It is optional.

Remarks

DriverStatisticsStruct is used by **GetShellDriverStatistics**.

IPXAddress

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct {
   BYTE   network[4];
   BYTE   node[6];
   BYTE   socket[2];
} IPXAddress;
```

Fields

network

node

socket

IPXSPXVersion

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructIPXSPXVersion {
    BYTE  IPXMajorVersion;
    BYTE  IPXMinorVersion;
    BYTE  SPXMajorVersion;
    BYTE  SPXMinorVersion;
} IPXSPXVersion;
```

Fields

IPXMajorVersion

The major version of IPX.

IPXMinorVersion

The minor version of IPX.

SPXMajorVersion

The major version of SPX.

SPXMinorVersion

The minor version of SPX.

Remarks

IPXSPXVersion is used by **GetIPXSPXVersion**.

IPXStatisticsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructIPXStatistics {
    long   sendPacketCount;
    WORD   malformedPacketCount;
    long   getECBRequestCount;
    long   getECBFailureCount;
    long   AESEventCount;
    WORD   postponedAESEventCount;
    WORD   maxConfiguredSocketsCount;
    WORD   maxOpenSocketsCount;
    WORD   openSocketFailureCount;
    long   listenECBCount;
    WORD   ECBCancelFailureCount;
    WORD   findRouteFailureCount;;
} IPXStatisticsStruct;
```

Fields

sendPacketCount

Indicates the number of times (since IPX was loaded) applications have called IPX to send a packet.

malformedPacketCount

Indicates the number of times (since IPX was loaded) applications have passed malformed packets to IPX. A packet is malformed if the value in its ECB's *fragmentCount* field is 0 or if the value of the size field within ECB's first *fragmentDescriptor* is less than 30 bytes.

getECBRequestCount

Indicates the number of times (since IPX was loaded) IPX has supplied Receive ECBs for incoming packets.

getECBFailureCount

Indicates the number of times (since IPX was loaded) IPX has been unable to supply a Receive ECB for an incoming packet.

AESEventCount

Indicates the number of times (since IPX was loaded) IPX has used the AES to schedule an event.

postponedAESEventCount

Indicates the number of times (since IPX was loaded) IPX has been unable to service an AES event on time. For example, IPX cannot send

an outgoing packet to a driver busy with another packet.

maxConfiguredSocketsCount

Indicates the maximum number of open sockets possible on the target node. (This value is configurable.)

maxOpenSocketsCount

Indicates the maximum number of sockets open simultaneously since IPX was loaded.

openSocketFailureCount

Indicates the number of times (since IPX was loaded) applications have unsuccessfully called **IPXOpenSocket**. IPX cannot open a socket if the socket table is full or if the socket is already open.

listenECBCount

Indicates the number of times (since IPX was loaded) applications have given IPX a Listen ECB.

ECBCancelFailureCount

Indicates the number of times (since IPX was loaded) IPX has been unable to cancel an ECB. For example, IPX cannot cancel an ECB if the driver and the ECB have entered a critical section just prior to sending a packet. In this case, the cancellation is too late.

findRouteFailureCount

Indicates the number of times (since IPX was loaded) IPX has been unable to find a route to a requested network address.

Remarks

IPXStatisticsStruct is used by **GetIPXStatistics**.

LocalTablesStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructLocalTables {
    NumberStruct        localNetworkNumber[16];
    NodeAddressStruct   localNodeAddress[16];
} LocalTablesStruct;
```

Fields

localNetworkNumber

Specifies an array of 16 structures of type NumberStruct.

Specifies the network defined for each LAN board. Normally, a bridge connects from one to four networks. However, in the case of remote LAN-to-LAN networks involving virtual LAN boards with their associated network numbers, a bridge can connect up to 16 networks.

localNodeAddress

Specifies an array of 16 structures of type NodeAddressStruct.

localNodeAddress specifies the LAN board installed in the bridge. In the case of LAN-to-LAN networks, the node address also identifies virtual LAN boards. Although it is 8 bytes long, the actual node address is only 6 bytes long.

Remarks

LocalTablesStruct is used by **GetLocalTables**.

NetworkAddressStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructNetworkAddress {
    BYTE   address[4];
} NetworkAddressStruct;
```

Fields

address

Holds one 4-byte network address.

NodeAddressStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructNodeAddress {
    BYTE   address[6]
    BYTE   reserved[2];
} NodeAddressStruct;
```

Fields

address

The address of the node.

reserved

Is reserved.

NumberStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructNumber {
    BYTE   number[4];
} NumberStruct;
```

Fields

number

Holds the number of a network.

OSVersionStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructOSVersion {
    BYTE   machineID;
    BYTE   versionData[41];
} OSVersionStruct;
```

Fields

machineID

Returns a value of 0x00 if the target machine is an IBM PC computer or compatible.

versionData

Remarks

OSVersionStruct is used by **GetOSVersionInfo**.

ReturnReceivedPacketStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructReturnReceivedPacket {
    WORD   packetsReceived;
} ReturnReceivedPacketStruct;
```

Fields

packetsReceived

It returns the number of point-to-point packets received by the destination node during a test.

Remarks

ReturnReceivedPacketStruct is used by **ReturnReceivedPacketCount**.

RoutingInfoStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructRoutingInfo {
    BYTE   routerForwardingAddress[6];
    BYTE   routerBoardNumber;
    BYTE   reserved[2];
    BYTE   routeHops;
    BYTE   routeTime;
} RoutingInfoStruct;
```

Fields

routerForwardingAddress

Indicates the node address of the LAN board receiving packets from the source bridge being queried.

routerBoardNumber

Indicates which LAN board (inside a router) to which packets should be routed. 0x00 indicates LAN A; 0x01 indicates LAN B, and so on.

reserved

Is not pertinent to **GetSpecificNetworkInfo**.

routeHops

Indicates the number of hops a packet makes traveling between the source bridge and the destination network on its associated route.

routeTime

Indicates the time it takes a packet to travel between the source bridge and the destination network on that particular route. If more routers exist, information pertaining to these routers appears next on the list.

RouteSourceInfoStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructRouteSourceInfo {
    BYTE   routeSourceAddress[6];
    WORD   routeHopsToSource;
    BYTE   reserved[2];
} RouteSourceInfoStruct;
```

Fields

routeSourceAddress

Indicates the node address of a LAN board (inside a router) capable of routing the received packet to the server specified. Up to 47 routes can be returned.

routeHopsToSource

Indicates the number of hops a packet makes traveling between the source bridge and the destination server specified.

reserved

Is not pertinent to **GetSpecificServerInfo**.

SendPacketsRequestStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct SPReq {
    BeginDiagnosticStruct   target;
    BYTE                    immediateAddress[6];
    WORD                    numberOfPackets;
    BYTE                    timerTickInterval;
    BYTE                    packetsPerTickInterval;
    WORD                    packetSize;
    WORD                    changeSize;
} SendPacketsRequestStruct;
```

Fields

target

Indicates a structure of type BeginDiagnosticStruct.

immediateAddress

Indicates the node address of the first bridge a packet encounters as it travels from the source node to the socket node. If the application sets this field to 0xFFFFFFFFFFFF (-1), IPX chooses a bridge.

numberOfPackets

Indicates the total number of packets to be sent to the destination node during the diagnostic test.

timerTickInterval

Indicates how often the source node should send a specified number of packets to the destination node. The *timerTickInterval* is measured in units of 1/18 second.

packetsPerTickInterval

Indicates how many packets the source node should send to the destination node as each send interval expires. If the send (tick) interval is 3 and the packets per tick interval is 5, the source node sends 5 packets every 3/18 second.)

packetSize

Indicates the size of the first packet to be sent. The size must be between 30 and 512 bytes inclusive. If the packet size shrinks below 30 bytes or grows beyond 512, IPX automatically adjusts the size to a valid value.

changeSize

Indicates a value to increase or decrease the size of the next packet.

This feature allows the packet size to vary during the diagnostic test.

Remarks

SendPacketsRequestStruct is used by **StartSendingPktsTimed**.

SendPacketsResponseStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct SPResp {
    WORD  numberOfTransmitErrors;
} SendPacketsResponseStruct;
```

Fields

numberOfTransmitErrors

Remarks

SendPacketsResponseStruct is used by **StartSendingPktsTimed**.

ServerAddressTableStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructServerAddressTable {
    AddressTableStruct   addressTable[8];
} ServerAddressTableStruct;
```

Fields

addressTable

An array of structures of type AddressTableStruct.

Remarks

ServerAddressTableStruct is used by **GetServerAddressTable**.

ServerInfoStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StrSrvrInfo {
    WORD   serverType;
    BYTE   serverName[48];
} ServerInfoStruct;
```

Fields

serverType

The object type of the server whose name appears in the *serverName* field. A file server's object type is 0x0004. A print server's object type is 0x0003.

serverName

A 48-byte NULL-terminated string containing the name of the server whose object type appears in the serverType field.

Remarks

ServerInfoStruct is used by **GetSpecificServerInfo**.

ShellAddressStruct

is a type definition of IPXAddress
Service: Diagnostic
Defined In: diag.h and nwdiag.h

Remarks

ShellAddressStruct is used by **GetShellAddress**.

ShellStatisticsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructShellStatistics {
    long   shellRequestsCount;
    WORD   operatorAbortsCount;
    WORD   operatorRetriesCount;
    WORD   timeoutsCount;
    WORD   writeErrorCount;
    WORD   invalidReplyHeaderCount;
    WORD   invalidSlotCount;
    WORD   invalidSequenceNumberCount;
    WORD   errorReceivingCount;
    WORD   noRouterFoundCount;
    WORD   beingProcessedCount;
    WORD   unknownErrorCount;
    WORD   invalidServerSlotCount;
    WORD   networkGoneCount;
    WORD   reserved1;
    WORD   allocateCannotFindRouteCount;
    WORD   allocateNoSlotsAvailableCount;
    WORD   allocateServerIsDownCount;
} ShellStatisticsStruct;
```

Fields

shellRequestsCount

Indicates the number of times (since the shell was activated) the shell has made requests to a file server.

operatorAbortsCount

Indicates the number of times (since the shell was activated) the user has aborted the shell-server connection by entering "A" in reply to a "Network Error" message.

operatorRetriesCount

Indicates the number of times (since the shell was activated) the user has instructed the shell to retry an operation.

timeoutsCount

Indicates the number of times (since the shell was activated) the shell sent a request to a server and then timed out without receiving a reply. (Normally, the shell sends another request packet.)

writeErrorCount

Indicates the number of times (since the shell was activated) the driver has been unable to send a request to a file server (after repeated retries). In this case, the shell displays the message "Error writing to network" on the workstation screen. The shell does not increment this counter if, after repeated retries, the driver is able to send the request.

invalidReplyHeaderCount

Indicates the number of times (since the shell was activated) the shell has received a reply packet header whose *Checksum* was -1 or whose *PacketType* indicated the packet was not a file server reply.

invalidSlotCount

Indicates the number of times (since the shell was activated) the shell has received a file server reply packet specifying an incorrect connection ID.

invalidSequenceNumberCount

Indicates the number of times (since the shell was activated) the shell received a file server reply packet specifying an incorrect sequence number. It usually indicates the reply was unnecessary.

errorReceivingCount

Indicates the number of times (since the shell was activated) IPX has indicated an error even though a packet was received on the socket. It usually indicates an "Overrun" error.

noRouterFoundCount

Indicates the number of times (since the shell was activated) the shell tried and failed to find a route to a destination node. The shell attempts to reroute a packet when a connection seems to fail and the user requests a "Retry."

beingProcessedCount

Indicates the number of times (since the shell was activated) the shell received a "being processed" reply from a file server. A file server sends this reply to a shell when the server, while processing the shell's request, receives duplicate requests from the shell for the same service.

unknownErrorCount

Indicates the number of times (since the shell was activated) the shell received a packet containing an undefined error value.

invalidServerSlotCount

Indicates the number of times the shell attempted to communicate on a particular client connection number, and the server indicated the connection number is invalid.

networkGoneCount

Indicates the number of times (since the shell was activated) the shell received a packet from a file server indicating the target network has gone away. Only a 68000 file server can generate this kind of packet.

reserved1

Is reserved for future use.

allocateCannotFindRouteCount

Indicates the number of times (since the shell was activated) the shell, asked by an application to establish a connection with a file server, could not find a route to the destination network.

allocateNoSlotsAvailablseCount

Indicates the number of times (since the shell was activated) the shell, asked by an application to establish a connection with a file server, could not establish the connection because the file server's connection table was full.

allocateServerIsDownCount

Indicates the number of times (since the shell was activated) the shell, asked by an application to establish a connection with a file server, could not establish the connection because the target file server was down.

Remarks

ShellStatisticsStruct is used by **GetShellStatistics**.

ShellVersionStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructShellVersion {
    BYTE                minor;
    BYTE                major;
    BYTE                rev;
} ShellVersionStruct;
```

Fields

minor

The minor version of the shell.

major

The major version of the shell.

rev

The revision of the shell.

Remarks

For a shell version number of 1.0, 1 indicates the major version number and 0 indicates the minor version number.

ShellVersionStruct is used by **GetShellVersionInfo**.

SpecificNetworkInfoStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructSpecificNetInfo {
    BYTE                networkAddress[4];
    BYTE                hopsToNet;
    BYTE                reservedA[7];
    WORD                routeTimeToNet;
    WORD                numberOfKnownRouters;
    RoutingInfoStruct   routingInfo[MAX_ROUTES];
} SpecificNetworkInfoStruct;
```

Fields

networkAddress

Indicates the network address of the destination network. It is the same address the application passes in the Request packet's Specific Network Address.

hopsToNet

Indicates the number of hops (between the source bridge and the destination network) of the most efficient route. This value and the value returned in *routeHops* are the same.

reservedA

Indicates information not pertinent to **GetSpecificNetworkInfo**.

routeTimeToNet

Indicates the route time (between the source bridge and the destination network) of the most efficient route. This value and the value returned in *routeTime1* are the same.

numberOfKnownRouters

Indicates the number of routes between the source bridge and the destination network. If only one route exists, it returns 0x01.

routingInfo

Indicates an array of RoutingInfoStruct structures.

Remarks

SpecificNetworkInfoStruct is used by **GetSpecificNetworkInfo**.

SpecificServerInfoStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StrSpecSrvrInfo {
    ServerInfoStruct         serverInfo;
    BYTE                     serverAddress[12];
    WORD                     hopsToServer;
    BYTE                     reserved1[2];
    WORD                     numberOfRoutes;
    RouteSourceInfoStruct    routeSourceInfo[MAX_ROUTES];
} SpecificServerInfoStruct;
```

Fields

serverInfo

Indicates a structure of type ServerInfoStruct. Must be set prior to calling **GetSpecificServerInfo**.

serverAddress

Indicates the destination server's 12-byte internetworkaddress.

hopsToServer

Indicates the number of hops (between the source bridge and the destination server) of the most efficient route.

reserved1

Is not pertinent to **GetSpecificServerInfo**.

numberOfRoutes

Indicates the number of routes between the source bridge and the destination server. If only one route exists, it returns 0x01.

routeSourceInfo

Indicates an array of structures of type RouteSourceInfoStruct.

Remarks

SpecificServerInfoStruct is used by **GetSpecificServerInfo**.

SPXStatisticsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructSPXStatistics {
    WORD   maxConnectionsCount;
    WORD   maxUsedConnectionsCount;
    WORD   establishConnectionRequest;
    WORD   establishConnectionFailure;
    WORD   listenConnectionRequestCount;
    WORD   listenConnectionFailureCount;
    long   sendPacketCount;
    long   windowChokeCount;
    WORD   badSendPacketCount;
    WORD   sendFailureCount;
    WORD   abortConnectionCount;
    long   listenPacketCount;
    WORD   badListenPacketCount;
    long   incomingPacketCount;
    WORD   badIncomingPacketCount;
    WORD   suppressedPacketCount;
    WORD   noSessionListenECBCount;
    WORD   watchdogDestroySessionCount;
} SPXStatisticsStruct;
```

Fields

maxConnectionsCount

Indicates the maximum number of SPX connections possible on the target node. (It is configurable.)

maxUsedConnectionsCount

Indicates the maximum number of SPX connections open simultaneously since IPX was loaded.

establishConnectionRequest

Indicates the number of times (since SPX was loaded) applications called **SPXEstablishConnection**.

establishConnectionFailure

Indicates the number of times (since SPX was loaded) **SPXEstablishConnection** calls failed because the SPXPacketHeader was too small, the SPX Connection Table was full, or no router was found to the target network.

listenConnectionRequestCount

Indicates the number of times (since SPX was loaded) applications called **SPXListenForConnection**.

listenConnectionFailureCount

Indicates the number of times (since SPX was loaded) **SPXListenForConnection** calls failed because the SPX Connection Table was full.

sendPacketCount

Indicates the number of times (since SPX was loaded) applications called **SPXSendSequencedPacket**.

windowChokeCount

Indicates the number of times (since SPX was loaded) SPX failed to send a packet because the target station had not allocated a receive buffer.

badSendPacketCount

Indicates the number of times (since SPX was loaded) **SPXSendSequencedPacket** was incorrectly called by passing an invalid connection ID or bypassing the address of an ECB indicating a packet header size of less than 42 bytes.

sendFailureCount

Indicates the number of times (since SPX was loaded) SPX was unable to send a packet across an SPX connection and receive acknowledgment. In such a case, SPX aborts the connection and informs the calling application.

abortConnectionCount

Indicates the number of times (since SPX was loaded) **SPXAbortConnection** was called.

listenPacketCount

Indicates the number of times (since SPX was loaded) applications gave listen ECBs to SPX.

badListenPacketCount

Indicates the number of times (since SPX was loaded) applications passed SPX a packet whose associated ECB is malformed. An ECB is malformed if the value in *FragmentCount* is 0, if the value in its first *FragmentDescriptorSize* is less than 42, or if the listen socket is not open.

incomingPacketCount

Indicates the number of times (since SPX was loaded) the node's driver gave an incoming packet to SPX.

suppressedPacketCount

Indicates the number of times (since SPX was loaded) SPX discarded inbound packets because they were duplicates of previously received packets.

noSessionListenECBCount

Indicates the number of times (since SPX was loaded) SPX was forced

to discard an inbound *SPXEstablishConnection* packet because SPX lacked a corresponding *SPXListenForConnection* ECB.

watchdogDestroySessionCount

Indicates the number of times (since SPX was loaded) watchdog destroyed a connection because the connection was no longer valid.

Remarks

SpxStatisticsStruct is used by **GetSPXStatistics**.

StartCountingPacketsStruct

Service: Diagnostic
Defined In: diag.h and nwdiag.h

Structure

```
typedef struct StructStartCountingPackets {
    WORD   destinationSocket;
} StartCountingPacketsStruct;
```

Fields

destinationSocket

The destination node to which packets will be sent.

Remarks

StartCountingPacketsStruct is used by **StartCountingPkts**.

Internet Network Library

Communication Service Group

Internet Network Library: Guides

Internet Network Library: General Guide

Internet Network Library Overview

Internet Network Library Function List

Internet and NetWare

Additional Links

Internet Network Library: Functions

Internet Network Library: Structures

Internet Network Library: Concepts

Internet and NetWare

While the Internet support functions emulate the corresponding 4.3BSD functions as closely as possible, the NetWare® OS environment requires some changes from the 4.3BSD specifications. As an application programmer, you must be aware of these differences while porting TCP/IP applications from the UNIX* to the NetWare OS, and also when designing new NetWare TCP/IP applications.

The BSD Internet support functions assume a single thread per data space; in contrast, the NetWare environment allows multiple threads to share the same data space. Therefore, the NetWare Internet support functions require a more flexible association of context with support functions. To provide this flexibility, each function requiring context takes a context block explicitly from the caller instead of allocating it implicitly.

To avoid confusion between the functions in the 4.3BSD environment and the NetWare environment, the NetWare function names are prefixed with either **NW** or **NetDB**. For example, the NetWare versions of **gethostbyname** are called **NWgethostbyname** and **NetDBgethostbyname**, and their first argument is a pointer to a context block. Your application allocates a structure of type nwsockent and passes the pointer to either **NWgethostbyname** or **NetDBgethostbyname**. The application must allocate a context block for each different context it requires. Each context normally corresponds to a separate process in the UNIX system.

Your application does not look at a nwsockent context block explicitly. The support functions maintain this structure and extract information from it. Each context block can remember the file position of each of the four database files. Whenever a support function returns the address of a file entry, it points to a location in the context block provided in the call. This is the same area for each type of support function, so the information in this area remains valid until the next call is made, regardless of the category.

Your application must take care of the following:

It must initialize the context block with zeros before calling any of the support functions for the first time.

It must not modify the context block later, since the library preserves some context information in the context block. Any such modifications can result in unpredictable behavior.

The sequence of operations that can be allowed depend on the context

block that is being passed, not on the thread that is calling. For example, in two subsequent calls to **NWgethostent** on the **same** thread, each instance of the call is associated with a different context block and results in the return of the same entry from the HOSTS file.

If the application uses the "NetDB" functions (that is, **NetDBgethostbyname** rather than **NWgethostbyname**), each thread must maintain its own separate context block. The "NetDB" functions maintain low-level information on a per-thread basis, and it is important that each thread not share the context block with any other thread or unexpected results can occur.

In the NetWare implementation, h_errno is actually a C preprocessor macro that behaves like a normal variable. However, your application must not declare h_errno as an extern int.

The 4.3BSD support functions are defined as macros in NETDB.H. These macros have been defined to help port simple applications involving a single thread and a single context. Applications with multiple threads must not call upon these macros, since they could call upon the support functions concurrently. These macros expand to either functions prefixed with **NW** (such as **NWgethostent**) or **NetDB** (such as **NetDBgethostent**).

The functions prefixed with **NW** access only the local hosts file (SYS:ETC\HOSTS). No Internet host access is provided with these functions.

The functions prefixed with **NetDB** also use SYS:ETC\HOSTS, but in addition provide transparent access to DNS and/or NIS databases if they are installed on the system and available on the network. For the host lookup functions (**gethostbyname** and **gethostbyaddr**), the search order is always local file first, followed by DNS, then NIS. For sequential access of the hosts database (**gethostent**), all of the hosts in SYS:ETC\HOSTS are returned first, followed by all the entries in the NIS hosts.byaddr map, if available. Use of these functions requires NETDB.NLM. NETDB.NLM is included with the NetWare SDK. Contact Novell for information about distribution of this NLM.

Support for DNS requires a RESOLV.CFG file in the SYS:ETC directory on the NetWare server in the standard format of /etc/resolv.conf on UNIX systems. A typical RESOLV.CFG follows:

```
domain dnsdomain.com          ;name of the DNS domain
nameserver 130.57.1.1         ;primary name server
nameserver 130.57.1.2         ;secondary name server
```

Support for NIS requires the NIS binder, called NISBIND.NLM, which is provided with certain Novell TCP/IP interoperability products. Once the NIS binder is installed and configured on the system, the application gains NIS host access automatically.

For backwards compatibility, the 4.3BSD function names are macros expanding to the functions that only access the local files (that is, **gethostbyname**

to expand to the internet functions (**NetDBgethostbyname**), the application needs to define the symbol **NETDB_USE_INTERNET** before it includes NETDB.H. All of the internet functions (prefixed by **NetDB**) require NETDB.NLM on the server before the application NLM can be loaded.

To use these macros, your application must include the **NETDB_DEFINE_CONTEXT** macro, which declares a single context block named nwSocketCtx. For each support function, a macro bearing the original 4.3BSD name calls upon the corresponding NetWare Internet support function and supplies the nwSocketCtx block. It is sufficient for your application to call upon **NETDB_DEFINE_CONTEXT** in any of its source modules that include NETDB.H.

The Internet address conversion function **inet_ntoa** has been renamed **NWinet_ntoa** to reflect the NetWare version of this function. Your application can use **NWinet_ntoa** when multiple threads attempt to convert address information. The **inet_ntoa** function is defined as a macro in ARPA/ INET.H. Use this function for simple applications involving single threads. Your application must call upon the **NETINET_DEFINE_CONTEXT** macro in any one of the source modules that include ARPA/INET.H.

Detailed information on each function can be found in Internet Network Library: Functions.

Parent Topic: Internet Network Library Overview

Internet Network Library Function List

Table auto. Internet Network Library Services Functions

Function	Purpose
endhostent	Ends sequential access of the HOSTS database
endnetent	Closes the SYS:ETC\NETWORKS file
endprotoent	Closes the SYS:ETC\PROTOCOL file
endservent	Closes the SYS:ETC\SERVICES file
gethostbyaddr	Returns information about a host at the given IP address
gethostbyname	Returns information about a host given its name
gethostent	Returns the next sequential entry from the HOSTS database
gethostid	Returns the system's default local IP address
gethostname	Returns the official host name for a system
getnetbyaddr	Returns information about a network given its IP address

getnetbyname	Returns information about a network given its name
getnetent	Returns the next entry from the SYS:ETC\NETWORKS file
getprotobyname	Returns information about a protocol given its name
getprotobynumber	Returns information about host given its protocol number
getprotoent	Returns the next entry from the SYS:ETC\PROTOCOL file
getservbyname	Returns information about a service given its name
getservbyport	Returns information about a service given its port number
getservent	Returns the next entry from the SYS:ETC\SERVICES file
htonl	Converts 32-bit quantities from host to network byte order
htons	Converts 16-bit quantities from host to network byte order
inet_addr	Converts a character string representing an IP address expressed in standard dotted notation to a long value that can be used as an IP address
inet_makeaddr	Takes an IP network number and a local network address and constructs an IP address from them
inet_network	Converts a character string representing an IP address in standard dotted notation to a numeric value
inet_ntoa	Converts a long value in in_addr format to an ASCII string representing the address in dotted notation
NetDBendhostent	Closes the SYS:ETC\HOSTS file and makes sure that the next call to **NetDBgethostent** returns the first entry from the file
NetDBgethostbyaddr	Returns information about a host at the given IP address using SYS:ETC\HOSTS and internet name services
NetDBgethostbyname	Returns information about a host given its name using SYS:ETC\HOSTS and internet name services
NetDBgethostent	Returns the next entry in the SYS:ETC\HOSTS file. Once the file is exhausted and if the NIS binder is installed on the system, host entries are returned from the NIS database

NetDBsethostent	Opens the SYS:ETC\HOSTS file and makes sure that the next call to **NetDBgethostent** returns the first entry in the file
ntohl	Converts 32-bit quantites from network to host byte order
ntohs	Converts 16-bit quantities from network to host byte order
NWendhostent	Closes the SYS:ETC\HOSTS file
NWendnetent	Closes the SYS:ETC\NETWORKS file
NWendprotoent	Closes the SYS:ETC\PROTOCOL file
NWendservent	Closes the SYS:ETC\SERVICES file
NWgethostbyaddr	Returns information about a host at the given IP address, using SYS:ETC\HOSTS to locate the entry
NWgethostbyname	Returns information about a host given its name, using SYS:ETC\HOSTS to locate the entry
NWgethostent	Returns the next entry in the SYS:ETC\HOSTS file
NWgetnetbyaddr	Returns information about a network given its IP address
NWgetnetbyname	Returns information about a network given its name
NWgetnetent	Returns the next entry from the SYS:ETC\NETWORKS file
NWgetprotobynam e	Returns information about a protocol given its name
NWgetprotobynum ber	Returns information about host given its protocol number
NWgetprotoent	Returns the next entry from the SYS:ETC\PROTOCOL file
NWgetservbyname	Returns information about a service given its name
NWgetservbyport	Returns information about a service given its port number
NWgetservent	Returns the next entry from the SYS:ETC\SERVICES file
NWinet_ntoa	Converts a long value in in_addr format to an ASCII string representing the address in dotted notation
NWsethostent	Initializes sequential access to the HOSTS database

NWsetnetent	Opens the SYS:ETC\NETWORKS file
NWsetprotoent	Opens the SYS:ETC\PROTOCOL file
NWsetservent	Opens the SYS:ETC\SERVICES file
sethostent	Initializes sequential access to the HOSTS database
setnetent	Opens the SYS:ETC\NETWORKS file
setprotoent	Opens the SYS:ETC\PROTOCOL file
setservent	Opens the SYS:ETC\SERVICES file

Parent Topic: Internet Network Library Overview

Internet Network Library Overview

This chapter describes support functions that simplify TCP/IP application programming. The functions explained in this chapter provide the following services:

Byte-Order Conversion Functions: Convert between host data order and network data order.

Internet Address Conversion Functions: Manipulate IP addresses in both numeric and string formats and convert between the two forms.

HOSTS File Access Functions: Provide access to hostname and address mappings stored in the SYS:ETC\HOSTS file. Internet name services access using DNS and NIS is also available and described. The functions for hostname and host ID are also described.

NETWORKS File Access Functions: Provide access to network name and number mappings stored in the SYS:ETC\NETWORKS file.

PROTOCOL File Access Functions: Provide access to protocol name and number mappings stored in the SYS:ETC\PROTOCOL file.

SERVICES File Access Functions: Provide access to service name and port number mappings stored in the SYS:ETC\SERVICES file.

Related Topics

Internet Network Library Function List

Internet and NetWare

Communication Service Group

Internet Network Library: Functions

endhostent

Ends the sequential access of the HOSTS database
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void endhostent ();
```

Return Values

None.

Remarks

The **endhostent** function is defined as a macro in NETDB.H. You can use this macro to access the HOSTS file or host information from the Internet name services. **endhostent** expands to **NWendhostent**, providing only access to the file SYS:ETC\HOSTS, or to **NetDBendhostent**, providing transparent access to the file in addition to Internet name services such as NIS. See Internet and NetWare in Internet Network Library: Concepts for more information. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

This function closes the SYS:ETC\HOSTS file and makes sure that the next call to **gethostent** begins reading from the beginning of the SYS:ETC\HOSTS file.

NOTE: If your application spawns multiple threads, use either **NWendhostent** or **NetDBendhostent**.

See Also

NetDBendhostent, **NWendhostent**, **sethostent**

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

endnetent

Closes the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

void endnetent ();
```

Return Values

None.

Remarks

The **endnetent** function is defined as a macro in NETDB.H. You can use this macro to access the NETWORKS file. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **endnetent** function closes the SYS:ETC\NETWORKS file.

NOTE: If your application spawns multiple threads, use **NWendnetent**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

NWendnetent, **setnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

endprotoent

Closes the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>

#include <netdb.h>
void endprotoent ();
```

Return Values

None.

Remarks

The **endprotoent** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any one of the source files that include NETDB.H.

The **endprotoent** function closes the SYS:ETC/PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

NOTE: If your application spawns multiple threads, call **NWendprotoent**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

NWendprotoent, **NWsetprotoent**, **setprotoent**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

endservent

Closes the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void endservent ();
```

Return Values

None.

Remarks

The **endservent** function closes SYS:ETC\SERVICES.

NOTE: If your application spawns multiple threads, call **NWendservent**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

NWendservent, **NWsetservent**, **setservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

gethostbyaddr

Returns information about a host, given its Internet address
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *gethostbyaddr (
   char  *addr,
   int    len,
   int    type);
```

Parameters

addr

(IN) Points to a sockaddr_in structure containing the host's Internet address.

len

(IN) Indicates the length of the Internet address, in bytes.

type

(IN) Indicates the value corresponding to the type of Internet address. Currently, the type is always AF_INET.

Return Values

If **gethostbyaddr** succeeds, it returns a pointer to a structure of type hostent. The **gethostbyaddr** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND	No such host exists.

Remarks

The **gethostbyaddr** function is defined as a macro in NETDB.H. You can use this macro to obtain host information from the HOSTS file or using the Internet name services. **gethostbyaddr** expands to **NWgethostbyaddr** , providing only access to the SYS:ETC\HOSTS file, or to **NetDBgethostbyaddr**, providing transparent access to this file in addition to Internet name services such as DNS. See Internet and NetWare in Internet Network Library: Concepts for more information. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **gethostbyaddr** function accepts a pointer to a sockaddr_in structure containing the Internet address, an integer value representing the length, and an integer value representing the application family type. The numeric Internet address is expressed in network byte order; the Internet address length is expressed in bytes.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

Calling any database routine overwrites the results of the last call to this function.

NOTE: If your application spawns multiple threads, use either **NWgethostbyaddr** or **NetDBgethostbyaddr**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostent, **NetDBgethostbyaddr**, **NetDBgethostent**, **NWgethostbyaddr**, **NWgethostent**

gethostbyname

Returns information about a host, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *gethostbyname (
   char   *name);
```

Parameters

name

(IN) Indicates the official name of the host.

Return Values

If **gethostbyname** succeeds, it returns a pointer to a structure of type hostent. The **gethostbyname** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND	No such host exists

Remarks

The **gethostbyname** function is defined as a macro in NETDB.H. You can use this macro to obtain information from the SYS:ETC\HOSTS file or using Internet name services. **gethostbyname** expands to **NWgethostbyname**, providing only access to the SYS:ETC\HOSTS file, or to **NetDBgethostbyname**, providing transparent access to this file in addition to name services such as DNS. See Internet and NetWare in Internet Network Library: Concepts for more information. Your

application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **gethostbyname** function accepts a pointer to a character string representing a hostname.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

Calling any database routine overwrites the results of the last routine that this NLM™ application called.

NOTE: If your application spawns multiple threads, use either **NWgethostbyname** or **NetDBgethostbyname**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostent, **NetDBgethostbyname**, **NetDBgethostent**, **NWgethostbyname**, **NWgethostent**

gethostent

Returns the next sequential entry from the HOSTS database
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *gethostent ( );
```

Return Values

If **gethostent** succeeds, it returns a pointer to a structure of typehostent. The **gethostent** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND	No more hosts exist

Remarks

The **gethostent** function is defined as a macro in NETDB.H. You can use this macro to access the HOSTS file or host information from Internet name services. **gethostent** expands to **NWgethostent**, providing only access to the SYS:ETC\HOSTS file, or to **NetDBgethostent**, providing transparent access to this file in addition to Internet name services such as NIS. See Internet and NetWare in Internet Network Library: Concepts for more information. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **gethostent** function returns the next sequential entry from the SYS:ETC\HOSTS file, opening the file if it is not already open. If the end of the file is reached, Internet name services are being used, and the NIS binder is installed on your system, enumeration continues with the first entry from the NIS HOSTS database.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

Calling any database routine overwrites the results of the last call to this function.

NOTE: If your application spawns multiple threads, call either **NWgethostent** or **NetDBgethostent**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, **gethostbyname**, **NetDBgethostent**, **NWgethostent**

gethostid

Returns the system's default local IP address
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

long gethostid ();
```

Return Values

The **gethostid** function returns the system's default local IP address if it succeeds; it returns a value of -1 if an error occurs.

Remarks

The **gethostid** function returns the system's default local IP address of the current host in network order.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

gethostname

Returns the official hostname for the system
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

int gethostname (
   char  *name,
   int    namelen);
```

Parameters

name

(IN/OUT) Indicates the official name of the host.

namelen

(IN) Specifies the size of the array pointed to by name.

Return Values

If the **gethostname** function succeeds, it returns a value of 0. If the call fails, it returns a value of -1.

Remarks

The **gethostname** function returns the standard hostname for the current host machine. The parameter *namelen* specifies the size of the array pointed to by *name*. The returned name is terminated by null, unless insufficient space is provided. Hostnames are limited to MAXNAMESIZE (from NETDB.H) characters, which is 64.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

getnetbyaddr

Returns information about a network, given its IP network number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *getnetbyaddr (
   long   net,
   int    type);
```

Parameters

net

(IN) Specifies the IP network number in host order.

type

(IN) Indicates the network type (only currently supported type is AF_INET).

Return Values

The **getnetbyaddr** function returns NULL when an error occurs. Upon success, **getnetbyaddr** returns a pointer to a structure of type netent.

Remarks

The **getnetbyaddr** function has been defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function and returns a pointer to a netent structure in the SYS:ETC\NETWORKS file.

The **getnetbyaddr** function accepts a long integer value representing the Internet network number and an integer value representing the application family type. Using **getnetent**, the function begins searching for the network number from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getnetbyaddr**.

If your application spawns multiple threads, call **NWgetnetbyaddr**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getnetbyname, **getnetent**, **NWgetnetent**, **NWgetnetbyaddr**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

getnetbyname

Returns information about a network, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *getnetbyname (
   char   *name);
```

Parameters

name

(IN) Indicates the official name of the network.

Return Values

The **getnetbyname** function returns NULL when an error occurs. Upon success, **getnetbyname** returns a pointer to a structure of type netent.

Remarks

The **getnetbyname** function has been defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function and returns a pointer to a netent structure in the SYS:ETC\NETWORKS file.

The **getnetbyname** function accepts a pointer to a character string representing a network name. Using **getnetent**, the function searches for the character string from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
```

```
    unsigned long   n_net;          /* network number */
    unsigned long   n_mask;         /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getnetbyname**.

If your application spawns multiple threads, call **NWgetnetbyname**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getnetbyaddr, **getnetent**, **NWgetnetbyname**, **NWgetnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

getnetent

Returns the next sequential entry from the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *getnetent ();
```

Return Values

The **getnetent** function returns NULL when an error occurs. Upon success, **getnetent** returns a pointer to a structure of type netent.

Remarks

The **getnetent** function has been defined as a macro in NETDB.H. This macro returns a pointer to a netent structure in the SYS:ETC\NETWORKS file.

The **getnetent** function returns the next sequential entry from the SYS:ETC\NETWORKS file, opening the file if necessary.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetnetent**.

If your application spawns multiple threads, call **NWgetnetent**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

endnetent, getnetbyaddr, getnetbyname, NWgetnetent, setnetent

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

getprotobyname

Returns information about a protocol, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *getprotobyname(
   char   *name);
```

Parameters

name

(IN) Indicates the official name of the protocol.

Return Values

getprotobyname returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **getprotobyname** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any one of the source files that include NETDB.H.

getprotobyname returns a pointer to a protoent structure in the SYS:ETC\PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

The **getprotobyname** function accepts a pointer to a character string representing a protocol name. Using **getprotoent**, the function begins searching for the character string from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to this function.

If your application spawns multiple threads, call **NWgetprotobyname**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getprotobynumber, **getprotoent**, **NWgetprotoent**, **NWgetprotobyname**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

getprotobynumber

Returns information about a protocol, given its protocol number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *getprotobynumber (
   int   *proto);
```

Parameters

proto

(IN) Indicates the protocol number.

Return Values

getprotobynumber returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **getprotobynumber** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any one of the source files that include NETDB.H.

getprotobynumber returns a pointer to a protoent structure in the SYS:ETC\PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

The **getprotobynumber** function accepts an integer value representing the protocol number. Using **getprotoent**, the function begins searching for the protocol number from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getprotobynumber**.

If your application spawns multiple threads, call **NWgetprotobynumber**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getprotobyname, **getprotoent**, **NWgetprotoent**, **NWgetprotobynumber**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

getprotoent

Returns the next sequential entry from the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *getprotoent ();
```

Return Values

getprotoent returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **getprotoent** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any one of the source files that include NETDB.H.

The **getprotoent** function returns the next sequential entry from the SYS:ETC\PROTOCOL file, opening the file if necessary. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getprotoent**.

If your application spawns multiple threads, call **NWgetprotoent**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport*

Supervisor's Guide.

See Also

getprotobyname, **getprotobynumber**, **NWgetprotoent**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

getservbyname

Returns information about a service, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct servent *getservbyname (
   char   *name,
   char   *proto);
```

Parameters

name

(IN) Indicates the official name of the service.

proto

(IN) Indicates the name of the protocol to use when contacting a service.

Return Values

getservbyname returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remarks

The **getservbyname** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

getservbyname returns a pointer to a servent structure in the SYS:ETC\SERVICES file. If *proto* is NULL, **getservbyname** returns the first entry matching the service name.

The **getservbyname** function accepts pointers to two character strings representing the service and protocol name for which to search. Using **getservent**, the function searches for the service from the beginning of

the file and continues until it finds a matching entry or reaches end-of-file.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getservbyname**.

If your application spawns multiple threads, call **NWgetservbyname**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

bind, **connect**, **getservbyport**, **getservent**, **NWgetservbyname**, **NWgetservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

getservbyport

Returns information about a service, given its port number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct servent *getservbyport (
   int   port,
   char *proto);
```

Parameters

port

(IN) Indicates the port number at which the service resides, in network order.

proto

(IN) Points to the name of the protocol to use when contacting a service.

Return Values

getservbyport returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remarks

The **getservbyport** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

getservbyport returns a pointer to a servent structure in the SYS:ETC\SERVICES file.

The **getservbyport** function accepts a pointer to a character string representing the protocol name and an integer representing the port number. Using **getservent**, the function searches from the beginning of

the file and continues until it reaches end-of-file or finds a service matching the port number. If a protocol name is also specified, then the searches must also match the protocol name. (The numeric protocol number is expressed in network byte order). If the protocol name is NULL, the first matching entry with the given port number is returned.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getservbyport**.

If your application spawns multiple threads, call **NWgetservbyport**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

bind, **connect**, **getservbyname**, **getservent**, **NWgetservbyport**, **NWgetservent**, **setservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

getservent

Returns the next sequential entry from the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>
struct servent *getservent ( );
```

Return Values

getservent returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remark

The **getservent** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

getservent returns a pointer to a servent structure in the SYS:ETC\SERVICES file.

The **getservent** function returns the next sequential entry from the SYS:ETC\SERVICES file, opening the file if necessary.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **getservbyport**.

If your application spawns multiple threads, call **NWgetservbyport**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

bind, **connect**, **getservbyname**, **getservbyport**, **NWgetservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

htonl

Converts 32-bit quantities from host to network byte order
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netinet/in.h>

unsigned long htonl (
   unsigned long   hostlong);
```

Parameters

hostlong

(IN) Indicates a 32-bit quantity in host byte order.

Return Values

htonl returns a value that has been converted from host to network byte order.

Remarks

htonl is frequently used to convert Internet addresses from host byte order to network byte order for socket calls requiring an Internet address as a parameter.

NOTE: Network data order places the most significant byte at the lower memory address, while 80386 data order places the least significant byte at the lower address.

See Also

accept, **bind**, **connect**, **inet_addr**, **inet_network**, **ntohl**

htons

Converts 16-bit quantities from host to network byte order
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netinet/in.h>

unsigned short htons (
   unsigned short   hostshort);
```

Parameters

hostshort

(IN) Indicates a 16-bit quantity in host byte order.

Return Values

htons returns a value that has been converted from host to network byte order.

Remarks

htons is frequently used to convert port numbers from host byte order to network byte order for socket calls requiring a port number as a parameter.

NOTE: Network data order places the most significant byte at the lower memory address, while 80386 data order places the least significant byte at the lower address.

See Also

accept, **bind**, **connect**, **inet_addr**, **inet_network**, **ntohs**

inet_addr

Converts a character string representing an IP address expressed in the Internet standard dotted notation to a long value that can be used as an Internet address

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Standard: BSD

Service: Internet Network Library

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr (
   char   *cp);
```

Parameters

cp

(IN) Points to the character string (in Internet Standard Dotted Notation) that represents the Internet address.

Return Values

Upon success, **inet_addr** returns an Internet address value. **inet_addr** returns -1 upon detection of malformed requests. It does not update the integer *errno*.

Remarks

The Internet addresses are returned in network byte order (bytes ordered from left to right).

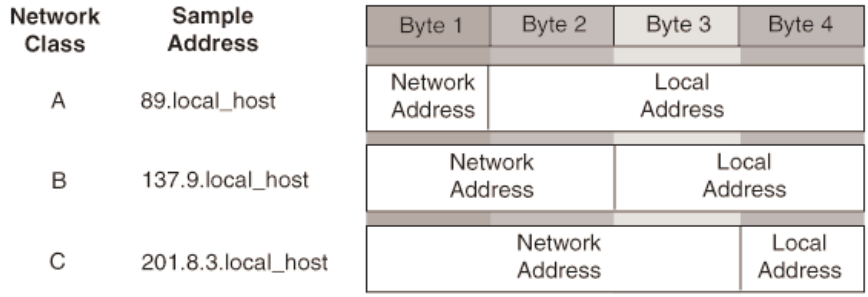
Internet Standard Dotted Notation

Values specified using dotted notation take one of the following forms:

a	When only one part of the address is given, the value is stored directly in the network address without any byte rearrangement.
a.b	When a two-part address is supplied, the last part is

	interpreted as a 24-bit quantity and placed in the rightmost 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses in net.host format (for example, 89.1 where the network address is 89 and the host address is 1).
a.b.c	When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses in net.host format (for example, 128.1.1 where the network address is 128.1 and the host address is 1).
a.b.c.d	When four parts are specified, each part is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

The following illustrates the Internet address form.



All numbers supplied as parts in dotted notation can be expressed in decimal, octal, or hexadecimal, as specified in the C language. (A leading 0x or 0X implies a hexadecimal number, and a leading 0 implies an octal number; otherwise, the number is interpreted as decimal.)

See Also

inet_network

inet_makeaddr

Takes an Internet network number and a local network address and constructs an Internet address from them

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Standard: BSD

Service: Internet Network Library

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr (
   int   net,
   int   lna);
```

Parameters

net

(IN) Internet network number.

lna

(IN) The local network address part of an Internet address.

Return Values

If successful, **inet_makeaddr** returns an in_addr structure representing the Internet address. **inet_makeaddr** does not return an error value.

Remarks

The Internet address is returned in the structure in_addr.

inet_network

Converts a character string representing an IP network number expressed in the Internet standard dotted notation to a numeric value

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Standard: BSD

Service: Internet Network Library

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network (
   char   *cp);
```

Parameters

cp

(IN) Indicates the character string that represents the IP network number.

Return Values

Upon success, **inet_network** returns a network number. **inet_network** returns -1 upon detection of malformed requests. It does not update the integer *errno*.

Remarks

The network number is returned as a long value in network order.

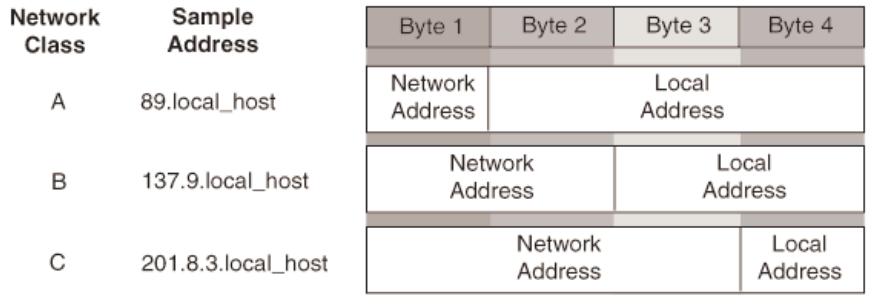
Internet Standard Dotted Notation

Values specified using dotted notation take one of the following forms:

a	When only one part of the address is given, the value is stored directly in the network address without any byte rearrangement.
a.b	When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost 3 bytes of the network address. This makes the two-part

	address format convenient for specifying Class A network addresses in net.host format (for example, 89.1 where the network address is 89 and the host address is 1).
a.b.c	When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses in net.host format (for example, 128.1.1 where the network address is 128.1 and the host address is 1).
a.b.c.d	When four parts are specified, each part is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

The following illustrates the Internet address form.



All numbers supplied as parts in dotted notation can be expressed in decimal, octal, or hexadecimal, as specified in the C language. (A leading 0x or 0X implies a hexadecimal number, and a leading 0 implies an octal number; otherwise, the number is interpreted as decimal.)

See Also

inet_addr

inet_ntoa

Converts a long value in in_addr format into an ASCII string representing the address in dotted notation

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Standard: BSD

Service: Internet Network Library

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa (
   struct in_addr   in);
```

Parameters

in

(IN) IP address, in in_addr format.

Return Values

inet_ntoa returns a pointer to the array containing the ASCII string representing the address in dotted notation. **inet_ntoa** does not return an error value.

Remarks

The **inet_ntoa** function has been defined as a macro in ARPA/INET.H. To use this macro, your application must include the NETDB_DEFINE_CONTEXT macro (defined in ARPA/INET.H) in at least one of the modules that is being linked to form an NLM. NETDB_DEFINE_CONTEXT declares an array of characters.

The library fills this array with the converted address and returns a pointer to this array. The contents are destroyed if **inet_ntoa** is called subsequently by the same or a different thread in the NLM. Simple applications involving a single thread can use this macro. Applications with multiple threads should call **NWinet_ntoa** instead of **inet_ntoa**.

See Also

NWinet_ntoa

NetDBendhostent

Closes the SYS:ETC\HOSTS file and makes sure the next call to **NetDBgethostent** returns the first entry from the file

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NetDBendhostent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent

(IN) Points to a context block.

Return Values

None.

Remarks

NetDBendhostent closes the SYS:ETC\HOSTS file and rewinds it if the file is already open. In addition, this function resets the thread's context so that the next call to **NetDBgethostent** begins reading from the beginning of the SYS:ETC\HOSTS file. **NetDBgethostent** returns all the host entries in the SYS:ETC\HOSTS file first, followed by all hosts in the NIS HOSTS database if available. See Internet and NetWare in Internet Network Library: Concepts for more information.

NetDBendhostent is identical **endhostent** except your application must pass the address of an nwsockent structure.

If you don't want to use Internet name services, call **NWendhostent** instead (see Internet and NetWare in Internet Network Library: Concepts).

The result of **NetDBendhostent** stays intact until the next call is made. **NetDBendhostent** is useful for applications spawning multiple threads, where each thread accesses the HOSTS file. Your application can declare

separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare Library uses the same database pointers for this function as for **NetDBgethostbyaddr**, **NetDBgethostbyname**, **NetDBgethostent**, and **NetDBsethostent** for each context block to access the HOSTS databases. In addition, each thread in an application maintains state information specific to that thread. Therefore, each thread should maintain its own context block. Calling these functions indiscriminately can result in unpredictable behavior from **NetDBgethostent** because it relies on the current value of the database pointers.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

endhostent, **NWendhostent**, **sethostent**

NetDBgethostbyaddr

Returns information about a host at a given Internet address
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NetDBgethostbyaddr  (
   struct nwsockent  *nwsktent,
   char              *addr,
   int                len,
   int                type);
```

Parameters

nwsktent

(IN) Points to a context block.

addr

(IN) Points to a sockaddr_in structure containing the host's Internet address.

len

(IN) Indicates the length of the Internet address, in bytes.

type

(IN) Indicates the value corresponding to the type of Internet address. Currently, the type is always AF_INET.

Return Values

If **NetDBgethostbyaddr** succeeds, it returns a pointer to a structure of type hostent. The **NetDBgethostbyaddr** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND---No such host exists.

Remarks

NetDBgethostbyaddr accepts a pointer to a sockaddr_in structure containing the Internet address, an integer value representing the address length, and an integer value representing the application family type. The local SYS:ETC\HOSTS file is consulted first. If the host entry is not found and Internet services are enabled, DNS is then consulted. If it is still not found, NIS is consulted. See Internet and NetWare in Internet Network Library: Concepts for more information.

NetDBgethostbyaddr is identical to **gethostbyaddr** except your application must pass the address of a nwsockent structure.

If you don't want to use Internet name services, call **NWgethostbyaddr** instead (see Internet and NetWare in Internet Network Library: Concepts).

The result of **NetDBgethostbyaddr** stays intact until the next call is made. **NetDBgethostbyaddr** is useful for applications spawning multiple threads, where each thread accesses the HOSTS database. Your application must declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare of Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare Library uses the same database pointers for **NetDBgethostbyaddr** as for **NetDBgethostent**, **NetDBgethostbyname**, **NetDBsethostent**, and **NetDBendhostent** for each context block to access the HOSTS databases. In addition, each thread in an application maintains state information specific to that thread. Therefore, each thread should maintain its own context block. Calling these functions indiscriminately can result in unpredictable behavior from **NetDBgethostent** because it relies on the current value of the database pointers.

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

See hostent.

NOTE: Calling any database routine overwrites the results of the last call to ***NetDBgethostbyaddr** .

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, gethostbyname, gethostent, NetDBgethostent, NWgethostbyaddr, NWgethostent

NetDBgethostbyname

Returns information about a host, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NetDBgethostbyname (
   struct nwsockent   *nwsktent,
   char               *name);
```

Parameters

nwsktent

(IN) Points to a context block.

name

(IN) Points to the official name of the host.

Return Values

If **NetDBgethostbyname** succeeds, it returns a pointer to a structure of type hostent. **NetDBgethostbyname** returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following value:

HOST_NOT_FOUND	No such host exists.

Remarks

NetDBgethostbyname accepts a pointer to a character string representing a hostname. The local SYS:ETC\HOSTS file is consulted first. If the host entry is not found and Internet services are enabled, DNS is then consulted. If it is still not found, NIS is consulted. SeeInternet and NetWare in Internet Network Library: Concepts for more information.

NetDBgethostbyname is identical to **gethostbyname** except your application must pass the address of a nwsockent structure.

If you don't want to use Internet name services, call **NWgethostbyname** instead (see Internet and NetWare in Internet Network Library: Concepts).

The result of **NetDBgethostbyname** stays intact until the next call is made. **NetDBgethostbyname** is useful for applications spawning multiple threads, where each thread accesses the HOSTS database. Your application must declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare Library uses the same database pointers for this function as for **NetDBgethostent**, **NetDBgethostbyaddr**, **NetDBsethostent**, and **NetDBendhostent** for each context block to access the HOSTS databases. In addition, each thread in an application maintains state information specific to that thread. Therefore, each thread should maintain its own context block. Calling these functions indiscriminately can result in unpredictable behavior from **NetDBgethostent** because it relies on the current value of the database pointers.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

See hostent.

NOTE: Calling any database routine overwrites the results of the last call to ***NetDBgethostbyname**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyname, **gethostent**, **NetDBgethostent**, **NWgethostbyname**, **NWgethostent**

NetDBgethostent

Returns the next sequential entry from the HOSTS database
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NetDBgethostent (
   struct nwsockent   *nwsktent,
   short              *ploc);
```

Parameters

nwsktent

(IN) Points to a context block.

ploc

(IN/OUT) Indicates whether this entry is from the local SYS:ETC\HOSTS file (1) or from the NIS HOSTS database (2). Pass in NULL if you do not want this information.

Return Values

If **NetDBgethostent** succeeds, it returns a pointer to a structure of type hostent. The **NetDBgethostent** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND---No more hosts exist.

Remarks

NetDBgethostent returns the next sequential entry from the SYS:ETC\HOSTS file, opening the file if it is not already open. After all the file's hosts are returned, all the hosts from the NIS HOSTS database, if any are returned. See Internet and NetWare in Internet Network Library: Concepts for more information.

NetDBgethostent is identical to **gethostent** except your application must pass the address of a nwsockent structure, as well as a second parameter that returns which database the host entry came from.

If you don't want to use Internet name services, call **NWgethostbyname** instead (see Internet and NetWare in Internet Network Library: Concepts).

The result of **NetDBgethostent** stays intact until the next call is made. **NetDBgethostent** is useful for applications spawning multiple threads, where each thread accesses the HOSTS database. Your application must declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare Library uses the same database pointers for this function as for **NetDBgethostbyaddr**, **NetDBgethostbyname**, **NetDBsethostent**, and **NetDBendhostent** for each context block to access the HOSTS databases. In addition, each thread in an application maintains state information specific to that thread. Therefore, each thread should maintain its own context block. Calling these functions indiscriminately can result in unpredictable behavior from **NetDBgethostent** because it relies on the current value of the database pointers.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NetDBgethostent**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostent, **gethostbyaddr**, **gethostbyname**, **NWgethostent**

NetDBsethostent

Opens the SYS:ETC\HOSTS file and makes sure that the next call to **NetDBgethostent** returns the first entry in the file

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NetDBsethostent(
   struct nwsockent  *nwsktent,
   int                stayopen);
```

Parameters

nwsktent

(IN) Points to a context block.

stayopen

(IN) If nonzero, causes SYS:ETC\HOSTS to remain open after a call to **NetDBgethostbyname** or **NetDBgethostbyaddr**.

Return Values

None.

Remarks

NetDBsethostent opens the SYS:ETC\HOSTS file and rewinds it if the file is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\HOSTS file is not closed after each call made to **NetDBgethostent** by **NetDBgethostbyname** or **NetDBgethostbyaddr**. In addition, this function resets the thread's context so that the next call to **NetDBgethostent** begins reading from the beginning of the SYS:ETC\HOSTS file. **NetDBgethostent** returns all the host entries in the SYS:ETC\HOSTS file first, followed by all hosts in the NIS HOSTS database if available. See Internet and NetWare in Internet Network Library: Concepts for more information.

The **NetDBsethostent** is identical to **sethostent** except your application must pass the address of an nwsockent structure.

If you don't want to use Internet name services, use **NWsethostent** instead (see Internet and NetWare in Internet Network Library: Concepts).

The result of **NetDBsethostent** stays intact until the next call is made. **NetDBsethostent** is useful for applications spawning multiple threads, where each thread is accessing the HOSTS databases. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare Library uses the same database pointers for this function as for **NetDBgethostbyaddr**, **NetDBgethostbyname**, **NetDBgethostent**, and **NetDBendhostent** for each context block to access the HOSTS databases. In addition, each thread in an application maintains state information specific to that thread. Therefore, each thread should maintain its own context block. Calling these functions indiscriminately can result in unpredictable behavior from **NetDBgethostent** because it relies on the current value of the database pointers.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, **gethostbyname**, **gethostent**, **NetDBgethostbyaddr**, **NetDBgethostbyname**, **NetDBgethostent**, **NWgethostbyaddr**, **NWgethostbyname**, **NWgethostent**, **NWsethostent**, **sethostent**

ntohl

Converts 32-bit quantities from network to host byte order
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netinet/in.h>

unsigned long ntohl (
   unsigned long   netlong);
```

Parameters

netlong

(IN) A 32-bit quantity in network byte order.

Return Values

ntohl returns a value that has been converted from network to host byte order.

Remarks

NOTE: Network data order places the most significant byte at the lower memory address, while 80386 data order places the least significant byte at the lower address.

See Also

htonl

ntohs

Converts 16-bit quantities from network to host byte order
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netinet/in.h>

unsigned short ntohs (
   unsigned short   netshort);
```

Parameters

netshort

(IN) Indicates a 16-bit quantity in network byte order.

Return Values

ntohs returns a value that has been converted from network to host byte order.

Remarks

NOTE: Network data order places the most significant byte at the lower memory address, while 80386 data order places the least significant byte at the lower address.

See Also

htons

NWendhostent

Closes the SYS:\ETC\HOSTS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWendhostent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(IN) Points to a context block.

Return Values

None.

Remarks

The **NWendhostent** function closes the SYS:ETC\HOSTS file.

The **NWendhostent** is identical **endhostent** except your application must pass the address of an nwsockent structure.

NWendhostent accesses the SYS:ETC\HOSTS file only. Call **NetDBendhostent** to access Internet name services transparently in addition to accessing the SYS:ETC\HOSTS file. See Internet and NetWare in Internet Network Library: Concepts for more information.

The result of **NWendhostent** call stays intact until the next call is made. **NWendhostent** is useful for applications spawning multiple threads, where each thread is accessing the HOSTS file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer **NWgethostent**,

NWgethostbyaddr, **NWgethostbyname** and **NWsethostent** for each context block to access the HOSTS file. Using these functions indiscriminately can result in unpredictable behavior from **gethostent** because it relies on the current value of the file pointer.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

endhostent, **NetDBendhostent**, **sethostent**

NWendnetent

Closes the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

void NWendnetent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(IN) Points to a context block.

Return Values

None.

Remarks

The **endnetent** function closes the SYS:ETC\NETWORKS file.

The **NWendnetent** function is identical to **endnetent** except your application must pass the address of a structure of type nwsockent.

The result of **NWendnetent** stays intact until the application makes the next call. **NWendnetent** is useful for applications spawning multiple threads, where each thread accesses the NETWORKS file.

Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in Internet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for this function as for **NWgetnetent**, **NWgetnetbyaddr**, **NWgetnetbyname** and **NWsetnetent** for each context block to access the NETWORKS file. Calling these functions indiscriminately can result in unpredictable behavior from **getnetent** because it relies on the current value of the file pointer.

For a description of the NETWORKS file, refer to the *TCP/IP Transport*

Supervisor's Guide.

See Also

endnetent, setnetent

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

NWendprotoent

Closes the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWendprotoent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(IN) Points to a context block.

Return Values

None.

Remarks

The **NWendprotoent** function closes the SYS:ETC/PROTOCOL file. This function is identical to **endprotoent** except your application must pass the address of an nwsockent structure.

The result of the function stays intact until the application makes the next call. **NWendprotoent** is useful for applications spawning multiple threads, where each thread is accessing the PROTOCOL file. Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in Internet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for this function as for **NWgetprotoent**, **NWgetprotobyname**, **NWgetprotobynumber** and **NWsetprotoent** for each context block to access the PROTOCOL file. Calling these functions indiscriminately can result in unpredictable behavior from **getprotoent** because it relies on the current value of the file pointer.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport*

Supervisor's Guide.

See Also

endprotoent, **NWsetprotoent**, **setprotoent**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

NWendservent

Closes the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWendservent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(OUT) Points to a context block.

Return Values

None.

Remarks

The **endservent** function closes SYS:ETC\SERVICES. This function is identical to **endservent** except your application must pass the address of an nwsockent structure. The result of the function call stays intact until the application makes the next call.

NWendservent is useful for applications spawning multiple threads, where each thread is accessing the SERVICES file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWendservent** as for **NWgetservbyport**, **NWgetservbyname**, **NWgetservent**, and **NWsetservent** for each context block to access the SERVICES file. Using these functions indiscriminately can result in unpredictable behavior from **getservent** because it relies on the current value of the file pointer.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

endservent, **NWendservent**, **NWsetservent**, **setservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

NWgethostbyaddr

Returns information about a host at a given Internet address
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NWgethostbyaddr (
   struct nwsockent  *nwsktent,
   char              *addr,
   int                len,
   int                type);
```

Parameters

nwsktent

(IN) Points to a context block.

addr

(IN) Points to a sockaddr_in structure containing the host's Internet address.

len

(IN) Indicates the length of the Internet address, in bytes.

type

(IN) Indicates the value corresponding to the type of Internet address. Currently, the type is always AF_INET.

Return Values

If **NWgethostbyaddr** succeeds, it returns a pointer to a structure of type hostent. The **NWgethostbyaddr** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND---No such host exists in the file SYS:ETC\HOSTS.

Remarks

The **NWgethostbyaddr** function accepts a pointer to a sockaddr_in structure containing the Internet address, an integer value representing the address length, and an integer value representing the application family type. Calling **NWgethostent**, **NWgethostbyaddr** begins searching for the character string from the beginning of the file and continues until it finds a matching entry or reaches end- of-file. The numeric Internet address is expressed in network byte order; the Internet address length is expressed in bytes.

The **NWgethostbyaddr** function is identical to **gethostbyaddr** except your application must pass the address of a nwsockent structure.

NWgethostbyaddr accesses the SYS:ETC\HOSTS file only. Use **NetDBgethostbyaddr** to transparently access Internet name services in addition to the SYS:ETC\HOSTS file. See Internet and NetWare in Internet Network Library: Concepts for more information.

The result of **NWgethostbyaddr** function stays intact until the next call is made. ***NWgethostbyaddr** is useful for applications spawning multiple threads, where each thread is accessing the HOSTS file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgethostbyaddr** as for **NWgethostent**, **NWgethostbyname**, **NWsethostent** and **NWendhostent** for each context block to access the HOSTS file. Using these functions indiscriminately can result in unpredictable behavior from **gethostent** because it relies on the current value of the file pointer.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgethostbyaddr**.

For a description of the HOSTS file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, **gethostbyname**, **gethostent**, **NetDBgethostbyaddr**,

NetDBgethostent, NWgethostent

NWgethostbyname

Returns information about a host, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NWgethostbyname(
   struct nwsockent   *nwsktent,
   char               *name);
```

Parameters

nwsktent

(IN) Points to a context block.

name

(IN) Points to the official name of the host.

Return Values

If **NWgethostbyname** succeeds, it returns a pointer to a structure of type hostent. **NWgethostbyname** returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND	No such host exists in the SYS:ETC\HOSTS file.

Remarks

The **NWgethostbyname** function accepts a pointer to a character string representing a hostname. Using **NWgethostent**, the function begins searching for the character string from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The **NWgethostbyname** function is identical to **gethostbyname** except your application must pass the address of a nwsockent structure.

NWgethostbyname accesses the SYS:ETC\HOSTS file only. Use **NetDBgethostbyname** to access Internet name services transparently in addition to accessing the SYS:ETC\HOSTS file. See Internet and NetWare in Internet Network Library: Concepts for more information.

The result of **NWgethostbyname** stays intact until the next call is made. **NWgethostbyname** is useful for applications spawning multiple threads, where each thread is accessing the HOSTS file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgethostbyname** as for **NWgethostent**, **NWgethostbyaddr**, **NWsethostent** and **NWendhostent** for each context block to access the HOSTS file. Using these functions indiscriminately can result in unpredictable behavior from **gethostent** because it relies on the current value of the file pointer.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgethostbyname**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyname, **gethostent**, **NetDBgethostbyname**, **NetDBgethostent**, **NWgethostent**

NWgethostent

Returns the next sequential entry from the SYS:\ETC\HOSTS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct hostent *NWgethostent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent

(IN) Points to a context block.

Return Values

If **NWgethostent** succeeds, it returns a pointer to a structure of type hostent. The **NWgethostent** function returns NULL when an error occurs. The integer h_errno, which is a C preprocessor macro, can be checked to determine the nature of the error.

The integer h_errno can have the following values:

HOST_NOT_FOUND---No such host exists in the file SYS:ETC\HOSTS.

Remarks

The **NWgethostent** function returns the next sequential entry from the SYS:ETC\HOSTS file, opening the file if it is not already open.

The **NWgethostent** function is identical to **gethostent** except that your application must pass the address of a nwsockent structure.

***NWgethostent** accesses the SYS:ETC\HOSTS file only. Use **NetDBgethostent** to access Internet name services transparently in addition to accessing the SYS:ETC\HOSTS file. See Internet and NetWare in Internet Network Library: Concepts for more information.

The result of ***NWgethostent** call stays intact until the next call is made. ***NWgethostent** is useful for applications spawning multiple threads,

where each thread is accessing the HOSTS file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgethostent** as for **NWgethostbyaddr**, **NWgethostbyname**, **NWsethostent** and **NWendhostent** for each context block to access the HOSTS file. Using these functions indiscriminately can result in unpredictable behavior from **gethostent** because it relies on the current value of the file pointer.

The hostent structure has the following format:

```
struct    hostent {
   char   *h_name;               /* official name of host */
   char   ** h_aliases;          /* alias list */
   int    h_addrtype;            /* host address type */
   int    h_length;              /* length of address */
   char   ** h_addr_list;        /* list of addresses from name server
#define h_addr   h_addr_list[0]  /* address, for backward compatibility
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgethostent**.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostent, **gethostbyaddr**, **gethostbyname**, **NetDBgethostent**

NWgetnetbyaddr

Returns information about a network, given its IP network number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *NWgetnetbyaddr (
   struct nwsockent  *nwsktent,
   int                net,
   int                type);
```

Parameters

nwsktent

(IN) Points to a context block.

net

(IN) Specifies the IP network number in host order.

type

(IN) Currently, the only network type supported is AF_INET.

Return Values

The **NWgetnetbyaddr** function returns NULL when an error occurs. Upon success, this function returns a pointer to a structure of type netent.

Remarks

The **NWgetnetbyaddr** function accepts a long integer value representing the Internet network number and an integer value representing the application family type. Using **NWgetnetent**, the function begins searching for the network number from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The **NWgetnetbyaddr** function is identical to **getnetbyaddr** except your application must pass the address of an nwsockent structure.

The result of **NWgetnetbyaddr** stays intact until the application makes

the next call. **NWgetnetbyaddr** is useful for applications spawning multiple threads, where each thread accesses the NETWORKS file.

Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified inInternet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetnetbyaddr** as for **NWgetnetent**, **NWgetnetbyname**, **NWsetnetent** and **NWendnetent** for each context block to access the NETWORKS file. Using these functions indiscriminately can result in unpredictable behavior from **getnetent** because it relies on the current value of the file pointer.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetnetbyaddr**.

For a description of the NETWORKS file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

getnetbyname, **getnetent**, **NWgetnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

NWgetnetbyname

Returns information about a network, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *NWgetnetbyname (
   struct nwsockent   *nwsktent,
   char               *name);
```

Parameters

nwsktent

(IN) Points to a context block.

name

(IN) Points to the official name of the network.

Return Values

The **NWgetnetbyname** function returns NULL when an error occurs. Upon success, **NWgetnetbyname** returns a pointer to a structure of type netent.

Remarks

The **NWgetnetbyname** function accepts a pointer to a character string representing a network name. Using **NWgetnetent**, the function searches for the character string from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

The **NWgetnetbyname** function is identical to **getnetbyname** except your application must pass the address of an nwsockent structure.

The result of **NWgetnetbyname** stays intact until the application makes the next call. **NWgetnetbyname** is useful for applications spawning multiple threads, where each thread accesses the NETWORKS file.

Your application can declare separate nwsockent structures for each

thread. It must follow the instructions specified inInternet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetnetbyname** as for **NWgetnetent**, **NWgetnetbyaddr**, **NWsetnetent** and **NWendnetent** for each context block to access the NETWORKS file. Using these functions indiscriminately can result in unpredictable behavior from **getnetent** because it relies on the current value of the file pointer.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetnetbyname**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getnetbyaddr, **getnetent**, **NWgetnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

NWgetnetent

Returns the next sequential entry from the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

struct netent *NWgetnetent (
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(IN) Points to a context block.

Return Values

The **NWgetnetent** function returns NULL when an error occurs. Upon success, **NWgetnetent** returns a pointer to a structure of type netent.

Remarks

The **NWgetnetent** function returns the next sequential entry from the SYS:ETC\NETWORKS file, opening the file if necessary.

The **NWgetnetent** is identical to **getnetent** except your application must pass the address of an nwsockent structure.

The result of **NWgetnetent** stays intact until the application makes the next call. **NWgetnetent** is useful for applications spawning multiple threads, where each thread accesses the NETWORKS file.

Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified inInternet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetnetent** as for **NWgetnetbyaddr**, **NWgetnetbyname**, **NWsetnetent** and **NWendnetent** for each context block to access the NETWORKS file. Using these functions indiscriminately can result in unpredictable behavior from

getnetent because it relies on the current value of the file pointer.

The netent structure has the following format:

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetnetent**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getnetbyaddr, **getnetbyname**, **getnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

NWgetprotobyname

Returns information about a protocol, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *NWgetprotobyname (
   struct nwsockent   *nwsktent,
   char               *name);
```

Parameters

nwsktent

(IN) Points to a context block.

name

(IN) Points to the official name of the protocol.

Return Values

NWgetprotobyname returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **NWgetprotobyname** function accepts a pointer to a character string representing a protocol name. Using **NWgetprotoent**, the function begins searching for the character string from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

NWgetprotobyname is identical to **getprotobyname** except your application must pass the address of an nwsockent structure.

The result of **NWgetprotobyname** stays intact until the application makes the next call. This function is useful for applications spawning multiple threads, where each thread is accessing the PROTOCOL file. Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified inInternet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetprotobyname** as for **NWgetprotoent**, **NWgetprotobynumber**, **NWsetprotoent** and **NWendprotoent** for each context block to access the PROTOCOL file. Calling these functions indiscriminately can result in unpredictable behavior from **getprotoent** because it relies on the current value of the file pointer.

NWgetprotobyname returns a pointer to a protoent structure in the SYS:ETC\PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.).

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetprotobyname**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getprotobynumber, **getprotoent**, **NWgetprotoent**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

NWgetprotobynumber

Returns information about a protocol, given its protocol number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *NWgetprotobynumber (
   struct nwsockent  *nwsktent,
   int                proto);
```

Parameters

nwsktent

(IN) Points to a context block.

proto

(IN) Indicates the protocol number.

Return Values

NWgetprotobynumber returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **NWgetprotobynumber** function accepts an integer value representing the protocol number. Using **NWgetprotoent**, the function begins searching for the protocol number from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

You can call **NWgetprotobynumber** to access the PROTOCOL file. This function is identical to **getprotobynumber** except your application must pass the address of an nwsockent structure.

The result of **NWgetprotobynumber** stays intact until the application makes the next call. **NWgetprotobynumber** is useful for applications spawning multiple threads, where each thread is accessing the PROTOCOL file. Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in

Internet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetprotobynumber** as for **NWgetprotoent**, **NWgetprotobyname**, **NWsetprotoent** and **NWendprotoent** for each context block to access the PROTOCOL file. Using these functions indiscriminately can result in unpredictable behavior from **getprotoent** because it relies on the current value of the file pointer.

NWgetprotobynumber returns a pointer to a protoent structure in the SYS:ETC\PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetprotobynumber**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getprotobyname, **getprotobynumber**, **getprotoent**, **NWgetprotoent**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

NWgetprotoent

Returns the next sequential entry from the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct protoent *NWgetprotoent(
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent
(IN) Points to a context block.

Return Values

NWgetprotoent returns a pointer to a structure of type protoent if successful or NULL if an error occurs.

Remarks

The **NWgetprotoent** function returns the next sequential entry from the SYS:ETC\PROTOCOL file, opening the file if necessary.

This function is identical to **getprotoent** except your application must pass the address of a structure of type nwsockent.

The result of the call stays intact until the application makes the next call. **NWgetprotoent** is useful for applications spawning multiple threads, where each thread is accessing the PROTOCOL file. Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in Internet Network Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetprotoent** as for **NWgetprotobyname**, **NWgetprotobyaddr**, **NWsetprotoent** and **NWendprotoent** for each context block to access the PROTOCOL file. Using these functions indiscriminately can result in unpredictable behavior from **getprotoent** because it relies on the current value of the

file pointer.

This function returns a pointer to a protoent structure in the SYS:ETC\PROTOCOL file. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.);

The protoent structure has the following format:

```
struct  protoent
{
   char   *p_name;       /* official protocol name */
   char   **p_aliases;   /* alias list */
   int    p_proto;       /* protocol # */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetprotoent**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getprotobyname, **getprotobynumber**, **getprotoent**, **socket**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

NWgetservbyname

Returns information about a service, given its name
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct servent *NWgetservbyname (
   struct nwsockent   *nwsktent,
   char               *name,
   char               *proto);
```

Parameters

nwsktent

(IN) Points to a context block.

name

(IN) Points to the official name of the service.

proto

(IN) Points to the name of the protocol to use when contacting a service.

Return Values

NWgetservbyname returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remarks

The **NWgetservbyname** function accepts pointers to two character strings representing the service and protocol name for which to search. Using **NWgetservent**, the function searches for a service from the beginning of the file and continues until it finds a matching entry or reaches end-of-file.

NWgetservbyname returns a pointer to a servent structure in the SYS:ETC\SERVICES file. If *proto* is NULL, **getservbyname** returns the first entry matching the service name.

NWgetservbyname is identical to **getservbyname** except your application must pass the address of an nwsockent structure. The result of **NWgetservbyname** stays intact until the application makes the next call.

NWgetservbyname is useful for applications spawning multiple threads, where each thread is accessing the SERVICES file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetservbyname** as for **NWgetservent**, **NWgetservbyport**, **NWsetservent**, and **NWendservent** for each context block to access the SERVICES file. Using these functions indiscriminately can result in unpredictable behavior from **getservent** because it relies on the current value of the file pointer.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetservbyname**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

bind, **connect**, **getservbyname**, **getservbyport**, **getservent**, **NWgetservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

NWgetservbyport

Returns information about a service, given its port number
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct servent *NWgetservbyport (
   struct nwsockent  *nwsktent,
   int                port,
   char              *proto);
```

Parameters

nwsktent

(IN) Points to a context block.

port

(IN) Indicates the port number at which the service resides, in network order.

proto

(IN) Indicates the name of the protocol to use when contacting a service.

Return Values

NWgetservbyport returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remarks

The **NWgetservbyport** function accepts a pointer to a character string representing the protocol name and an integer representing the port number. Calling **NWgetservent**, **NWgetservbyport** searches from the beginning of the file and continues until it finds a service matching the port number or reaches end-of-file. If a protocol name is also specified, then the searches must also match the protocol name. (The numeric protocol number is expressed in network byte order). If the protocol name is NULL, the first matching entry with the given port number is

returned.

You can call **NWgetservbyport** to access the SERVICES file. **NWgetservbyport** is identical to **getservbyport** except your application must pass the address of an nwsockent structure. The result of **NWgetservbyport** stays intact until the application makes the next call.

NWgetservbyport is useful for applications spawning multiple threads, where each thread is accessing the SERVICES file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetservbyport** as for **NWgetservent**, **NWgetservbyname**, **NWsetservent**, and **NWendservent** for each context block to access the SERVICES file. Using these functions indiscriminately can result in unpredictable behavior from **getservent** because it relies on the current value of the file pointer.

NWgetservbyport returns a pointer to a servent structure in the SYS:ETC\SERVICES file.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetservbyport**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getservbyname, **getservbyport**, **getservent**, **setservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

NWgetservent

Returns the next sequential entry from the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

struct servent *NWgetservent(
   struct nwsockent   *nwsktent);
```

Parameters

nwsktent

(IN) Points to a context block.

Return Values

NWgetservent returns a pointer to a structure of type servent if successful or NULL if an error occurs.

Remarks

The **NWgetservent** function returns the next sequential entry from the SYS:ETC\SERVICES file, opening the file if necessary.

NWgetservent returns a pointer to a servent structure in the SYS:ETC\SERVICES file.

NWgetservent is identical to **getservent** except your application must pass the address of an nwsockent structure. The result of **NWgetservent** stays intact until the application makes the next call.

NWgetservent is useful for applications spawning multiple threads, where each thread is accessing the SERVICES file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWgetservent** as for

NWgetservbyport, **NWgetservbyname**, **NWsetservent**, and **NWendservent** for each context block to access the SERVICES file. Using these functions indiscriminately can result in unpredictable behavior from **getservent** because it relies on the current value of the file pointer.

The servent structure has the following format:

```
struct servent
{
   char   *s_name;      /* official service name */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port # */
   char   *s_proto;     /* protocol to use */
};
```

NOTE: Calling any database routine overwrites the results of the last call to **NWgetservent**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

getservbyname, **getservbyport**, **getservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

NWinet_ntoa

Converts an Internet address in in_addr format into an ASCII string representing the address in dotted notation

Local Servers: nonblocking

Remote Servers: N/A

NetWare Server: 2.x, 3.x, 4.x

Platform: NLM

Service: Internet Network Library

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
NETINET_DEFINE_CONTEXT

char *NWinet_ntoa (
   char            *cp,
   struct in_addr   in);
```

Parameters

cp

(IN) Points to the character string that represents the Internet address.

in

(IN) Indicates the Internet address, in in_addr format.

Return Values

NWinet_ntoa returns a pointer to a character array containing the ASCII string representing the Internet address in dotted notation. **NWinet_ntoa** does not return any error value.

Remarks

Your application must pass a pointer to a character array of at least 18 bytes. **NWinet_ntoa** fills the array with the converted address and returns the pointer to the character array. Your application must pass the address to be converted in the structure in_addr. Since the application is allocating the character array, it can call **NWinet_ntoa** for multiple threads where each thread can define its own array.

See Also

inet_ntoa

inet_ntoa

NWsethostent

Opens the SYS:\ETC\HOSTS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWsethostent(
   struct nwsockent  *nwsktent,
   int                stayopen);
```

Parameters

nwsktent

(IN) Points to a context block.

stayopen

(IN) Causes SYS:ETC\HOSTS to remain open (if nonzero) after a calling **NWgethostbyname** or **NWgethostbyaddr**.

Return Values

None.

Remarks

The **NWsethostent** function opens the SYS:ETC\HOSTS file and rewinds it if the file is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\HOSTS file is not closed after each call made to **NWgethostent** by **NWgethostbyname** or **NWgethostbyaddr**.

The **NWsethostent** is identical to **sethostent** except your application must pass the address of an nwsockent structure.

NWsethostent accesses the SYS:ETC\HOSTS file only. Use **NetDBsethostent** to transparently access Internet name services in addition to the SYS:ETC\HOSTS file. See Internet and NetWare in Internet Network Library: Concepts for more information.

The result of **NWsethostent** stays intact until the next call is made.

NWsethostent is useful for applications spawning multiple threads, where each thread is accessing the HOSTS file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in Internet and NetWare in Internet Network Library: Concepts. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWsethostent** as for **NWgethostent**, **NWgethostbyaddr**, **NWgethostbyname** and **NWendhostent** for each context block to access the HOSTS file. Calling these functions indiscriminately can result in unpredictable behavior from **gethostent** because it relies on the current value of the file pointer.

For a description of the HOSTS file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, **gethostbyname**, **gethostent**, **NetDBgethostbyaddr**, **NetDBgethostbyname**, **NetDBgethostent**, **NetDBsethostent**, **NWgethostbyaddr**, **NWgethostbyname**, **NWgethostent**, **sethostent**

NWsetnetent

Opens the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

void NWsetnetent (
   struct nwsockent  *nwsktent,
   int                stayopen);
```

Parameters

nwsktent

(IN) Points to a context block.

stayopen

(IN) Causes SYS:ETC\NETWORKS to remain open (if nonzero) after a call to **NWgetnetbyname** or **NWgetnetbyaddr**.

Return Values

None.

Remarks

The **NWsetnetent** function opens the SYS:ETC\NETWORKS file and rewinds the file if it is open. If the *stayopen* flag is set (nonzero), the SYS:ETC\NETWORKS file is not closed after each call made to **NWgetnetent** by **NWgetnetbyname** or **NWgetnetbyaddr**.

The **NWsetnetent** function is identical to **setnetent** except your application must pass the address of a structure of type nwsockent.

The result of **NWsetnetent** stays intact until the application makes the next call. **NWsetnetent** is useful for applications spawning multiple threads, where each thread accesses the NETWORKS file.

Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in Internet Network

Library. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWsetnetent** as for **NWgetnetent**, **NWgetnetbyaddr**, **NWgetnetbyname** and **NWendnetent** for each context block to access the NETWORKS file. Calling these functions indiscriminately can result in unpredictable behavior from **getnetent** because it relies on the current value of the file pointer.

For a description of the NETWORKS file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

endnetent, **getnetbyaddr**, **getnetbyname**, **getnetent**, **NWgetnetent**, **NWgetnetbyaddr**, **NWgetnetbyname**, **setnetent**

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

NWsetprotoent

Opens the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWsetprotoent (
   struct nwsockent  *nwsktent,
   int                stayopen);
```

Parameters

nwsktent

(IN) Points to a context block.

stayopen

(IN) Causes SYS:ETC\PROTOCOL to remain open (if nonzero) after a call to **NWgetprotobyname** or **NWgetprotobynumber**.

Return Values

None.

Remarks

The **NWsetprotoent** function opens the SYS:ETC/PROTOCOL file and rewinds the file if it is already open. If the *stayopen* flag is set (nonzero), the PROTOCOL file is not closed after each call made to **NWgetprotoent** by **NWgetprotobyname** or **NWgetprotobynumber**.

NWsetprotoent is identical to **setprotoent** except your application must pass the address of an nwsockent.

The result of **NWsetprotoent** stays intact until the application makes the next call. **NWsetprotoent** is useful for applications spawning multiple threads, where each thread is accessing the PROTOCOL file. Your application can declare separate nwsockent structures for each thread. It must follow the instructions specified in TBS. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWsetprotoent** as for **NWgetprotoent**, **NWgetprotobyname**, **NWgetprotobynumber** and **NWendprotoent** for each context block to access the PROTOCOL file. Calling these functions indiscriminately can result in unpredictable behavior from **getprotoent** because it relies on the current value of the file pointer.

For a description of the PROTOCOL file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

endprotoent, **getprotobyname**, **getprotobynumber**, **getprotoent**, **NWendprotoent**, **NWgetprotoent**, **NWgetprotobyname**, **NWgetprotobynumber**, **setprotoent**

Example

See the sample file in SYS:ETC\SAMPLES\PROTOCOL.

NWsetservent

Opens the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void NWsetservent (
   struct nwsockent  *nwsktent,
   int                stayopen);
```

Parameters

nwsktent

(IN) Points to a context block.

stayopen

(IN) Causes SYS:ETC\SERVICES to remain open (if nonzero) after a call by **NWgetservbyname** or **NWgetservbyport**.

Return Values

None.

Remarks

The **NWsetservent** function opens the SYS:ETC\SERVICES file and rewinds the file if it is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\SERVICES file is not closed after each call made to **NWgetservent** by **NWgetservbyname** or **NWgetservbyport**.

You can call **NWsetservent** to access the SERVICES file. **NWsetservent** is identical to **setservent** except your application must pass the address of an nwsockent structure. The result of **NWsetservent** stays intact until the application makes the next call.

NWsetservent is useful for applications spawning multiple threads, where each thread is accessing the SERVICES file. Your application can declare separate nwsockent structures for each thread.

Your application must follow the instructions specified in TBS. Any deviation can result in unpredictable behavior.

The NetWare library uses the same file pointer for **NWsetservent** as for **NWgetservbyport**, **NWgetservbyname**, **NWgetservent**, and **NWendservent** for each context block to access the SERVICES file. Using these functions indiscriminately can result in unpredictable behavior from **getservent** because it relies on the current value of the file pointer.

For a description of the SERVICES file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

endservent, **NWendservent**, **setservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

sethostent

Initializes sequential access to the HOSTS database
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void sethostent (
   int   stayopen);
```

Parameters

stayopen

(IN) Causes SYS:ETC\HOSTS to remain open (if nonzero) after a call to **gethostbyname** or **gethostbyaddr**.

Return Values

None.

Remarks

The **sethostent** function is defined as a macro in NETDB.H. You can use this macro to access the HOSTS file or host information from Internet name services. **sethostent** expands to **NWsethostent**, providing only access to the SYS:ETC\HOSTS file, or to **NetDBsethostent**, providing transparent access to this file in addition to Internet name services such as NIS. See Internet and NetWare in Internet Network Library: Concepts for more information. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **sethostent** function opens the SYS:ETC\HOSTS file and rewinds the file if it is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\HOSTS file is not closed after each call made to **gethostent** by **gethostbyname** or **gethostbyaddr**. **sethostent** also makes sure that the next call to **gethostent** reads from the beginning of the SYS:ETC\HOSTS file.

NOTE: If your application spawns multiple threads, call either **NWsethostent** or **NetDBsethostent**.

For a description of the HOSTS file, refer to the*TCP/IP Transport Supervisor's Guide*.

See Also

gethostbyaddr, **gethostbyname**, **gethostent**, **NetDBgethostbyname**, **NetDBgethostent**, **NetDBsethostent**, **NWgethostent**, **NWgethostbyaddr**, **NWgethostbyname**, **NWsethostent**

setnetent

Opens the SYS:\ETC\NETWORKS file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <netdb.h>
#include <sys/types.h>

void setnetent (
   int   stayopen);
```

Parameters

stayopen

(IN) Causes SYS:ETC\NETWORKS to remain open (if nonzero) after a call to **getnetbyname** or **getnetbyaddr**.

Return Values

None.

Remarks

The **setnetent** function is defined as a macro in NETDB.H. You can use this macro to access the NETWORKS file. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any of the source files that include NETDB.H.

The **setnetent** function opens the SYS:ETC\NETWORKS file and rewinds the file if it is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\NETWORKS file is not closed after each call made to **getnetent** by **getnetbyname** or **getnetbyaddr**.

NOTE: If your application spawns multiple threads, call **NWsetnetent**.

For a description of the NETWORKS file, refer to the *TCP/IP Transport TCP/IP Supervisor's Guide*.

See Also

endnetent, getnetbyaddr, getnetbyname, getnetent, NWgetnetbyaddr, NWgetnetbyname, NWgetnetent, NWsetnetent

Example

See the sample files in SYS:ETC\SAMPLES\NETWORKS.

setprotoent

Opens the SYS:\ETC\PROTOCOL file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void setprotoent (
   int   stayopen);
```

Parameters

stayopen

(IN) Causes SYS:ETC\PROTOCOL to remain open (if nonzero) after a call to **getprotobyname** or **getprotobynumber**.

Return Values

None.

Remarks

The **setprotoent** function is defined as a macro in NETDB.H. This macro calls upon the corresponding NetWare Internet support function. Your application must include NETDB_DEFINE_CONTEXT in any one of the source files that include NETDB.H.

The **setprotoent** function opens the SYS:ETC/PROTOCOL file if it is not already open. It rewinds the file if it is open. If the *stayopen* flag is set (nonzero), the PROTOCOL file is not closed after each call made to **getprotoent** by **getprotobyname** or **getprotobynumber**. (This filename differs from the 4.3BSD filename PROTOCOLS because of the 8-character filename limitation imposed by NetWare.)

NOTE: If your application spawns multiple threads, call **NWsetprotoent**.

For a description of the PROTOCOL file, refer to the *TCP/IP Transport TCP/IP Supervisor's Guide*.

See Also

endprotoent, **getprotobyname**, **getprotobynumber**, **getprotoent**, **NWendprotoent**, **NWgetprotoent**, **NWgetprotobyname**, **NWgetprotobynumber**, **NWsetprotoent**

Example

See the sample files in SYS:ETC\SAMPLES\PROTOCOL.

setservent

Opens the SYS:\ETC\SERVICES file
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 2.x, 3.x, 4.x
Platform: NLM
Standard: BSD
Service: Internet Network Library

Syntax

```
#include <sys/types.h>
#include <netdb.h>

void setservent (
   int   stayopen);
```

Parameters

stayopen

(IN) Causes SYS:ETC\SERVICES to remain open (if nonzero) after a call by **getservbyname** or **getservbyport**.

Return Values

None.

Remarks

The **setservent** function opens the SYS:ETC\SERVICES file and rewinds the file if it is already open. If the *stayopen* flag is set (nonzero), the SYS:ETC\SERVICES file is not closed after each call made to **getservent** by **getservbyname** or **getservbyport**.

NOTE: If your application spawns multiple threads, call **NWsetservent**.

For a description of the SERVICES file, refer to the *TCP/IP Transport Supervisor's Guide*.

See Also

endservent, **NWendservent**, **NWsetservent**

Example

See the sample files in SYS:ETC\SAMPLES\SERVICES.

Communication Service Group

Internet Network Library: Structures

hostent

Defines an internet host
Service: Internet Network Library
Defined In: netdb.h

Structure

```
struct    hostent {
   char   *h_name;
   char   ** h_aliases;
   int    h_addrtype;
   int    h_length;
   char   ** h_addr_list;
#define h_addr   h_addr_list[0]
};
```

Fields

h_name

Points to the official name of the host.

h_aliases

Points to the first element in a list of addresses of alternative names for the host. A zero (0) terminates the list.

h_addrtype

Contains the type of address being returned. Currently, this is always AF_INET.

h_length

Contains the length in bytes of the host's address.

h_addr_list

Points to the first element in an array of pointers to in_addr structures containing the addresses for the host. A zero (0) terminates the list. Host addresses are returned in network byte order.

h_addr

Points to the first address in *h_addr_list*. This `define` is provided for backward compatibility.

in_addr

Defines an internet address
Service: Internet Network Library
Defined In: in.h

Structure

```
struct {
   union {
   struct {
         unsigned char s_b1, s_b2,s_b3,s_b4;
      } S_un_b;
      struct {
         unsigned short s_w1,s_w2;
      } S_un_w;
      unsigned long S_addr;
   } S_un;
} in_addr;
```

Fields

s_b1

s_b2

s_b3

.s_b4

.s_w1

.s_w2

S_addr

netent

Defines an internet network
Service: Internet Network Library
Defined In: netdb.h

Structure

```
struct netent
{
   char            *n_name;      /* official name of network */
   char            **n_aliases;  /* list of network aliases */
   int             n_addrtype;   /* network number type */
   unsigned long   n_net;        /* network number */
   unsigned long   n_mask;       /* net mask--Novell extension */
};
```

Fields

n_name

Points to the official name of the network.

n_aliases

Points to the first element in a list of pointers to alternate names (aliases) for the network. A NULL character terminates the list.

n_addrtype

Contains the type of the network number returned. Currently, this is always AF_INET.

n_net

Contains the network number, in host order.

n_mask

Contains the net mask. This field is a Novell extension of the structure.

nwsockent

Provides context in NetWare 386 environment
Service: Internet Network Library
Defined In: netdb.h

Structure

```
typedef struct nwsockent{
   FILE *nse_hostctx;
   FILE *nse_netctx;
   FILE *nse_protoctx
   FILE *nse_servctx
   int  nse_h_errno;
   union sockent {
      struct hostent nsu_hst;
      struct netent nsu_net;
      struct.netent nsu_proto;
      struct.servent nsu_serv;
   }      nse_sockent_un;
char      nse_scratch[SRATCHBUFSIZE];
} nwsockent;
```

Fields

No fields listed

protoent

Defines an internet protocol
Service: Internet Network Library
Defined In: netdb.h

Structure

```
struct  protoent {
   char   *p_name;
   char   **p_aliases;
   int    p_proto;
};
```

Fields

p_name

Points to the official name of the protocol.

p_aliases

Points to the first element in a list of pointers to alternate names (aliases) for the protocol. A NULL character terminates the list.

p_proto

Contains the protocol number.

servent

Defines an internet service
Service: Internet Network Library
Defined In: netdb.h

Structure

```
struct servent {
   char   *s_name;
   char   **s_aliases;
   int    s_port;
   char   *s_proto;
};
```

Fields

s_name

Points to the official name of the service.

s_aliases

Points to the first element in a list of pointers to alternate names (aliases) for the service. A NULL pointer terminates the list.

s_port

Contains the port number at which the service resides. Port numbers are returned in network byte order.

s_proto

Points to the name of the protocol to use when contacting the service.

IPX/SPX

IPX/SPX: Guides

IPX/SPX: Concept Guide

IPX/SPX: Concepts

Checksumming and Ethernet 802.3

Checksum values are stored in the checksum field of the IPX header. A value of 0FFFFh in the checksum field identifies ETHERNET_802.3 packets to LAN drivers. Consequently, applications can't use IPX checksums when transmitting packets across a network that uses ETHERNET_802.3. Specifically, applications should not use checksumming if the local DOS IPX engine is bound to an ETHERNET_802.3 board/frame type.

Parent Topic: Packet Checksums

Checksums and Packet Transmission

DOS IPX supports two functions that send packets:

IPXFastSend

IPXSendPacket

IPXFastSend speeds delivery by bypassing the normal error checking performed by **IPXSendPacket**. **IPXFastSend** leaves the IPX header unmodified, assuming the caller has included all the necessary data.

In contrast, **IPXSendPacket** initializes several IPX header fields before sending the packet, including the checksum field, which it always sets to 0FFFFh. In other words, this function always overwrites any current checksum value. To allow for unmodified checksums, call **IPXSendWithChecksum**.

Parent Topic: Packet Checksums

Connection-Mode Service: SPX

Communicating with SPX involves much of the same preparation required by IPX. You must initialize SPX, open a socket, and set up send and receive ECBs just as you would for IPX. SPX, however, requires a few additional steps to build the connection-oriented resources.

Related Topics

Establishing an SPX Connection

Transmitting Data with SPX

Releasing an SPX Connection

Connectionless Service: IPX

Once you have initialized IPX locally, you are ready to send and receive IPX packets.

Related Topics

Managing ECBs

Listening for IPX Packets

Sending IPX Packets

Deallocating IPX Resources

Scheduling Asynchronous Events with IPX

IPX Connectionless Mode Client: Example and IPX Connectionless Mode Server: Example

Deallocating IPX Resources

When you have finished communicating with a particular node, call **IPXDisconnectFromTarget**. This function allows network drivers to deallocate resource at the network protocol level.

Parent Topic: Connectionless Service: IPX

ECB Management

ECBs can be be used individually or can be used out of a queue. In other words multiple ECBs can be queued up for use by IPX/SPX in turn. You need to decide whether to use individual ECBs or the queueing feature of IPX/SPX. Another decision you must address is where to store the data that will be placed in the packet that goes out onto the network wire. You can use a single fragment of memory that contains all the data (including the header) of the packet that will be sent or you can have multiple fragments of memory that will contain the data that will be sent.

This section discusses both ECB queue usage and managing packet fragments.

Packet Fragments

IPX assembles a packet from the buffers or fragments referenced by the ECB. The ECB stores these values as an array of fragment descriptors along with a fragment count that indicates the length of the array.

You can use as many fragments as are necessary. The only requirement is that the first 30 bytes of the first buffer must be devoted to the IPX header (42 bytes for an SPX header) and the entire packet length cannot exceed 576 bytes.

In most situations two buffers are adequate for managing a packet. The first buffer usually contains the IPX or SPX header, and the second buffer contains the message data.

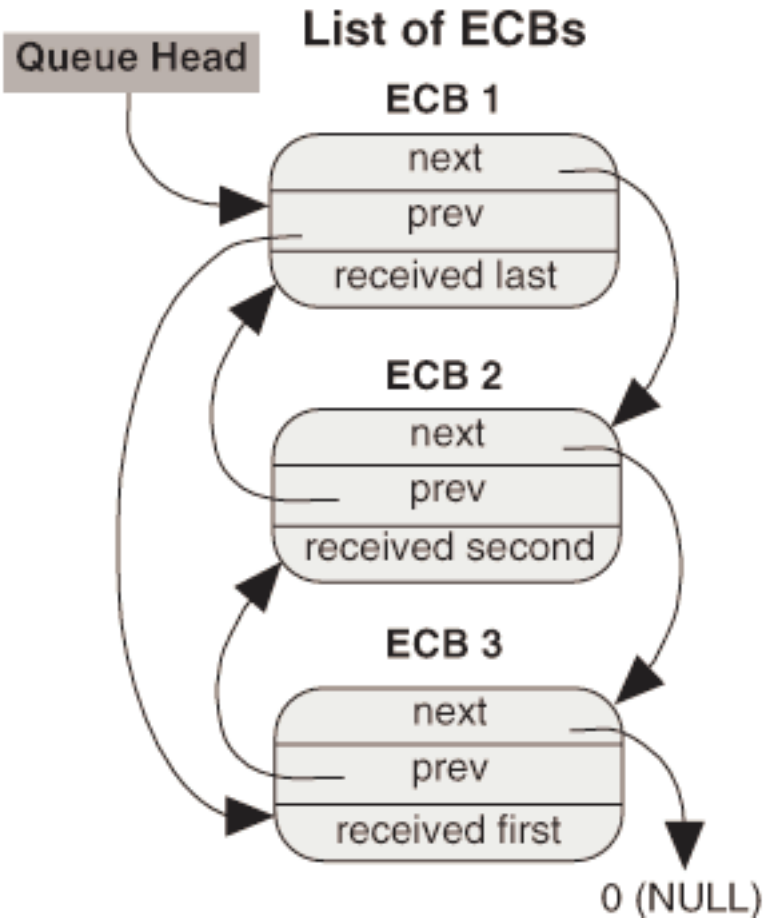
ECB Queues

When an ECB with a nonNULL *queueHead* field is used, a queue (or list) of ECBs is built from each ECB which shares a queue head as the event associated with the ECB occurs.

This list of ECBs, which are linked in reverse order of the completion of the associated events, are doubly linked by means of a *next* and *prev* pointer. The first ECB in the list is for the most recent event and the last ECB is for the first event.

The *prev* pointer in the list's first (most recent) ECB points to the last ECB in the list and the last ECB's *next* pointer is NULL. To process the ECBs in the order received, the application uses the first ECB's prev pointer to get a pointer to the last ECB and then follows the prev pointer until it reaches the first ECB (see the following figure).

Figure 11. ECB List



Most applications have several functionally equivalent ECBs outstanding simultaneously. If the application uses the queue head feature, it typically gives all of the pending ECBs (those that are dedicated to a particular use) the same queue head. Thus, the same queue head can process the completed ECBs from any and all of several listens.

Whenever the application does process the ECBs queued on a particular queue head, the application should call **IpxGetAndClearQ** to retrieve the pointer to the most recent ECB on the queue and to simultaneously clear the queue head.

> **NOTE:** An application should set a queue head to NULL before it is used the first time.

Parent Topic:

Enabling Interrupts

An event service routine can execute for longer periods if it enables interrupts or invokes a procedure that temporarily enables interrupts. However, if IPX calls the event service routine, problems can occur if the routine is not reentrant and enables interrupts.

Under these circumstances, the event service routine should take measures to prevent itself from recurring. If the AES calls the event service routine, enabling interrupts will prevent the AES from calling other event service routines scheduled during the same clock interval. These routines will be delayed until the next system clock tick.

Parent Topic: Event Service Routines

Establishing an SPX Connection

The explanation of how to set up ECBs for IPX applies equally to SPX (see Managing ECBs). SPX needs an additional ECB to establish the SPX connection. Generally, you will want to allocate a separate ECB for this purpose since the initial ECB requires no event service routine and references no data other than the SPX header.

To set up the connection, one node acts as the caller and the other acts as the listener. Once the connection has been established, SPX makes no distinction between the caller and sender.

The calling node calls **SPXEstablishConnection**. This function takes the initial ECB, a retry count, and a watchdog flag as input. The retry count specifies how many times SPX will continue sending unacknowledged packets.

The watchdog flag allows the caller to enable a local watchdog process to monitor the SPX connection after it is established. (If you enable the watchdog process, you should allocate an additional receive ECB.) If the request is successful, the caller receives an SPX connection ID that identifies the connection.

At the other end of the connection, the listening node calls **SPXListenForConnection**. This function takes parameters similar to **SPXEstablishConnection**. **SPXListenForConnection** can be cancelled by calling **IPXCancelEvent**.

Related Topics:

Connection-Mode Service: SPX

SPX TLI Multiple Connection Server: Example

Event Control Blocks

Event Control Blocks (ECBs) are the link between your application and IPX. Each ECB contains the information IPX needs to send or receive a packet tailored to the needs of your application. Setting up the ECBs is the main hurdle you face when programming IPX or SPX.

ECBs can be socket-based or socketless.

Socket-based ECBs handle IPX/SPX events.

Socketless ECBs schedule events without reference to IPX/SPX packet transmissions.

A socket-based ECB includes the following fields:

Link address

ESR address

In use flag

Completion code

Socket number

IPX workspace

Driver workspace

Immediate address

Fragment count

ECB fragment descriptor list

ECB contains the ECB data.

Socketless ECBs are timed by the Asynchronous Event Scheduler (AES) and consequently don't require a socket. The fields in a socketless ECB are a subset of those found in the socket-based ECB:

Link address

ESR address

In-use flag

AES workspace

The structure IPX_ECB, which is identical to the SPX_ECB structure, is

defined in NWIPXSPX.H and consists of the following fields:

```
unsigned long          semHandleSave;       /*R*/
struct IPX_ECBStruct   **queueHead;         /*sr*/
struct IPX_ECBStruct   *next;               /*A*/
struct IPX_ECBStruct   *prev;               /*A*/
short                  status;              /*q*/
unsigned long          semHandle;           /*sr (ignored by
                                              IpxSend) */
unsigned short        IProtID;              /*R*/
unsigned char         protID [6];           /*R*/
unsigned long         boardNumber;          /*R*/
unsigned char         immediateAddress [6]; /*s*/
unsigned char         driverWS [4];         /*R*/
unsigned long         ESREBXValue;          /*R*/
unsigned short        socket;               /*sr (ignored if
                                              the socket
                                              parameter is 0
                                              and on SPX
                                              functions which
                                              take the
                                              connection
                                              number as 0) */
unsigned short        protocolWorkspace;    /*R*/
unsigned long         dataLen;              /*q*/
unsigned long         fragCount;            /*sr*/
ECBFrag               fragList [2];         /*sr*/
```

In the preceding comments:

R	Indicates fields that are reserved.
s	Indicates fields the application must set when using the ECB to send a packet.
r	Indicates fields the application must set when using the ECB to receive a packet.
A	Indicates fields that may be used when the ECB is not in use by IPX/SPX.
q	Indicates fields the application may read.

IPX and SPX use the *semHandleSave* field for internal purposes and must not be modified.

The *queueHead* field is set by the application either to a pointer to an ECB pointer (double pointer to an ECB) or else to NULL. The *queueHead* field is set to an ECB pointer pointer to cause IPX and SPX to queue ECBs for completed events, using the ECB pointer pointed to by the *queueHead* field as the head of the queue. The application can set this field to NULL if it does not need this feature.

IPX/SPX uses the *next* and *prev* fields for internal purposes. IPX and SPX maintain these fields while the ECB is in use. When the ECB is not in use, the application can use these fields (if necessary). Most commonly, the *next* and *prev* fields are used by the application as link fields for keeping the ECB in a free list. These fields are also used by IPX/SPX to queue ECBs off of the queue head if the *queueHead* field is nonNULL.

The *status* field indicates the current state of the event (for example, SPX is listening on a socket) and to indicate whether the event was completed successfully. If this field is positive, the event has not yet occurred. If this field is zero, the event completed successfully. If this field is negative, the event completed with an error. The value of status in this case indicates the type of error.

The *semHandle* field is set by the application either to a semaphore handle returned by **OpenLocalSemaphore** or to NULL. The purpose of the semaphore handle is to allow the application to block pending the completion of one or more IPX/SPX events. Whenever the event (associated with any of the IPX/SPX functions that take an ECB as a parameter) occurs, and the *semHandle* field is nonNULL, IPX/SPX calls **SignalLocalSemaphore** with the specified semaphore handle. This unblocks the application if the application was blocked on the semaphore.

NOTE: The semaphore handle is undefined until the ECB completes.

The *IProtID*, *boardNumber*, *driverWS*, *ESREBXValue*, *protocolWorkspace*, and *dataLen* fields are used internally by IPX/SPX and the driver and must not be modified. *IProtID*, *protID*, *boardNumber*, *driverWS*, *ESREBXValue*, and *protocolWorkspace* are reserved fields.

The field *immediateAddress* holds the address of the node to which the packet is sent or from which it arrived. This is the address of an internetwork bridge on the local network if the packet is not sent to or received from a node on the local network. **GetLocalTarget** can be used to get the information for this field. This field needs to be initialized only for **IpxSend**.

The *socket* field identifies the sending or receiving socket with which the ECB is associated. This field can be used two ways:

The application sets this field to the desired socket number and passes zero as the socket number to those functions that take both a socket number and an ECB as parameters.

The application does not fill this field, but passes the actual socket to the above-mentioned functions.

The *fragCount* field indicates the number of buffers from which an outbound packet is built or into which an inbound packet is dispersed. The number of buffers (one or more) is given by *fragCount*. The application provides a list of fragment descriptors, at the end of the ECB, which contains the address and size of the buffers.

The first *fragList* must describe a buffer large enough to hold at least the

packet header for the service being used. That is, for IPX packets the first *fragList* must describe a buffer of at least length 30, and for SPX the first *fragList* must describe a buffer of at least length 42. Note that the IPX_ECB (and SPX_ECB) defines two fragments.

Event Service Routine Processing

An event service routine behaves like an interrupt service routine. IPX calls the event service routine with interrupts disabled. Consequently, if you use an event service routine, it should execute as quickly as possible. When the routine is finished, IPX returns control to your application.

An event service routine should not perform tasks that the main body of your application can perform just as effectively. Typically, an event service routine places its associated ECB in a queue and returns. The main body of the application monitors the queue, processes the data, and hands the ECB back to IPX.

Parent Topic: Event Service Routines

Event Service Routines

The ESR Address field in the ECB can contain the address of an event service routine that your application wants to execute in response to an IPX event. The event triggering the event service routine might be the sending of a packet, the receiving of a packet, the recurrence of an IPX event rescheduling itself, or an event scheduled by your application.

After processing the event, IPX resets the ECB's In-use flag to zero, enters a completion code, removes the ECB from its internal lists, and calls the event service routine. At that point, IPX is finished and you are free to handle the ECB and any associated data as you see fit.

Related Topics

The Event Service Routine Interface

Event Service Routine Processing

Enabling Interrupts

Event Service Routines and IPX Requests

Parent Topic: IPX/SPX Local Management Issues

Event Service Routines and IPX Requests

An event service routine called by IPX can freely call any IPX or AES function except these functions:

IPXCloseSocket

IPXDisconnectFromTarget

These functions should not be called by an event service routine.

An event service routine can reschedule itself for delayed execution by calling either of these functions:

IPXScheduleIPXEvent (if it was called by IPX)

IPXScheduleSpecialEvent (if it was called by the AES)

In either case, the event service routine passes the address of the procedure to its associated ECB and then executes an RETF instruction.

All IPX Functions except for the following execute entirely in an interrupts-disabled state:

IPXSendPacket

IPXGetLocalTarget

IPXRelinquishControl

If you are calling any of these functions from inside code that is running with interrupts disabled, be sure your code can survive if interrupts are enabled temporarily.

Parent Topic: Event Service Routines

IPX and SPX Overview

IPX™ and SPX™ are the native communication services of the NetWare® OS. IPX is used by the NetWare workstation software to communicate with NetWare servers. SPX adds connection-oriented enhancements to IPX to provide a reliable transport for client applications. To use IPX or SPX, you must load the IPX communication software on the local workstation. Other workstation software can also be loaded but is not required.

IPX Features

IPX deals in datagrams, lone packets of data requiring no acknowledgements or sequencing. Datagrams reduce network traffic and streamline network performance. In the case of IPX the success rate for datagram delivery is typically about 95%. Consequently, if you use IPX you need to develop a strategy for confirming deliveries and, possibly, keeping packets in order.

If you are transmitting brief messages that fit into a single packet and don't require an acknowledgement, you may be able to use IPX as-is. Also, if you are using IPX to perform simple request/reply transactions, the reply itself can serve as an acknowledgement. If no reply is received within a certain period of time, you can assume the delivery failed and react accordingly.

SPX Features

On the other hand, if you need to know with certainty that data is being received in the order it is sent, you must build such guarantees into your application or use SPX. SPX provides the packet sequencing and guaranteed delivery associated with a connection-mode service. These features add some overhead to network communications, but they come ready-to-use and require no additional work on your part.

IPX Function List

These functions perform IPX operations.

Function	Header	Comment
IPXCancelEvent	nxtd.h nxtw.h	Cancels a pending event.
IPXCloseSocket	nxtd.h nxtw.h	Closes the specified IPX socket and cancels any events defined by ECBs associated with the socket.
IPXDisconnectFromTarget	nxtd.h nxtw.h	Informs IPX that the connection with the specified node is no longer needed.
IPXGetInternetworkAddress	nxtd.h nxtw.h	Returns the network and node address of the requesting workstation.
IPXGetIntervalMarker	nxtd.h nxtw.h	Returns an IPX time marker. This function lets applications measure the elapsed time between two events.
IPXGetLocalTarget	nxtd.h nxtw.h	Gets the value to be placed in an ECB's immediate address field.
IPXGetMaxPacketSize	nxtd.h nxtw.h	Finds the maximum packet size allowed by a workstation's network topology.
IPXInitialize	nxtd.h nxtw.h	Gets the entry address to the IPX interface.

IPXListenForPacket	nxtd.h nxtw.h	Prepares IPX to receive a packet.
IPXOpenSocket	nxtd.h nxtw.h	Opens an IPX socket.
IPXRelinquishControl	nxtd.h nxtw.h	Relinquishes control of the workstation's CPU.
IPXScheduleIPXEvent	nxtd.h nxtw.h	Schedules an IPX event.
IPXSendPacket	nxtd.h nxtw.h	Sends an IPX packet to the ECB-defined node and socket.

IPX Initialization

For Assembly programmers IPX and SPX functions are accessed by making a far call to the IPX interface. To obtain this address, you execute the DOS multiplex interrupt. For example, the following instructions will return the address:

```
IPXLocation dd
mov   AX, 7A00h
int   2Fh
cmp   AL, 0FFh
jne   NoIPX
mov   AX, ES
mov   IPXLocation, DI
mov   IPXLocation + 2, AX
```

If IPX is loaded, it sets the low byte of AX to FFh and returns its far call entry address in ES:DI. If IPX has not been loaded, the interrupt leaves AL unmodified. You can make IPX and SPX calls by preparing the appropriate registers and then calling the entry point. The BP register is not preserved on SPX calls.

C programmers can call **IPXInitialize** to check whether IPX is loaded. If it is, this function stores the IPX entry address internally for the API library to use.

Parent Topic: IPX/SPX Local Management Issues

IPX Packet Structure

The structure of an IPX packet is identical to the structure of a Xerox Network Standard (XNS) packet. The packet consists of a 30-byte header followed by 0 to 546 bytes of data. The minimum packet size is 30 bytes (the header only) and the maximum packet size is 576 bytes. The content and

structure of the data portion are entirely the responsibility of the application using IPX and can take any format.

The NWIPXSPX.H include file contains definitions for all the C structures needed for IPX, SPX headers, and ECB structures.

The structure IPX_HEADER is defined in NWIPXSPX.H and consists of the following fields:

```
unsigned short   checksum;       /* high/low */
unsigned short   packetLen;      /* high/low */
unsigned char    transportCtl;
unsigned char    packetType;
unsigned long    destNet;        /* high/low */
unsigned char    destNode[6];    /* high/low */
unsigned short   destSocket;     /* high/low */
unsigned long    sourceNet;      /* high/low */
unsigned char    sourceNode[6];  /* high/low */
unsigned short   sourceSocket;   /* high/low */
```

The *checksum* field contains a dummy checksum of the packet contents and is always set by IPX to 0xFFFF.

The *packetLen* field contains the length of the complete IPX packet (30 to 576 bytes) and is set by IPX.

The application must set the *packetType* and *destNet, destNode, destSocket* fields before sending an IPX packet. The remaining fields are set by IPX.

NetWare internetwork bridges use the *transportCtl* field to monitor the number of bridges or routers that a packet has crossed. The packets are discarded by the 16th bridge they encounter. IPX sets this field to zero before sending the packet.

The *packetType* field identifies the type of service offered or required by the packet.

Xerox* has defined the following packet types:

0	Unknown Packet Type
1	Routing Information Packet (RIP)
2	Echo Packet
3	Error Packet

Novell® has defined the following packet types:

4	Packet Exchange Packet (IPX)
5	Sequenced Packet Protocol Packet (SPX)

16-31	Experimental Protocols
17	NetWare Control Protocol™ (NCP™) Packet
20	NetBIOS Name Packet

IPX users should set *packetType* to either 0 or 4. SPX sets the packet type to 5 for packets of that protocol.

The *destNet* and *sourceNet* fields identify the network address of the target or source application. The network address is a 4-byte number assigned to each physical cabling segment. This address is determined and assigned by the network administrator.

The *destNode* and *sourceNode* fields contain the 6-byte number that identifies the LAN board within the target or source network station (or node). A value of 0xFFFFFFFFFFFF is placed in the node field to indicate a broadcast.

The *destSocket* and *sourceSocket* fields contain the socket address, a 2-byte number that identifies a process within a node. This is an IPX socket and must be opened using the IPX open socket function. If the destination socket is not opened communication will not occur.

Parent Topic: Packet Information

IPX/SPX Local Management Issues

IPX and SPX local management issues include initializing the API library (for C programmers), opening and closing sockets, scheduling asynchronous events, developing event service routines, and managing ECB queues and fragments.

Initializing IPX and SPX: Before you can call the functions in the IPX/SPX API library, you must initialize the IPX environment. Optionally, you can initialize SPX to ensure local compatibility with SPX functions. This initializaiton does not apply if programming an NLM. See the following:

IPX Initialization

SPX Initialization

Related Topics

Managing IPX Sockets

Event Service Routines

ECB Management

IPX/SPX NLM Function List

These functions are available for NLMs. They compare to the functions shown above and for the most part are the same, but there are some differences.

Table auto. Communication Services Functions

Function	Purpose
IpxCancelEvent/ IpxCancelPacket	Cancels an ECB being used by **IpxReceive**
IpxCheckSocket	Determines if the specified socket is open
IpxCloseSocket	Closes an IPX socket
IpxConnect	Creates a virtual connection to the specified address
IpxDisconnect	Destroys a virtual connection to the specified address
IpxGetAndClearQ	Returns a pointer to the list of ECBs queued off of a specified queue head and then sets the queue head to NULL
IpxGetInternetworkAddress	Returns the network and node address of the requesting NLM
IpxGetLocalTarget	Fills the *immediateAddress* field in an ECB
IpxGetStatistics	Returns diagnostic statistics maintained by IPX (Not implemented; included for compatibility)
IpxGetVersion	Returns the major and minor version and the revision of IPX
IpxOpenSocket	Opens an IPX socket
IpxReceive	Initiates the receiving of an IPX packet
IpxResetStatistics	Resets diagnostic statistics maintained by IPX (Not implemented; included for compatibility)
IpxSend	Initiates the sending of an IPX packet
SpxAbortConnection	Aborts an SPX connection
SpxCancelEvent/ SpxCancelPacket	Cancels a pending SPX event
SpxCheckSocket	Checks the status of a specified socket number

SpxCloseSocket	Closes an SPX socket
SpxEstablishConnection	Attempts to establish an SPX connection with a listening socket
SpxGetConfiguration	Determines the maximum number of SPX connections and the number of available SPX connections
SpxGetConnectionStatus	Returns the status of an SPX connection
SpxGetTime	Gets a time marker from SPX
SpxGetVersion	Returns the major and minor version, revision, and revision date of SPX
SpxListenForConnection	Attempts to receive an Establish Connection packet and thereby establish an SPX connection with a remote partner
SpxListenForSequencedPacket	Passes an ECB to SPX for the purpose of receiving a sequenced packet
SpxOpenSocket	Opens an SPX socket
SpxSendSequencedPacket	Sends an SPX packet
SpxTerminateConnection	Terminates an SPX connection

Listening for IPX Packets

To listen for packets, call **IPXListenForPacket**. As input, this function takes an ECB prepared for receiving packets. Call this function for each ECB you want to post on a socket. There is no limit to the number of ECBs you can post.

While IPX is using the ECB, the In-use flag is a nonzero value. Once a packet has been received on an ECB, the In-use flag is set to zero and an appropriate value is entered in the completion code field. When IPX is finished with the receive ECB it invokes the ECB's event service routine if you have assigned one.

Parent Topic: Connectionless Service: IPX

Look Ahead Socket Restrictions

Sockets opened by **IPXOpenLookAheadSocket** are handled a little differently from sockets opened by **IPXOpenSocket**. The following restrictions apply to look ahead sockets:

All look ahead sockets are opened long term. Therefore, an application that opens a look ahead socket should always close the socket before the application terminates.

Receive ECBs cannot be posted on a look ahead socket.

SPX functions cannot be used with a look ahead socket.

Despite these differences, you close a look ahead socket as you would a normal socket by calling **IPXCloseSocket**. Also, like a normal socket, a look ahead socket can be used to send packets or to schedule asynchronous events.

Parent Topic: Look Ahead Sockets

Look Ahead Sockets

The function **IPXOpenLookAheadSocket** has recently been added to NetWare. They are not available for NLM applications. This function lets an application open a look ahead socket and register a **ReceiveLookAheadHandler** with IPX.

Related Topics

The Receive Look Ahead Handler

Look Ahead Socket Restrictions

The Look Ahead Structure

Managing ECBs

The first step in exchanging packets is to set up a pool of ECBs to receive incoming data and a pool of send ECBs for outgoing data. The number of ECBs you allocate depends on what your application needs to do with the data associated with the ECBs. For example, at some point you must remove the incoming data from the input buffer referenced by the ECB's fragment list.

Allocating ECBs

Until you process the ECB and return it to IPX, the ECB can no longer receive incoming data. Other ECBs must be available to pick up the slack. The more processing your application performs on the input buffer, the more ECBs you will want to allocate. It is better to have a few too many than not enough.

Queueing ECBs

You need to devise some strategy for queuing and scanning the ECBs, such as a linked list. For the receive ECBs, your event service routine can process the ECB. When IPX uses the ECB, it invokes the event service routine, which at that time can enter the ECB into a queue. The application can check the queue periodically and process the ECBs.

Initializing ECBs

As you allocate and prepare the ECBs, several fields need to be initialized. For receive ECBs, fill in the following fields:

Event service routine address

Socket number

Fragment list fields

For send ECBs, fill in these fields:

The event service routine address

Socket number

Immediate address

Packet type

Destination address

Fragment list fields

The immediate address field can be set by calling **IPXGetLocalTarget**. If you are not using an event service routine, place a NULL in the event service routine address field.

Maintaining ECBs

Receive and send ECBs are generally handled separately:

Receive ECBs are passed to IPX by calling **IPXListenForPacket**. This approach lets your application forget about the receive ECBs until they need attention.

Send ECBs must be maintained by the application. Once IPX uses a send ECB it is finished with it, and the application is responsible for keeping the ECB in circulation. You can place available send ECBs in a free list or you could scan the entire pool of send ECBs searching for a free ECB each time you need one.

Parent Topic:

Connectionless Service: IPX

Managing IPX Sockets

A socket is a two-byte value that associates an application or a process with the transmission of data. Both IPX and SPX require you open a socket in order to receive data. SPX also requires an open socket to send data. If you are sending data with IPX, you do not have to open a socket. IPX will allocate a socket if you do not specify one. You cannot use the same socket to handle both IPX and SPX transmissions.

Opening an IPX Socket

To open a socket, call **IPXOpenSocket**. This function takes a socket number and socket type as input. IPX will select a dynamic socket value if you pass a socket number of 0000h. IPX chooses dynamic sockets between 4000h and 5000h.

Socket types can be short-lived or long-lived. Typically, most applications use short-lived sockets. A long-lived socket remains open after your application terminates and can be advantageous for memory-resident programs.

Closing an IPX Socket

When you are finished with a socket, call **IPXCloseSocket**. This function cancels events associated with the socket. It is especially important that your application closes any long-lived sockets it has opened. These sockets will persist after your application terminates. IPX closes short-lived sockets upon program termination.

Parent Topic:

IPX/SPX Local Management Issues

Negotiating Checksums

Communicating nodes must negotiate whether to use checksums. Such negotiations should be conducted using connection-oriented protocols such as SPX or NetBIOS. To determine whether checksumming is supported at the local workstation, call **GetIPXInformation**.

Parent Topic: Packet Checksums

Packet Checksum Function List

These assembly language functions support IPX Checksums and Look Ahead Receive handlers.

Function	Value	Comment
Get IPX Information	31	Indicates whether checksumming is supported on the local node.
IPX Generated Checksum	33	Generates an appropriate checksum value for an IPX packet.
IPX Open Look Ahead Socket	35	Opens a look ahead socket and registers a **Look Ahead Receive** handler.
IPX Send With Checksum	32	Generates a checksum in an IPX packet's checksum field before sending the packet.
IPX Verify Checksum	34	Determines whether or not a received packet's data has been corrupted.

Packet Checksums

Packet checksum functions have recently been added to the NetWare API. They are not available for NLM programs at this time. These functions generate and verify checksums for packet transmissions, providing a high level of data integrity across the network. Packet checksumming is an optional feature that must be supported at both ends of a transmission. The checksum requests are implemented under the DOS IPX protocol only (IPXODI).

Related Topics

Negotiating Checksums

Checksums and Packet Transmission

Checksumming and Ethernet 802.3

Packet Information

Both IPX and SPX are adaptations of the Xerox* Network Systems (XNS*)

architecture. IPX conform to the Xerox Internetwork Datagram Protocol (IDP), and SPX conforms to the Sequenced Packet Protocol (SPP). To use either protocol you must think in terms of packets. A packet is a specially formatted data item to be transmitted from one workstation to another workstation.

A packet contains a header and a message. The header contains the data needed to transport the packet from its source to its destination. The message is the application data. IPX defines the basic format of the packet header. SPX augments this format with the data it needs to maintain connections. In the discussions that follow, you can assume that what is said about IPX is also true for SPX unless otherwise noted.

Related Topics

IPX Packet Structure

SPX Packet Structure

The Receive Look Ahead Handler

IPX calls **ReceiveLookAheadHandler** when it receives packets on the associated socket so the handler can preview the incoming data. If the application wants the packet, the handler supplies IPX with the addresses of application buffers to receive the data. Using this approach, the application can arrange to receive a burst of packets directly into application buffers.

Normally the receive ECB buffers serve as temporary storage for incoming data. Before posting the ECB again, the application must copy the data out of these buffers and into an area where it can operate on the data. By using a look ahead socket, an application is no longer required to maintain this pool of intermediate buffers.

To understand how the **ReceiveLookAheadHandler** works, consider how incoming packets are handled. When a workstation receives a packet, the LAN driver gives a portion of it to the Link Support Layer (LSL). From this data, the LSL determines which network layer the packet is destined for. If the destination is IPX, LSL calls the IPX internal look ahead handler.

The IPX internal handler determines the status of the destination socket. If the destination socket is a normal socket on which an application has posted receive ECBs, the handler passes an ECB to the LSL, which in turn passes it to the LAN driver. The LAN driver fills in the ECB to reference the received data and then calls the associated event service routine. At this point, the application must retrieve the data from the ECB buffers and post the ECB again.

If the destination socket is a look ahead socket, the IPX internal handler calls the socket's **ReceiveLookAheadHandler**. This handler can then determine whether the application wants the packet. If so, it must build an ODI-style ECB to reference the application's receive buffers and event service routine.

ReceiveLookAheadHandler returns by passing IPX a pointer to this ECB. (For the **ReceiveLookAheadHandler** interface specifications, see the Assembly Transport Function Reference in the NetWare Limited Support SDK.)

Parent Topic: Look Ahead Sockets

Releasing an SPX Connection

There are two functions that close an SPX connection:

SPXTerminateConnection informs the partner node that the connection is being closed.

SPXAbortConnection makes no attempt to alert the partner node that it is closing the connection.

SPXTerminateConnection is the preferred method for most situations.

Parent Topic: Connection-Mode Service: SPX

Scheduling Asynchronous Events with IPX

The workstation IPX software includes an Asynchronous Event Scheduler (AES). This device allows you to schedule IPX events to occur after a specified time interval. To schedule an asynchronous event, call **IPXScheduleIPXEvent**. As input, this function takes the amount of time to delay the event and an ECB that will control the execution of the event.

The ECB should include the socket number that the event will occur on and the event service routine that will receive control when the time period expires. The time unit is expressed in clock ticks (from zero to 65,353). You can schedule an event up to one hour in advance. To cancel a scheduled event, call **IPXCancelEvent**.

Parent Topic: Connectionless Service: IPX

Sending IPX Packets

To send a packet, call **IPXSendPacket**. As input, this function takes a send ECB whose IPX header specifies the destination node and socket. To broadcast to all stations, the header's destination field should be filled with FFh. If your application is broadcasting to the network that it resides on, the ECB's immediate address field should also be filled with FFh.

While IPX attempts to send the packet, it sets the ECB's In-use flag to FFh.

Afterward, IPX sets this field to zero and enters an appropriate value in the completion code field. A completion code of FEh indicates that the packet is not deliverable. This may be because IPX cannot find a bridge to the destination network or because the target node address does not exist.

Parent Topic: Connectionless Service: IPX

SPX Function List

These functions perform SPX operations.

Function	Header	Comment
SPXAbortConnection	nxtd.h nxtw.h	Unilaterally aborts the connection associated with the specified SPX connection ID number.
SPXEstablishConnection	nxtd.h nxtw.h	Establishes an SPX connection with a the ECB-defined listening node and socket.
SPXGetConnectionStatus	nxtd.h nxtw.h	Returns the status of an SPX connection.
SPXInitialize	nxtd.h nxtw.h	Checks whether SPX is installed at the workstation.
SPXListenForConnection	nxtd.h nxtw.h	Attempts to receive an establish connection packet from the ECB-defined node and socket.
SPXListenForSequencedPacket	nxtd.h nxtw.h	Listens for an SPX packet using the specified ECB.
SPXSendSequencedPacket	nxtd.h nxtw.h	Sends an SPX packet using the specified SPX connection ID number and ECB.
SPXTerminateConnection	nxtd.h nxtw.h	Terminates the specified SPX connection.

SPX Initialization

Before using SPX, you can check whether SPX is supported locally. (Early versions of the NetWare workstation software did not support SPX.) To check for SPX support, call **SPXInitialize**. This function returns the SPX major and minor revision number and the number of maximum and available SPX connections. SPX is supported by NetWare 2.1 and higher.

Parent Topic: IPX/SPX Local Management Issues

SPX Packet Structure

The SPX packet is identical to the IPX packet except that it has an additional 12 bytes in the header. An SPX packet consists of a 42-byte header followed by 0 to 534 bytes of data. The minimum packet size is 42 bytes (the header only) and the maximum size is 576 bytes. The content and structure of the data portion are entirely the responsibility of the application using SPX and can take any format.

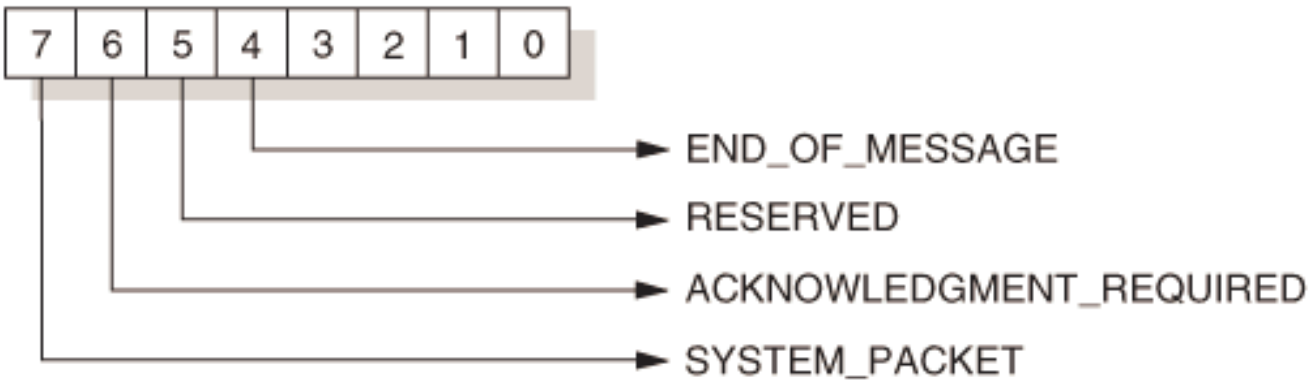
The SPX packet header consists of an IPX header (30 bytes) and 7 additional fields as follows:

```
unsigned char    connectionCtl;
unsigned char    dataStreamType;
unsigned short   sourceConnectID;  /* high/low */
unsigned short   destConnectID;    /* high/low */
unsigned short   sequenceNumber;   /* high/low */
unsigned short   ackNumber;        /* high/low */
unsigned short   allocNumber;      /* high/low */
```

SPX sets all fields in the SPX header except the destination fields (in some functions), *connectionCtl,* and *dataStreamType*.

The *connectionCtl* field controls the bidirectional flow of data across an SPX connection. The defined bits are shown in the following figure.

Figure 12. Connection Control Bits



The END_OF_MESSAGE bit is the only bit the application sets or clears.

The *dataStreamType* field indicates the type of data included in the packet.

Values of 0x00 through 0xFD are defined by the client and ignored by SPX. A value of 0xFE indicates an End-Of-Connection packet. When a client makes a call to terminate an active connection, SPX generates an End-Of-Connection packet. This packet is then delivered to the connection partner as the last message on the connection.

A value of 0xFF indicates an End-Of-Connection-Acknowledgment packet. SPX generates an End-Of-Connection-Acknowledgment packet automatically. It is marked as a system packet and not delivered to the partner clients. The values 0xFE and 0xFF are reserved for use by SPX in connection maintenance and should not be used by an application.

The *sourceConnectID* and *destConnectID* fields specify the connection identification number assigned to the SPX connection by the source node and destination node respectively.

The *sequenceNumber* field keeps a count of packets exchanged in one direction on the connection. Each side of the connection keeps its own count. The number wraps to 0x0000 after reaching 0xFFFF. Since SPX manages this field, client processes need not be concerned with it.

The *ackNumber* field indicates the next packet that an SPX connection expects to receive. The values in this field increment from 0x0000 to 0xFFFF and then wrap to zero again.

The *allocNumber* (in conjunction with the *ackNumber*) indicates the number of outstanding packet receive buffers (posted listens) available for a given SPX connection. It is used by SPX to implement flow control between communicating applications. The *allocNumber* minus the *ackNumber* equals the number of posted listens outstanding on the connection socket. SPX sends packets only until the local sequence number equals the *allocNumber* of the remote partner. The *allocNumber* increments from 0xFFFF and wraps to 0x0000.

Related Topics:

Packet Information

SPX TLI Client: Example and SPX TLI Server: Example

The Event Service Routine Interface

On entry, your event service routine should assume the following conditions:

The AL register contains the identity of the calling process: 00h for the AES and FFh for IPX.

A pointer to the ECB associated with the event service routine is in the register pair ES:SI.

The state of the processor flags and all registers except SS and SP have

been saved on the user's stack. The event service routine is responsible for saving SS and SP so it can return properly after execution.

Interrupts are disabled.

The DS register needs to be initialized before the event service routine makes reference to any application variables.

The event service routine is called by a long procedure call. Consequently, it must return with an RETF instruction and with interrupts disabled. It can't return a value.

For programs written in C, the event service routine can be implemented as a C function called by an assembly front end:

```
_ESRHandler proc far
mov   ax, DGroup
mov   ds, ax
push   es
push   si
call   _EventServiceRoutine
retf
_ESRHandler endp
```

If your ESR handler needs to pass more than a few bytes of data to the event service routine, the handler should set up its own stack. In that case, it should restore the original stack segment and stack pointer before returning.

Parent Topic: Event Service Routines

The Look Ahead Structure

When IPX calls your application's **ReceiveLookAheadHandler**, it passes a LookAhead referencing the received packet. The structure includes the following fields:

Look ahead address

Look ahead length

look ahead address is the address of a buffer holding the first n bytes of the received packet.

look ahead length contains the length of the look ahead buffer.

look ahead length is equal to or greater than the look ahead size requested by the application when it called **IPXOpenLookAheadSocket**. However, if the packet's total length is less than the requested size, *look ahead length* contains the length of the packet. If this value is less than the minimum size requested, discard the packet.

Parent Topic: Look Ahead Sockets

Transmitting Data with SPX

There are two functions used to exchange data once a connection has been set up:

SPXListenForSequencedPacket

SPXSendSequencedPacket

As input, both functions take an ECB to conduct the associated event. Your application must manage the ECBs in a manner similar to that described for IPX operations. To receive status information about an SPX connection, call **SPXGetConnectionStatus**.

Parent Topic: Connection-Mode Service: SPX

Communication Service Group

IPX: Functions

IPX for DOS

IPXCancelEvent (DOS)

Cancels a pending event by passing an ECB address to the IPX™ protocol
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IPXCancelEvent(
  ECB far   *eventControlBlock);
```

Parameters

eventControlBlock
(IN/OUT) Points to an ECB structure.

Return Values

0x0000	Successful
0xF9	ECB cannot be canceled
0xFF	ECB not in use

ECB Return Values

0xFC	Event canceled

Remarks

IPXCancelEvent (DOS) returns a completion code upon returning control to the application program.

ECB might be performing an event such as send or listen under IPX, a schedule or reschedule under AES, or a listen under the SPX™ protocol.

When an ECB is submitted to IPX to coordinate an event, IPX sets ECB's *inUseFlag* to a nonzero value, indicating the ECB is unavailable to other applications. Possible nonzero values for *inUseFlag* are listed below:

Decimal	Hex	Flag Value Description
224	0xE0	AES Temporary Indicator
248	0xF8	Critical Holding (IPX in critical process)
250	0xFA	Processing
251	0xFB	Holding (in processing after an event occurred)
252	0xFC	AES Waiting
253	0xFD	Waiting
254	0xFE	Receiving
255	0xFF	Sending

After attempting to cancel the event defined by the ECB, IPX sets ECB's *completionCode* to an appropriate value. It also sets ECB's *inUseFlag* to 0x00 (available for use). IPX does not call the ESR referenced by ECB's *ESRAddress*.

IPXCancelEvent (DOS) cannot cancel packets already sent by the node's driver.

See Also

IPXScheduleIPXEvent (DOS)

IPXCloseSocket (DOS)

Closes the specified IPX socket and cancels all associated ECBs
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void cdecl IPXCloseSocket(
   WORD   socketNumber);
```

Parameters

socketNumber

(IN) Indicates the socket number of the socket to be closed (high-low).

Remarks

IPXCloseSocket (DOS) returns a value of 0xFC in each ECB's *completionCode*, indicating that the event has been canceled. Then IPX sets each ECB's *inUseFlag* to 0x00 (available for use). IPX does not call ECBs' associated ESRs.

NOTE: Any socket number can be closed; no error is generated if the specified socket is not opened. Before an application program terminates and destroys ESR code, it must close all sockets it has opened as long-lived. Otherwise, ESR could be called after it is no longer available, permanently hanging the workstation. Sockets opened as short-lived will automatically close upon program termination if the shell is loaded.

Applications waiting to receive a packet on the closed socket are awakened with 0x8007 code (Cancel) in ECB *status*.

Transient applications should close sockets before terminating.

IPXCloseSocket (DOS) must not be called from within an ESR.

See Also

IPXOpenSocket (DOS)

IPXDisconnectFromTarget (DOS)

Informs the communication driver the application does not intend to send any more packets to the specified station

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: IPX

Syntax

```
#include <nwipxspx.h>

void IPXDisconnectFromTarget(
   BYTE far   *networkAddress);
```

Parameters

networkAddress

(IN) Points to the 12-byte local network address (high-low). The 12 bytes are organized as follows:

Bytes 0-3 = Network number

Bytes 4-9 = Node number

Bytes 10-11 = Socket number

Remarks

IPXDisconnectFromTarget (DOS) is a courtesy to network communications drivers operating strictly on a point-to-point basis at the physical transport level. Once informed, the driver can dismantle any board-level virtual connection with the node specified.

networkAddress identifies the node with which communication will be terminated. After calling **IPXDisconnectFromTarget (DOS)**, any virtual connection between the machine on which the application is running and the target machine may be dismantled by the driver.

An application should never call **IPXDisconnectFromTarget (DOS)** from within an ESR.

See Also

SPXAbortConnection (DOS), **SPXTerminateConnection (DOS)**

IPXGetInternetworkAddress (DOS)

Returns the network and node address of the requesting workstation
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void IPXGetInternetworkAddress(
   BYTE far   *networkAddress);
```

Parameters

networkAddress

(OUT) Points to the 10-byte local network address. The 10 bytes are organized as follows:

Bytes 0-3 = Network number
Bytes 4-9 = Node number

Remarks

IPXGetInternetworkAddress (DOS) is especially useful to any application that must inform other nodes of its address on the internetwork.

The buffer supplied by the application is filled with a 4-byte network address followed by a 6-byte node address. Both fields are in high-low order.

IPXGetInternetworkAddress (DOS) does not return a socket number. The socket number is determined when an application opens a socket. When an application builds a complete 12-byte network address, it must append the appropriate socket number.

See Also

IPXGetLocalTarget (DOS)

IPXGetIntervalMarker (DOS)

Returns a time marker from IPX
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

WORD cdec IPXGetIntervalMarker(
   void);
```

Parameters

IPXGetIntervalMarker (DOS) requires no parameters.

Return Values

timeUnits as a WORD.

Remarks

IPXGetIntervalMarker (DOS) may be used to measure the elapsed time between two events. To do this, call **IPXGetIntervalMarker (DOS)** twice, once before each of the two events. Subtract the first interval marker from the second to derive the elapsed time (in clock ticks). This result remains accurate even if the interval marker wraps through zero between the first and second event. The timer is not intended for use with time intervals greater than 1 hour due to the precision of the interval marker (16 bits unsigned).

The interval marker is a value between 0 and 65,535 [(0x0000) and (0xFFFF)]. Each interval represents one IBM* PC clock tick, which is approximately 1/18 second.

IPXInitialize (DOS) must be called before calling **IPXGetIntervalMarker (DOS)**.

See Also

IPXInitialize (DOS)

IPXGetLocalTarget (DOS)

Returns the value to be placed in ECB's *immediateAddress*
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

int cdecl IPXGetLocalTarget(
   BYTE far   *networkAddress,
   BYTE far   *immediateAddress,
   int far    *transportTime);
```

Parameters

networkAddress

(IN) Points to the 12-byte destination network address (high-low).

immediateAddress

(OUT) Points to the ECB's 6-byte immediate address (high-low).

transportTime

(OUT) Indicates the estimated amount of time (in system clock ticks) for sending a 576-byte packet to the destination node.

Return Values

0x0000	Successful
0xFA	No Local Target Identified

Remarks

IPXGetLocalTarget (DOS) passes the internetwork address of a destination node to IPX. IPX returns an estimated packet transport time and either the node address of the destination node or the node address of the local bridge. It will use this information to route a packet to the destination node.

The 12 bytes of *networkAddress* are defined as:

```
Bytes 0-3      =   Network Number
Bytes 4-9      =   Node Number
Bytes 10-11    =   Socket Number
```

transportTime contains an estimate of the time (in clock ticks of approximately 1/18 second) a 576-byte packet takes to travel from the source node to the destination node. Since the returned value is only an estimate, actual travel time can vary depending on network traffic and packet size.

IPXGetLocalTarget (DOS) can be used with destination addresses using broadcast values (all 0xFFs) in the node field of *networkAddress*.

When IPX receives a packet, it records the node address of the sending node in *immediateAddress* of the receive ECB. The 6 bytes of *immediateAddress* are the local target node number, not the local source node number. The local source node number is the number of the calling workstation.

Therefore, once an application begins exchanging packets with an application on another node, either application can use *immediateAddress* returned with a receive ECB to obtain the local target value instead of continually calling **IPXGetLocalTarget (DOS)**.

An application can use an ECB that has received a packet to send a packet to the originating address without modifying *immediateAddress* in ECB.

IPXGetLocalTarget (DOS) can be called from within IPX or an AES ESR or directly from the main portion of an application. It must not be called from any other kind of interrupt service routine.

See Also

IPXSendPacket (DOS), **IPXListenForPacket (DOS)**

IPXGetMaxPacketSize (DOS)

Finds the maximum packet size allowed by a workstation's topology (LAN card and wiring)

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: IPX

Syntax

```
#include <nwipxspx.h>

WORD cdecl IPXGetMaxPacketSize(
   void);
```

Parameters

IPXGetMaxPacketSize (DOS) requires no parameters.

Return Values

Packet Size	Successful

Remarks

IPXInitialize (DOS) must be called successfully before calling **IPXGetMaxPacketSize (DOS)**.

When **IPXGetMaxPacketSize (DOS)** fails, it returns 0xF0 (240 decimal); 240 decimal can also be a valid maximum packet size in some situations.

Packet size on a local network is limited only by that network's topology. The content and structure of an IPX packet's data portion are entirely the responsibility of the application using IPX. The application may be able to take advantage of a larger allowable packet size if the topology permits. Call **IPXGetMaxPacketSize (DOS)** to check what size packets the topology allows.

NOTE: Some bridges will not route packets that are longer than 576 bytes. Because bridges will not fragment packets and reassemble them later, the application must not exceed the 576-byte packet size limit if it is to work across a bridge.

If SHELL.CFG uses *IPXMaxPacket*, the packet size returned by **IPXGetMaxPacketSize (DOS)** will be supported by theLAN, not by the individual workstation.

See Also

IPXInitialize (DOS), **SPXInitialize (DOS)**

IPXInitialize (DOS)

Returns the entry address for the IPX interface
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

BYTE cdecl IPXInitialize(
   void);
```

Parameters

IPXInitialize (DOS) does not requires any parameters.

Return Values

0x0000	Successful
0xF0	Ipx Not Installed

Remarks

IPXInitialize (DOS) initializes a variable (*IPXLocation*) in the library with the address of the IPX services.

IPXInitialize (DOS) or **SPXInitialize (DOS)** must be called before any IPX function can be performed.

See Also

SPXInitialize (DOS)

IPXListenForPacket (DOS)

Prepares IPX to receive an IPX packet
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void IPXListenForPacket(
   ECB far   *eventControlBlock);
```

Parameters

eventControlBlock
(IN/OUT) Points to an ECB structure.

ECB Return Values

0x0000	Successful
0xFC	Request Cancelled
0xFD	Bad Packet
0xFF	Socket Not Open

Remarks

IPXListenForPacket (DOS) passes an ECB address to IPX for the purpose of receiving an IPX packet. It then returns control to the calling application. Meanwhile, IPX listens for and attempts to receive a packet.

Before calling **IPXListenForPacket (DOS)**, an application must open the socket and initialize ECB's *ESRAddress*, *socketNumber*, *immediateAddress*, *fragmentCount*, and *fragmentDescriptor*. If no routine is to be called, *ESRAddress* should contain a NULL.

Initially, IPX sets ECB's *inUseFlag* to (0xFE) indicating that the ECB is waiting to receive a packet. IPX also adds the ECB to a buffer pool of ECBs all listening for IPX packets on the same socket. IPX imposes no limit on the number of ECBs that can be listening concurrently on a socket.

When IPX detects an incoming packet, IPX uses one of the listening ECBs to receive the packet. Listen ECBs are not filled in the order they were

submitted to IPX. IPX records an appropriate value in the selected ECB's *completionCode* and places a node address in ECB's *immediateAddress*. The value in *immediateAddress* identifies either the sending node (if the sending node resides on the local network), or the local bridge that routed the packet to the receiving node (if the sending node does not reside on the local network).

Finally, IPX sets ECB's *inUseFlag* to 0x00 (available for use) and calls the ESR referenced by ECB's *ESRAddress* (if applicable).

See Also

IPXSendPacket (DOS)

IPXOpenSocket (DOS)

Opens an IPX socket
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

int cdecl IPXOpenSocket(
   BYTE far   *socketNumber,
   BYTE        socketType);
```

Parameters

socketNumber

(IN/OUT) Points to a socket number (high-low) or a 0x00 value.

socketType

(IN) Indicates the type of socket to open. *socketType* can be either:

0x00 = Stay open until program closes it or program terminates (short-lived)
0xFF = Stay open until program closes it (long-lived)

Return Values

0x0000	Successful
0x00F0	IPX Not Installed
0x00FE	Socket Table Full
0x00FF	Socket Already Open

Remarks

An application must call **IPXOpenSocket (DOS)** to open a socket before receiving a packet on the socket.

socketNumber contains the number of the socket to be opened. Passing 0x0000 allows IPX to open an available socket of its choice in the range 0x4000 to 0x5000. This is known as a **dynamic** socket. The dynamic socket opened will be returned in *socketNumber*, which is a 2-byte variable.

If **IPXOpenSocket (DOS)** returns a completion code of 0x00, the socket

has been opened as expected. If the completion code is 0xFE, the socket table is already full. If **IPXOpenSocket (DOS)** returns a completion code of 0xFF, the specified socket is already open.

socketType specifies how long the socket should remain open. Unless the program intends to terminate and stay resident, the socket type should always be temporary (0x00---SHORT_LIVED). The exception to this is for programming with MS Windows, in which case you should always open sockets as permanent (0xFF---LONG_LIVED) sockets. Sockets that are opened as temporary (SHORT_LIVED) are automatically closed upon program termination (if the NetWare® shell is loaded), and all associated events pending on that socket are canceled. If the application program opens a socket as permanent, it must guarantee that the ESR is available even after it terminates. Otherwise, the workstation could hang if the ESR is called after it is no longer available.

By default, IPX supports up to 20 open sockets on one workstation. The maximum number of simultaneously opened sockets for a workstation can be as high as 150 and is configurable in SHELL.CFG.

See Also

IPXCancelEvent (DOS), **IPXCloseSocket (DOS)**

IPXRelinquishControl (DOS)

Relinquishes control of a workstation's CPU
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void cdecl IPXRelinquishControl(
   void);
```

Parameters

IPXRelinquishControl (DOS) needs no parameters.

Remarks

IPXRelinquishControl (DOS) can temporarily relinquish control of the workstation CPU so that other processing can be done while the application is waiting for input.

IPX applications should make repeated calls to **IPXRelinquishControl (DOS)** during idle time to allow other applications in the computer to use the CPU. This is especially important when the application is resident with NetWare server or bridge software, since **IPXRelinquishControl (DOS)** greatly improves the efficiency of the server or bridge.

IPXRelinquishControl (DOS) also allows the communications driver in IPX to run. This is important if the driver is not interrupt driven. On a normal workstation, **IPXRelinquishControl (DOS)** invokes a polling procedure provided by the network communications driver. It represents the only opportunity the driver has to use the CPU to send and receive packets, events essential to the application itself.

IPXScheduleIPXEvent (DOS)

Schedules an IPX event
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void IPXScheduleIPXEvent(
   WORD      timeUnits,
   ECB far   eventControlBlock);
```

Parameters

timeUnits

(IN) Indicates the time interval.

eventControlBlock

(IN) Points to ECB.

Remarks

After initializing ECB's *ESRAddress* and *socketNumber*, calling **IPXScheduleIPXEvent (DOS)** passes a delay time and an ECB address to IPX for the purpose of scheduling an IPX event. It then returns control to the calling application. Meanwhile, IPX attempts to schedule the event.

The scheduled event is directly associated with an IPX socket. If **IPXCloseSocket (DOS)** is executed with this socket as its argument, the event scheduled by **IPXScheduleIPXEvent (DOS)** is canceled along with all other IPX packet events for the socket. To cancel only an event scheduled by **IPXScheduleIPXEvent (DOS)**, call **IPXCancelEvent (DOS)**.

timeUnits specifies the waiting time (in clock ticks) before a scheduled event takes place. It must be set to a value between 0 and 65,535 (0x0000 and 0xFFFF). The interval between each IBM PC clock tick is approximately 1/18 second. This allows an event to be scheduled for up to one hour.

IPXScheduleIPXEvent (DOS) can reschedule an ECB previously submitted for scheduling. The application passes a pointer to the same ECB to the function, along with a new expiration time. This is the only situation in which it is legal to pass IPX (AES) to an ECB that is still in use. *inUseFlag* is set to a value of 0xFD while the AES is in control of the IPX ECB.

An ESR can also call **IPXScheduleIPXEvent (DOS)**. For example, when

IPX calls an ESR, the ESR can first check conditions in the system, determine that conditions are not acceptable for executing at the moment, and then call **IPXScheduleIPXEvent (DOS)** to reschedule itself. After the specified delay time, the ESR can execute again, check conditions, and continue the cycle until conditions are favorable.

An application should never call **IPXScheduleIPXEvent (DOS)** to pass the address of an ECB currently being used by IPX for packet events.

See Also

IPXCancelEvent (DOS)

IPXSendPacket (DOS)

Initiates the sending of an IPX packet
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: IPX

Syntax

```
#include <nwipxspx.h>

void IPXSendPacket(
   ECB far   *eventControlBlock);
```

Parameters

eventControlBlock
(IN/OUT) Points to ECB.

ECB Return Values

0x00	Successfully Sent But Not Necessarily Received
0xFC	Request Canceled
0xFD	Given Packet Did Not Have a 30-byte Packet Header as the First Fragment, or Its Total Length Exceeded 567 Bytes
0xFE	Packet Not Deliverable
0xFF	Hardware Failure

Remarks

IPXSendPacket (DOS) passes an ECB address to IPX for the purpose of sending an IPX packet. It then returns control to the calling application. Meanwhile, IPX attempts to send the packet.

Before calling **IPXSendPacket (DOS)**, the application must initialize ECB's *ESRAddress*, *socketNumber*, *immediateAddress*, *fragmentCount*, and *fragmentDescriptor*. The application must also prepare the IPX header of the associated packet by filling in *packetType* and *destination*. **IPXSendPacket (DOS)** then passes the ECB to the network communication drivers to initiate the send operation.

Although *socketNumber* in ECB is significant (IPX uses it as the source socket number on the packet header), the socket need not be open to perform a send on it.

immediateAddress in ECB can be set by calling **NWIPXGetLocalTarget (DOS)**.

Initially, IPX sets ECB's *inUseFlag* to 0xFF, indicating that the ECB is sending a packet. After attempting to send the packet, IPX sets ECB's *completionCode* to an appropriate value and sets *inUseFlag* to 0x00, indicating that the ECB is available for use. Finally, IPX calls the ESR referenced by ECB's *ESRAddress* (if applicable).

A completion code of 0x00 indicates the packet was sent successfully but does not guarantee the packet was received successfully by the destination node. For example, the transmission media may lose or garble the packet, or the destination socket may not be open or listening. IPX does not inform the sending node if these problems occur.

0xFE indicates that the packet is undeliverable. The ECB returns this completion code for one of two reasons:

IPX cannot find a bridge with a path to the destination network.

The target node address does not exist.

An application can send or broadcast an IPX packet to any socket on the internetwork including the socket on which the application sends the packet. Packets sent to sockets residing in the same node as the application are called intra-node packets.

To send a broadcast packet (one that will be received by every listening station on a network), *node* of *destination* in the packet header should be initialized to all 0xFFs (-1). If the broadcast is to the network on which the application resides, *immediateAddress* in the ECB should also be set to all 0xFFs.

See Also

IPXGetLocalTarget (DOS), **IPXListenForPacket (DOS)**

IPX for NLM

IpxCancelEvent, IpxCancelPacket (NLM)

Cancels an ECB being used by **IpxReceive**
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Platform: NLM
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxCancelEvent(
   IPX_ECB   *eventControlBlock);
```

Parameters

eventControlBlock

(IN/OUT) Points to the IPX_ECB structure.

Return Values

0	(0x00)
ESUCCESS	249
(0xF9)	ERR_ECB_CANNOT_BE_CANCELLED

Remarks

IpxCancelEvent (NLM) cancels a pending event by passing an Event Control Block (ECB) address to IPX. The function returns a completion code upon returning control to the application program.

The ECB should be busy waiting for an IPX packet (by calling **IpxReceive (NLM)**).

When an ECB is submitted to **IpxReceive (NLM)** to receive a packet, IPX sets the ECB's *status* field to a positive value, indicating that the ECB is unavailable to other applications. The possible values are listed in NWIPXSPX.H. After attempting to cancel the event defined by the ECB, IPX sets the ECB's *status* field to STS_SPX_EVENT_CANCELLED. IPX neither posts the ECB's associated semaphore handle nor queues the ECB on the queue head.

For compatibility with the OS implementation of the IPX/SPX™ protocols, **IpxCancelEvent (NLM)** has the alternate name **IpxCancelPacket (NLM)**.

IpxCheckSocket (NLM)

Determines whether a socket is open
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxCheckSocket (
   unsigned short   socketNumber);
```

Parameters

socketNumber

(IN) Specifies the socket number of the socket to be checked (high/low).

Return Values

0	(0x00)
ESUCCESS	240
(0xF0)	ERR_SOCKET_NOT_OPEN

Remarks

IpxCheckSocket (NLM) determines whether the socket is open.

IpxCloseSocket (NLM)

Closes an IPX socket
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxCloseSocket (
   unsigned short   socketNumber);
```

Parameters

socketNumber

(IN) Specifies the socket number of the socket to be closed (high/low)

Return Values

0	(0x00)	ESUCCESS

Remarks

IpxCloseSocket (NLM) closes the IPX socket and cancels events defined by the ECBs associated with the socket. IPX also returns a value of 0xFFFC in each ECB's *status* field, indicating that the event has been cancelled.

IPX posts the ECBs' associated semaphore handles and then queues the ECBs on the associated queue heads for ECBs that are pending on the socket. Any socket number can be closed; no error is generated if the specified socket was not open.

Applications should close all sockets they open before terminating.

See Also

IpxOpenSocket (NLM)

IpxConnect (NLM)

Creates a virtual connection to an address
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxConnect (
   IPX_ECB   *eventControlBlock);
```

Parameters

eventControlBlock

(IN) Points to the IPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS

Remarks

IpxConnect (NLM) is nonoperational in the NetWare 3.x and 4.x environment. It is included only for compatibility with the OS implementation of IPX/SPX.

IpxDisconnect (NLM)

Destroys a virtual connection to an address
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxDisconnect (
   IPX_ECB   *eventControlBlock);
```

Parameters

eventControlBlock

(IN) Points to the IPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS

Remarks

IpxDisconnect (NLM) is nonoperational in the NetWare 3.x and 4.x environment. It is included only for compatibility with the OS implementation of IPX/SPX.

IpxGetAndClearQ (NLM)

Returns a pointer to the list of completed ECBs queued off of a queue head and then sets the queue head to NULL

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x

Service: IPX

Syntax

```
#include <nwipxspx.h>

IPX_ECB * IpxGetAndClearQ (
   IPX_ECB   **queueHeadPtr);
```

Parameters

queueHeadPtr

(IN/OUT) Points to the queue head.

Return Values

IpxGetAndClearQ (NLM) returns a pointer to the first ECB in the list of ECBs queued from the queue head. If NULL is returned, the list is empty.

Remarks

The queue head (to which *queueHeadPtr* points) is cleared (set to NULL) by **IpxGetAndClearQ (NLM)**. It is used with both SPX and IPX (although IPX_ECB is used in the declaration).

Before NW SDK Release 9, **IpxGetAndClearQ (NLM)** was implemented as a macro. An **IpxGetAndClearQ (NLM)** symbol was not exported.

Beginning with NW SDK Release 11, **IpxGetAndClearQ (NLM)** is prototyped as an external function. The **IpxGetAndClearQ (NLM)** symbol will be exported at load time by the static libraries (prelude.obj) being used.

Example

IpxGetAndClearQ

```
#include <nwipxspx.h>

IPX_ECB   *ptr;
IPX_ECB   *queueHeadPtr;
```

```
IPX_ECB   *queueHeadPtr;
ptr = IpxGetAndClearQ (&queueHeadPtr);
```

IpxGetInternetworkAddress (NLM)

Returns the network and node address of the requestingNLM™ application
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxGetInternetworkAddress (
   unsigned char   *networkAddress);
```

Parameters

networkAddress

(OUT) Points to the 10-byte local network address. The bytes are organized as follows:

Bytes 0-3 = Network number
Bytes 4-9 = Node number

Return Values

0	(0x00)	ESUCCESS

Remarks

The **IpxGetInternetworkAddress (NLM)** function is especially useful to an application that must inform other nodes of its address on the internetwork. It provides the internetwork address of the node on which an application is executing. The buffer supplied by an application is filled with a 4-byte network number followed by a 6-byte node address. Both fields are in high/low order.

IpxGetInternetworkAddress (NLM) does not return a socket number. The socket number is determined when an application opens a socket. When an application builds a complete 12-byte network address, it must append the appropriate socket number.

See Also

IpxGetLocalTarget (NLM)

IpxGetLocalTarget (NLM)

Fills the *immediateAddress* field in an ECB
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxGetLocalTarget (
   unsigned char   *networkAddress,
   IPX_ECB         *eventControlBlock,
   unsigned long   *transportTime);
```

Parameters

networkAddress

(IN) Points to the 12-byte destination network address (high/low).

eventControlBlock

(OUT) Points to the ECB into which the 6-byte immediate address (high/ low) of the target is returned.

transportTime

(OUT) Points to the estimated amount of time (in system clock ticks) for sending a 576-byte packet to the destination node.

Return Values

0	(0x00)	ESUCCESS
250	(0xFA)	ERR_NO_LOCAL_TARGET_IDENTIFIED

Remarks

The **IpxGetLocalTarget (NLM)** function passes the internetwork address of a destination node to IPX. IPX returns an estimated packet transport time and either the node address of the destination node or the node address of the local bridge it uses to route a packet to the destination node.

The 12 bytes of the *networkAddress* are defined as follows:

Bytes 0-3 = Network Number

Bytes 4-9 = Node Number

Bytes 10-11 = Socket Number

The *transportTime* parameter contains an estimate of the time (in clock ticks of approximately 1/18 of a second) a 576-byte packet takes to travel from the source node to the destination node. Since the returned value is only an estimate, actual travel time can vary depending on network traffic and packet size.

The **IpxGetLocalTarget (NLM)** function can be used with destination addresses using broadcast values (all 0xFFs) in the node field.

When IPX receives a packet, it records the node address of the sending node in the *immediateAddress* field of the receive ECB. The 6 bytes of *immediateAddress* are the local target node number, not the local source node number. The local source node number is the number of the calling workstation. Therefore, once an application begins exchanging packets with an application on another node, either application can use the *immediateAddress* from a receive ECB to obtain the local target value instead of continually making calls to the **IpxGetLocalTarget (NLM)** function.

An application can use an ECB that has received a packet to send a packet to the originating address without modifying the *immediateAddress* field in the ECB.

See Also

IpxSend (NLM)

IpxGetStatistics (NLM)

Returns diagnostic statistics maintained by IPX
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxGetStatistics (
   IPX_STATS   *ipxStats);
```

Parameters

ipxStats
(OUT) Points to the diagnostic statistics.

Return Values

-1	EFAILURE

Remarks

The **IpxGetStatistics (NLM)** function is nonoperational in the NetWare 3.x and 4.x environment. This function is included only for compatibility with the OS implementation of IPX/SPX.

See Also

IpxResetStatistics (NLM)

IpxGetVersion (NLM)

Returns the major and minor version and the revision of the IPX
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxGetVersion (
   unsigned char    *majorVersion,
   unsigned char    *minorVersion,
   unsigned short   *revision);
```

Parameters

majorVersion

(OUT) Points to major revision of the IPX.

minorVersion

(OUT) Points to minor revision of the IPX.

revision

(OUT) Points to the revision of the IPX.

Return Values

0	(0x00)	ESUCCESS

Remarks

Versions of IPX are identified by major version number, minor version number, and (if applicable) revision number. **IpxGetVersion (NLM)** returns the major and minor version of IPX. The revision is always returned as 0.

If any of the parameters is NULL, the value for that parameter is not returned.

See Also

SpxGetVersion (NLM)

IpxOpenSocket (NLM)

Opens an IPX socket
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxOpenSocket (
   unsigned short   *socketNumber);
```

Parameters

socketNumber

(IN/OUT) Points to a socket number (high/low).

Return Values

0	(0x00)	ESUCCESS
240	(0xF0)	ERR_IPX_NOT_INSTALLED
241	(0xF1)	ERR_SOCKET_ALREADY_OPEN
254	(0xFE)	ERR_SOCKET_TABLE_FULL

Remarks

An application must call **IpxOpenSocket (NLM)** to open a socket before receiving a packet on the socket.

Blocking InformationIpxOpenSocket (NLM) blocks for the initial call and is nonblocking for subsequent calls.

The *socketNumber* parameter points to the number of the socket to be opened. If the socket number to be opened is set to zero, IPX opens an available socket of its choice in the range 0x4000 to 0x7FFF, returning the value of the new socket in the variable pointed to by *socketNumber*. This is known as a **dynamic** socket open.

If **IpxOpenSocket (NLM)** returns a completion code of 0x00, the socket has been opened as expected. If the completion code is 0xFE, the socket table is already full. If **IpxOpenSocket (NLM)** returns a completion code of 0xF1, the specified socket is already open.

of 0xF1, the specified socket is already open.

At present, IPX/SPX supports up to 100 open sockets on a NetWare 3.x and 4.x server.

See Also

IpxCloseSocket (NLM)

IpxReceive (NLM)

Initiates the receiving of an IPX packet
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxReceive (
   unsigned short   socketNumber,
   IPX_ECB         *eventControlBlock);
```

Parameters

socketNumber

(IN) Specifies the socket number of the socket to receive the packet.

eventControlBlock

(IN/OUT) Points to the IPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS
240	(0xF0)	ERR_SOCKET_NOT_OPEN
252	(0xFC)	ERR_REQUEST_CANCELLED
253	(0xFD)	ERR_PACKET_OVERFLOW
254	(0xFE)	ERR_TIMEOUT_FAILURE
254	(0xFE)	ERR_BAD_PACKET
255	(0xFF)	ERR_SOCKET_CLOSED

Remarks

IpxReceive (NLM) passes the ECB to the LAN drivers to execute the receive operation. An application can specify a semaphore handle in the ECB and then wait (**WaitOnLocalSemaphore**) on the handle after calling **IpxReceive (NLM)**. The application is awakened when a packet is received. A queue head can also be specified in the ECB (for more information about queue heads, see ECB Queues).

The ECB is used to receive a packet. The socket must be open before any receive requests can be made on that socket. IPX ignores all incoming packets destined for unopened sockets.

An application does not have to execute a receive before executing a send. This implementation of IPX does not buffer incoming packets destined for open sockets with no pending receive.

If the *socketNumber* parameter is zero, the *socket* field in the ECB is assumed to contain the socket number to receive the packet on.

IPX places packet contents (including the 30-byte IPX header) into memory according to the ECB's fragment list. If available buffer space is large enough to contain the packet, IPX disburses the packet among as many fragments as needed. Any remaining buffer space is undisturbed. If a packet is larger than the combined available space specified by the fragment list, IPX disburses as much of the packet as can be handled, and remaining data is lost.

IpxResetStatistics (NLM)

Resets diagnostic statistics maintained by IPX
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxResetStatistics (
   void);
```

Return Values

-1	EFAILURE

Remarks

IpxResetStatistics (NLM) is nonoperational in the NetWare 3.x and 4.x environment. It is included only for compatibility with the OS implementation of IPX/SPX.

See Also

IpxGetStatistics (NLM)

IpxSend (NLM)

Initiates the sending of an IPX packet
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: IPX

Syntax

```
#include <nwipxspx.h>

int IpxSend (
   unsigned short   socket,
   IPX_ECB         *ECBp);
```

Parameters

socket

(IN) Indicates the socket number of the socket to perform a send on; can be 0.

ECBp

(IN/OUT) Points to the IPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS
250	(0xFA)	ERR_NO_KNOWN_ROUTE_TO_DESTINATION
252	(0xFC)	ERR_REQUEST_CANCELLED
254	(0xFE)	ERR_BAD_PACKET
255	(0xFF)	ERR_HARDWARE_FAILURE

Remarks

IpxSend (NLM) passes an ECB address to IPX for the purpose of sending an IPX packet. It then returns control to the calling application. Meanwhile, IPX attempts to send the packet.

Before calling **IpxSend (NLM)**, the application must initialize the ECB's

socketNumber (if the *socketNumber* parameter is 0), *immediateAddress*, *fragCount*, and *fragList* fields. If the *socketNumber* parameter is nonzero, it is used rather than the *socketNumber* field in the ECB.

NOTE: IpxSend (NLM) is the only IPX/SPX function that takes an ECB as a parameter and does not use a semaphore handle or a queue head if specified in the ECB. The application has no way to determine when the packet is actually sent by the LAN driver.

The application must also prepare the IPX header of the associated packet by filling in the *packetType* and *dest* fields. The function then passes the ECB to the network communication drivers to initiate the send operation.

Although the *socketNumber* parameter in the ECB is significant (IPX uses it as the source socket number on the packet header), the socket does **not** need be open to perform a send on it.

The *immediateAddress* field in the ECB can be set using **IpxGetLocalTarget (NLM)**.

Initially, IPX sets the ECB's *status* field to a positive value, indicating that the ECB is sending a packet. After attempting to send the packet, IPX sets the ECB's *status* field to an appropriate value.

The 0x00 completion code indicates the packet was sent successfully but does not guarantee the packet was received successfully by the destination node. For example, the transmission media might lose or garble the packet, or the destination socket might not be open or listening. IPX does not inform the sending node if these problems occur.

The 0xFA completion code indicates the packet is undeliverable. The ECB returns this completion code for one of two reasons:

IPX cannot find a bridge with a path to the destination network.

The target node address does not exist.

An application can send or broadcast an IPX packet to any socket on the internetwork, including the socket on which the application sends the packet. Packets sent to sockets that reside in the same node as the application are called intra-node packets.

To send a broadcast packet (one to be received by every listening station on a network), the node portion of the destination field in the packet header should be initialized to all 0xFFs (-1). If the broadcast is to the network on which the application resides, the *immediateAddress* field in the ECB should also be set to all 0xFFs.

See Also

IpxGetLocalTarget (NLM), **IpxReceive (NLM)**

Communication Service Group

IPX for OS/2

For the 32-bit OS/2* interface to IPX, see NWSIPX.

IPX for Windows

IPXCancelEvent (Win)

Cancels a pending event by passing an ECB address to IPX
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXCancelEvent(
   DWORD      IPXTaskID,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	Successful
0xF9	Ecb Cannot Be Canceled
0xFF	Ecb Not In Use

Ecb Return Values

0xFC	Event Canceled

Remarks

IPXCancelEvent (Win) returns a completion code upon returning control to the application program.

ECB might be performing an event such as send or listen under IPX, a schedule or reschedule under AES, or a listen under SPX.

When an ECB is submitted to IPX to coordinate an event, IPX sets ECB's *inUseFlag* to a nonzero value, indicating the ECB is unavailable to other applications. Possible nonzero values for *inUseFlag* are listed below:

Decimal	Hex	Flag Value Description
224	0xE0	AES Temporary Indicator
248	0xF8	Critical Holding (IPX in critical process)
250	0xFA	Processing
251	0xFB	Holding (in processing after an event occurred)
252	0xFC	AES Waiting
253	0xFD	Waiting
254	0xFE	Receiving
255	0xFF	Sending

After attempting to cancel the event defined by the ECB, IPX sets ECB's *completionCode* to an appropriate value. It also sets ECB's *inUseFlag* to 0x00 (available for use). IPX does not call the ESR referenced by ECB's *ESRAddress*.

IPXCancelEvent (Win) cannot cancel packets already sent by the node's driver.

See Also

IPXScheduleIPXEvent (Win)

IPXCloseSocket (Win)

Closes the specified IPX socket and cancels any events defined by the ECBs associated with the socket

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXCloseSocket(
   DWORD    IPXTaskID,
   WORD     socketNumber);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

socketNumber

(IN) Indicates the socket number of the socket to be closed (high-low).

Remarks

IPX returns a value of 0xFC in each ECB's *completionCode*, indicating the event has been canceled. IPX sets each ECB's *inUseFlag* to 0x00 (available for use). IPX does not call ECBs' associated ESRs.

NOTE: Windows allows no short-lived sockets. When a windows application quits, no DOS end-of-task is sent; therefore, IPX cannot close any short-lived sockets. To prevent the application from leaving open sockets after exiting, **IPXCloseSocket (Win)** must be called prior to exiting the application.

IPXTaskID must contain the task ID assigned by the intialization function to this IPX process.

Any socket number can be closed; no error is generated if the specified socket is not opened. Before an application program terminates and destroys the ESR code, it must close all sockets that it has opened. Otherwise the ESR could be called after it is no longer available, permanently hanging the workstation.

IPXCloseSocket (Win) must not be called from within an ESR.

NOTE: After all sockets are closed, **IPXSPXDeinit (Win)** must be called to clean up any resources allocated to the application by

NWIPXSPX.DLL.

See Also

IPXOpenSocket (Win), **IPXSPXDeinit (Win)**

IPXDisconnectFromTarget (Win)

Informs the communication driver the application does not intend to send any more packets to the specified station

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXDisconnectFromTarget(
   DWORD       IPXTaskID,
   BYTE FAR   *networkAddress);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

networkAddress

(IN) Points (high-low) to the 12-byte local network address. The 12 bytes are organized as follows:

Bytes 0---3 = Network number

Bytes 4---9 = Node number

Bytes 10---11 = Socket number

Remarks

IPXDisconnectFromTarget (Win) is a courtesy to network communications drivers operating strictly on a point-to-point basis at the physical transport level. Once informed, the driver can dismantle any board-level virtual connection with the node specified.

networkAddress identifies the node with which communication will be terminated. After **IPXDisconnectFromTarget (Win)** is made, any virtual connection between the machine the application is running on and the target machine may be dismantled by the driver.

An application should never call **IPXDisconnectFromTarget (Win)** from within an ESR.

See Also

SPXAbortConnection (Win), **SPXTerminateConnection (Win)**

IPXGetInternetworkAddress (Win)

Returns the network and node address of the requesting workstation
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXGetInternetworkAddress(
   DWORD       IPXTaskID,
   BYTE FAR   *networkAddress);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

networkAddress

(OUT) Points to the 10-byte local network address. The 10 bytes are organized as follows:

Bytes 0---3 = Network number
Bytes 4---9 = Node number

Remarks

IPXGetInternetworkAddress (Win) is especially useful to any application that must inform other nodes of its address on the internetwork.

IPXGetInternetworkAddress (Win) does not return a socket number. The socket number is determined when an application opens a socket. When an application builds a complete 12-byte network address, it must append the appropriate socket number.

See Also

IPXGetLocalTarget (Win)

IPXGetIntervalMarker (Win)

Returns a time marker from IPX
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

WORD FAR PASCAL IPXGetIntervalMarker(
   DWORD   IPXTaskID);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

Return Values

Time as a WORD.

Remarks

IPXGetIntervalMarker (Win) may be used to measure the elapsed time between two events. To do this, call**IPXGetIntervalMarker (Win)** twice, once before each of the two events. Subtract the first interval marker from the second. The difference represents the time interval between the two events (in clock ticks). This result remains accurate even if the interval marker wraps through zero between the first and second event. The timer is not intended for use with time intervals greater than 1 hour due to the precision of the interval marker (16 bits unsigned).

The interval marker is a value between 0 and 65,535 [(0x0000) and (0xFFFF)]. Each interval represents one IBM PC clock tick, which is approximately 1/18 second.

IPXInitialize (Win) must be called before calling **IPXGetIntervalMarker (Win)**.

See Also

IPXInitialize (Win)

IPXGetLocalTarget (Win)

Returns the value to be placed in an ECB's *immediateAddress*
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXGetLocalTarget(
   DWORD        IPXTaskID,
   BYTE FAR    *networkAddress,
   BYTE FAR    *immediateAddress,
   int FAR     *transportTime);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

networkAddress

(IN) Points to the 12-byte destination network address (high-low).

immediateAddress

(OUT) Points to the ECB's 6-byte immediate address (high-low).

transportTime

(OUT) Indicates the estimated amount of time (in system clock ticks) for sending a 576-byte packet to the destination node.

Return Values

0x0000	Successful
0xF1	Ipx/Spx Not Initialized
0xFA	No Local Target Identified

Remarks

IPXGetLocalTarget (Win) passes the internetwork address of a destination node to IPX. IPX returns an estimated packet transport time and either the node address of the destination node or the node address of the local bridge it will use to route a packet to the destination node.

The 12 bytes of *networkAddress* are defined as:

Bytes 0---3 = Network Number
Bytes 4---9 = Node Number
Bytes 10---11 = Socket Number

transportTime contains an estimate of the time (in clock ticks of approximately 1/18 second) a 576-byte packet takes to travel from the source node to the destination node. Since the returned value is only an estimate, actual travel time can vary depending on network traffic and packet size.

IPXGetLocalTarget (Win) can be used with destination addresses using broadcast values (all 0xFFs) in *node* of *networkAddress*.

When IPX receives a packet, it records the node address of the sending node in *immediateAddress* of the receive ECB. The 6 bytes of *immediateAddress* are the local target node number, not the local source node number. The local source node number is the number of the calling workstation.

Therefore, once an application begins exchanging packets with an application on another node, either application can use *immediateAddress* returned with a receive ECB to obtain the local target value instead of continually calling **IPXGetLocalTarget (Win)**.

An application can use an ECB that has received a packet to send a packet to the originating address without modifying *immediateAddress* in ECB.

IPXGetLocalTarget (Win) can be called from within IPX, or an AES ESR, or directly from the main portion of an application. It must not be called from any other kind of interrupt service routine.

See Also

IPXSendPacket (Win), **IPXListenForPacket (Win)**

IPXGetLocalTargetAsync (Win)

Returns the value to be placed in an ECB's *immediateAddress* field
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

WORD FAR PASCAL IPXGetLocalTargetAsync(
   AGLT FAR   *listenAGLT,
   AGLT FAR   *sendAGLT,
   BYTE FAR   *destAddr);
```

Parameters

listenAGLT

(IN) Pointer to a listen AGLT.

sendAGLT

(IN) Pointer to a send AGLT.

destAddr

(IN) Points to the 12-byte destination network address (high-low).

Return Values

0x0000	Successful
0xF1	Ipx/Spx Not Initialized
0xFA	No Local Target Identified

Messages

0x6666	IMMEDIATE_ADDRESS_FAILED
0x7777	IMMEDIATE_ADDRESS_COMPLETE

Remarks

The address is placed in the ECB specified in *listenAGLT*.

IPXGetLocalTargetAsync(Win) passes the internetwork address of a destination node to IPX. IPX returns an estimated packet transport time and either the node address of the destination node or the node address of a local bridge that can be used to route a packet to the destination node.

IPXGetLocalTargetAsync (Win) calls the listen ESR.

Before calling **IPXGetLocalTargetAsync (Win)** set the fields of *listenAGLT* as follows:

listenAGLT.ecb.fragmentDescriptor[0].address

Set to point to a buffer that is filled with zeros. The buffer must be 64 bytes or larger.

listenAGLT.ecb.fragmentDescriptor[0].size

Set to the size of the buffer pointed to by *listenAGLT.ecb.fragmentDescriptor[0].address.*

listenAGLT.taskID

Set to the IPX task ID that was returned when **IPXInitialize (WIN)** was called.

listenAGLT.retry

This field can be ignored.

listenAGLT.hWnd

Set to the handle of window from which **IPXGetLocalTargetAsync (Win)** is called.

In addition, the fields of *sendAGLT* must be filled out as follows before calling **IPXGetLocalTargetAsync (Win)**:

sendAGLT.ecb.fragmentDescriptor[0].address

Points to a buffer that is filled with zeros. The buffer must be 64 bytes or larger.

sendAGLT.ecb.fragmentDescriptor[0].size

Set to the size of the buffer pointed to by *sendAGLT.ecb.fragmentDescriptor[0].address.*

sendAGLT.taskID

The IPX task ID that was returned when **IPXInitialize (WIN)** was called.

sendAGLT.retry

This field can be ignored.

sendAGLT.hWnd

The handle of window from which **IPXGetLocalTargetAsync (Win)** is called.

If **IPXGetLocalTargetAsync (Win)** can determine the local target, it will

send your application an IMMEDIATE_ADDRESS_COMPLETE message. Otherwise, it will send your application an IMMEDIATE_ADDRESS_FAILED message. The message will be sent to the window specified by *listenAGLT.hWnd*.

If the IMMEDIATE_MESSAGE_COMPLETE message is received, the *lParam* parameter of your Windows **WndProc** procedure will contain a pointer to the received ECB. You can then use this pointer to extract the immediate address from the ECB in the following manner:

```
...
static BYTE immediate[6];
ECB *receivedImmediate;
...

case IMMEDIATE_ADDRESS_COMPLETE:
    receivedImmediate = (ECB FAR*) lParam;
    memcpy(&immediate, &receivedImmediate->immediateAddress, 6);
...
```

CAUTION: To be safe, you should not allow your application to terminate until one of the above messages is received. Failure to allow NWIPXSPX.DLL to clean up after calling IPXGetLocalTargetAsync (Win).could compromise system stability.

See Also

IPXSendPacket (Win), **IPXListenForPacket (Win)**

IPXGetMaxPacketSize (Win)

Finds the maximum packet size allowed by a workstation's topology (LAN card and wiring)

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXGetMaxPacketSize(
   void);
```

Parameters

IPXGetMaxPacketSize (Win) does not require any parameters.

Return Values

0x0000	Successful

Remarks

Packet size on a local network is limited only by that network's topology. The content and structure of an IPX packet's data portion are entirely the responsibility of the application using IPX. The application may be able to take advantage of a larger allowable packet size if the topology permits. Use **IPXGetMaxPacketSize (Win)** to see what size packets the topology allows.

NOTE: Bridges will not route packets larger than 576 bytes. Because bridges will not fragment packets and reassemble them later, the application must not exceed the 576-byte packet size limit if it is to work across a bridge.

See Also

IPXInitialize (Win), **SPXInitialize (Win)**

IPXInitialize (Win)

Returns the entry address for the IPX interface
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXInitialize(
   DWORD FAR   *IPXTaskID,
   WORD         maxECBs,
   WORD         maxPacketSize);
```

Parameters

IPXTaskID

(IN/OUT) On input, points to how resources are allocated. On return, points to the assigned task ID.

maxECBs

(IN) Indicates the maximum number of outstanding ECBs that can be submitted to IPX.

maxPacketSize

(IN) Indicates the maximum size packet that can be sent by the application. The default value is 576 bytes: 30 bytes for the IPX header, and 546 bytes for the data portion of the packet.

Return Values

0x0000	Successful
0xF0	Ipx Not Installed
0xF1	Ipx/Spx Not Initialized
0xF2	No Dos Memory
0xF3	No Free Ecb
0xF4	Lock Failed
0xF5	Over The Maximum Limit
0xF6	Ipx/Spx Previously Initialized

Remarks

IPXInitialize (Win) initializes a variable (*IPXLocation*) in the NIT.C library with the address of the IPX services. It must be executed before any of the IPX functions in this manual can be performed.

As an input parameter, *IPXTaskID* determines how resources are allocated in the following way:

0x00000000 Resources allocated directly to the calling application.

0xFFFFFFFE Resources allocated directly to the calling application; however, multiple initializations are allowed.

0xFFFFFFFF Resources allocated in a pool for access by multiple applications.

However, DLLs frequently manage their own resources, regardless of the number of clients they service. For example, a DLL may manage a pool of ECBs for multiple applications wanting to access an SPX server.

As an output parameter, *IPXTaskID* receives the task ID assigned by the initialization function to this IPX or SPX process. The application must store this task ID for later use. Most Communication Services functions require *IPXTaskID*, returned by **IPXInitialize (Win)**, as one of their input parameters.

NOTE: For every *IPXTaskID* assigned by **IPXInitialize (Win)**, a matching call to **IPXSPXDeinit (Win)** must be provided. **IPXSPXDeinit (Win)** releases the resources allocated by NWIPXSPX.DLL for each task ID.

See Also

SPXInitialize (Win), **IPXSPXDeinit (Win)**

IPXListenForPacket (Win)

Prepares IPX to receive an IPX packet
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXListenForPacket(
   DWORD      IPXTaskID,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

eventControlBlock

(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xF7	ECB in use by VIPX
0xFC	Request Cancelled
0xFD	Bad Packet
0xFF	Socket Not Open

Remarks

IPXListenForPacket (Win) passes an ECB address to IPX for the purpose of receiving an IPX packet. It then returns control to the calling application. Meanwhile, IPX listens for and attempts to receive a packet.

Before calling **IPXListenForPacket (Win)**, an application must open the socket and initialize ECB's *ESRAddress*, *socketNumber*, *immediateAddress*, *fragmentCount*, and *fragmentDescriptor*. If no routine is to be called, *ESRAddress* should contain a NULL.

Initially, IPX sets ECB's *inUseFlag* to (0xFE) indicating the ECB is waiting to receive a packet. IPX also adds the ECB to a buffer pool of ECBs all

listening for IPX packets on the same socket. IPX imposes no limit on the number of ECBs that can be listening concurrently on a socket.

When IPX detects an incoming packet, IPX uses one of the listening ECBs to receive the packet. Listen ECBs are not filled in the order they were submitted to IPX. IPX records an appropriate value in the selected ECB's *completionCode* and places a node address in ECB's *immediateAddress*. The value in *immediateAddress* identifies either the sending node (if the sending node resides on the local network), or the local bridge that routed the packet to the receiving node (if the sending node does not reside on the local network).

Finally, IPX sets ECB's *inUseFlag* to 0x00 (available for use) and calls the ESR referenced by ECB's *ESRAddress* (if applicable).

NOTE: Existing non-Windows applications frequently verify whether or not an ECB has been released by using while(ecb.inUseFlag). However, using such a statement from the Window's environment prevents the LAN driver from setting ECB's *inUseFlag* to zero after the packet is sent. If *inUseFlag* is never set to zero, while will remain in an infinite loop.

To avoid this situation, check *inUseFlag* from an ESR. If you prefer not to use an ESR, either have the application post a message, or have it activate a Window's timer to check *inUseFlag*.

See Also

IPXSendPacket (Win)

IPXOpenSocket (Win)

Opens an IPX socket before receiving a packet on the socket
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXOpenSocket(
   DWORD        IPXTaskID,
   WORD FAR    *socketNumber,
   BYTE         socketType);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

socketNumber

(IN/OUT) Points to a socket number (high-low) or a 0x00 value.

socketType

(IN) Indicates the type of socket to be opened. *socketType* can be either:

0x00 = Stays open until program closes it or the program terminates (short-lived)
0xFF = Stays open until program closes it (long-lived)

Return Values

0x0000	Successful
0xF0	Ipx Not Installed
0xF1	Ipx/Spx Not Initialized
0xFE	Socket Table Full
0xFF	Socket Already Open

Remarks

Call **IPXOpenSocket (Win)** to open a socket before receiving a packet on the socket.

IPXTaskID must contain the task ID assigned by **IPXInitialize (Win)**.

socketNumber contains the number of the socket to be opened. Passing 0x0000 in this field allows IPX to open an available socket of its choice, known as a dynamic socket, in the range 0x4000 to 0x5000. The dynamic socket opened will be returned in *socketNumber*, which is a 2-byte variable.

> **NOTE:** If the application can have more than one instance on a node, be sure the application uses dynamic sockets. Conversely, if the application uses a well-known, statically-defined socket, do not allow more than one instance of the application to run on a node. This restriction is due to the pooling of ECBs by socket, and to the necessity of using the socket to demultiplex communications.

If **IPXOpenSocket (Win)** returns a completion code of 0x00, the socket has been opened as expected. If the completion code is 0xFE, the socket table is already full. If **IPXOpenSocket (Win)** returns a completion code of 0xFF, the specified socket is already open.

socketType specifies how long the socket should remain open. Unless the program intends to terminate and stay resident, the socket type should always be temporary (0x00---SHORT_LIVED). The exception to this is if programming in MS Windows you should always open sockets as permanent (0xFF---LONG_LIVED) sockets. Sockets that are opened as temporary (SHORT_LIVED) are automatically closed upon program termination (if the NetWare shell is loaded), and all associated events pending on that socket are canceled. If the application program opens a socket as permanent, it must guarantee that the ESR is available even after it terminates. Otherwise, the workstation could hang if the ESR is called after it is no longer available.

By default, IPX supports up to 20 open sockets on one workstation. The maximum number of simultaneously opened sockets for a workstation can be as high as 150 and is configurable in SHELL.CFG.

After all sockets are closed, **IPXSPXDeinit (Win)** must be called to clean up any resources allocated to the application by NWIPXSPX.DLL.

See Also

IPXCancelEvent (Win), **IPXCloseSocket (Win)**, **IPXSPXDeinit (Win)**

IPXRelinquishControl (Win)

Relinquishes control of a workstation's CPU
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXRelinquishControl(
   void);
```

Parameters

IPXRelinquishControl (Win) requires no parameters.

Remarks

IPXRelinquishControl (Win) can temporarily relinquish control of the workstation CPU so that other processing can be done while the application is waiting for input.

IPX applications should make repeated calls to **IPXRelinquishControl (Win)** during idle time to allow other applications in the computer to use the CPU. This is especially important when the application is resident with NetWare server or bridge software since **IPXRelinquishControl (Win)** greatly improves the efficiency of the server or bridge.

IPXRelinquishControl (Win) also allows the communications driver in IPX to run. This is important if the driver is not interrupt driven. On a normal workstation, **IPXRelinquishControl (Win)** invokes a polling procedure provided by the network communications driver. It represents the only opportunity the driver has to use the CPU to send and receive packets, events essential to the application itself.

IPXScheduleIPXEvent (Win)

Schedules an IPX event
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL IPXScheduleIPXEvent(
   DWORD      IPXTaskID,
   WORD       time,
   ECB FAR    eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

time

(IN) Indicates the time interval.

eventControlBlock

(IN) Points to ECB.

Remarks

After initializing ECB's *ESRAddress* and *socketNumber*, executing **IPXScheduleIPXEvent** passes a delay time and an ECB address to IPX for the purpose of scheduling an IPX event. The function then returns control to the calling application. Meanwhile, IPX attempts to schedule the event.

The scheduled event is directly associated with an IPX socket. If **IPXCloseSocket (Win)** is executed with this socket as its argument, the event scheduled by **IPXScheduleIPXEvent (Win)** is canceled along with all other IPX packet events for the socket. To cancel only an event scheduled by **IPXScheduleIPXEvent (Win)**, call **IPXCancelEvent (Win)**.

time specifies the waiting time (in clock ticks) before a scheduled event takes place. It must be set to a value between 0 and 65,535 (0x0000 and 0xFFFF). The interval between each IBM PC clock tick is approximately 1/18 second. This allows an event to be scheduled for up to one hour.

IPXScheduleIPXEvent (Win) can reschedule an ECB previously submitted for scheduling. The application passes a pointer to the same ECB to the function, as well as a new expiration time. This is the only situation in which it is legal to pass IPX (AES) to an ECB that is still in

use. *inUseFlag* is set to a value of 0xFD while the AES is in control of the IPX ECB.

An ESR can also call **IPXScheduleIPXEvent (Win)**. For example, when IPX calls an ESR, the ESR can first check conditions in the system, determine that conditions are not acceptable for executing at the moment, and invoke **IPXScheduleIPXEvent (Win)** to reschedule itself. After the specified delay time, the ESR can execute again, check conditions, and continue the cycle until conditions are favorable.

An application should never call **IPXScheduleIPXEvent (Win)** to pass the address of an ECB currently being used by IPX for packet events.

See Also

IPXCancelEvent (Win)

IPXSendPacket (Win)

Initiates the sending of an IPX packet
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: IPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCALIPXSendPacket(
   DWORD      IPXTaskID,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

eventControlBlock

(IN/OUT) Points to the ECB structure.

ECB Return Values

0x0000	Successfully Sent But Not Necessarily Received
0xF7	ECB in use by VIPX
0xFC	Request Canceled
0xFD	Given Packet Did Not Have a 30-byte Packet Header As the First Fragment, or Its Total Length Exceeded 567 Bytes
0xFE	Packet Not Deliverable
0xFF	Hardware Failure

Remarks

IPXSendPacket (Win) passes an ECB address to IPX for the purpose of sending an IPX packet. It then returns control to the calling application. Meanwhile, IPX attempts to send the packet.

Before calling **IPXSendPacket (Win)**, the application must initialize ECB's *ESRAddress*, *socketNumber*, *immediateAddress*, *fragmentCount*, and *fragmentDescriptor*. The application must also prepare the IPX header of the associated packet by filling in *packetType* and *destination*.

IPXSendPacket (Win) then passes the ECB to the network communication drivers to initiate the send operation.

Although *socketNumber* in ECB is significant (IPX uses it as the source socket number on the packet header), the socket need not be open to perform a send on it.

immediateAddress in ECB can be set by calling **NWIPXGetLocalTarget (Win)**.

Initially, IPX sets ECB's *inUseFlag* to 0xFF, indicating that the ECB is sending a packet. After attempting to send the packet, IPX sets ECB's *completionCode* to an appropriate value and sets *inUseFlag* to 0x00, indicating that the ECB is available for use. Finally, IPX calls the ESR referenced by ECB's *ESRAddress* (if applicable).

0x00 completion code indicates that the packet was sent successfully but does not guarantee the packet was received successfully by the destination node. For example, the transmission media may lose or garble the packet, or the destination socket may not be open or listening. IPX does not inform the sending node if these problems occur.

0xFE indicates that the packet is undeliverable. The ECB returns this completion code for one of two reasons:

IPX cannot find a bridge with a path to the destination network
The target node address does not exist

An application can send or broadcast an IPX packet to any socket on the internetwork including the socket on which the application sends the packet. Packets sent to sockets residing in the same node as the application are called intra-node packets.

To send a broadcast packet (one that will be received by every listening station on a network), the node portion of *destination* in the packet header should be initialized to all 0xFFs (-1). If the broadcast is to the network on which the application resides, ECB's *immediateAddress* should also be set to 0xFF.

See Also

IPXGetLocalTarget (Win), **IPXListenForPacket (Win)**

IPXSPXDeinit (Win)

Releases any resources allocated to an application by NWIPXSPX.DLL for use by other applications

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: IPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL IPXSPXDeinit(
   DWORD   IPXTaskID);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

Return Values

0x0000	Successful
0xF1	IPX/SPX Not Initialized

See Also

IPXCancelEvent (Win), **IPXCloseSocket (Win)**, **IPXRelinquishControl (Win)**, **SPXAbortConnection (Win)**, **SPXTerminateConnection (Win)**

IPX for Windows 95

For the 32-bit Windows 95* interface to IPX, see NWSIPX.

IPX for Windows NT

For the 32-bit Windows NT* interface to IPX, see NWSIPX.

SPX: Functions

SPX for DOS

SPXAbortConnection (DOS)

Passes an SPX™ connection ID to SPX for the purpose of unilaterally aborting an SPX connection

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: SPX

Syntax

```
#include <nwipxspx.h>

void cdecl SPXAbortConnection(
   WORD   connectionIDNumber);
```

Parameters

connectionIDNumber

(IN) Indicates the connection ID number assigned by SPX when the connection was established.

Remarks

SPXAbortConnection (DOS) returns control to the calling application. Meanwhile, SPX aborts the connection.

SPX makes no attempt to inform the connection partner of the decision to abort the connection. The partner discovers the connection is no longer valid when it attempts to send a packet on the connection or when its watchdog checks the connection after the inactivity timer expires.

Any **SPXEstablishConnection (DOS)**, **SPXTerminateConnection (DOS)**, **SPXListenForPacket (DOS)**, or **SPXSendSequencedPacket (DOS)** calls that are outstanding on the connection are aborted. The completion codes in the ECBs of such calls are set to 0xED, indicating an abnormal connection termination. The ESRs of the affected ECBs are called (unless *ESRAddress* is NULL).

If any of the affected ECBs are in a state not allowing them to be canceled (the network interface card is in the middle of sending the packet, for example), they will be canceled at the earliest possible time. In this case, the completion code of the ECB is still set to 0xED, and the ESR is called if it exists.

SPXAbortConnection (DOS) is included to allow a connection partner to unilaterally dismantle the connection if some catastrophic condition is detected. Under normal conditions **SPXTerminateConnection (DOS)** should be used to ensure both connection partners break the connection in a controlled fashion.

See Also

SPXEstablishConnection (DOS), **SPXTerminateConnection (DOS)**

SPXEstablishConnection (DOS)

Establishes a connection with a listening socket
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: SPX

Syntax

```
#include <nwipxspx.h>

int cdecl SPXEstablishConnection(
   BYTE         retryCount,
   BYTE         watchDog,
   WORD FAR    *connectionID,
   ECB FAR     *eventControlBlock);
```

Parameters

retryCount

(IN) Indicates values from 0 to 255.

watchDog

(IN) Indicates the flag controlling watchdog connection supervision (0=disable, !0=enable).

connectionID

(OUT) Points to an unsigned integer that will receive the connection ID number.

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	SPX Connection Started
0xEF	Connection Table Full
0xFD	Malformed Packet
0xFF	Socket Not Opened

ECB Return Values

0xED	SPX_NO_ANSWER_FROM_TARGET (either a hardware failure has occurred, or the application has used

	SPXAbortConnection)
0xEF	SPX_CONNECTION_TABLE_FULL (no further connections may be initiated until an active connection is terminated. The SPX Connection Table has room for entries) 100
0xFC	SPX_SOCKET_CLOSED (connection socket was closed before the command was completed, and ECB's ESR was *not* called)
0xFD	SPX_MALFORMED_PACKET (either fragment count was not 1 or the buffer size was not 42)
0xFF	Socket Not Opened

Remarks

SPXEstablishConnection (DOS) passes a *retryCount*, SPX *watchDog*, and *eventControlBlock* address to SPX to establish an SPX connection. **SPXEstablishConnection (DOS)** then returns control to the calling application. Meanwhile, SPX attempts to establish the connection.

NOTE: Sockets used by SPX connections must be dedicated solely for use by SPX and cannot be used with **IPXSendPacket (DOS)** or **IPXListenForPacket (DOS)** for sending or receiving packets directly.

If an SPX socket is closed, all connections on that socket automatically terminate and any pending events (sends or listens) are canceled.

Do not cancel **SPXEstablishConnection (DOS)** by calling **IPXCancelEvent (DOS)**. If an application does not want to establish a connection, call **SPXAbortConnection (DOS)** and **SPXEstablishConnection (DOS)**.

Before calling **SPXEstablishConnection (DOS)**, the application must do the following:

1. Set *retryCount*, *watchDog*, and *ECBAddress* (described below) to the appropriate values.

2. Use **IPXOpenSocket (DOS)** to open the socket specified in ECB's *socketNumber*.

3. Initialize ECB's *ESRAddress*, *socketNumber*, *fragmentCount*, and *fragmentDescriptor*. *fragmentCount* should be initialized to 0x01. *fragmentDescriptor* must point to a 42-byte buffer containing the header of an SPX packet.

4. Initialize SPX header's *destination*. None of these fields can be set to -1 (broadcast).

5. Create at least two listen ECBs, and pass them to SPX with **SPXListenForSequencedPacket (DOS)**. Once SPX sends the

SPXEstablishConnection (DOS) packet mentioned above, SPX uses one of the listen ECBs to receive a confirmation packet from the destination socket. The other can be used by the watchdog process to maintain the SPX connection once it is established.

To establish the connection, SPX creates a local connection half and verifies that the local connection half is fully functional. **SPXEstablishConnection (DOS)** then returns to the calling procedure.

Once SPX creates and verifies the local connection half, **SPXEstablishConnection (DOS)** returns a completion code. SPX also records this completion code in *completionCode* of the sending ECB. If **SPXEstablishConnection (DOS)** returns anything but 0x00 (SUCCESSFUL), the attempt to establish a connection stops at this point.

Meanwhile, SPX sends an Establish Connection packet through the local connection half to a specified (listening) destination socket to establish a connection. To receive the Establish Connection packet, the destination socket must pass an ECB to its SPX process with **SPXListenForConnection (DOS)**.

More extensive parameter descriptions follow:

retryCount

Specifies how many times SPX resends unacknowledged packets before concluding that the destination node is not functioning properly. The application can set this field to 0x00, which instructs SPX to use its own internal retry count. A value of 1 through 255 (inclusive) indicates SPX should resend packets the specified number of times.

watchDog

Monitors an SPX connection, ensuring the connection is functioning properly when traffic is not passing through the connection. To enable *watchDog*, set *watchDog* to 0xFF ENABLE_WATCHDOG. A value of 0x00 disables the feature.

If *watchDog* determines that an SPX connection has failed, it aborts the connection and posts an **SPXListenSequencedPacket (DOS)** ECB with a *completionCode* value of SPX_CONNECTION_FAILED (0xED). The connection ID of the failed connection is in the first word of ECB's *IPXWorkspace*. When ECB's *inUseFlag* is reset to zero, *completionCode* contains 0x00 if the SPX connection was established.

sourceConnectionID

If **SPXEstablishConnection (DOS)** returns a completion code of 0x00 (SUCCESSFUL), it also returns a connection ID in *sourceConnectionID*. Although no connection is established yet with a destination socket, a connection occupies one entry in the node's SPX Connection Table. SPX then sends an **SPXEstablishConnection (DOS)** packet to the destination node.

See Also

SPXAbortConnection (DOS), **SPXTerminateConnection (DOS)**

SPXGetConnectionStatus (DOS)

Returns the status of an SPX connection
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SPXGetConnectionStatus(
   WORD                    connectionIDNumber,
   CONNECTION_INFO FAR    *connectionInfo);
```

Parameters

connectionIDNumber

(IN) Indicates the number assigned by SPX when the connection was established.

connectionInfo

(OUT) Points to a 44-byte connection information structure (CONNECTION_INFO).

Return Values

0x0000	Connection OK
0xEE	Invalid Connection

Remarks

SPXGetConnectionStatus (DOS) allows an application to check on the current status of an SPX connection. If the specified connection exists, it returns the connection ID number and a buffer of information on the connection. If the specified connection does not currently exist, **SPXGetConnectionStatus (DOS)** returns a completion code (0xEE).

CONNECTION_INFO is defined in NXT.H, and contains the following:

```
typedef struct
{
   BYTE   connectionState;
   BYTE   connectionFlags;
   WORD   sourceConnectionID;/*high-low*/
   WORD   destinationConnectionID;/*high-low*/
```

```
    WORD    sequenceNumber;/*high-low*/
    WORD    acknowledgeNumber;/*high-low*/
    WORD    allocationNumber;/*high-low*/
    WORD    remoteAcknowledgeNumber;/*high-low*/
    WORD    remoteAllocationNumber;/*high-low*/
    WORD    connectionSocket;/*high-low*/
    BYTE    immediateAddress[6]IPXAddress destination;
    WORD    retransmissionCount;/*high-low*/
    WORD    estimatedRoundTripDelay;/*high-low*/
    WORD    retransmittedPackets;/*high-low*/
    WORD    suppressedPackets;/*high-low*/;
} CONNECTION_INFO
```

connectionState

indicates the current state of the specified connection, as follows:

Hex	Name	Description
0x01	WAITING	SPX is listening on the connection, waiting to receive an SPX Establish Connection packet. See **SPXListenForConnection (DOS)**.
0x02	STARTING	SPX is attempting to create a full connection with a remote workstation by sending SPX Establish Connection packets on its half of the connection. See **SPXEstablishConnection (DOS)**
0x03	ESTABLISHED	SPX has established a connection with a remote workstation, and the connection is open for two-way packet transmission.
0x04	TERMINATING	The remote SPX has terminated the connection. However, the local SPX has not yet terminated its half of the connection. See **SPXTerminateConnection (DOS)**.

sourceConnectionID

Indicates the connection ID assigned by the local client's SPX package.

destinationConnectionID

Indicates the cconnection ID assigned by the remote client's SPX package. It is valid only if the connection state is ESTABLISHED or TERMINATING.

sequenceNumber

Indicates the next packet the local SPX sends to the remote workstation. SPX assigns a sequence number (0x0000 to 0xFFFF) to each packet it sends to the remote workstation. This ensures sequenced transmission at the local end and sequenced reception at

the remote end. When the sequence number reaches 0xFFFF, it wraps to 0x0000. *sequenceNumber* is not valid if the connection state is WAITING.

acknowledgeNumber

Indicates the sequence number of the next packet that the local SPX expects to receive from the remote SPX. When this sequence number reaches 0xFFFF, it wraps to 0x0000. It is not valid if the connection state is WAITING.

allocationNumber

Indicates (in conjunction with *acknowledgeNumber*) the number of outstanding packet receive buffers (posted listens) available for a given SPX connection. It is used by SPX to implement flow control between communicating applications. *allocationNumber* minus *acknowledgeNumber* equals the number of posted listens outstanding on the connection socket. SPX sends packets only until the local sequence number equals *allocationNumber* of the remote partner. *allocationNumber* increments from 0xFFFF and wraps to 0x0000. It is not valid if the connection state is WAITING. The number is based on the number of Listen ECBs outstanding.

remoteAcknowledgeNumber

Indicates the sequence number of the next packet the remote SPX expects to receive from the local SPX. When this sequence number reaches 0xFFFF, it wraps to 0x0000. It is not valid if the connection state is WAITING.

The local SPX is allowed to send packets with sequence numbers up to and including *remoteAllocationNumber*. Meanwhile, the remote SPX increments the remote allocation number as the remote workstation generates listen ECBs. In this way, the remote SPX regulates the number of packets the local SPX sends and avoids being inundated with packets it is not ready to receive. When this number reaches 0xFFFF, it wraps back to 0x0000. *remoteAcknowledgeNumber* is not valid if the connection state is WAITING. The number is based on the number of ECBs outstanding.

connectionSocket

Indicates the socket number the local SPX is using to send and receive packets.

immediateAddress

Indicates the node address of the bridge (on the local network) routing the packets to and from the remote workstation. If the local and remote workstations reside on the same local network, the immediate address is the node address of the remote workstation. (In this case, a bridge is unnecessary.) It is not valid if the connection state is WAITING.

destinationSocket

Indicates the socket address through which the remote SPX expects to receive packets pertaining to this connection, and from which packets are sent to the local SPX. The two SPX packages do not need to use the

same socket; this number does not need to be the same as *connectionSocket*. It is not valid if the connection state is WAITING.

retransmissionCount

Indicates the number of times SPX attempts to retransmit an unacknowledged packet before it determines the remote SPX has become inoperable or unreachable.

estimatedRoundTripDelay

Indicates the amount of time (in 1/18th second units) SPX should wait for an acknowledgment to arrive from the remote SPX partner. SPX includes both dynamic flow control and dynamic routing. This means the value of *estimatedRoundTripDelay* may change from time to time as SPX adjusts to observed fluctuations in packet throughput on the underlying internetwork. It is not valid if the connection state is WAITING.

retransmittedPackets

Indicates the number of times SPX had to retransmit a packet on this connection before it received an expected acknowledgment. When this field reaches a value of 0xFFFF, it wraps to 0x0000. It is valid only if the connection state is ESTABLISHED or TERMINATING.

suppressedPackets

Indicates the number of times SPX received a data packet on the connection that was not delivered to the connection client because the packet was either a duplicate of previously delivered data or was out-of-bounds for the current receive window. When it reaches a value of 0xFFFF, it wraps to 0x0000. *suppressedPackets* is valid only if the connection state is ESTABLISHED or TERMINATING.

See Also

SPXEstablishConnection (DOS)

SPXInitialize (DOS)

Checks to see if SPX is installed
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: SPX

Syntax

```
#include <nwipxspx.h>

int cdecl SPXInitialize(
   BYTE FAR   *majorRevisionNumber,
   BYTE FAR   *minorRevisionNumber,
   WORD FAR   *maxConnections,
   WORD FAR   *availableConnections);
```

Parameters

majorRevisionNumber

(OUT) Points to a byte receiving the SPX major revision number.

minorRevisionNumber

(OUT) Points to a byte receiving the SPX minor revision number.

maxConnections

(OUT) Points to an unsigned integer receiving the maximum number of SPX connections supported.

availableConnections

(OUT) Points to an unsigned integer receiving the number of available SPX connections.

Return Values

0x0000	SPX_NOT_INSTALLED
0x00FF	SPX_IS_INSTALLED

Remarks

SPXInitialize (DOS) determines whether SPX is installed on the node where the application is running. If SPX is installed, SPXInitialize returns the major and minor revision numbers of SPX, the maximum number of connections supported by SPX, and the number of SPX connections available.

Early versions of the NetWare® OS did not support SPX. Specifically, only the last shell (ANET3.COM, 2.01-4) of NetWare 2.0a supports the SPX. All versions of NetWare 2.1 and higher support SPX.

NOTE: In NetWare 2.0a, SPX is supported for workstations only, not for servers.

majorRevisionNumber and *minorRevisionNumber*

Indicate which SPX revision is installed. For example, Revision 1.0 returns a 1 in the first field and a 0 in the second.

maxConnections

Indicates the maximum number of SPX connections this particular version of SPX supports.

availableConnections

Indicates how many SPX connections are available to the application.

SPXListenForConnection (DOS)

Receives an Establish Connection packet and thereby establishes an SPX connection with a remote partner

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: SPX

Syntax

```
#include <nwipxspx.h>

void cdecl SPXListenForConnection(
   BYTE       retryCount,
   BYTE       watchDog,
   ECB FAR  *eventControlBlock);
```

Parameters

retryCount

(IN) Indicates the retry count (from 0 to 255).

watchDog

(IN) Indicates the flag controlling watchdog connection supervision (0=disable, !0=enable).

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	Successful

ECB Return Values

0xEF	SPX Connection Table Full
0xF	SPX Command Canceled (with **IPXCancelEvent (DOS)**)
0xF	Socket Not Opened

Remarks

SPXListenForConnection (DOS) passes a *retryCount*, SPX *watchDog* flag,

and *eventControlBlock* address to SPX to listen for and receive an **SPXEstablishConnection (DOS)**packet. When the **SPXEstablishConnection (DOS)** packet is received, a connection is established with the sending node.

SPXListenForConnection (DOS) then returns control to the calling function. Meanwhile, SPX attempts to receive an **SPXEstablishConnection (DOS)** packet and establish a connection.

NOTE: ECB's *ESRAddress* and *socketNumber* must be intialized. The other parameters are passed by **SPXListenForConnection (DOS)**.

Before **SPXListenForConnection (DOS)** can be called, **IPXOpenSocket (DOS)** must be called to open the socket to be used for the connection. When ECB's *inUseFlag* is reset to zero, *completionCode* contains 0x00 if the SPX connection was established.

SPX_CONNECTION_TABLE_FULL (0xFE) indicates all positions in the SPX connection table are occupied and no more connections are available. The application must wait until an active connection is terminated.

SPX_COMMAND_CANCELLED (0xFC) indicates the command was canceled by **IPXCancelEvent (DOS)**.

SPX tries to establish a connection in two steps as follows:

1. The function listens for an **SPXEstablishConnection (DOS)** packet.
2. After an establish connection packet is received, the function attempts to create a local connection.

If successful, SPX sends a confirmation packet to the source node, and both sides are ready to either send or receive data on the connection.

To complete the second step, SPX requires *retryCount*, SPX *watchDog*, and *eventControlBlock*.

retryCount specifies how many times SPX resends unacknowledged packets before concluding the partner node is not functioning properly. The application should set *retryCount* to 0x00; 0x00 instructs SPX to use its own internal retry count. A value of 1 through 255 (inclusive) indicates that SPX should resend packets the specified number of times.

SPX *watchDog* monitors an SPX connection, ensuring the connection is functioning properly when traffic is not passing through the connection. If *watchDog* determines an SPX connection has failed, it signals the application by recording a value of 0xED (Failed Connection) in *completionCode* of any listening ECB. *watchDog* also records the failed connection ID number in the same ECB's *IPXWorkspace* and calls the ECB's ESR.

The ECB (*eventControlBlock*) passed to **SPXListenForConnection (DOS)** needs no associated packets or fragments.

SPX-based applications must prepost at least two ECBs by calling **SPXListenForSequencedPacket (DOS)**. SPX uses one of these ECBs to receive the Establish Connection packet, and uses the other to acknowledge the Establish Connection packet has been received.

After the connection is established, SPX sends a Confirmation packet back to the node that initiated the connection. SPX records the following information in that packet's associated ECB fields:

IPXWorkspace (first 2 bytes)---Connection ID number

driverWorkspace---12-byte address of the partner node

completionCode---A value of 0x00 (Connection established)

inUseFlag---A value of 0x00 (available for use)

To cancel a connection attempted by **SPXListenForConnection (DOS)**, an application can call **IPXCancelEvent (DOS)**. In this situation, SPX records a value of 0xFC (SPX_COMMAND_CANCELLED) in ECB's *completionCode*.

The following ECBs occupy one entry each in the node's SPX Connection Table:

ECBs attempting to establish a connection

ECBs participating in a connection

See Also

IPXOpenSocket (DOS), **SPXEstablishConnection (DOS)**, **SPXListenForSequencedPacket (DOS)**

SPXListenForSequencedPacket (DOS)

Passes an ECB to SPX for the purpose of receiving a sequenced packet
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS
Service: SPX

Syntax

```
#include <nwipxspx.h>

void cdecl SPXListenForSequencedPacket(
   ECB FAR   *eventControlBlock);
```

Parameters

eventControlBlock
(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xED	Connection Failed
0xFC	SPX Socket Closed
0xFD	Packet Overflow
0xFF	SPX Socket Not Opened

Remarks

SPXListenForSequencedPacket (DOS) delivers an ECB address and the buffer space it identifies to SPX for the purpose of receiving a sequenced packet. It then returns control to the calling application. SPX allows applications to receive sequenced packets in a fully asynchronous fashion. Therefore, **SPXListenForSequencedPacket (DOS)** only dedicates the given ECB for use in receiving packets; it does not wait for an actual packet to be received.

Before calling **SPXListenForSequencedPacket (DOS)**, the application must initialize ECB's *ESRAddress*, *SocketNumber*, *fragmentCount*, and *fragmentDescriptor*. The socket specified by *socketNumber* must be previously opened by calling **IPXOpenSocket (DOS)**.

When ECB's *inUseFlag* is reset to zero, *completionCode* contains the codes listed in the ECB Completion Code table above.

If the completion code is 0xED, SPX *watchDog* determined a connection has failed and aborted the connection. If the code is 0xFD, a sequenced packet was received; but the available space defined by ECB's fragment descriptor list was not large enough to contain the entire packet.

Because both system and user packets arrive via the buffers in the listen ECB pool, SPX cannot function properly unless the SPX client makes available an adequate supply of listen ECBs. SPX imposes no limits on the number of ECBs that can be used concurrently for listening on a given socket. If a listening ECB is canceled with **IPXCancelEvent (DOS)**, ECB's *ESRAddress* must be reinitialized to an appropriate value before the ECB is reused.

See Also

SPXListenForConnection (DOS), **SPXSendSequencedPacket (DOS)**

SPXSendSequencedPacket (DOS)

Passes a connection ID and an ECB address to SPX for the purpose of sending an SPX packet and returns control to the calling application

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: SPX

Syntax

```
#include <nwipxspx.h>

void SPXSendSequencedPacket(
   WORD       connectionIDNumber,
   ECB FAR   *eventControlBlock);
```

Parameters

connectionIDNumber

(IN) Indicates the number assigned by SPX when the connection was established.

eventControlBlock

(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xEC	Connection Terminated
0xED	SPX Terminated Poorly
0xEE	Invalid Connection
0xFC	Socket Closed
0xFD	Malformed Packet

Remarks

Before calling **SPXSendSequencedPacket (DOS)**, ECB's *ESRAddress*, *fragmentCount*, and *fragmentDescriptor* must be initialized. The fragment buffer referenced by the first *fragmentDescriptor* must contain at least a 42-byte SPX packet header. The application can also initialize the SPX packet. header's *endOfMessage* bit in *connectionControl* and should initialize *dataStreamType*. All other fields in the SPX packet header are initialized by SPX.

Initially, SPX sets ECB's *inUseFlag*to a non-zero value indicating that the ECB is sending a packet. SPX also queues the ECB/packet combination for transmission.

When SPX completes its attempt to send the packet, SPX records a completion code in ECB's *completionCode*, sets *inUseFlag* to 0x00 (AVAILABLE_FOR_USE), and calls the ESR defined in ECB's *ESRAddress* (if applicable).

0x00 indicates the packet was successfully sent and received in order by the connection partner. An acknowledgment from the partner was returned.

0xEC indicates the remote partner terminated the connection without acknowledging this packet. SPX cannot guarantee the remote partner received this packet before the connection was destroyed.

0xED indicates the connection ended abnormally. One of the two partners used **SPXAbortConnection (DOS)** to abort the connection, or the connection partner failed to acknowledge receipt of this packet. This error is also reported if the network hardware fails or if the packet cannot be delivered to the specified destination.

0xEE indicates *SPXConnectionID* does not reference an established connection.

0xFC indicates the connection socket was closed. In this case, *EventServiceRoutine* is not called.

0xFD indicates the fragment count is zero, the first fragment is less than 42 bytes long, or the entire packet is greater than 576 bytes long.

WARNING: 0xFD causes the connection to be aborted.

Sockets used by SPX connections cannot be used to send or receive packets directly using IPX™ calls **IPXSendPacket (DOS)** and **IPXListenForPacket (DOS)**. The sockets must be dedicated solely for use by the SPX protocol.

When an application passes several ECB/packet combinations to SPX, SPX queues the packets and sends them in the order it receives them.

When an application or some other agent closes an SPX socket, SPX terminates all connections associated with the socket.

An application can establish a connection between two sockets residing in the same node. An application should use **SPXAbortConnection (DOS)** (not **IPXCancelEvent (DOS)**) to cancel an **SPXSendSequencedPacket (DOS)** event.

See Also

SPXListenForSequencedPacket (DOS)

SPXTerminateConnection (DOS)

Terminates an SPX connection by passing a connection ID and an ECB address to SPX and returns control to the calling application

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS

Service: SPX

Syntax

```
#include <nwipxspx.h>

void SPXTerminateConnection(
   WORD       connectionIDNumber,
   ECB FAR   *eventControlBlock);
```

Parameters

connectionIDNumber

(IN) Indicates the number assigned by SPX when the connection was established.

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	Successful

ECB Return Values

0x0000	SPX Connection Terminated
0xEC	Terminated By Remote Partner
0xED	SPX Terminated Poorly
0xEE	Invalid Connection
0xFD	Malformed Packet

Remarks

Before calling **SPXTerminateConnection (DOS)**, ECB's *ESRAddress*, *fragmentCount*, and *fragmentDescriptor* must be initialized. The application

must set *fragmentCount* to 1. *fragmentDescriptor* must point to a 42-byte buffer.

connectionIDNumber is the local number assigned to the connection upon creation by SPX. If the connection was created with **SPXEstablishConnection (DOS)**, it is the number returned from that call. If the connection was created with **SPXListenForConnection (DOS)**, it is the number obtained from the first 2 bytes of **SPXListenForConnection (DOS)** ECB's *IPXWorkspace* after the connection was made.

1. SPX records a value of 0xFE (TERMINATE_CONNECTION) in *dataStreamType* of the SPX header referenced in the ECB. This tells the receiver the connection is terminated.
2. SPX delivers the packet to the partner node.
3. SPX returns the appropriate completion code in ECB's *completionCode*, and records 0x00 (AVAILABLE_FOR_USE) in ECB's *inUseFlag*.
4. SPX calls the ESR referenced by ECB's *ESRAddress* (if applicable).

A completion code of 0xED indicates the remote connection partner failed to acknowledge the request within an appropriate amount of time. The connection is terminated on the local side, but SPX cannot guarantee the connection partner saw this **SPXTerminateConnection (DOS)** request. This error is also returned if the network hardware fails or the packet cannot be delivered to the specified destination.

A completion code of 0xFD indicates the fragment count was not 1 or the buffer size was not 42. Because of the asynchronous functionality of SPX, the termination request may not be complete when **SPXTerminateConnection (DOS)** returns. Thus, the ECB is not available until *inUseFlag* is reset to zero.

Once SPX terminates a connection, the position *connectionIDNumber* occupied in the Connection Table becomes available for use by a new connection. Applications can initiate new connections with **SPXEstablishConnection (DOS)**.

An application should call **SPXAbortConnection (DOS)** (not **IPXCancelEvent (DOS)**) to cancel **SPXTerminateConnection (DOS)**.

See Also

SPXAbortConnection (DOS), **SPXEstablishConnection (DOS)**, **SPXListenForConnection (DOS)**

SPX for NLM

SpxAbortConnection (NLM)

Aborts an SPX connection
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Platform: NLM
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxAbortConnection (
   unsigned short   connection);
```

Parameters

connection

(IN) Specifies the connection ID number assigned by SPX when the connection was established.

Return Values

0	(0x00)
ESUCCESS	238
(0xEE)	ERR_SPX_INVALID_CONNECTION

Remarks

SpxAbortConnection (NLM) passes a connection ID to SPX for the purpose of unilaterally aborting an SPX connection. It then returns control to the calling application. Meanwhile, SPX aborts the connection.

SPX makes no attempt to inform the connection partner of the decision to abort the connection. The partner discovers the connection is no longer valid when it attempts to send a packet on the connection or when its watchdog checks the connection after the inactivity timer expires.

Any **SpxEstablishConnection (NLM)**, **SpxTerminateConnection (NLM)**, or **SpxSendSequencedPacket (NLM)** commands that are outstanding on the connection are aborted. The completion codes in the ECBs of such commands are set to 0xFFED, indicating an abnormal connection termination. The semaphore handles of any incomplete

SpxSendSequencedPacket (NLM) ECBs on the affected session are posted (unless the *semHandle* field is NULL). The ECBs are queued off of their queue heads (unless the *queueHead* field is NULL).

If any of the affected ECBs are in a state that does not allow them to be cancelled (for example, if the network interface card is in the middle of sending the packet), they are cancelled at the earliest possible time. In this case, the completion code of the ECB is still set to 0xFFED, and the semaphore handles and queue heads of the affected ECBs are still acted on.

This function is included to allow a connection partner to unilaterally dismantle the connection if some catastrophic condition is detected. Under normal conditions, the **SpxTerminateConnection (NLM)** function should be used to ensure that both connection partners break the connection in a controlled fashion.

See Also

SpxEstablishConnection (NLM), **SpxTerminateConnection (NLM)**

SpxCancelEvent, SpxCancelPacket (NLM)

Local Servers: nonblocking
Remote Servers: N/A
NetWare Versions: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxCancelEvent (
   SPX_ECB   *ecb);
```

Parameters

ecb

(IN/OUT) Points to the SPX_ECB structure to be cancelled.

Return Values

0	(0x00)	ESUCCESS
238	(0xEE)	ERR_SPX_INVALID_CONNECTION
249	(0xF9)	ERR_ECB_CANNOT_BE_CANCELLED
255	(0xFF)	ERR_ECB_NOT_IN_USE

Remarks

SpxCancelEvent (NLM) cancels the action requested by a previously submitted ECB.

If this function is made to an ECB performing an **SpxSendSequencedPacket (NLM)**, **SpxEstablishConnection (NLM)**, or **SpxListenForConnection (NLM)**, the following events occur:

The pending application is aborted.

The semaphore handle is **not** posted.

The ECB is not queued off of the queue head.

The following conditions cause a cancel request to fail:

Conditions are determined to be in an unexpected state.

The ECB is being processed by the LAN board.

The ECB has been processed and is waiting for an acknowledgment.

SpxCancelEvent (NLM) can also be called as **SpxCancelPacket (NLM)** (for compatibility with the OS implementation of SPX).

SpxCheckSocket (NLM)

Checks the status of a specified socket number
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxCheckSocket (
   unsigned short   socket);
```

Parameters

socket

(IN) Specifies the socket number to be checked (high/low).

Return Values

0	(0x00)	ESUCCESS
240	(0xF0)	ERR_SOCKET_NOT_OPENED

Remarks

SpxCheckSocket (NLM) allows an application to check on the current status of an SPX socket.

SpxCloseSocket (NLM)

Closes an SPX socket
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxCloseSocket (
   unsigned short   socket);
```

Parameters

socket

(IN) Specifies the socket number of the socket to be closed (high/low).

Return Values

0	(0x00)	ESUCCESS

Remarks

SpxCloseSocket (NLM) closes the specified SPX socket and cancels events defined by the ECBs associated with the socket. SPX also returns a value of 0xFFFC in each ECB's *status* field, indicating that the event has been cancelled. SPX posts the ECBs' associated semaphore handles and then queues the ECBs on the associated queue heads for all **SpxSendSequencedPacket (NLM)**, **SpxListenForSequencedPacket (NLM)**, **SpxListenForConnection (NLM)**, and **SpxEstablishConnection (NLM)** ECBs that are pending on the socket.

Any socket number can be closed; no error is generated if the specified socket was not open.

See Also

SpxOpenSocket (NLM)

SpxEstablishConnection (NLM)

Attempts to establish a connection with a listening socket
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxEstablishConnection (
   unsigned short   socket,
   SPX_ECB         *ecb,
   unsigned char    retryCount,
   unsigned char    watchDogFlag,
   unsigned short  *connection);
```

Parameters

socket

(IN) Specifies the socket number of the socket for which to establish a connection.

ecb

(IN/OUT) Points to the SPX_ECB structure.

retryCount

(IN) Specifies a value from 0 to 255.

watchDogFlag

(IN) Specifies a flag that controls watchdog connection supervision (zero disables; nonzero enables).

connection

(OUT) Points to an unsigned integer that receives the connection ID number.

Return Values

0	(0x00)	ESUCCESS
237	(0xED)	ERR_SPX_NO_ANSWER_FROM_TARGET
239	(0xEF)	ERR_SPX_CONNECTION_TABLE_FULL
240	(0xF0)	ERR_SOCKET_NOT_OPENED
250	(0xFA)	ERR_NO_LOCAL_TARGET_IDENTIFIED

252	(0xFC)	ERR_SPX_COMMAND_CANCELLED
254	(0xFE)	ERR_SPX_MALFORMED_PACKET

Remarks

SpxEstablishConnection (NLM) passes a retry count, SPX watchdog flag, and an ECB address to SPX to of establish an SPX connection. The function then returns control to the calling application. Meanwhile, SPX attempts to establish the connection.

Before calling **SpxEstablishConnection (NLM)**, the application must do the following:

Initialize an ECB's *semHandle* field, *queueHead* field, *socketNumber* field (only necessary if the socket number is zero), *fragCount* field, and *fragList* field. The *fragCount* field should be initialized to 0x01. The *fragList* field must point to a 42-byte buffer containing the header of an SPX packet.

Initialize the SPX header's destination field. It cannot be set to -1 (broadcast).

SPX tries to establish the connection in two steps. First, SPX creates a local connection half and verifies that the local connection half is fully functional. The **SpxEstablishConnection (NLM)** function returns to the calling procedure at this point.

Meanwhile, SPX sends an Establish Connection packet through the local connection half to a specified (listening) destination socket to establish a connection. To receive the Establish Connection packet, the destination socket must pass an ECB to its SPX with **SpxListenForConnection (NLM)**.

For the first step, SPX requires the *retryCount* and *watchDogFlag* parameters be set to the appropriate values.

The *retryCount* parameter specifies how many times SPX resends unacknowledged packets before concluding that the destination node is not functioning properly. The application can set this field to 0x00, which instructs SPX to use its own internal retry count. A value of 1 to 255 indicates that SPX should resend packets the specified number of times.

The watchdog process monitors an SPX connection, ensuring that the connection is functioning properly when traffic is not passing through the connection. To enable the watchdog feature, an application should set the *watchDogFlag* parameter to ENABLE_WATCHDOG (0XFF). A value of 0x00 disables the feature.

If the watchdog process determines that an SPX connection has failed, the watchdog process aborts the connection and posts an **SpxListenForSequencedPacket (NLM)** ECB with 0xFFED (SPX

connection failed) in the *status* field. The connection ID of the failed connection is placed in the ECB's *protocolWorkspace* field.

The socket specified in the ECB's *socket* field must have been opened prior to this call using the **SpxOpenSocket (NLM)** or **IpxOpenSocket (NLM)** function. Sockets that is used by SPX connections must not be used with the **IpxSendPacket (NLM)** or **IpxReceive (NLM)** function for sending or receiving packets directly. The socket must be dedicated solely for use by SPX. If an SPX socket is closed, then all connections on that socket are terminated and pending events (sends or listens) are cancelled.

Once SPX creates and verifies the local connection half, the function returns a completion code. If the function returns anything but 0x00 (ESUCCESS), the attempt to establish a connection stops at this point.

If the function returns a completion code of 0x00 (ESUCCESS), the function also returns a connection ID. Although no connection is established yet with a destination socket, a connection occupies one entry in the node's SPX connection table. SPX then sends an SPX Establish Connection packet to the destination node.

The ECB's *status* field contains 0x00 if the SPX connection is established.

An **SpxEstablishConnection (NLM)** event must not be cancelled by calling **IpxCancelEvent (NLM)**. If an application does not want to establish a connection, the **SpxAbortConnection (NLM)** function should be called to prevent it.

See Also

SpxAbortConnection (NLM), **SpxTerminateConnection (NLM)**

SpxGetConfiguration (NLM)

Determines the maximum number of SPX connections and the number of available SPX connections

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x

Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxGetConfiguration (
   unsigned   *maxConn,
   unsigned   *availConn);
```

Parameters

maxConn

(OUT) Receives the maximum number of SPX connections supported.

availConn

(OUT) Receives the number of available SPX connections.

Return Values

0	(0x00)	ESUCCESS

Remarks

SpxGetConfiguration (NLM) returns the maximum number of connections supported by SPX and the number of SPX connections available to the application.

SpxGetConnectionStatus (NLM)

Returns the status of an SPX connection
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxGetConnectionStatus (
   unsigned short   connection,
   SPX_SESSION     *buffer);
```

Parameters

connection

(IN) Specifies the number assigned by SPX when the connection was established.

buffer

(OUT) Points to a 60-byte connection information structure.

Return Values

0	(0x00)	ESUCCESS
238	(0xEE)	ERR_SPX_INVALID_CONNECTION

Remarks

SpxGetConnectionStatus (NLM) allows an application to check on the status of an SPX connection. If the connection exists, the function returns the connection ID number and a buffer of information on the connection. If the connection does not exist, then the completion code 0xEE is returned.

The SPX_SESSION structure contains the following fields:

```
unsigned char    sStatus;
unsigned char    sFlags;
unsigned short   sSourceConnectID;      /* high/low */
unsigned short   sDestConnectID;        /* high/low */
unsigned short   sSequenceNumber;       /* high/low */
unsigned short   sAckNumber;            /* high/low */
```

```
unsigned short   sAllocNumber;          /* high/low */
unsigned short   sRemoteAckNumber;      /* high/low */
unsigned short   sRemoteAllocNumber;    /* high/low */
unsigned short   sLocalSocket;          /* low-high */
unsigned  char   sImmediateAddres[6];
unsigned long    sRemoteNet;            /* high/low */
unsigned char    sRemoteNode[6];        /* high/low */
unsigned short   sRemoteSocket;         /* high/low */
unsigned char    sRetransmitCount;
unsigned char    sRetransmitMax;
unsigned short   sRoundTripTimer;
unsigned short   sRetransmittedPackets;
unsigned short   sSuppressedPackets;
unsigned short   sLastReceiveTime;
unsigned short   sLastSendTime;
unsigned short   sRoundTripMax;
unsigned short   sWatchdogTimeout;
unsigned long    sSessionXmitQHead;
unsigned long    sSessionXmitECBp;
```

The SPX_SESSION structure is described in the "IPX and SPX Protocols" chapter of *NetWare Library Reference for C: Structures*.

See Also

SpxEstablishConnection (NLM)

SpxGetTime (NLM)

Gets a time marker from SPX
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxGetTime (
   unsigned long   *marker);
```

Parameters

marker

(OUT) Specifies the time interval marker.

Return Values

0	(0x00)	ESUCCESS

Remarks

The time interval marker is a value between 0 and 65,535 (0x0000 and 0xFFFF). The interval value represents units of milliseconds.

An application can call **SpxGetTime (NLM)** to measure the elapsed time between two events. To do so, the application makes two requests to **SpxGetTime (NLM)**, each function returning an interval marker that corresponds to the time of an event. The application then subtracts the first interval marker from the second. The difference represents the time interval between the two events (in clock ticks). This result remains accurate even if the interval marker wraps through zero between the first and second event. This timer is not intended for use with large time intervals (greater than 1 hour) due to the precision of the interval marker (16 bits unsigned).

SpxGetVersion (NLM)

Returns the major and minor version, revision, and revision date of SPX
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxGetVersion (
   unsigned char   *major,
   unsigned char   *minor,
   unsigned short  *revision,
   unsigned long   *revDate);
```

Parameters

major

(OUT) Returns major revision of the SPX.

minor

(OUT) Returns minor revision of the SPX.

revision

(OUT) Returns revision of the SPX.

revDate

(OUT) Returns revision date of the SPX.

Return Values

0	(0x00)	ESUCCESS

Remarks

SpxGetVersion (NLM) enables an application to determine the SPX version installed on the server where the application is running. Call **SpxGetVersion (NLM)** before making other SPX calls.

If the SPX interface changes, the SPX major version number or SPX minor version number changes. SPX client programs should verify that they are running on the SPX version which they were written for.

SpxGetVersion (NLM) always returns the revision as 0 and revision date as 0.

If any of the parameters is NULL, the value of that parameter is not returned.

SpxListenForConnectedPacket (NLM)

Passes an ECB to SPX so that it can receive a sequenced packet for an SPX session

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 4.x

Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxListenForConnectedPacket (
   unsigned short   socket,
   SPX_ECB         *ecb,
   unsigned short   connection);
```

Parameters

socket

(IN) Specifies the number of the open socket.

ecb

(IN/OUT) Points to the SPX_ECB structure.

connection

(IN) Specifies the SPX connection ID number assigned when the connection was established (see **SpxEstablishConnection (NLM)**).

Return Values

0	(0x00)	ESUCCESS
237	(0xED)	ERR_SPX_CONNECTION_FAILED
240	(0xF0)	ERR_SPX_SOCKET_NOT_OPENED
252	(0xFC)	ERR_SPX_COMMAND_CANCELLED
253	(0xFD)	ERR_SPX_PACKET_OVERFLOW
254	(0xFE)	ERR_SPX_MALFORMED_PACKET

Remarks

SpxListenForConnectedPacket (NLM) is identical to **SpxListenForSequencedPacket (NLM)** except that it allows you to post an ECB for a specific SPX session (specified by *connection*).

SpxListenForConnectedPacket (NLM) delivers an ECB address and the buffer space it identifies to SPX for the purpose of receiving a sequenced packet. It then returns control to the calling application. SPX allows applications to receive sequenced packets in a fully asynchronous fashion. Therefore, **SpxListenForConnectedPacket (NLM)** only dedicates the given ECB for use in receiving packets; it does not wait for an actual packet to be received.

Before calling **SpxListenForConnectedPacket (NLM)**, the application must initialize the ECB's *semHandle*, *queueHead*, *socket* (if the *socket* parameter is zero), *fragCount*, and *fragList* fields. The socket must have been previously opened by calling **SpxOpenSocket (NLM)**.

If the completion code is 0xED, the SPX watchdog process determined that a connection has failed and aborted the connection. If the code is 0xFE, a sequenced packet was received, but the available space defined by the ECB's fragment descriptor list was not large enough to contain the entire packet.

When the listen completes (either because a packet arrived or an error occurred), the watchdog process signals the ECB's semaphore handle (if nonNULL) and queues the ECB on the queue head (if nonNULL).

If a listening ECB is cancelled with the **SpxCancelEvent (NLM)** function, the ECB's *semHandle* field must be reinitialized to an appropriate value before the ECB is reused.

See Also

SpxListenForConnection (NLM), **SpxListenForSequencedPacket (NLM)**, **SpxSendSequencedPacket (NLM)**

SpxListenForConnection (NLM)

Attempts to receive an Establish Connection packet and establish an SPX connection with a remote partner

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x

Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxListenForConnection (
   unsigned short   socket,
   SPX_ECB         *ecb,
   unsigned char    retryCount,
   unsigned char    watchDogFlag,
   unsigned short  *connection);
```

Parameters

socket

(IN) Specifies the socket number.

ecb

(IN/OUT) Points to the SPX_ECB structure.

retryCount

(IN) Specifies a value from 0 to 255.

watchDogFlag

(IN) Specifies a flag that controls watchdog connection supervision (zero disables, nonzero enables).

connection

(OUT) Receives the connection ID of the established connection.

Return Values

0	(0x00)	ESUCCESS
239	(0xEF)	ERR_SPX_CONNECTION_TABLE_FULL
240	(0xF0)	ERR_SPX_SOCKET_NOT_OPENED
252	(0xFC)	ERR_SPX_COMMAND_CANCELLED

Remarks

SpxListenForConnection (NLM) passes a retry count, SPX watchdog flag, and ECB address to SPX to listen for and receive an SPX Establish Connection packet. This establishes a connection with the sending node. The function then returns control to the calling function. Meanwhile, SPX attempts to receive an SPX Establish Connection packet and establish a connection.

Before this function can be used, call **SpxOpenSocket (NLM)** or **IpxOpenSocket (NLM)** to open the socket to be used for the connection. The ECB's *status* field is set to zero (0x00) if the SPX connection is established. The *status* field is set negative if an error occurred. The *status* field is set positive while the listen for connection is pending.

SPX tries to establish a connection in two steps. First, the function listens for an SPX Establish Connection packet. Second, after an establish connection packet is received, the function attempts to create a local connection. If it is successful, a confirmation packet is sent to the source node, and both sides are ready to either send or receive data on the connection.

To complete the second step, SPX requires a retry count, an SPX watchdog flag, and an ECB address.

The *retryCount* parameter specifies how many times SPX resends unacknowledged packets before concluding that the partner node is not functioning properly. The application can set this parameter to 0x00, which instructs SPX to use its own internal retry count. A value of 1 to 255 indicates that SPX should resend packets that many of times.

The SPX watchdog process monitors SPX connections, ensuring that the connection is functioning properly when traffic is not passing through the connection. If the watchdog process determines that an SPX connection has failed, the watchdog process signals the application by recording a value of 0xFFED (Failed Connection) in the status field of any listening ECB. The watchdog process also records the failed connection ID number in the same ECB's *protocolWorkspace* field and signals the ECB's semaphore handle (if nonNULL) and queues the ECB on the queue head (if nonNULL).

Then SPX sends a confirmation packet back to the node that initiated the connection. Next, SPX records the connection ID number in the ECB's *protocolWorkspace* field (this is the same value returned in the *connection* parameter).

If the application cancels (by calling **SpxCancelEvent (NLM)**) the **SpxListenForConnection (NLM)** attempt, SPX records a value of 0xFFFC in the ECB's *status* field.

The following ECBs occupy one entry each in the node's SPX Connection Table:

ECBs attempting to establish a connection

ECBs participating in a connection

An application can call **SpxCancelEvent (NLM)** to cancel **SpxListenForConnection (NLM)**.

See Also

SpxEstablishConnection (NLM), **SpxListenForSequencedPacket (NLM)**, **SpxOpenSocket (NLM)**

SpxListenForSequencedPacket (NLM)

Passes an ECB to SPX for the purpose of receiving a sequenced packet
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxListenForSequencedPacket (
   unsigned short   socket,
   SPX_ECB         *ecb);
```

Parameters

socket

(IN) Specifies the socket number.

ecb

(IN/OUT) Points to the SPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS
237	(0xED)	ERR_SPX_CONNECTION_FAILED
240	(0xF0)	ERR_SPX_SOCKET_NOT_OPENED
252	(0xFC)	ERR_SPX_COMMAND_CANCELLED
253	(0xFD)	ERR_SPX_PACKET_OVERFLOW
254	(0xFE)	ERR_SPX_MALFORMED_PACKET

Remarks

SpxListenForSequencedPacket (NLM) delivers an ECB address and the buffer space it identifies to SPX for the purpose of receiving a sequenced packet. It then returns control to the calling application. SPX allows applications to receive sequenced packets in a fully asynchronous fashion. Therefore, **SpxListenForSequencedPacket (NLM)** only dedicates the given ECB for use in receiving packets; it does not wait for an actual packet to be received.

Before calling **SpxListenForSequencedPacket (NLM)**, the application

must initialize the ECB's *semHandle*, *queueHead*, *socket* (if the *socket* parameter is zero), *fragCount*, and *fragList* fields. The specified socket must have been previously opened with a call to **SpxOpenSocket (NLM)** .

If the completion code is 0xED, the SPX watchdog process determined that a connection has failed and aborted the connection. If the code is 0xFE, then a sequenced packet was received, but the available space defined by the ECB's fragment descriptor list was not large enough to contain the entire packet.

When the listen completes (either because a packet arrived or an error occurred), the watchdog process signals the ECB's semaphore handle (if nonNULL) and queues the ECB on the queue head (if nonNULL).

Because both system and user packets arrive via the buffers in the listen ECB pool, SPX cannot function properly unless the SPX client makes available an adequate supply of listen ECBs. SPX imposes no limits on the number of ECBs that can be used concurrently for listening on a given socket.

If a listening ECB is cancelled with the **SpxCancelEvent (NLM)** function, the ECB's *semHandle* field must be reinitialized to an appropriate value before the ECB is reused.

See Also

SpxListenForConnection (NLM), **SpxSendSequencedPacket (NLM)**

SpxOpenSocket (NLM)

Opens an SPX socket
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxOpenSocket (
   unsigned short   *socket);
```

Parameters

socket

(IN/OUT) Contains a socket number (high/low).

Return Values

0	(0x00)	ESUCCESS
241	(0xF1)	ERR_SOCKET_ALREADY_OPEN
254	(0xFE)	ERR_SOCKET_TABLE_FULL

Remarks

An application must call **SpxOpenSocket (NLM)** to open a socket before receiving a packet on the socket.

The *socket* parameter points to the number of the socket to be opened. If the socket number to be opened is set to zero, SPX opens an available socket of its choice in the range 0x4000 to 0x7FFF, returning the value of the new socket in the variable pointed to by *socketNumber*. This is known as a *dynamic* socket open.

If **SpxOpenSocket (NLM)** returns a completion code of 0x00, then the socket has been opened as expected. If the completion code is 0xFE, then the socket table is already full. If the function returns a completion code of 0xF1, then the specified socket is already open.

See Also

SpxCloseSocket (NLM)

SpxSendSequencedPacket (NLM)

Sends an SPX packet
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxSendSequencedPacket (
   unsigned short   connection,
   SPX_ECB         *ecb);
```

Parameters

connection

Specifies the number assigned by SPX when the connection was established.

ecb

Points to the SPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS
236	(0xEC)	ERR_TERMINATED_BY_REMOTE_PARTNER
237	(0xED)	ERR_SPX_CONNECTION_FAILED
238	(0xEE)	ERR_SPX_INVALID_CONNECTION
254	(0xFE)	ERR_MALFORMED_PACKET

Remarks

SpxSendSequencedPacket (NLM) passes a connection ID and an ECB address to SPX for the purpose of sending an SPX packet. It then returns control to the calling application. Meanwhile, SPX attempts to send an SPX packet.

Before calling **SpxSendSequencedPacket (NLM)**, the application must initialize the ECB's *semHandle*, *queueHead*, *fragCount*, and *fragList* fields. The fragment buffer referenced by the first *fragList* field must contain at least a 42-byte SPX packet header. The application must also set or clear the SPX packet header's endOfMessage bit in the *connectionCtl* field and must initialize the *dataStreamType* field. All other fields in the SPX packet header are initialized by SPX.

Initially, SPX sets the ECB's *status* field to a positive value indicating the ECB is sending a packet. SPX also queues the ECB/packet combination for transmission.

When SPX completes its attempt to send the packet, SPX records a completion code in the ECB's *status* field, and signals the ECB's semaphore handle (if nonNULL) and queues the ECB on the queue head (if nonNULL).

Sockets used by SPX connections cannot be used to send or receive packets directly using the IPX calls **IpxSend (NLM)** and **IpxReceive (NLM)** functions. The sockets must be dedicated solely for use by the SPX protocol.

When an application passes several ECB/packet combinations to SPX, SPX queues the packets and sends them in the order that it received them.

When an SPX socket is closed, SPX terminates all connections associated with the socket.

An application can establish a connection between two sockets residing in the same node.

An application should call **SpxCancelEvent (NLM)** to cancel an **SpxSendSequencedPacket (NLM)** event.

See Also

SpxListenForSequencedPacket (NLM)

SpxTerminateConnection (NLM)

Terminates an SPX connection
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: SPX

Syntax

```
#include <nwipxspx.h>

int SpxTerminateConnection (
   unsigned short  connection,
   IPX_ECB        *ecb);
```

Parameters

connection

(IN) Specifies the number assigned by SPX when the connection was established.

ecb

(IN/OUT) Points to the SPX_ECB structure.

Return Values

0	(0x00)	ESUCCESS
236	(0xEC)	ERR_TERMINATED_BY_REMOTE_PARTNER
237	(0xED)	ERR_SPX_TERMINATED_POORLY
238	(0xEE)	ERR_SPX_INVALID_CONNECTION
254	(0xFE)	ERR_SPX_MALFORMED_PACKET

Remarks

SpxTerminateConnection (NLM) terminates an SPX connection by passing a connection ID and an ECB address to SPX. It then returns control to the calling application. Meanwhile, SPX attempts to break the connection.

Before calling **SpxTerminateConnection (NLM)**, the application must initialize the ECB's *semHandle*, *queueHead*, *fragCount*, and *fragList* fields. The application must set the *fragCount* field to 1. The *fragList* field must point to a 42-byte buffer. At least one **SpxListenForSequencedPacket (NLM)** must be outstanding on the socket associated with the connection.

The *connection* parameter is the local number assigned to the connection upon creation by SPX. If the connection was created with **SpxEstablishConnection (NLM)** or **SpxListenForConnection (NLM)**, this is the number returned from that call.

SPX records a value of 0xFE (TERMINATE_CONNECTION) in the *datastreamType* field of the SPX header referenced in the ECB. This tells the receiver that the connection is terminated. SPX then delivers the packet to the partner node, returns the appropriate completion code in the ECB's *status* field, and signals the ECB's semaphore handle (if nonNULL) and queues the ECB on the queue head (if nonNULL).

Once SPX terminates a connection, the position that the connection ID number occupied in the connection table becomes available for use by a new connection. Applications can initiate new connections with **SpxEstablishConnection (NLM)**.

An application should call **SpxAbortConnection (NLM)** to cancel **SpxTerminateConnection (NLM)**.

See Also

SpxAbortConnection (NLM), **SpxEstablishConnection (NLM)**, **SpxListenForConnection (NLM)**

SPX for OS/2

For the 32-bit OS/2* interface to SPX, see NWSIPX.

SPX for Windows

SPXAbortConnection (Win)

Aborts an SPX connection
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: SPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL SPXAbortConnection(
   WORD   SPXConnectionID);
```

Parameters

SPXConnectionID

(IN) Indicates the connection ID number assigned by SPX when the connection was established.

Remarks

SPXAbortConnection (Win) is included to allow a connection partner to unilaterally dismantle the connection if some catastrophic condition is detected.

Under normal conditions, **SPXTerminateConnection (Win)** should be used to ensure that both connection partners break the connection in a controlled fashion.

SPXAbortConnection (Win) passes a connection ID to SPX for the purpose of unilaterally aborting an SPX connection. It then returns control to the calling application. Meanwhile, SPX aborts the connection.

SPX makes no attempt to inform the connection partner of the decision to abort the connection. The partner will discover the connection is no longer valid when it attempts to send a packet on the connection or when its watchdog checks the connection after the inactivity timer expires.

Any outstanding **SPXEstablishConnection (Win)**, **SPXTerminateConnection (Win)**, or **SPXSendSequencedPacket (Win)** commands on the connection are aborted. The completion codes in the ECBs of such commands are set to 0xED, indicating an abnormal connection termination. The ESRs of the affected ECBs are called (unless *ESRAddress* is NULL).

If any of the affected ECBs are in a state not allowing them to be canceled (the network interface card is in the middle of sending the packet, for example), they are canceled at the earliest possible time. In this case, the completion code of the ECB is still set to 0xED, and the ESR is called if it

exists.

See Also

SPXEstablishConnection (Win), **SPXTerminateConnection (Win)**

SPXEstablishConnection (Win)

Establishes a connection with a listening socket
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: SPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL SPXEstablishConnection(
   DWORD       IPXTaskID,
   BYTE        retryCount,
   BYTE        watchDog,
   WORD FAR   *SPXConnectionID,
   ECB FAR    *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

retryCount

(IN) Indicates the retry count (0-255).

watchDog

(IN) Indicates the flag controlling watchdog connection supervision (0=disable, 0!=enable).

SPXConnectionID

(OUT) Points to an unsigned integer to receive the SPX connection ID number.

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	Successful
0xEF	Connection Table Full
0xF1	IPS/SPX Not Initialized
0xF7	ECB in use by VIPX
0xFD	Malformed Packet
0xFF	Socket Not Opened

ECB Return Values

0x0000	Successful
0xED	No Answer From Target
0xEF	Connection Table Full
0xFC	Socket Closed
0xFD	Malformed Packet
0xFF	Socket Not Opened

Remarks

SPXEstablishConnection (Win) passes a task ID, retry count, *SPXwatchDog* flag, and *eventControlBlock* address to SPX for the purpose of establishing an SPX connection. It then returns control to the calling application. Meanwhile, SPX attempts to establish the connection.

Before calling **SPXEstablishConnection (Win)**, the application must perform the following:

1. Initialize ECB's *ESRAddress*, *socketNumber*, *fragmentCount*, and *fragmentDescriptor.fragmentCount* should be initialized to 0x01. *fragmentDescriptor* must point to a 42-byte buffer containing the header of an SPX packet.

2. Initialize the SPX header's destination field. None of these fields can be set to -1 (broadcast).

3. Create at least two listen ECBs, and pass them to SPX with **SPXListenForSequencedPacket (Win)**. Once SPX sends the **SPXEstablishConnection (Win)** packet mentioned above, SPX will use one of the listen ECBs to receive a confirmation packet from the destination socket. The other may be used by *WatchDog* to maintain the SPX connection once it is established.

SPX tries to establish the connection in two steps as follows:

1. SPX creates a local connection half and verifies that it is fully functional. **SPXEstablishConnection (Win)**returns to the calling procedure at this point.

2. Meanwhile, SPX sends an Establish Connection packet through the local connection half to a specified (listening) destination socket in hopes of establishing a connection. To receive the Establish Connection packet, the destination socket must pass an ECB to its SPX process with **SPXListenForConnection (Win)**.

For the first step, SPX requires *IPXtaskID*, *retryCount*, *watchDog*, and *ECBAddress* be set to the appropriate values.

IPXTaskID is a value returned from **SPXInitialize (Win)**.

retryCount specifies how many times SPX will resend unacknowledged packets before concluding the destination node is not functioning properly. The application can set this field to 0x00, which instructs SPX to use its own internal retry count. A value of 1-255 (inclusive) indicates SPX should resend packets the specified number of times.

watchDog monitors an SPX connection, ensuring the connection is functioning properly when traffic is not passing through the connection. To enable the *watchDog* feature, an application should set *watchDog* to 0xFF, ENABLE_WATCHDOG. A value of 0x00 disables *watchDog*.

If *watchDog* determines an SPX connection has failed, *watchDog* will abort the connection and post an **SPXListenSequencedPacket (Win)** ECB with *completionCode* SPX_CONNECTION_FAILED (0xED). The connection ID of the failed connection will be in the first word of ECB's *IPXWorkspace*. When ECB's *inUseFlag* is reset to zero, *completionCode* will contain 0x00 if the SPX connection was established.

The socket specified in ECB's *socketNumber* must be opened prior to calling **SPXEstablishConnection (Win)** by calling **IPXOpenSocket (Win)** . Sockets used by SPX connections must not be used with **IPXSendPacket (Win)** or **IPXListenForPacket (Win)** for sending or receiving packets directly. The socket must be dedicated solely for use by SPX. If an SPX socket is closed, all connections on that socket automatically terminate and any pending events (sends or listens) are canceled.

Once SPX creates and verifies the local connection half, **SPXEstablishConnection (Win)** returns a completion code. SPX also records this completion code in *completionCode* of the sending ECB. If **SPXEstablishConnection (Win)** returns anything but 0x00 (SUCCESSFUL), the attempt to establish a connection stops at this point.

If **SPXEstablishConnection (Win)** returns a completion code of 0x00 (SUCCESSFUL), it will also return a connection ID in *sourceConnectionID*. Although no connection is established yet with a destination socket, a connection occupies one entry in the node's SPX Connection Table SPX then sends an **SPXEstablishConnection (Win)** packet to the destination node.

An **SPXEstablishConnection (Win)** event must not be canceled by calling **IPXCancelEvent (Win)**. If an application does not want to establish a connection, **SPXAbortConnection (Win)** should be called to prevent it.

A completion code 0xED (SPX_NO_ANSWER_FROM_TARGET) indicates either a hardware failure has occurred, or the application has used **SPXAbortConnection (Win)**.

A completion code 0xEF (SPX_CONNECTION_TABLE_FULL) indicates no further connections may be initiated until an active connection is terminated. The SPX Connection Table has room for 100 entries.

A completion code of 0xFC (SPX_SOCKET_CLOSED) indicates the connection socket was closed before the command was completed, and ECB's ESR was not called.

A completion code of 0xFD (SPX_MALFORMED_PACKET) indicates the fragment count was not 1 or the buffer size was not 42.

See Also

SPXAbortConnection (Win), **SPXTerminateConnection (Win)**

SPXGetConnectionStatus (Win)

Returns the status of an SPX connection
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: SPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL SPXGetConnectionStatus(
   DWORD                  IPXTaskID,
   WORD                   SPXConnectionID,
   CONNECTION_INFO FAR   *connectionInfo);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

SPXConnectionID

(IN) Indicates the number assigned by SPX when the connection was established.

connectionInfo

(OUT) Points to a 44-byte connection information structure.

Return Values

0x0000	Connection OK
0xEE	Invalid Connection
0xF1	IPS/SPX Not Initialized

Remarks

SPXGetConnectionStatus (Win) allows an application to check on the current status of an SPX connection. If the specified connection exists, **SPXGetConnectionStatus (Win)** returns the connection ID number and a buffer of information on the connection. If the specified connection does not currently exist, the completion code (0xEE) is returned.

CONNECTION_INFO is defined in NXT.H, and contains the following:

```
typedef struct
```

```
{
   BYTE          connectionState;
   BYTE          connectionFlags;
   WORD          sourceConnectionID;/*high-low*/
   WORD          destinationConnectionID;/*high-low*/
   WORD          sequenceNumber;/*high-low*/
   WORD          acknowledgeNumber;/*high-low*/
   WORD          allocationNumber;/*high-low*/
   WORD          remoteAcknowledgeNumber;/*high-low*/
   WORD          remoteAllocationNumber;/*high-low*/
   WORD          connectionSocket;
   BYTE          immediateAddress[6];
   IPXAddress    destination;
   WORD          retransmissionCount;/*high-low*/
   WORD          estimatedRoundTripDelay;/*high-low*/
   WORD          retransmittedPackets;/*high-low*/
   WORD          suppressedPackets;/*high-low*/
```

connectionState indicates the current state of the specified connection as follows:

Hex	Name	Description
0x01	WAITING	SPX is listening on the connection, waiting to receive an SPX Establish Connection packet. See **SPXListenForConnection (Win)**.
0x02	STARTING	SPX is attempting to create a full connection with a remote workstation by sending SPX Establish Connection packets on its half of the connection. See **SPXEstablishConnection (Win)**.
0x03	ESTABLISHED	SPX has established a connection with a remote workstation, and the connection is open for two-way packet transmission.
0x04	TERMINATING	The remote SPX has terminated the connection. However, the local SPX has not yet terminated its half of the connection. See **SPXTerminateConnection (Win)**.

sourceConnectionID	Indicates the connection ID assigned by the local client's SPX package.
destinationConnectionID	Indicates the connection ID assigned by the remote client's SPX package. It is valid only if the connection state is ESTABLISHED or TERMINATING.
sequenceNumber	Indicates the next packet the local SPX will send to the remote workstation. SPX assigns a

	sequence number (0x0000 to 0xFFFF) to each packet it sends to the remote workstation. This ensures sequenced transmission at the local end and sequenced reception at the remote end. When the sequence number reaches 0xFFFF, it wraps to 0x0000. *sequenceNumber* is not valid if the connection state is WAITING.
acknowledgeNumber	Indicates the sequence number of the next packet the local SPX expects to receive from the remote SPX. When the sequence number reaches 0xFFFF, it wraps to 0x0000. *acknowledgeNumber* is not valid if the connection state is WAITING.
allocationNumber	Indicates (in conjunction with *acknowledgeNumber*) number of outstanding packet receive buffers (posted listens) available for a given SPX connection. It is used by SPX to implement flow control between communicating applications. *allocationNumber* minus *acknowledgeNumber* equals the number of posted listens outstanding on the connection socket. SPX will send packets only until the local sequence number equals *allocationNumber* of the remote partner. The *allocationNumber* increments from 0xFFFF and wraps to 0x0000. It is not valid if the connection state is WAITING. This number is based on the number of Listen ECBs outstanding.
remoteAcknowledgeNumber	Indicates the sequence number of the next packet the remote SPX expects to receive from the local SPX. When the sequence number reaches 0xFFFF, it wraps to 0x0000. *remoteAcknowledgeNumber* is not valid if the connection state is WAITING. The local SPX is allowed to send packets with sequence numbers up to and including, but not exceeding, *remoteAllocationNumber*. Meanwhile, the remote SPX increments the remote allocation number as the remoteworkstation generates listen ECBs. In this way, the remote SPX regulates the number of packets that the local SPXsends and avoids being inundated with packets it is not ready to receive. When this number reaches 0xFFFF, it wraps back to 0x0000. The number is based on the number of ECBs outstanding.
connectionSocket	Indicates the socket number the local SPX is

	using to send and receive packets.
immediateAddress	Indicates the node address of the bridge (on the local network) that routes the packets to and from the remote workstation. If the local and remote workstations reside on the same local network, the immediate address is the node address of the remote workstation. (In this case, a bridge is unnecessary.) *immediateAddress* is not valid if the connection state is WAITING.
retransmissionCount	Indicates the number of times SPX attempts to retransmit an unacknowledged packet before it determines the remote SPX has become inoperable or unreachable.
estimatedRoundTripDelay	Indicates the amount of time (in 1/18th second units) SPX should wait for an acknowledgment to arrive from the remote SPX partner. SPX includes both dynamic flow control and dynamic routing. This means the value of *estimatedRoundTripDelay* may change from time to time as SPX adjusts to observed fluctuations in packet throughput on the underlying internetwork. *estimatedRountTripDelay* is not valid if the connection state is WAITING.
retransmittedPackets	Indicates the number of times SPX had to retransmit a packet on the connection before it received an expected acknowledgment. When it reaches a value of 0xFFFF, it wraps to 0x0000. *retransmittedPackets* is valid only if the connection state is ESTABLISHED or TERMINATING.
suppressedPackets	Indicates the number of times SPX received a data packet on the connection that was not delivered to the connection client because the packet was either a duplicate of previously delivered data or was out-of-bounds for the current receive window. When it reaches a value of 0xFFFF, it wraps to 0x0000. *suppressedPackets* is valid only if the connection state is ESTABLISHED or TERMINATING.

See Also

SPXEstablishConnection (Win)

SPXInitialize (Win)

Checks to see if SPX is installed, and if not, initializes SPX
Server: 2.2, 3.11 and above, 4.0 and above
Platform: Win
Service: SPX

Syntax

```
#include <nwipxspx.h>

int FAR PASCAL SPXInitialize(
   DWORD FAR   *IPXTaskID,
   WORD         maxECBs,
   WORD         maxPacketSize,
   BYTE FAR    *majorRevisionNumber,
   BYTE FAR    *minorRevisionNumber,
   WORD FAR    *maxConnections,
   WORD FAR    *availableConnections);
```

Parameters

IPXTaskID

(IN/OUT) Inputs how resources are allocated. Outputs the assigned task ID.

maxECBs

(IN) Indicates the maximum number of outstanding ECBs that can be submitted to SPX.

maxPacketSize

(IN) Indicates the maximum size packet that can be sent by the application. The default value is 576 bytes: 30 bytes for the IPX header, and 546 bytes for the data portion of the packet.

majorRevisionNumber

(OUT) Points to a byte receiving the SPX major revision number.

minorRevisionNumber

(OUT) Points to a byte receiving the SPX minor revision number.

maxConnections

(OUT) Points to an unsigned integer receiving the maximum number of SPX connections supported.

availableConnections

(OUT) Points to an unsigned integer receiving the number of available SPX connections.

Return Values

0x0000	SPX Not Installed
0xF1	IPX/SPX Not Initialized
0xF2	No DOS Memory
0xF3	No Free ECB
0xF4	Lock Failed
0xF5	Over the Maximum Limit
0xF6	IPX/SPX Previously Initialized
0xFF	SPX Installed

Remarks

Applications must initialize SPX before calling other SPX functions.

SPXInitialize (Win) determines whether SPX is installed on the node where the application is running. If SPX is installed, **SPXInitialize (Win)** returns the major and minor revision numbers of SPX, the maximum number of connections supported by SPX, and the number of SPX connections available.

IPXTaskID

acts both as an input and as an output parameter for the initialization functions. As input, *IPXTaskID* specifies how NWIPXSPX.DLL allocates resources as follows:

0x00000000

Resources allocated directly to the calling application.

0xFFFFFFFE

Resources allocated directly to the calling application; however, multiple initializations are allowed.

0xFFFFFFFF

Resources allocated in a pool for access by multiple applications.

However, DLLs frequently manage their own resources,regardless of the number of clients they service. Forexample, a DLL may manage a pool of ECBs for multipleapplications wanting to access an SPX server.

As output, *IPXTaskID* receives the task ID assigned by the initialization function to this IPX or SPX process. The application must store this task ID for later use. Most Communication Services functions require *IPXTaskID* bereturned by the initialization function as one of their inputparameters.

For every *IPXTaskID* assigned by the initialization function, an application must provide a matching call to **IPXSPXDeinit (Win)** . **IPXSPXDeinit (Win)** releases the resourcesallocated by NWIPXSPX.DLL for each task ID.

completionCode

indicates whether SPX is installed on the node where the application is running (0x00 = not installed; 0xFF = installed).

Early versions of NetWare did not support SPX. Specifically, only the last shell (ANET3.COM, 2.01-4) of NetWare 2.0a supports the SPX protocol. All versions of NetWare 2.1 and higher support SPX.

NOTE: In NetWare 2.0a, SPX is supported for workstations only; not for servers.

majorRevisionNumber

and *minorRevisionNumber* indicate which SPX revision is installed. For example, Revision 1.0 returns a 1 in the first field and a 0 in the second.

maxConnections

Indicates the maximum number of SPX connections this particular version of SPX supports.

availableConnections

Indicates how many SPX connections are available to the application.

See Also

IPXSPXDeinit (Win)

SPXListenForConnection (Win)

Receives an Establish Connection packet and thereby establishes an SPX connection with a remote partner

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: SPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL SPXListenForConnection(
   DWORD      IPXTaskID,
   BYTE       retryCount,
   BYTE       watchDog,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

retryCount

(IN) Indicates the retry count (0-255).

watchDog

(IN) Indicates the flag controlling watchdog connection supervision (0=disable, 0!=enable).

eventControlBlock

(IN/OUT) Points to ECB.

Return Values

0x0000	Successful

ECB Return Values

0xEF	SPX Connection Table Full
0xF7	ECB in use by VIPX
0xFC	SPX Command Canceled (with **IPXCancelEvent (Win)**)
0xFF	Socket Not Opened

Remarks

SPXListenForConnection (Win) passes a task ID, *retryCount*, *SPXwatchDog*, and *eventControlBlock* address to SPX for listening and receiving an **SPXEstablishConnection (Win)** packet. When the **SPXEstablishConnection (Win)** packet is received, a connection is established with the sending node.

SPXListenForConnection (Win) then returns control to the calling function. Meanwhile, SPX attempts to receive an **SPXEstablishConnection (Win)** packet and establish a connection.

NOTE: ECB's *ESRAddress* and *socketNumber* must be initialized. The other parameters are passed by **SPXListenForConnection (Win)**.

Before **SPXListenForConnection (Win)** can be used, **IPXOpenSocket (Win)** must be called to open the socket to be used for the connection. When ECB's *inUseFlag* is reset to zero, *completionCode* contains 0x00 if the SPX connection was established.

SPX tries to establish a connection in two steps:

1. The function listens for an **SPXEstablishConnection (Win)** packet.
2. After an establish connection packet is received it attempts to create a local connection.

If successful, a confirmation packet is sent to the source node, and both sides are ready to either send or receive data on the connection.

To complete the second step, SPX requires a task ID, *retryCount*, *SPXwatchDog*, and an *eventControlBlock* address.

IPXTaskID

Indicates a value returned from **SPXInitialize (Win)**.

retryCount

specifies how many times SPX resends unacknowledged packets before concluding the partner node is not functioning properly. The application should set *retryCount* to 0x00; 0x00 instructs SPX to use its own internal retry count. A value of 1 through 255 (inclusive) indicates SPX should resend packets the specified number of times.

SPX *watchDog* monitors an SPX connection, ensuring the connection is functioning properly when traffic is not passing through the connection. If *watchDog* determines an SPX connection has failed, it signals the application by recording a value of 0xED (Failed Connection) in *completionCode* of any listening ECB. *watchDog* also records the failed connection ID number in the same ECB's *IPXWorkspace* and calls ECB's ESR.

The application must prepost at least two **SPXListenForSequencedPacket (Win)** functions (see **SPXEstablishConnection (Win)**). These are used by SPX to actually receive the establish connection packet and to acknowledge the receive. The ECB passed to this function does not need packets or fragments associated with it.

1. SPX sends a confirmation packet back to the node initiating the connection.
2. SPX records the connection ID number in the first 2 bytes of ECB's *IPXWorkspace*.
3. SPX also records the 12-byte address of the partner node in ECB's *driverWorkspace*.
4. SPX records a value of 0x00 (*ConnectionEstablished*) in ECB's *completionCode* and a value of 0x00 (AVAILABLE_FOR_USE) in ECB's *inUseFlag*.

If the application cancels (with **IPXCancelEvent (Win)**) **SPXListenForConnection (Win)** attempt at any time, SPX records a value of 0xFC (SPX_COMMAND_CANCELLED) in ECB's *completionCode*.

The following ECBs occupy one entry each in the node's SPX Connection Table:

ECBs attempting to establish a connection

ECBs participating in a connection

An application can call **IPXCancelEvent (Win)** to cancel **SPXListenForConnection (Win)**.

See Also

IPXOpenSocket (Win), **SPXEstablishConnection (Win)**, **SPXListenForSequencedPacket (Win)**

SPXListenForSequencedPacket (Win)

Delivers an ECB address and the buffer space it identifies to SPX for the purpose of receiving a sequenced packet and then returns control to the calling application

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: SPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL SPXListenForSequencedPacket(
   DWORD      IPXTaskID,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

eventControlBlock

(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xED	SPX Connection Failed
0xF7	ECB in use by VIPX
0xFC	SPX Socket Closed
0xFD	SPX Packet Overflow
0xFF	SPX Socket Not Opened

Remarks

SPX allows applications to receive sequenced packets in a fully asynchronous fashion. Therefore, **SPXListenForSequencedPacket (Win)** dedicates only the given ECB for use in receiving packets; it does not wait for an actual packet to be received.

Before calling **SPXListenForSequencedPacket (Win)**, the application must initialize ECB's *ESRAddress*, *SocketNumber*, *fragmentCount*, and

fragmentDescriptor. The socket specified by *socketNumber* must be previously opened by calling **IPXOpenSocket (Win)**.

When ECB's *inUseFlag* is reset to zero, *completionCode* contains the codes listed in the ECB Completion Code table above.

If the completion code is 0xED, SPX *watchDog* determined a connection failed and aborted the connection. If the code is 0xFD, a sequenced packet was received, but the available space defined by the ECB's fragment descriptor list was not large enough to contain the entire packet.

Because both system and user packets arrive via the buffers in the listen ECB pool, SPX cannot function properly unless the SPX client makes available an adequate supply of listen ECBs. SPX imposes no limits on the number of ECBs that can be used concurrently for listening on a given socket. If a listening ECB is canceled with **IPXCancelEvent (Win)**, ECB's *ESRAddress* must be reinitialized to an appropriate value before the ECB is reused.

See Also

SPXListenForConnection (Win), **SPXSendSequencedPacket (Win)**

SPXSendSequencedPacket (Win)

Passes a connection ID and an ECB address to SPX for the purpose of sending an SPX packet and then returns control to the calling application

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: SPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL SPXSendSequencedPacket(
   DWORD      IPXTaskID,
   WORD       SPXConnectionID,
   ECB FAR   *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

SPXConnectionID

(IN) Indicates the number assigned by SPX when the connection was established.

eventControlBlock

(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xEC	SPX Terminated Poorly
0xED	SPX Connection Terminated
0xEE	Invalid Connection
0xF7	ECB in use by VIPX
0xFC	Socket Closed
0xFD	Malformed Packet

Remarks

Before calling **SPXSendSequencedPacket (Win)**, ECB's *ESRAddress, fragmentCount*

buffer referenced by the first *fragmentDescriptor* must contain at least a 42-byte SPX packet header. The application can also initialize SPX packet header's *endOfMessage* in *connectionControl* and should initialize *dataStreamType*. All other fields in the SPX packet header are initialized by SPX.

Initially, SPX sets ECB's *inUseFlag* to a non-zero value indicating the ECB is sending a packet. SPX also queues the ECB/packet combination for transmission.

When SPX completes its attempt to send the packet, SPX records a completion code in ECB's *completionCode*, sets *inUse Flag* to 0x00 (AVAILABLE_FOR_USE), and calls the ESR defined in ECB's *ESRAddress* (if applicable).

0x00 indicates the packet was successfully sent and received in order by the connection partner. An acknowledgment from the partner was returned.

0xEC indicates the remote partner terminated the connection without acknowledging this packet. SPX cannot guarantee the remote partner received this packet before the connection was destroyed.

0xED indicates the connection ended abnormally. One of the two partners called **SPXAbortConnection (Win)** to abort the connection, or the connection partner failed to acknowledge receipt of this packet. It is also reported if the network hardware fails or if the packet cannot be delivered to the specified destination.

0xEE indicates *SPXConnectionID* does not reference an established connection.

0xFC indicates the connections socket was closed. In this case, *EventServiceRoutine* is not called.

0xFD indicates the fragment count is zero, the first fragment is less than 42 bytes long, or the entire packet is greater than 576 bytes long.

WARNING: An ECB completion error 0xFD causes the connection to be aborted.

Sockets used by SPX connections cannot be used to send or receive packets directly using the IPX calls **IPXSendPacket (Win)** and **IPXListenForPacket (Win)**. The sockets must be dedicated solely for use by the SPX protocol.

When an application passes several ECB/packet combinations to SPX, SPX queues the packets and sends them in the order it receives them.

When an application or some other agent closes an SPX socket, SPX terminates all connections associated with the socket.

An application can establish a connection between two sockets residing in the same node. An application should call **SPXAbortConnection**

(Win) (not **IPXCancelEvent (Win)**) to cancel an **SPXSendSequencedPacket (Win)** event.

See Also

SPXListenForSequencedPacket (Win)

SPXTerminateConnection (Win)

Terminates an SPX connection by passing a connection ID and an ECB address to SPX and then returns control to the calling application

Server: 2.2, 3.11 and above, 4.0 and above

Platform: Win

Service: SPX

Syntax

```
#include <nwipxspx.h>

void FAR PASCAL SPXTerminateConnection(
   DWORD       IPXTaskID,
   WORD        SPXConnectionID,
   ECB FAR    *eventControlBlock);
```

Parameters

IPXTaskID

(IN) Indicates the task ID obtained from **IPXInitialize (Win)** or **SPXInitialize (Win)**.

SPXConnectionID

(IN) Indicates the number assigned by SPX when the connection was established.

eventControlBlock

(IN/OUT) Points to ECB.

ECB Return Values

0x0000	Successful
0xEC	SPX Terminated Poorly
0xED	SPX Connection Terminated By Remote Partner
0xEE	Invalid Connection
0xFD	Malformed Packet

Remarks

Before calling **SPXTerminateConnection (Win)**, ECB's *ESRAddress*, *fragmentCount*, and *fragmentDescriptor* must be initialized. The application must set *fragmentCount* to 1. *fragmentDescriptor* must point to a 42-byte buffer.

IPXTaskID is a value returned from **SPXInitialize (Win)**.

SPXConnectionID is the local number assigned to the connection upon creation by SPX. If the connection was created with **SPXEstablishConnection (Win)**, it is the number returned from that call. If the connection was created with **SPXListenForConnection (Win)**, it the number obtained from the first 2 bytes of **SPXListenForConnection (Win)** ECB's *IPXWorkspace* after the connection was made.

SPX records a value of 0xFE (TERMINATE_CONNECTION) in *dataStreamType* of the SPX header referenced in the ECB telling the receiver the connection is terminated. SPX then delivers the packet to the partner node, returns the appropriate completion code in ECB's *completionCode*, and records 0x00 (AVAILABLE_FOR_USE) in ECB's *inUseFlag*. Finally, SPX calls the ESR referenced by ECB's *ESRAddress* (if applicable).

0xED indicates the remote connection partner failed to acknowledge the request within an appropriate amount of time. The connection is terminated on the local side, but SPX cannot guarantee the connection partner saw the **SPXTerminateConnection (Win)** request. It is also returned if the network hardware fails or the packet cannot be delivered to the specified destination.

0xFD indicates the fragment count was not 1 or the buffer size was not 42. Due to the asynchronous functionality of SPX, the termination request may not be complete when **SPXTerminateConnection (Win)** returns. Thus, the ECB is not available until *inUseFlag* is reset to zero.

Once SPX terminates a connection, the position *SPXConnectionID* occupied in the connection table becomes available for use by a new connection. Applications can initiate new connections with **SPXEstablishConnection (Win)**.

An application should call **SPXAbortConnection (Win)** (not **IPXCancelEvent (Win)**) to cancel **SPXTerminateConnection (Win)**.

See Also

SPXAbortConnection (Win), **SPXEstablishConnection (Win)**, **SPXListenForConnection (Win)**

SPX for Windows 95

For the 32-bit Windows 95* interface to SPX, see NWSIPX.

SPX for Windows NT

Communication Service Group

For the 32-bit Windows NT* interface to SPX, see NWSIPX.

Communication Service Group

IPX/SPX and TLI IPX: Structures

ECB

Contains the information required by IPX and SPX to send and receive a packet

Service: IPX/SPX and TLI IPX

Defined In: nxtw.h

Structure

```
typedef struct ECB {
   void far       *linkAddress;
   void           (far *ESRAddress) ();
   BYTE           inUseFlag;
   BYTE           completionCode;
   WORD           socketNumber;        /* high-low */
   BYTE           IPXWorkspace[4];     /* N/A */
   BYTE           driverWorkspace[12]; /* N/A */
   BYTE           immediateAddress[6]; /* high-low */
   WORD           fragmentCount;       /* low-high */
   ECBFragment    fragmentDescriptor[5];
} ECB;
```

Fields

inUseFlag

Defines availability for the ECB. If the ECB is set by IPX or SPX to a non-zero value, then the ECB is unavailable to other applications.

completionCode

Records an appropriate value in the ECB to denote whether the packet was sent successfully. If anything but a 0x00 is sent, the attempt to establish a connection ceases.

socketNumber

Identifies the sending or receiving socket with which the ECB is associated.

IPXWorkspace

Displays the connection ID number.

driverWorkSpace

Reserved for NetWare. Do not modify.

immediateAddress

Reserved for NetWare. Do not modify.

fragmentCount

Reserved for NetWare. Do not modify.

fragmentDescriptor

Communication Service Group

Reserved for NetWare. Do not modify.

IPX_ADDR

Contains an IPX address
Service: IPX/SPX and TLI IPX
Defined In: tispxipx.h

Structure

```
typedef struct ipxaddr_s {
   unsigned char   ipxa_net[4];
   unsigned char   ipxa_node[6];
   unsigned char   ipxa_socket[2];
} IPX_ADDR;
```

Fields

ipxa_net

Contains a four-character array that holds the network number.

ipxa_node

Contains a six-character array that holds the node number.

ipxa_socket

Contains a two-character array that holds the socket number.

Remarks

This structure is specific to the NetWare implementation of TLI using IPX, SPX and SPX II. No functions require this structure, but it has been provided to facilitate IPX addressing. For example, it would be difficult to deal with the 4-byte network address (*ipxa_net*) as a long variable because the Intel* CPU stores the variable in word-swapped order. The character arrays of IPX_ADDR are provided to facilitate the reordering of these numbers.

The IPX_ADDR structure can be used with any structures that are defined with the field

```
struct netbuf addr;
```

which means it can be used with the t_bind, t_call, t_uderr, and t_unitdata structures. For example, to make the t_bind *addr.buf* field point to an IPX_ADDR structure, cast the address of the IPX_ADDR structure to be a char pointer, as shown in the following examples:

```
struct t_bind T_bind;
IPX_ADDR ipx_addr;
T_bind.addr.buf=(char *)&ipx_addr;
      OR
```

```
(IPX_ADDR *)tbind.addr.buf=&ipx_addr;
```

IPX_ECB

Contains information that links IPX to your application (SPX_ECB is identical)

Service: IPX/SPX and TLI IPX

Defined In: nwipxspx.h

Structure

```
typedef struct IPX_ECBStruct {
   unsigned long           semHandleSave;       /* R */
   struct IPX_ECBStruct  **queueHead;           /* sr */
   struct IPX_ECBStruct   *next;                /* A */
   struct IPX_ECBStruct   *prev;                /* A */
   unsigned short          status;              /* q */
   unsigned long           semHandle;           /* sr (ignored for IpxS
   unsigned short          lProtID;             /* R */
   unsigned char           protID [6];          /* R */
   unsigned long           boardNumber;         /* R */
   unsigned char           immediateAddress [6];/* s (IpxSend only) */
   unsigned char           driverWS [4];        /* R */
   unsigned long           ESREBXValue;         /* R */
   unsigned short          socket;           /* sr ignored if socket pa
   unsigned short          protocolWorkspace;   /* R */
   unsigned long           dataLen;             /* q */
   unsigned long           fragCount;           /* sr */
   ECBFrag                 fragList [2];        /* sr */
} IPX_ECB;
```

Fields

In the structure comments:

R	Indicates fields that are reserved.
s	Indicates fields the application must set when using the ECB to send a packet.
r	Indicates fields the application must set when using the ECB to receive a packet.
A	Indicates fields that may be used when the ECB is not in use by IPX/SPX.
q	Indicates fields the application may read.

semHandleSave

Used internally; must not be modified.

queueHead

Set by the application either to a pointer to an ECB pointer (double pointer to an ECB) or else to NULL. The*queueHead* field is set to an ECB pointer pointer to cause IPX and SPX to queue ECBs for completed events, using the ECB pointer pointed to by the*queueHead* field as the head of the queue. The application can set this field to NULL if it does not need this feature.

next

Used for internal purposes. IPX (and SPX) maintains this field while the ECB is in use. When the ECB is not in use, the application can use this field (if necessary). Most commonly, this field is used by the application as link fields for keeping the ECB in a free list. This field is also used by IPX (and SPX) to queue ECBs off of the queue head if the *queueHead* field is nonNULL.

prev

Used for internal purposes. IPX (and SPX) maintains this field while the ECB is in use. When the ECB is not in use, the application can use this field (if necessary). Most commonly, this field is used by the application as link fields for keeping the ECB in a free list. This field is also used by IPX (and SPX) to queue ECBs off of the queue head if the *queueHead* field is nonNULL.

status

Indicates the current state of the event (for example, SPX is listening on a socket) and to indicate whether the event was completed successfully. If this field is positive, the event has not yet occurred. If this field is zero, the event completed successfully. If this field is negative, the event completed with an error. The value of status in this case indicates the type of error.

semHandle

Set by the application either to a semaphore handle returned by **OpenLocalSemaphore** or to NULL. The purpose of the semaphore handle is to allow the application to block pending the completion of one or more IPX (SPX) events. Whenever the event (associated with any of the IPX (SPX) functions that take an ECB as a parameter) occurs, and the*semHandle* field is nonNULL, IPX (SPX) calls **SignalLocalSemaphore** with the specified semaphore handle. This unblocks the application if the application was blocked on the semaphore.

> **NOTE:** The semaphore handle is undefined until the ECB completes.

IProtID

Reserved for NetWare. Do not modify.

protID

Reserved for NetWare. Do not modify.

boardNumber

Reserved for NetWare. Do not modify.

immediateAddress

Holds the address of the node to which the packet is sent or from which it arrived. This is the address of an internetwork bridge on the local network if the packet is not sent to or received from a node on the local network. **GetLocalTarget** can be used to get the information for this field. This field needs to be initialized only for **IpxSend**.

driverWS

Reserved for NetWare. Do not modify.

ESREBXValue

Reserved for NetWare. Do not modify.

socket

Identifies the sending or receiving socket with which the ECB is associated. This field can be used two ways:

The application sets this field to the desired socket number and passes zero as the socket number to those functions that take both a socket number and an ECB as parameters.

The application does not fill this field, but passes the actual socket to the above-mentioned functions.

protocolWorkspace

Reserved for NetWare. Do not modify.

dataLen

Reserved for NetWare. Do not modify.

fragCount

Indicates the number of buffers from which an outbound packet is built or into which an inbound packet is dispersed. The number of buffers (one or more) is given by *fragCount*. The application provides a list of fragment descriptors, at the end of the ECB, which contains the address and size of the buffers.

fragList

The first *fragList* must describe a buffer large enough to hold at least the packet header for the service being used. That is, for IPX packets the first *fragList* must describe a buffer of at least length 30, and for SPX the first *fragList* must describe a buffer of at least length 42. Note that the IPX_ECB (and SPX_ECB) defines two fragments.

IPX_HEADER

Service: IPX/SPX and TLI IPX
Defined In: nwipxspx.h

Structure

```
typedef struct tagIPX_HEADER
{
   unsigned short   checksum;          /* hi-lo */
   unsigned short   packetLen;         /* hi-lo */
   unsigned char    transportCtl;
   unsigned char    packetType;
   unsigned long    destNet;           /* hi-lo */
   unsigned char    destNode[6];
   unsigned short   destSocket;        /* hi-lo */
   unsigned long    sourceNet;         /* hi-lo */
   unsigned char    sourceNode[6];
   unsigned short   sourceSocket;      /* hi-lo */
} IPX_HEADER;
```

Fields

checksum

Contains a dummy checksum of the packet contents and is always set by IPX to 0xFFFF.

packetLen

Contains the length of the complete IPX packet (30 to 576 bytes) and is set by IPX.

transportCtl

Used by NetWare internetwork bridges to monitor the number of bridges or routers that a packet has crossed. The packets are discarded by the 16th bridge they encounter. IPX sets this field to zero before sending the packet.

packetType

Identifies the type of service offered or required by the packet.

Xerox* has defined the following packet types:

0	Unknown Packet Type
1	Routing Information Packet (RIP)
2	Echo Packet
3	Error Packet

Novell® has defined the following packet types:

4	Packet Exchange Packet (IPX)
5	Sequenced Packet Protocol Packet (SPX)
16-31	Experimental Protocols
17	NetWare Control Protocol™ (NCP™) Packet
20	NetBIOS Name Packet

IPX users should set *packetType* to either 0 or 4. SPX sets the packet type to 5 for packets of that protocol.

destNet

Identifies the network address of the target application. The network address is a 4-byte number assigned to each physical cabling segment. This address is determined and assigned by the network administrator.

destNode

Contains the 6-byte number that identifies the LAN board within the target network station (or node). A value of 0xFFFFFFFFFFFF is placed in the node field to indicate a broadcast.

destSocket

Contains the socket address, a 2-byte number that identifies a process within a node. This is an IPX socket and must be opened using the IPX open socket function. If the destination socket is not opened communication will not occur.

sourceNet

Identifies the network address of the source application. The network address is a 4-byte number assigned to each physical cabling segment. This address is determined and assigned by the network administrator.

sourceNode

Contains the 6-byte number that identifies the LAN board within the source network station (or node). A value of 0xFFFFFFFFFFFF is placed in the node field to indicate a broadcast.

sourceSocket

Contains the socket address, a 2-byte number that identifies a process within a node. This is an IPX socket and must be opened using the IPX open socket function.

IPX_OPTS

Used when accessing IPX through TLI
Service: IPX/SPX and TLI IPX
Defined In: ispxipx.h

Structure

```
typedef struct ipxopt_s
{
   unsigned char   ipx_type;
   unsigned char   ipx_pad1[3];
   unsigned char   ipx_hops;
   unsigned char   ipx_pad2[3];
} IPX_OPTS;
```

Fields

ipx_type

Holds the type field for the IPX header.

ipx_pad1

Contains a three-character array that pads the structure out to a double-word boundary; it should be set to 0.

ipx_hops

Contains the hop count.

ipx_pad2

Contains athree-character array that pads the structure to a double-word boundary; it should be set to 0.

Remarks

The address of the IPX_OPT structure can be used as one of the values in the t_unitdata structure, which is used as a parameter for **t_rcvudata**, **t_rcvuderr**, and **t_sndudata**. The t_uintdata structure contains a netbuf structure, named opt, whose *buf* field can be set to point to the IPX_OPTS structure.

To make the t_uintdata *opt.buf* field point to the IPX_OPTS structure, cast the address of the IPX_OPTS structure to be a char pointer, as shown in the following examples:

```
struct t_unitdata T_unitdata;
IPX_OPTS ipx_opts;
T_unitdata.opt.buf=(char *)&ipx_opts;
      OR
(IPX_OPTS *)T_unitdata.opts.buf=&ipx_opts;
```

```
(IPX_OPTS *)T_unitdata.opts.buf=&ipx_opts;
```

NOTE: The IPX_OPTS structure is used when /dev/nipx device is opened with **t_open**.

SPX_ECB

Contains information that links IPX to your application
Service: IPX/SPX and TLI IPX
Defined In: nwipxspx.h

Syntax

```
#define SPX_ECB struct IPX_ECBStruct
```

Remarks

See the structure description for IPX_ECB.

SPX_HEADER

Contains information about an IPX packet
Service: IPX/SPX and TLI IPX
Defined In: nwipxspx.h

Structure

```
typedef struct tagSPX_HEADER {
   unsigned short   checksum;          /* hi-lo */
   unsigned short   packetLen;         /* hi-lo */
   unsigned char    transportCtl;
   unsigned char    packetType;
   unsigned long    destNet;           /* hi-lo */
   unsigned char    destNode[6];
   unsigned short   destSocket;        /* hi-lo */
   unsigned long    sourceNet;         /* hi-lo */
   unsigned char    sourceNode[6];
   unsigned short   sourceSocket;      /* hi-lo */
   unsigned char    connectionCtl;
   unsigned char    dataStreamType;
   unsigned short   sourceConnectID;   /* hi-lo */
   unsigned short   destConnectID;     /* hi-lo */
   unsigned short   sequenceNumber;    /* hi-lo */
   unsigned short   ackNumber;         /* hi-lo */
   unsigned short   allocNumber;       /* hi-lo */
  SPX_HEADER;
```

Fields

checksum

Contains a dummy checksum of the packet contents and is always set by IPX to 0xFFFF.

packetLen

Contains the length of the complete IPX packet (30 to 576 bytes) and is set by IPX.

transportCtl

Used by NetWare internetwork bridges to monitor the number of bridges or routers that a packet has crossed. The packets are discarded by the 16th bridge they encounter. IPX sets this field to zero before sending the packet.

packetType

Identifies the type of service offered or required by the packet.

Xerox* has defined the following packet types:

0	Unknown Packet Type
1	Routing Information Packet (RIP)
2	Echo Packet
3	Error Packet

Novell® has defined the following packet types:

4	Packet Exchange Packet (IPX)
5	Sequenced Packet Protocol Packet (SPX)
16-31	Experimental Protocols
17	NetWare Control Protocol™ (NCP™) Packet
20	NetBIOS Name Packet

IPX users should set *packetType* to either 0 or 4. SPX sets the packet type to 5 for packets of that protocol.

destNet

Identifies the network address of the target application. The network address is a 4-byte number assigned to each physical cabling segment. This address is determined and assigned by the network administrator.

destNode

Contains the 6-byte number that identifies the LAN board within the target network station (or node). A value of 0xFFFFFFFFFFFF is placed in the node field to indicate a broadcast.

destSocket

Contains the socket address, a 2-byte number that identifies a process within a node. This is an IPX socket and must be opened using the IPX open socket function. If the destination socket is not opened communication will not occur.

sourceNet

Identifies the network address of the source application. The network address is a 4-byte number assigned to each physical cabling segment. This address is determined and assigned by the network administrator.

sourceNode

Contains the 6-byte number that identifies the LAN board within the source network station (or node). A value of 0xFFFFFFFFFFFF is placed in the node field to indicate a broadcast.

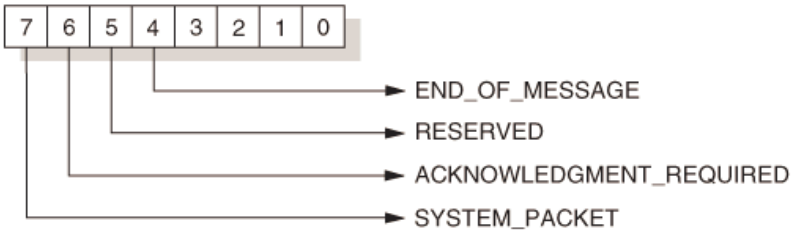
sourceSocket

Contains the socket address, a 2-byte number that identifies a process

within a node. This is an IPX socket and must be opened using the IPX open socket function.

connectionCtl

Controls the bi-directional flow of data across an SPX connection. The defined bits are as follows:



The END_OF_MESSAGE bit is the only bit the application sets or clears.

dataStreamType

Indicates the type of data included in the packet. Values of 0x00 through 0xFD are defined by the client and ignored by SPX. A value of 0xFE indicates an End-Of-Connection packet. When a client makes a call to terminate an active connection, SPX generates an End-Of-Connection packet. This packet is then delivered to the connection partner as the last message on the connection.

A value of 0xFF indicates an End-Of-Connection-Acknowledgment packet. SPX generates an End-Of-Connection-Acknowledgment packet automatically. It is marked as a system packet and not delivered to the partner clients. The values 0xFE and 0xFF are reserved for use by SPX in connection maintenance and should not be used by an application.

sourceConnectID

Specifies the connection identification number assigned to the SPX connection by the source node.

destConnectID

Specifies the connection identification number assigned to the SPX connection by the destination node.

sequenceNumber

Keeps a count of packets exchanged in one direction on the connection. Each side of the connection keeps its own count. The number wraps to 0x0000 after reaching 0xFFFF. Since SPX manages this field, client processes need not be concerned with it.

ackNumber

Indicates the next packet that an SPX connection expects to receive.

The values in this field increment from 0x0000 to 0xFFFF and then wrap to zero again.

allocNumber

Indicates (in conjunction with the *ackNumber*) the number of outstanding packet receive buffers (posted listens) available for a given SPX connection. It is used by SPX to implement flow control between communicating applications. The *allocNumber* minus the *ackNumber* equals the number of posted listens outstanding on the connection socket. SPX sends packets only until the local sequence number equals the *allocNumber* of the remote partner. The *allocNumber* increments from 0xFFFF and wraps to 0x0000.

SPX_OPTMGMT

Contains SPX timeout and retry information
Service: IPX/SPX and TLI IPX
Defined In: tispxipx.h

Structure

```
typedef struct spx_optmgmt {
   unsigned char   spxo_retry_count;
   unsigned char   spxo_watchdog_flag;
   unsigned char   spxo_min_retry_delay;
   unsigned char   spxo_pad2[2];
} SPX_OPTMGMT;
```

Fields

spxo_retry_count

Contains a retry value between 0 and 255 that specifies how many times SPX resends unacknowledged packets before concluding that the destination node is not functioning properly. The application can set this value to 0, which instructs SPX to use its own internal retry count. A nonzero value indicates that SPX should resend packets the specified number of times.

spxo_watchdog_flag

Contains a flag that controls watchdog connection supervision. The watchdog process monitors an SPX connection, ensuring that the connection is functioning properly when traffic is not passing through the connection. To enable the watchdog feature, an application should set this flag to FFh. Setting this flag to 0 disables the watchdog.

spxo_min_retry_delay

Contains the initial delay in milliseconds that SPX should wait between retries. SPX adjusts to transmission times to help in determining if a packet needs to be resent. The longest initial delay is approximately 65 seconds.

spxo_pad2

Pads the structure to 8 bytes; it should be set to 0.

Remarks

NOTE: SPX_OPTS and SPX_OPTMGMT will be changed in future releases to support other options.

SPX_OPTS

Used for accessing SPX through TLI
Service: IPX/SPX and TLI IPX
Defined In: ispxipx.h

Structure

```
typedef struct spxopt_s {
   unsigned char   spx_connectionID[2];
   unsigned char   spx_allocationNumber[2];
   unsigned char   spx_pad1[4];
} SPX_OPTS;
```

Fields

spx_connectionID

Contains the internal connection number that SPX uses to track the connection.

spx_allocationNumber

Contains the data window that SPX uses.

spx_pad1

Contains a three-character array that pads the structure to a double-word boundary; it should be set to 0.

Remarks

NOTE: The SPX_OPTS and SPX_OPTMGMT structures are used when the /dev/nspx device is opened with **t_open**.

SPX_SESSION

Defines an SPX session
Service: IPX/SPX and TLI IPX
Defined In: nwipxspx.h

Structure

```
typedef struct SPX_ConnStruct {
   unsigned char    sStatus;
   unsigned char    sFlags;
   unsigned short   sSourceConnectID;      /* hi-lo */
   unsigned short   sDestConnectID;        /* hi-lo */
   unsigned short   sSequenceNumber;       /* hi-lo */
   unsigned short   sAckNumber;            /* hi-lo */
   unsigned short   sAllocNumber;          /* hi-lo */
   unsigned short   sRemoteAckNumber;      /* hi-lo */
   unsigned short   sRemoteAllocNumber;    /* hi-lo */
   unsigned short   sLocalSocket;          /* lo-hi */
   unsigned char    sImmediateAddrees[6];
   unsigned long    sRemoteNet;            /* hi-lo */
   unsigned char    sRemoteNode[6];        /* hi-lo */
   unsigned short   sRemoteSocket;         /* hi-lo */
   unsigned char    sRetransmitCount;
   unsigned char    sRetransmitMax;
   unsigned short   sRoundTripTimer;       /* lo-hi */
   unsigned short   sRetransmittedPackets; /* lo-hi */
   unsigned short   sSuppressedPackets;    /* lo-hi */

   unsigned short   sLastReceiveTime;
   unsigned short   sLastSendTime;
   unsigned short   sRoundTripMax;
   unsigned short   sWatchdogTimeout;
   unsigned long    sSessionXmitQHead;
   unsigned long    sSessionXmitECBp;
} SPX_SESSION;
```

Fields

sStatus

ndicates the current state of the connection. Five states are defined:

0x00	ABORTED	The session has been aborted.
0x01	WAITING	SPX is listening on the connection, waiting to receive an SPX Establish Connection packet. See **SpxListenForConnection**

		(NLM).
0x02	STARTING	SPX is attempting to create a full connection with a remote workstation by sending SPX Establish Connection packets on its half of the connection. See **SPXEstablishConnection (NLM)**.
0x03	ESTABLISHED	SPX has established a connection with a remote workstation, and the connection is open for two-way packet transmission.
0x04	TERMINATING	The remote SPX has terminated the connection. However, the local SPX has not yet terminated its half of the connection. See **SPXTerminateConnection (NLM)**.

sFlags

sSourceConnectID

sDestConnectID

sSequenceNumber

sAckNumber

Indicates the sequence number of the next packet that the local SPX expects to receive from the remote SPX. When this sequence number reaches 0xFFFF, it wraps to 0x0000. This field is not valid if the connection state is WAITING.

sAllocNumber

Indicates (in conjunction with the *sAckNumber*) the number of outstanding packet receive buffers (posted listens) available for a given SPX connection. It is used by SPX to implement flow control between communicating applications. The *sAllocNumber* minus the *sAckNumber* equals the number of posted listens outstanding on the connection socket. SPX sends packets only until the local sequence number equals the *sAllocNumber* of the remote partner. The *sAllocNumber* increments from 0xFFFF and wraps to 0x0000. This field is not valid if the connection state is WAITING. This number is based on the number of Listen ECBs outstanding.

sRemoteAckNumber

Indicates the sequence number of the next packet that the remote SPX expects to receive from the local SPX. When this sequence number reaches 0xFFFF, it wraps to 0x0000. This field is not valid if the connection state is WAITING.

sRemoteAllocNumber

The local SPX is allowed to send packets with sequence numbers up to and including, but not exceeding, the *sRemoteAllocNumber*. Meanwhile, the remote SPX increments the remote allocation number as the remote workstation generates listen ECBs. In this way, the remote SPX

regulates the number of packets that the local SPX sends and avoids being inundated with packets it is not ready to receive. When this number reaches 0xFFFF, it wraps back to 0x0000. This parameter is not valid if the connection state is WAITING. This number is based on the number of ECBs outstanding.

sLocalSocket

sImmediateAddress

Contains the node address of the bridge (on the local network) that routes the packets to and from the remote workstation. If the local and remote workstations reside on the same local network, the immediate address is the node address of the remote workstation. (In this case, a bridge is unnecessary.) This field is not valid if the connection state is WAITING.

sRemoteNet

sRemoteNode

sRemoteSocket

sRetransmitCount

Indicates the number of times that SPX attempts to retransmit an unacknowledged packet before it determines that the remote SPX has become inoperable or unreachable.

sRetransmitMax

sRoundTripTimer

Indicates the time (in 1/18ths of a second) that SPX should wait for an acknowledgment to arrive from the remote SPX partner. SPX includes both dynamic flow control and dynamic routing. This means the value of *sRoundTripTimer* can change from time to time as SPX adjusts to observed fluctuations in packet throughput on the underlying internetwork. This field is not valid if the connection state is WAITING.

sRetransmittedPackets

Indicates the number of times that SPX had to retransmit a packet on this connection before it received an expected acknowledgment. When this field reaches a value of 0xFFFF, it wraps to 0x0000. This field is valid only if the connection state is ESTABLISHED or TERMINATING.

sSuppressedPackets

Indicates the number of times that SPX received a data packet on the connection that was not delivered to the connection client because the packet was either a duplicate of previously delivered data or was out-of-bounds for the current receive window. When this field reaches a value of 0xFFFF, it wraps to 0x0000. This field is valid only if the connection state is ESTABLISHED or TERMINATING.

sLastReceiveTime

sLastSendTime

Communication Service Group

sLastSendTime

sRoundTripMax

sWatchdogTimeout

sSessionXmitQHead

sSessionXmitECBp

SPX2_OPTIONS

Contains SPX II option information
Service: IPX/SPX and TLI IPX
Defined In: tispxipx.h

Structure

```
typedef struct spx2_options {
   unsigned long   versionNumber;
   unsigned long   spxIIOptionNegotiate;
   unsigned long   spxIIRetryCount;
   unsigned long   spxIIMinimumRetryDelay;
   unsigned long   spxIIMaximumRetryDelta;
   unsigned long   spxIIWatchdogTimeout;
   unsigned long   spxIIConnectTimeout;
   unsigned long   spxIILocalWindowSize;
   unsigned long   spxIIRemoteWindowSize;
   unsigned long   spxIIConnectionID;
   unsigned long   spxIIInboundPacketSize;
   unsigned long   spxIIOutboundPacketSize;
   unsigned long   spxIISessionFlags;
} SPX2_OPTIONS;
```

Fields

versionNumber

Contains the version number of the SPX2_OPTIONS structure. This number is increased each time the structure is enhanced. You must set this field to OPTIONS_VERSION.

For transparency reasons, an SPX II TLI-based application should use **t_alloc** and **t_getinfo** to allocate and determine the size of the SPX2_OPTIONS structure, rather than the `sizeof` operator.

spxIIOptionNegotiate

Specifies whether the application wants to exchange option information with the remote endpoint and whether to determine the largest packet size. This field can be set to SPX_NEGOTIATE_OPTIONS (the default) or to SPX_NO_NEGOTIATE_OPTIONS.

spxIIRetryCount

Specifies the number of times SPX II retries sending a data packet that has been involved in a transmission failure before it unilaterally aborts the connection. To use the current default value for the protocol stack, set this value to zero.

spxIIMinimumRetryDelay

Indicates whether the application wants to override the internal round-trip time calculation algorithm and wants to specify a minimum timeout value before SPX II resends a data packet.

Setting this field to zero tells the protocol stack to determine the round trip time (this is recommended). Setting the field to a nonzero value specifies a new minimum timeout value for SPX II to use. The time is specified in milliseconds.

spxIIMaximumRetryDelta

Specifies the amount of time (in milliseconds) to add to the current round-trip time to determine the maximum retry delay. A value of zero indicates to the protocol stack to use the current default.

spxIIWatchdogTimeout

Specifies the time (in milliseconds) that the watchdog algorithm allows to pass on a silent connection before it sends a watchdog query packet to determine if the other side is still available.

A value of zero indicates the protocol stack to use the current default value. A nonzero value overrides the default value.

spxIIConnectTimeout

Specifies time limit (in milliseconds) that a session setup packet must arrive in after a successful connection request has been made. A zero value specifies an infinite timeout.

spxIILocalWindowSize

Specifies the size (in packets) of the local endpoint receive window. A zero value indicates to the protocol stack to determine the receive window size.

spxIIRemoteWindowSize

Specifies the number of packets in the remote endpoint's receive window. This is an information only field and is valid only after a connection has been established.

spxIIConnectionID

Specifies the local endpoint connection ID. This is an information only field and is valid only after a connection has been established.

spxIIInboundPacketSize

Specifies the size (in bytes) of incoming packets. This value may change if SPX II renegotiates after a router change. If this occurs, there is no way for the application to receive the new packet size.

This is an information only field and is valid only after a connection has been established.

spxIIOutboundPacketSize

Specifies the size (in bytes) of outgoing packets. This value may change if SPX II renegotiates after a router change. If this occurs, there is no way for the application to receive the new packet size.

This is an information only field and is valid only after a connection has been established.

spxIISessionFlags

Contains a bit field that contains flags that are used to control the characteristics of the SPX II packets on the wire. These characteristics can include packet checksums, data signing, or data encryption. The currently defined values are listed in the following table.

Flag Definition	Meaning
SPX_SF_NONE	No special characteristics.
SPX_SF_IPX_CHECKSUM	Use IPX checksum if both ends support it.
SPX_SF_SPX2_SESSION	After the connection was established, it was a complete SPX II connection.

Remarks

NOTE: The SPX2_OPTIONS structure is used when the /dev/nspx2 device is opened with **t_open**.

Message

Message: Guides

Message: Concept Guide

Message: Concepts

Introduction to Message

Message let your application send broadcast messages to other workstations attached to a common NetWare server. The server stores the messages in buffers it maintains for this purpose and alerts each workstation that a broadcast has arrived. At the workstation, the NetWare workstation software automatically retrieves the message.

You can also send messages to the NetWare server console. The message is displayed in a single line on the console screen after the colon (:) prompt. The NetWare SEND, CASTON, and CASTOFF utilities are examples of how to apply these functions. (CASTON and CASTOFF are NetWare 2.2 and 3.11 utilities.)

Message Functions

These functions send and receive broadcast messages. They are declared in nwmsg.h. It is possible only a subset of these functions are supported by a specific client.

Function	Comment
NWBroadcastToConsole	Sends a message to the default NetWare server's system console.
NWDisableBroadcasts	Informs the server that a client doesn't want to receive messages from other clients.
NWEnableBroadcasts	Allows a client to receive broadcast messages after broadcasts have been disabled.
NWGetBroadcastMessage	Returns a message from the specified NetWare server. (Not supported on Unix.)
NWSendBroadcastMessage	Sends a message to the specified logical connections on the specified NetWare server.
NWSendConsoleBroadcast	Sends a console message to the specified logical connection. The functions requires operator rights.
NWSetBroadcastMode	Sets the message mode for the workstation on the specified NetWare server.

Message Modes

A workstation has a configurable message mode on the NetWare server to which it's connected. The message mode enables and disables the reception of messages, and lets the workstation discriminate between messages from other users and messages from the server console.

The mode also lets you control the notification feature that causes the workstation software to retrieve a message automatically. If notification is disabled, your application must poll the server for current messages. The following table shows possible values for the message mode. The default value is 0, enabling all broadcasts.

Table auto. Broadcast Message Modes

Mode	**Value**	**Comment**
Enable all	0	Receive all broadcasts.
Server only	1	Receive only server broadcasts. Discard user broadcasts.
Disable all	2	Disable all broadcasts. Discard user broadcasts. Store server broadcast but don't notify.
Disable notify	3	Store both user and server broadcasts but don't notify.

Message Size

All broadcast messages should be NULL-terminated. The message size and the number of connections to which you can send messages depends on the version of the server.

For NetWare 3.11b and below, a message can be from 1 to 58 bytes long including the null terminator and can be sent to between 1 and the maximum number of possible connections (configurable up to 256).

For NetWare 3.11c and above, the message can be 1 to 254 bytes long including a null terminator and be sent to as many as 62 connections.

When retrieving a message on networks running NetWare 3.11c and above, allocate a buffer at least 254 bytes in length.

Message: Functions

NWBroadcastToConsole

Sends a message to the default server's system console
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 2.2, 3.11, 3.12, 4.x
Platform: DOS, NLM, OS/2, Windows* 3.1, Windows NT*, Windows*95
Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWBroadcastToConsole (
   NWCONN_HANDLE   conn,
   pnstr8          message);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWBroadcastToConsole
  (conn : NWCONN_HANDLE;
   message : pnstr8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare® server connection handle.

message

(IN) Points to the NULL-terminated message to be sent.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION

Remarks

Under NETX, if an invalid connection handle is passed to *conn*, **NWBroadcastToConsole** will return 0x0000. NETX will pick a default connection handle if the connection handle cannot be resolved.

The message is displayed in a single line on the console screen after the colon (:) prompt. Messages longer than 58 bytes are truncated without notifying the broadcasting workstation. New messages overwrite previous messages at the console.

NCP Calls

0x2222 21 9 Broadcast To Console

NWDisableBroadcasts

Informs the server a client does not want to receive messages from other client

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 2.2, 3.11, 3.12, 4.x

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95

Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWDisableBroadcasts (
   NWCONN_HANDLE   conn);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWDisableBroadcasts
  (conn : NWCONN_HANDLE
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x89FF	Broadcast Disabled

Remarks

After calling **NWDisableBroadcasts**, the server does not allow other clients to log messages for forwarding to this client. If another client

clients to log messages for forwarding to this client. If another client attempts to broadcast to a client with broadcast disabled, 0x89FF (failed) is returned. **NWDisableBroadcasts** can be used by any client.

NCP Calls

0x2222 21 2 Disable Broadcasts

See Also

NWEnableBroadcasts

NWEnableBroadcasts

Allows a client to enable message reception after broadcast reception has been disabled by calling **NWDisableBroadcasts**

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 2.2, 3.11, 3.12, 4.x

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95

Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWEnableBroadcasts (
   NWCONN_HANDLE   conn);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWEnableBroadcasts
  (conn : NWCONN_HANDLE
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL

Remarks

Messages are enabled by default when the connection is first established. **NWEnableBroadcasts** can be called by any client.

NCP Calls

0x2222 21 3 Enable Broadcasts

See Also

NWDisableBroadcasts

NWGetBroadcastMessage

Returns a message from the specified server
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 2.2, 3.11, 3.12, 4.x
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95
Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetBroadcastMessage (
   NWCONN_HANDLE   conn,
   pnstr8          message);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWGetBroadcastMessage
  (conn : NWCONN_HANDLE;
   message : pnstr8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

message

(OUT) Points to the message buffer where the message will be stored.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x89FD	BAD_STATION_NUMBER

Remarks

Because some servers support 256-byte messages, the message buffer passed in should be large enough to contain messages of this size.

NCP Calls

0x2222 21 01 Get Broadcast Message
0x2222 21 11 Get Broadcast Message (new)
0x2222 23 17 Get File Server Information

NWGetBroadcastMode (obsolete 6/96)

Returns the receive message mode for the current workstation but is now obsolete. Call **NWCCGetConnInfo** instead.

Local Servers: blocking

Remote Servers: blocking

NetWare Server: PNW, 2.2, 3.11, 3.12, 4.x

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT

Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetBroadcastMode
   (NWCONN_HANDLE    conn,
    pnuint16         mode);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWGetBroadcastMode
   (conn : NWCONN_HANDLE;
    mode : pnuint16
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

mode

(OUT) Points to the broadcast mode.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL

Remarks

According to individual settings of broadcast mode, the broadcast mode can be any one of the following:

Broadcast Mode	Description
0x0000	Receive all broadcasts (default mode).
0x0001	Receive only server messages. User messages are not stored.
0x0002	Disable all broadcasts. User messages are not stored. Server messages are stored, but notification is not given to the workstation; the client can poll for server messages.
0x0003	Both user and server messages are stored, but message notification is not sent to the workstation. The client can poll for messages.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 22 Get Station's Logged Info (old)
0x2222 23 28 Get Station's Logged Info
0x2222 104 1 Ping for NDS NCP

NWSendBroadcastMessage

Allows a client to send a broadcast message to the specified logical connections on the specified NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 2.2, 3.11, 3.12, 4.x

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95

Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWSendBroadcastMessage (
   NWCONN_HANDLE   conn,
   pnstr8          message,
   nuint16         connCount,
   pnuint16        connList,
   pnuint8         resultList);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWSendBroadcastMessage
  (conn : NWCONN_HANDLE;
   message : pnstr8;
   connCount : nuint16;
   connList : pnuint16;
   resultList : pnuint8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

message

(IN) Points to the NULL-terminated message being sent.

connCount

(IN) Specifies the number of connections in the connection list.

connList

(IN) Points to an array containing the connection numbers of all

stations scheduled to receive the message.

resultList

(OUT) Points to an array containing result codes for all stations being sent the broadcast.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x89FB	INVALID_PARAMETERS
0x89FC	MESSAGE_QUEUE_FULL
0x89FD	BAD_STATION_NUMBER

Remarks

NWSendBroadcastMessage can be used by any client. The specified NetWare server attempts to store the broadcast message in the message buffer of each target connection. A result code for each target is returned by **NWSendBroadcastMessage** in *resultList*. Valid result codes are listed below:

Result Values	Description
0x0000	Successful. NetWare server stored the message in the target connection's message buffer.
0x0001	(4.0 only) Illegal station number---station number (conn) is invalid
0x0002	(4.0 only) Client not logged in---intended recipient of the message is not currently logged in to the default server, even though the client may be logged in to the network.
0x0003	(4.0 only) Client not accepting message---intended recipient of message not accepting incoming messages
0x0004	(4.0 only) Client already has message---server already has a message stored for intended recipient and cannot accept another message until that recipient clears the message from their screen
0x0150	(4.0 only) No allocation of space for the message on the server---message cannot be sent

These result codes indicate whether the NetWare server has successfully placed the message in the message buffer of the target connection. The NetWare server notifies the connection when a message arrives. However, placing the message in the message buffer and notifying the connection does not guarantee that the target station received the message. It is the target's responsibility to retrieve and display the message, depending on the broadcast mode of the connection.

A broadcast message can have the following sizes:

```
before 3.11         1-58 bytes
3.11 and later      1-250 bytes
```

A broadcast can be sent to the following maximum number of configured connections:

```
before 3.11         1-200
3.11 and later      1-62
```

Messages longer than the appropriate buffer size are truncated. The broadcasting workstation does not receive a message regarding truncated broadcasts.

NCP Calls

0x2222 21 00 Send Personal Message (3.11a or below)
0x2222 21 10 Send Personal Message (3.11b or above)
0x2222 23 17 Get File Server Information

NWSendConsoleBroadcast

Broadcasts a message to the specified logical connections on the specified NetWare server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 2.2, 3.11, 3.12, 4.x

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95

Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWSendConsoleBroadcast (
   NWCONN_HANDLE   conn,
   pnstr8          message,
   nuint16         connCount,
   pnuint16        connList);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWSendConsoleBroadcast
  (conn : NWCONN_HANDLE;
   message : pnstr8;
   connCount : nuint16;
   connList : pnuint16
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

message

(IN) Points to the NULL-terminated message being broadcast.

connCount

(IN) Specifies the number of connections in the connection list.

connList

(IN) Points to an array containing the connection number of all stations scheduled to receive the message.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x8801	INVALID_CONNECTION
0x89C6	NO_CONSOLE_PRIVILEGES
0x89FD	BAD_STATION_NUMBER

Remarks

The requesting client must have operator rights to call **NWSendConsoleBroadcast**.

Messages are not received by workstations that have disabled broadcasts or workstations that are not logged in. If *connCount* is set to 0, the message is sent to all connections.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 23 209 Send Console Broadcast
0x2222 23 253 Send Console Broadcast

See Also

NWSetBroadcastMode

NWSetBroadcastMode

Sets the message mode of the requesting workstation
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 2.2, 3.11, 3.12, 4.x
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT, Windows95
Service: Message

Syntax

```
#include <nwmsg.h>
or
#include <nwcalls.h>

NWCCODE N_API NWSetBroadcastMode (
   NWCONN_HANDLE   conn,
   nuint16         mode);
```

Pascal Syntax

```
#include <nwmsg.inc>

Function NWSetBroadcastMode
  (conn : NWCONN_HANDLE;
   mode : nuint16
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

mode

(IN) Specifies the broadcast mode to be set.

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL
0x8836	INVALID_PARAMETER

Remarks

NWSetBroadcastMode can be used by any client.

When a broadcast message is sent, the NetWare server attempts to store the message in the message buffer of each target connection. The result of this action depends on the broadcast mode of the target station. The default mode is 0x00, receive all broadcasts. However, broadcast modes can be set to any one of the following by calling **NWSetBroadcastMode**:

Broadcast Mode	Description
0x0000	Receive all broadcasts (default mode).
0x0001	Receive only server broadcasts. User messages are not stored.
0x0002	Disable all broadcasts. User messages are not stored. Server messages are stored but notification is not given to the workstation.
0x0003	Both user and server messages are stored but message notification is not sent to the workstation. The client can poll for messages.

NOTE: When using NETX, NWCONN_HANDLE is ignored and NWBROADCAST_MODE is set for all logged-in connections.

NCP Calls

0x2222 21 02 Disable Broadcasts
0x2222 21 03 Enable Broadcasts

See Also

NWSendConsoleBroadcast

NCP Extension

NCP Extension: Guides

NCP Extension: Task Guide

NCP Extension: Concept Guide

NCP Extension Identification

NCP Extension Name

NCP Extension Security

NCP Extension Server Components

NCP Extension Views

- Client View
- Provider View

NCP Extension: Tasks

Accessing an NCP Extension from the Client

The client takes the following steps when accessing an NCP Extension:

1. **Establish a connection with the server that has the NCP Extension registered.**

```
    char serverUserCombo[96];
printf("\nEnter login [fileserver/]username to access echo server:")
scanf("%s", serverUserCombo);
LoginToFileServer(serverUserCombo, 1, "");
```

   NCP Extension works through existing connections. This connection can be an attachment as well as a logged-in connection.

   Some of the functions that can be used to establish connections are:

   AttachByAddress

   AttachToFileServer

   LoginToFileServer

2. **Query to see if the desired NCP Extension has been registered.**

```
    struct queryDataStruct {
       LONG CharsEchoed;
       LONG unused[7];
    } queryData;

    LONG NCPExtID;
    NWGetNCPExtensionInfo("ECHO SERVER", &NCPExtID, NULL, NULL, NULL,
       &queryData);
    /* you should check the return value to see if the service exists
    printf("ECHO SERVER reports %ld characters echoed so far.\nKeystr
       "will be echoed on the ECHO SERVER's screen, and echoed locall
       "after they are returned from the ECHO SERVER\n(Enter ctrl-z t
       "quit)\n",queryData.CharsEchoed);
    ...
}
```

   Before you can use the NCP Extension, it must be registered. To see if the extension has been registered, call **NWGetNCPExtensionInfo**, passing in the name of the NCP Extension you are looking for, such as ECHO SERVER. This function returns SUCCESSFUL if the NCP

Extension is registered and gives you an ID to use when accessing the NCP Extension. This ID is valid until the NCP Extension is deregistered. If no extension within a given ID is found then ERR_NO_ITEMS_FOUND will be returned. It may not be necessary to query in this fashion at all if a well known ID is being used.

The example above not only checks to see if the NCP Extension is registered, but it also uses the queryData pointer to receive information about how many characters the ECHO SERVER has echoed back to clients.

3. **Access the NCP Extension.**

```
main()
{
   LONG NCPExtID;
   char chr, rtnChr;
   LONG rtnSize;

   struct queryDataStruct {
      LONG CharsEchoed;
      LONG unused[7];
   } queryData;

   ...
   /* A connection would be established before making the following
   NWGetNCPExtensionInfo("ECHO SERVER", &NCPExtID, NULL, NULL, NULL,
      &queryData);
   ...
   rtnSize = 1;
   while((chr = getch()) != CTRL_Z)   /* checking for ctrl-z to exit
   {
      NWSendNCPExtensionRequest(NCPExtID, &chr, sizeof(chr), &rtnChr
                                &rtnSize);
      putchar(rtnChr);
   }
}
```

Your client can access an NCP Extension by first setting its thread group's current connection to the server with the NCP Extension, and then calling the **NWSendNCPExtensionRequest**, with the NCP Extension ID as one of the parameters. The example sends a request buffer with one character to the NCP Extension and receives a character in its reply buffer.

Allocating Reply Buffers

Reply buffers are allocated in the following ways:

When a reply buffer manager is not used the NetWare API creates a

single reply buffer and passes its address to the NCP Extension handler.

The NCP Extension handler allocates a single reply buffer and returns a pointer to this buffer.

The NCP Extension handler allocates multiple reply buffers and returns a pointer to a NCP extension message fragment structure that has pointers to the reply buffers.

NetWare API Allocation of a Single Reply Buffer: If your NCP Extension does not use a reply buffer manager, the NetWare API allocates a reply buffer that is the size specified by the client. The NetWare API then passes a pointer to the allocated buffer as a parameter into your NCP Extension handler. Your NCP Extension handler places its reply into the buffer, and the NetWare API sends the data in the buffer to the client.

NCP Extension Handler Allocation of a Single Reply Buffer: If your NCP Extension uses a reply buffer manager, the NetWare API does not allocate a reply buffer. In this case, it is the responsibility of the NCP Extension handler to provide the buffer. The NCP Extension handler places its data in the buffer it has allocated and then returns a pointer to the buffer. The NetWare API then sends the data in that buffer to the client.

NCP Extension Handler Allocation of Multiple Reply Buffers: If you wish to reply with data from multiple buffers, you may do so using a reply buffer manager.. In this case, the NCP Extension handler sets its *replyData* parameter to point to a structure containing pointers to multiple fragments. The NCP Extension handler also sets its *replyDataLen* parameter to REPLY_BUFFER_IS_FRAGGED. The NetWare API then sends the information from the multiple buffers.

The structure that you use to point to the fragmented data must be similar to the NCPExtensionMessageFrag structure that is documented in the reference for **NWSendNCPExtensionFraggedRequest**. The difference is that the structure your NCP Extension handler returns can have more than four elements in its *fragList*. (The client is limited to four fragments, but there is no limit to the number of fragments that the NCP Extension handler can return.)

Deregistering Before Unloading

Before an NLM unloads, it must deregister all NCP Extensions that it has registered. Failure to deregister before unloading may cause the server to abend.

When an NCP Extension is deregistered, all new requests return with ERR_NO_ITEMS_FOUND, and existing requests may or may not be completed. Those that don't complete also return with the value of ERR_NO_ITEMS_FOUND.

If you need to be assured that all of your current requests are completed,

you can set a counter telling how many requests are outstanding. Outstanding requests are requests being processed by the exstension handler or the reply buffer manager. You decrement the counter when a request has completed. Before deregistering the NCP Extension, you return a failure return code immediately for all new requests and continue servicing the current requests. When the counter is set to zero, you call **NWDeRegisterNCPExtension**, then continue letting your NLM unload.

See Listing Registered NCP Extensions: Example.

Processing an NCP Extension

When you register an NCP Extension with **NWRegisterNCPExtension**, three of the parameters are functions that may be called as part of the service. These parameters are *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager*. When the client calls the NCP Extension, the order of processing is in the following manner.

1. **The client sends an NCP Extension request.**

   If the client sends an NCP Extension request with **NWSendNCPExtensionRequest** the clients request must be contained in one buffer. If the client sends a request with **NWSendNCPExtensionFraggedRequest**, the client's request can be placed in one buffer or in multiple buffers depending on the client's architecture. Either way, the data is sent across the wire as a stream of bytes.

2. **The NetWare API creates the needed buffer(s) on the server that is providing the NCP Extension.**

   If a reply buffer manager has not been specified, the NetWare API creates two buffers on the server; the size of these buffers are the size of the client's request and reply buffers respectively.

   If a buffer manager has been specified, the NetWare API creates only one buffer the size of the client's request buffer. The creation of the reply buffer is the responsibility of the function registered as *NCPExtensionHandler*.

   If the request buffer is large, it is be sent in fragments to the server. The server reassembles the fragments, making the fragmentation transparent to your program.

3. **The NetWare API calls the *NCPExtensionHandler* function.**

   This workhorse function interprets the data in the request buffer and processes the request.

   If a reply buffer manager is not being used, this function places its reply data in the buffer that the NetWare API created. If a reply buffer manager is being used, this function returns a pointer to a buffer where

it has stored the reply data.

4. **The NetWare API sends the reply information to the client.**

5. **If the reply data is large, the NetWare API sends it across the wire in fragments. The client's NWSendNCPExtensionRequest or NWSendNCPExtensionFraggedRequest function reassembles the packet, making the fragmentation transparent to the user.**

   NOTE: The data in the reply buffer is sent to the client only if the *NCPExtensionHandler* function returns SUCCESSFUL.

6. **If a buffer manager was specified, the NetWare API calls the BufferManager.**

   The buffer manager is called after that reply data has been sent to the client.

   In some ways, the reply buffer manager can be thought of as a second part of the NCP Extension handler. The reply buffer manager can free buffers and reset counters and semaphores that the NCP Extension handler has set. For example, if the NCP Extension handler has set a semaphore for a buffer, the buffer manager can signal or free the semaphore.

7. **The NetWare API frees the buffers it has created.**

   When the NCP Extension request is completed, the NetWare API frees the buffers it has allocated for the request and reply data. When new requests come in, the NetWare API allocates new buffers.

   NOTE: The *ConnectionEventHandler* is currently only called when a connection is freed, killed, logged out, or restarted.

Providing an NLM Service as an NCP Extension

You must take the following steps to provide your NLM service as an NCP Extension.

1. **Create your NCP Extension handler, connection event handler, and reply buffer manager functions, as well as a queryData update routine as needed (remember all the routines are optional).**

```
BYTE EchoServer(NCPExtensionClient *client, BYTE *requestData,
   LONG requestDataLen, BYTE *replyData, LONG *replyDataLen)
{
   int savedThreadGroupID;
   savedThreadGroupID = GetThreadGroupID();
   SetThreadGroupID(myThreadGroupID);

   /* echo character */
```

```
    putchar(*(char *)requestData);
    *replyDataLen = 1;

    /* return echoed character */
    *replyData = *requestData;
    queryData->CharsEchoed++;
    SetThreadGroupID(savedThreadGroupID);
    return 0;
}
void EchoServerConnEventHandler(LONG connection, LONG eventType)
{
    ConsolePrintf("\nECHO SERVER notified of connection %d
                  logged out or returned\n", connection);
}
/* A buffer manager is not used in this example. */
```

In the example code above, **EchoServer** is the function that serves as the NCP Extension handler, and **EchoServerConnDownHandler** is called when certain connection events occurs. This example does not use a buffer manager.

You do not need to supply all of the routines. This example has a NCP Extension handler, a connection event handler, and a queryData update routine, but it does not use a reply buffer manager.

2. **If needed, store the thread group ID so that it can be used for establishing CLIB context within your registered functions.**

```
int myThreadGroupID;
main()
{
    myThreadGroupID = GetThreadGroupID();
    SetThreadContextSpecifier(GetThreadID(), NO_CONTEXT);
    ...
}
```

The functions registered for NCP Extension run as callbacks which run as OS threads. If these threads are going to call the NetWare API functions, you should manually give them CLIB context.

For the NetWare 4.x OS, callback threads can be automatically given thread group context when they are registered. The context they are given is determined by the value of the registering thread's context specifier when the callbacks are registered. The context specifier can be set to give callbacks the thread group context of the calling thread, the thread group context of another thread group, or no context at all. When registering your callbacks for NCP Extension, it is recommended that you call **SetThreadContextSpecifier** with NO_CONTEXT as the *contextSpecifier* parameter so that the callbacks are not be automatically given context when they are registered. Then within your handler you would call getThreadGroupID() with the appropriate thread group ID. This is recommended for performance and compatability reasons.

For the NetWare 3.11 OS, threads do not have a context specifier, so you must manually set the context within each callback handler.

3. **Register your NLM service as an NCP Extension.**

```
struct queryDataStruct{
   LONG CharsEchoed;
   LONG unused[7];
} *queryData;
void main(void)
{
   ...
   NWRegisterNCPExtension("ECHO SERVER", EchoServer,\
      EchoServerConnDownHandler,NULL, 1, 0, 0, &queryData);
   queryData->CharsEchoed = 0;
   printf("Press any key to unload echo server.\n");
   getch();
   ...
}
```

An NLM can provide a service through NCP Extension by registering its service with the OS in one of the following ways:

Calling **NWRegisterNCPExtension** to register the NCP Extension by using the name of the NCP Extension. This method returns a dynamic ID that is valid until the service providing NLM is unloaded.

Calling **NWRegisterNCPExtensionByID** to register the NCP Extension using a well known ID that always associated with the NLM applications service.

After an NCP Extension has been registered, clients can access the NCP Extension. The Extension remains valid until the service-providing NLM deregisters the NCP Extension.

In this example, the NLM's service is registered with the server by calling **NWRegisterNCPExtension**.

The example above registers an extension handler with the name of "ECHO SERVER." **EchoServer** is the *NCPExtensionHandler*, **EchoServerConnEventHandler** is the *ConnectionEventHandler*, and NULL is passed in for *ReplyBufferManager*, meaning a reply buffer manager is not used. The *queryData* pointer becomes the handle to the NCP Extension.

4. **Provide the service when your NCP Extension is accessed.**

When the client requests service from your NCP Extension, the NetWare API first calls the function you registered in Step 3 as *NCPExtensionHandler*. This function is the workhorse that processes the request and fills a reply buffer that the NetWare API sends back to the client. After the buffer has been sent to the client, the NetWare API calls the function registered with *ReplyBufferManager*, if you have registered

one. Remember if programming an NLM you must establish CLIB context as needed within all handlers.

The *ConnectionEventHandler* is called whenever a connection event occurs. Currently, notification occurs when connections are freed, killed, logged out, or restarted. This information is helpful for NCP Extensions that need to know when connection events occur. (These event types are discussed in Connection Status.)

5. **Deregister the NCP Extension.**

```
struct queryDataStruct{
   LONG CharsEchoed;
   LONG unused[7];
} *queryData;

main()
{
   ...
   NWRegisterNCPExtension("ECHO SERVER", EchoServer,\
      EchoServerConnEventHandler,NULL, 1, 0, 0, &queryData);
   ...
   NWDeRegisterNCPExtension(queryData);
}
```

In most cases, you will choose to have your NLM provide its services as long as it is loaded. Before your NLM unloads, it must call **NWDeRegisterNCPExtension** to remove its NCP Extension from the list of NCP Extensions. If a NLM has more than one NCP Extension registered, it must call **NWDeRegisterNCPExtension** for each extension that it has registered.

> **NOTE:** You cannot guarantee that outstanding NCP Extension requests complete successfully after **NWDeRegisterNCPExtension** is called.

If a client makes a call to an NCP Extension after the Extension has been deregistered, the client's call fails, returning ERR_NO_ITEMS_FOUND.

Registering Multiple NCP Extensions

There might be times when your service-providing NLM offers more than one service. If your NLM is a database, you may have the following services:

Open the database

Add a record

Delete a record

Search for a record

Close the database

In the above case, you would have to make a decision: do you register five NCP Extensions to handle the requests, or do you register one NCP Extension that decodes a subfunction field within the request messages? If you choose to register five NCP Extensions, you must create five names for them. If you choose to use one NCP Extension, you only need to create one name (most NCPs operate this way).

If you choose to use a single NCP extension, your code might look like the following:

Registering a Single NCP Extension to Provide Multiple Services

```
typedef MyStruct MyStruct;
struct requestDataStruct{
   int operation;
   char data[1000];
}MyStruct;
BYTE DataBaseControl(NCPExtensionClient *client,MyStruct *requestData,
    LONG requestDataLen, BYTE replyData, LONG *replyDataLen)
{
   switch(requestData->operation)
   {
   case OPEN_DATABASE:
      OpenDatabase(requestData->data);
      break;
   case ADD_RECORD:
      AddRecord(requestData->data);
      break;
   case DELETE_RECORD:
      DELETE_RECORD(requestData->data);
      break;
   case SEARCH_FOR_RECORD:
      SearchForRecord(requestData->data);
      break;
   case CLOSE_DATABASE:
      CloseDatabase(requestData->data);
   }
}
```

Related Topics:

See Listing Registered NCP Extensions: Example.

NCP Extension: Concepts

Client-Server Applications

NCP Extension is an excellent use for client-server applications, since they allow the service-providing NLM, which is close to the resource, to do the work for the client. For example, with a database, the client could send a request to the NCP Extension to search the database for a certain record. The function registered as the NCP Extension handler would interpret the request, process the search, and return the related information to the client

Related Topics:

NCP Extension Client: Example

NCP Extension Server: Example

Connection Status

Your *connectionEventHandler* can keep track of when connections are freed, killed, logged out, or restarted. If keeping track of connection status is not important to you, you can pass NULL for the *ConnectionEventHandler* when you register the NCP Extension.

In some cases, this information is important; in other cases, it is not. A service that has a limit on the number of users would be interested in knowing when a connection was terminated, so it could allow another user to have access to the service. A service that allows unlimited access may not be concerned with who is using it.

> **CAUTION: If you are using a reply buffer manager, you should use a connection event handler. This is because the reply buffer manager is never called if the current client's connection goes down while the reply buffer is being sent to the client. Instead of calling the reply buffer manager, the OS eventually calls the connection event handler for that connection. It is then the responsibility of the connection event handler to recognize that the client has gone away and to free its resources accordingly.**

Your connection event handler is called when a connection is freed, killed, logged out, or restarted. For the NetWare 3.12 OS and above, this information in received in the *eventType* parameter. This parameter may be tested for the following values:

CONNECTION_BEING_FREED---This is returned when the client calls a

CONNECTION_BEING_FREED---This is returned when the client calls a function to return its connection, or an NLM is unloaded without returning its connection, or an attempt to create a connection fails.

CONNECTION_BEING_KILLED---This means that someone has asked to kill the connection either explicitly or via a call to bring down the server.

CONNECTION_BEING_LOGGED_OUT---The client has made a call to log out.

CONNECTION_BEING_RESTARTED---This is returned when the client is making a call to create a connection when it has not already freed the connection. This can happen when the client station locks up and is rebooted. When the client tries to log in to the server, the server sees that the client is trying to allocate a connection when it already has a connection. The server issues a notice of CONNECTION_BEING_RESTARTED, then a notice for CONNECTION_BEING_LOGGED_OUT. If the logout fails, the server issues a CONNECTION_BEING_FREED notice.

NOTE: The eventType parameter is not used for the NetWare 3.11 OS and previous versions. The prototype for the ConnectionEventHandler does not include this parameter. Do not attempt to interpret the eventType value if running on those versions of the OS.

Data Transfer

The data that is used by the client and the server is stored in buffers as it moves through the process. For example, the client can store its information in one location or in up to four locations. The data stored in multiple locations is known as fragmented data.

If the client wants to send data that is stored in one buffer, it sends the request by calling **NWSendNCPExtensionRequest**. If the client wants to send fragmented data, it sends the request by calling **NWSendNCPExtensionFraggedRequest**.

NWSendNCPExtensionFraggedRequest gathers the data from the multiple locations and sends it as a stream of bytes to the server, just as if the data had come from one location.

Once the server has received all the data, the NetWare API calls the NCP Extension handler, giving the handler the address of the request buffer where the client's request is stored. The client's request is stored in one buffer, even if the client's request has come from multiple locations on the client.

After the NCP Extension handler has serviced the request, it can return the reply from one buffer or from multiple buffers (fragmented data as with the client). In either case, the reply is sent as a stream of bytes to the client.

When the reply returns to the client, the client's code is still within the **NWSendNCPExtensionRequest** or within the **NWSendNCPExtensionFraggedRequest** function. These functions place the data in the buffer(s) specified as parameters to the functions. **NWSendNCPExtensionRequest** places the reply in one buffer. **NWSendNCPExtensionFragged Request** can place the reply in one buffer, or it can separate the reply, placing it in multiple buffers.

Introduction to NCP Extension

NCP Extension is a client-server paradigm, where the following events occur:

The client sends an NCP request to the server.

The NCP Extension on the server processes the request.

The server sends the results back to the client.

With NCP Extension, the client can be either a workstation application or an NLM acting as a client. The server is an NLM running on a NetWare server.

The fundamental NetWare® services are provided by a set of functions implemented by the NetWare Core Protocol™ (NCP™) software. Each routine is referred to as an NCP. Many of the NLM™ C Interface functions call the NCP routines.

The NetWare API allows you to register the services of an NLM as an NCP Extension, allowing you to extend the services provided by the NetWare OS while maintaining the advantages associated with NCPs. The main advantages of NCP Extension follow:

Easy to use.

Use an existing connection with a server (eliminating the need to establish a separate communications session with the server).

Allow use of arbitrary message sizes

There are two sides to NLM Extensions:

The service-providing side runs as an NLM on a server and registers its service as an NCP Extension. The NLM must be loaded on each server that provides the NCP Extension. An NLM that is loaded on one server cannot register an NCP Extension on a remote server.

The client side uses the service of the NLM by calling the NCP Extension. The client can be an NLM acting as a client, or it can be an application running on a workstation.

IPX/SPX Alternative

While NCP Extension does not replace every need for IPX and SPX, there are some cases where NCP Extension can simplify the communication between the client and the server. The advantage of an NCP Extension is it uses the existing connection of the client, freeing you from needing to set up communication sockets. You can use NCP Extension in many of the cases where you are currently using IPX or SPX.

The disadvantage is that NCP Extension takes a connection. If your application isn't already establishing a NetWare connection, and you don't want to establish one, you may choose to use IPX and SPX instead.

Another disadvantage to NCP Extension is that communication must always be initiated by the client. With IPX and SPX the client and/or the server can initiate communication.

NCP Extension Context

The *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager* functions that you provide as parameters to **NWRegisterNCPExtension** are registered as callbacks. These callbacks run as OS threads and are not able to call most of the NetWare API functions, unless they are given CLIB context. If context is not given to these callbacks and they call functions that need context, the server abends.

With the NetWare 4.x OS, threads have been given a context specifier that determines what CLIB context is given to the callbacks they register. You can determine the existing setting of the registering thread's context specifier by calling **GetThreadContextSpecifier**. Call **SetThreadContextSpecifier** to set the registering thread's context specifier to one of the following options:

NO_CONTEXT---Callbacks registered by threads with this option set are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context, the callback is not able to use NetWare API functions that require thread group level context.

USE_CURRENT_CONTEXT---Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread. This is the default setting for threads that are started with **BeginThread**, **BeginThreadGroup**, or **ScheduleWorkToDo**.

A valid thread group ID---This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

Although CLIB context can be given to these callbacks automatically (with

the NetWare 4.x OS) by setting the registering thread's context specifier to USE_CURRENT_CONTEXT, your NCP Extension processes faster if you set the context specifier to NO_CONTEXT and then manually establish the context inside your callback by calling **SetThreadGroupID**, and passing in the ID of a valid thread group. (Note: This behavior is peculiar to the NCP Extension-handling code, and does not apply to callbacks in general.)

NLM applications that run on the NetWare 3.11 OS must manually set the thread-group syntax within the callbacks, by calling **SetThreadGroupID** and passing in a valid thread group ID.

For more information on using CLIB context, seeContext Problems with OS Threads.

NCP Extension Functions

Table auto. NLM Service Provider Functions

Function	Task
NWDeRegisterNCPExtension	Remove an NCP Extension from the OS.
NWRegisterNCPExtension	Using a specific name, register an NCP Extension with the OS
NWRegisterNCPExtensionByID	Using a specific ID and name, register an NCP Extension with the OS.

Table auto. NLM Client Functions

Function	Task
NWGetNCPExtensionInfo	Return information about the NCP Extension associated with a specific name.
NWGetNCPExtensionInfoByID	Return information about the NCP Extension associated with a specific ID.
NWScanNCPExtensions	List all registered NCP Extensions.
NWSendNCPExtensionRequest	Send a request to an NCP Extension.Have this function send the data from one location and place the reply in another.
NWSendNCPExtensionFraggedRequest	Send a request to an NCP Extension.Have this function gather the data from multiple addresses and place the reply in more than one address.

NCP Extension Name

Every NCP Extension must have an identifying name. The following rules apply for naming an NCP Extension:

Case-sensitive.

Any text character string up to 32 bytes long, not counting the NULL terminator.

Unique. It should be cleared through Developer Support to guarantee uniqueness.

NOTE: Problems can occur if two service providing NLM applications use the same name for each NCP Extension. The clients accessing the extensions by calling **NWGetNCPExtensionInfo** and **NWScanNCPExtensions** would not know if the extension they see registered is the one they want. To avoid duplicate names, you should clear your NCP Extension name through Developer Support.

NCP Extension ID

An ID is also used to identify an NCP Extension. The following rules apply to an NCP Extension ID.

They are unique.

They are dynamically assigned by the OS when an NLM registers an NCP Extension with **NWRegisterNCPExtension**. These dynamic ID's are determined by the OS on a first-come, first-served basis, and they can be different each time an NCP Extension is registered.

If an NCP Extension that is using dynamic IDs is deregistered and then registered again, it has a different ID. NCP Extension IDs increase in a monotonic manner. For example, if IDs 1 through 5 are used and the NCP Extension with an ID of 3 is deregistered and then reregistered, it will have an ID greater than 5. The ID 3 is not used again until the server is brought down and restarted.

They can be "well known" IDs that NLM applications use to identify an NCP Extension when they register the Extensions with **NWRegisterNCPExtensionByID**. These IDs are the same each time an NCP Extension is registered, so they can be used to identify a specific NCP Extension.

NOTE: Well known IDs are assigned by Developer Support to guarantee uniqueness and that the IDs are in a valid range.

Dynamically assigned IDs are not assigned by Developer Support since these IDs are not attached to a specific NCP Extension.

NCP Extension Identification

After an NCP Extension is registered with the OS, it is available to service requests from clients. Before using an NCP Extension, the clients must verify that an NCP Extension is active for the service they want to use. An NCP Extension is identified by name or by ID.

NCP Extension Security

You must make sure that your NCP Extension does not violate the security of the network. Your service-providing NLM may have supervisor access to the server it is running on. If your NCP Extension handler provides a service that is sensitive to NetWare security issues (accesses requires NetWare security controls), it should take over the client's connection and make its requests using the client's connection. This helps ensure that the request is processed with the client's rights.

The following code fragment shows how to take over a client's connection:

```
// The current connection ID is assumed to be set to the local server.
oldConnection = SetCurrentConnection(clientsConnection);
// Server requests are made on behalf of the client, using his
// connection's security restrictions.
setCurrentConnection(oldConnection);
```

NCP Extension Server Components

The NCP Extension server resides on a NetWare server and consists of the following components:

NCP Extension handler (optional)

Reply buffer manager (optional)

Connection event handler (optional)

Query data buffer

The NCP Extension handler is a routine that runs on the server and is called by the NetWare API whenever the client calls **NWSendNCPExtensionRequest**, or **NWSendNCPExtensionFraggedRequest**. The NCP Extension handler interprets the message sent by the client, processes the request, and sends a

reply to the client.

A reply buffer manager is useful when the data to be transferred is already gathered or if the data should be kept in a specific memory location. The reply buffer manager is a routine that determines what to do with the reply buffer after the information in the buffer has been sent to the client. The OS calls this routine after it has sent the information in the buffer. Whether to use a reply buffer manager or not is a question of performance and function. The reply buffer manager might do things such as free the reply buffer, return it to a free list of buffers or unlock the data; the implementation is determined by the NCP Extension handler and the reply buffer manager.

The connection event handler is a routine that the server calls when any connection on the server is freed, killed, logged out, or restarted. One of the parameters to this routine is the connection that the event is happening on, and the other parameter is the event type. The connection event handler can use this information to determine if the connection belongs to a client that is being serviced by the NCP Extension handler, and if so, what action to take to clean up that connection's state.

The query data buffer is a 32-byte buffer that can be used to return information when **NWGetNCPExtensionInfo** or **NWScanNCPExtensions** is called. Calling these functions returns the contents of the update buffer to the client, which provides a one-way, passive information passing scheme.

> **NOTE:** The query data buffer becomes the sole communication mechanism if an NCP Extension handler is not registered.

NCP Extension Views

Client View

The view from the client is different than that from the NCP Extension. The client sends requests and receives replies. It does not need to know the details of how the NCP Extension works; it only needs to know the protocol for the communication between them.

The client accesses the services of an NCP Extension in the following ways:

Checks to see if the NCP Extension has been registered. A client cannot use an NCP Extension until it has been registered. The client can use **NWGetNCPExtensionInfo** or **NWScanNCPExtensions** to obtain the NCP Extension IDs of extensions that have been registered. If the NCP Extension ID is a well known ID it is not necessary to scan or get extension information because attempting to use the ID will return a failure if the extension is not registered.

Sends a request to the NCP Extension with **NWSendNCPExtensionRequest** or

NWSendNCPExtensionFraggedRequest and use the information that was returned.

Asks for the information in an NCP Extension's query data buffer by calling **NWGetNCPExtensionInfo** or **NWScanNCPExtensions** and uses the query data that is returned.

Provider View

The NCP Extension does not need to know what the client process looks like; it only needs to know the format of the request coming from the client and how to format the reply.

The NCP Extension does the following:

Registers the NCP Extension with the NCP Extension handler, the reply buffer manager, the connection event handler, and query data buffer.

When the NCP Extension handler is called, it finds the request in a buffer that the OS allocated when the request was received. The NCP Extension handler processes the request and places the reply in another buffer(s) that the OS will use when sending the reply to the requester.

If a reply buffer manager is used, it is called after the data in the reply buffer(s) has been sent to the client. When the OS calls the reply buffer manager the reply buffer address is passed to it and the reply buffer manager determines what to do with the buffer(s) where the reply is stored.

When the connection event handler is called, it determines if the event affects the NCP Extension, and takes appropriate action.

Updates the information in the query data buffer if there is a need..

Deregisters the NCP Extension handler when it no longer wants to provide the service or when the service-providing NLM is unloaded.

Reentrancy

You must make sure your NCP Extension handler is reentrant. You cannot be assured that your NCP Extension handler runs to completion before it is called again by another client. Because more than one request can be accessing the same code, you need to code with reentrancy in mind. Similar issues are of concern exist for the reply buffer manager and the connection event handler as well. For more information about reentrancy see Shared Memory.

Reply Buffer Manager

The reply buffer manager is a routine that the NetWare API calls after it has sent the NCP Extension handler's reply to the client. If you are going to use a reply buffer manager, you specify it when you register the NCP Extension with the OS.

If you specify that your NCP Extension uses a reply buffer manager, the NetWare API does not allocate reply buffers for your NCP Extension. In this case, the creation of the buffers is the responsibility of the NCP Extension handler.

The reply buffer manager does not allocate reply buffers. However, it can free the buffers that the NCP Extension handler allocates or manipulate data which controls access to those buffers.

Reasons to Use

The main reason for using a reply buffer manager is to avoid needless copying of reply data, thereby speeding up your application. It also minimizes possible failures due to Alloc Memory failures for copying data into another buffer when the data already exists in memory.

For example, if your NCP Extension handler maintains a buffer itself, it can save a copy cycle by returning a pointer to its buffer, rather than copying the buffer's contents into a buffer created by the NetWare API. If your NCP Extension is a game that maintains a screen buffer and returns the updated screen to the client after its player is moved, it would be best to send the screen data directly from the buffer it is maintained in.

Another example is with an NCP Extension that returns fragmented data. In this case the NCP Extension could have a routine that is constantly polling the server and placing information into various buffers. When the NCP Extension is called, the NCP Extension handler simply returns a structure that has fields pointing to the buffers where the information is located. This avoids copying the data from various locations and placing it in a single buffer.

Another reason for using a reply buffer manager is that, in some ways, it can be thought of as the second part of the NCP Extension handler. With the example in the previous paragraph, the NCP Extension handler could set a semaphore to stop the update routine from updating the buffers. Then, after the information in the buffers has been sent to the client, and the reply buffer manager is called, the reply buffer manager can reset the semaphore, allowing the update routine to continue with updating the buffers.

Tips for Using

A common issue when using the reply buffer management scheme presented above is that of associating the call to the reply buffer manager with the associated call to the extension handler. The parameters to each callback are helpful in this regard. Even though the reply buffer address

passed between calls is the same, this is sometimes insufficient. The connection number and task number are the same between calls, but this knowledge alone may require an additional private tracking mechanism to correctly associate the two callbacks.

To help eliminate the problem the NCPExtensionClient parameter has been constructed so that the same address is passed to both callbacks and the same contents are passed to both callbacks. In conjunction with this your extension handler can overwrite the two LONG members of the NCPExtension client structure with any values you like. These same values will then be returned to your reply buffer manager handler in the NCPExtension client parameter. This should be sufficient to allow you to accurately and efficiently coordinate the reply buffer between the extension handler and the reply buffer manager callbacks.

One other tip is that the reply buffer manager will not be called if the extension handler returns a nonzero return code. It will also not be called if no data was returned and REPLY_BUFFER_IS_FRAGGED was not used.

Communication Service Group

NCP Extension: Functions

NWDeRegisterNCPExtension

Deregisters an NCP Extension
Local Servers: nonblocking
Remote Servers: N/A
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWDeRegisterNCPExtension (
   void  *queryData);
```

Parameters

queryData

(IN) Points to the extension handle received from the **NWRegisterNCPExtension** function.

Return Values

0	0x00	SUCCESSFUL The extension was deregistered
255	0xFF	ERR_NO_ITEMS_FOUND The extension has already been deregistered

Remarks

NWDeRegisterNCPExtension is called by the service-providing NLM applications in conjunction with the **NWRegisterNCPExtension** function.

When an NCP Extension is registered with the **NWRegisterNCPExtension** function, the address of the *queryData* parameter is passed as one of the parameters. The pointer is then initialized to point to a 32-byte area of memory in which the service provider can place data. The*queryData* parameter is also used as a handle for deregistering the NCP Extension.

See Also

NWGetNCPExtensionInfo (NLM), NWRegisterNCPExtension, NWScanNCPExtensions, NWSendNCPExtensionRequest

NWFragNCPExtensionRequest

Sends and receives information from an NCP extension handle
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11 and above, 4.0 and above
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
#include <nwmisc.h>
or
#include <nwcalls.h>

NWCCODE N_API NWFragNCPExtensionRequest(
   NWCONN_HANDLE        conn,
   nuint32              NCPExtensionID,
   nuint16              reqFragCount,
   NW_FRAGMENT N_FAR   *reqFragList,
   nuint16              replyFragCount,
   NW_FRAGMENT N_FAR   *replyFragList);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWFragNCPExtensionRequest
  (conn : NWCONN_HANDLE;
   NCPExtensionID : nuint32;
   reqFragCount : nuint16;
   Var reqFragList : NW_FRAGMENT;
   replyFragCount : nuint16;
   Var replyFragList : NW_FRAGMENT
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

NCPExtensionID

(IN) Specifies the ID of the NCP extension handler to use for the request.

reqFragCount

(IN) Specifies the number of request fragments.

reqFragList

(IN) Points to the NW_FRAGMENT structure.

replyFragCount

(IN) Specifies the number of reply fragments.

replyFragList

(IN/OUT) Points to the NW_FRAGMENT structure.

Remarks

The fragment based protocol allows data up to 64K (a server imposed limitation) to be transferred to and from the NCP extension handler.

To increase packet efficiency, **NWFragNCPExtensionRequest** packs as many fragments as possible into a send buffer.

The reply data will be returned in the NW_FRAGMENT structure pointed to by the *replyFragList* parameter. The *fragSize* field of the NW_FRAGMENT structure will be updated to reflect the number of bytes copied into the buffer pointed to by the *fragAddress* field.

NCP Calls

0x2222 23 17 Get File Server Information

NWGetNCPExtensionInfo

Returns information about the specified NCP extension handler
NetWare Server: 3.11 and above, 4.0 and above
Platform: DOS, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetNCPExtensionInfo(
   NWCONN_HANDLE   conn,
   nuint32         NCPExtensionID,
   pnstr8          NCPExtensionName,
   pnuint8         majorVersion,
   pnuint8         minorVersion,
   pnuint8         revision,
   pnuint8         queryData);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWGetNCPExtensionInfo
  (conn : NWCONN_HANDLE;
   NCPExtensionID : nuint32;
   NCPExtensionName : pnstr8;
   majorVersion : pnuint8;
   minorVersion : pnuint8;
   revision : pnuint8;
   queryData : pnuint8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

NCPExtensionID

(IN) Specifies the ID of the NCP extension handler for which to get information.

NCPExtensionName

(OUT) Points to a buffer to receive NCP extension name (33 bytes, optional).

majorVersion

(OUT) Points to the major version number of the NCP extension handler (optional).

minorVersion

(OUT) Points to the minor version number of the NCP extension handler (optional).

revision

(OUT) Points to the revision number of the NCP extension handler (optional).

queryData

(OUT) Points to a 32-byte buffer of custom information that the NCP extension handler can use (optional).

Return Values

These are common return values; see Return Values for more information.

0x0000	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
0x89FE	Extension ID not found

Remarks

If an NCP extension with an ID higher than the one submitted was found, and its data was returned, **NWGetNCPExtensionInfo** returns 0x89FE.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 36 00 Scan NCP Extensions
0x2222 36 02 Scan Loaded Extensions By Name
0x2222 36 05 Get NCP Extension Info

See Also

NWNCPExtensionRequest, **NWFragNCPExtensionRequest**, **NWScanNCPExtensions**, **NWGetNCPExtensionInfoByName**, **NWGetNCPExtensionsList**, **NWGetNumberNCPExtensions**

NWGetNCPExtensionInfo (NLM)

Returns information about an NCP Extension specified by name
Local Servers: nonblocking
Remote Servers: blocking
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWGetNCPExtensionInfo (
   const char  *NCPExtensionName,
   LONG         NCPExtensionID,
   BYTE        *majorVersion,
   BYTE        *minorVersion,
   BYTE        *revision,
   void        *queryData);
```

Parameters

NCPExtensionName

(IN) Points to the name of the desired NCP Extension.

NCPExtensionID

(OUT) Specifies the ID of the desired NCP Extension (optional).

majorVersion

(OUT) Points to the major version number of the NCP Extension provider (optional).

minorVersion

(OUT) Points to the minor version number of the NCP Extension provider (optional).

revision

(OUT) Points to the revision number of the NCP Extension provider (optional).

queryData

(OUT) Points to 32 bytes of information from the NCP Extension (optional).

Return Values

0	0x00	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
255	0xFF	ERR_NO_ITEMS_FOUND The extension name was not found.
1-16		A communications error occurred. (See nwncpext.h.)

Remarks

NWGetNCPExtensionInfo (NLM), the **NWScanNCPExtensions (NLM)** function, and the **NWSendNCPExtensionRequest** function access NCP Extensions. For example, if you know the name of the NCP Extension you want to access, such as "ECHO SERVER," you can call **NWGetNCPExtensionInfo (NLM)** for the following purposes:

To see if the NCP Extension is registered.

To check the version of the NCP Extension.

To get the NCP Extension ID number used when calling the **NWSendNCPExtensionRequest** function.

To receive 32 bytes of information from the NCP Extension without calling the **NWSendNCPExtensionRequest** function.

Before a client can access an NCP Extension, **NWGetNCPExtensionInfo (NLM)** must be called to see if the Extension has been registered followed by calling the **NWScanNCPExtensions (NLM)** function to receive the Extension ID needed to call the NCP Extension. If the NCP Extension has been registered, **NWGetNCPExtensionInfo (NLM)** returns SUCCESSFUL; otherwise, it returns ERR_NO_ITEMS_FOUND. The **NWScanNCPExtensions (NLM)** function returns the same information but must be called iteratively until the NCP Extension name is found.

The *NCPExtensionName* parameter can be any character string, up to 32 bytes plus a NULL terminator. The NCP Extension names are case sensitive and must be unique for each NCP Extension. One suggestion is to name the NCP Extension the same as your NLM. To avoid naming conflicts, you should clear your NCP Extension's name through Developer Support.

You provide the *majorVersion*, *minorVersion*, and *revision* parameters when you call the **NWRegisterNCPExtension** function. If you have different versions or revisions of the NCP Extension, the client can use these parameters to verify that the extension is the correct version. If you do not want to use any of these parameters, pass NULL.

The server side and the client side of NCP Extensions should be implemented as matched sets so the client side knows what the server side is expecting and what it can return. The client side also needs to

know the name of the NCP Extension.

There are some cases where **NWGetNCPExtensionInfo (NLM)** can return all of the information your client needs, eliminating the need to call the **NWSendNCPExtensionRequest** function or to have an NCP Extension handler. This information is placed in the client's *queryData* buffer, whose address is passed as a parameter to **NWGetNCPExtensionInfo (NLM)**.

Use this method if the service-providing NLM is periodically updating its *queryData* buffer (with 32 bytes or less of information) and the buffer address was returned to the NLM when it called the **NWRegisterNCPExtension** function. If the information you want is in the NLM's *queryData* buffer, call **NWGetNCPExtensionInfo (NLM)** to copy the contents of the *queryData* buffer for the service-providing NLM into the *queryData* buffer for the client. This method is useful only if a one-way server-to-client message is sufficient.

If you are using the *queryData* buffer, pass NULL to the *queryData* parameter.

NOTE: If an NLM is unloaded, all NCP Extensions associated with it are deregistered. If the NLM is reloaded, its NCP Extensions do not have the same NCP Extension IDs, even though they have the same names.

If any of the client (NLM or workstation) NCP Extension functions return ERR_NO_ITEMS_FOUND (or ERR_NCPEXT_NO_HANDLER after previously working properly), call the **NCPGetExtensionInfo** function again. The **NCPGetExtensionInfo** function will return the new *NCPExtensionID* parameter if the NCP Extension has been deregistered and then reregistered.

See Also

NWDeRegisterNCPExtension, **NWGetNCPExtensionInfoByID**, **NWRegisterNCPExtension**, **NWScanNCPExtensions (NLM)**, **NWSendNCPExtensionRequest**

NWGetNCPExtensionInfoByID

Returns information about an NCP Extension specified by ID
Local Servers: nonblocking
Remote Servers: blocking
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWGetNCPExtensionInfoByID (
   LONG   NCPExtensionID,
   char  *NCPExtensionName,
   BYTE  *majorVersion,
   BYTE  *minorVersion,
   BYTE  *revision,
   void  *queryData);
```

Parameters

NCPExtensionID

(IN) Specifies the ID of the desired NCP Extension.

NCPExtensionName

(OUT) Points to the name of NCP Extension associated with the ID passed in the *NCPExtensionID* parameter (optional).

majorVersion

(OUT) Points to the major version number of the NCP Extension provider (optional).

minorVersion

(OUT) Points to the minor version number of the NCP Extension provider (optional).

revision

(OUT) Points to the revision number of the NCP Extension provider (optional).

queryData

(OUT) Points to 32 bytes of information from the NCP Extension (optional).

Return Values

0	0x00	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
255	0xFF	ERR_NO_ITEMS_FOUND The extension name was not found.
1-16		A communications error occurred. (See niterror.h.)

Remarks

The **NWGetNCPExtensionInfo (NLM)**, **NWGetNCPExtensionInfoByID**, **NWScanNCPExtensions (NLM)**, **NWSendNCPExtensionFraggedRequest**, and **NWSendNCPExtensionRequest** functions access NCP Extensions.

If you know the ID of the NCP Extension, you can call **NWGetNCPExtensionInfoByID** for the following purposes:

To see if the NCP Extension is registered.

To verify the name of the NCP Extension.

To check the version of the NCP Extension handler.

To receive 32 bytes of information from the NCP Extension without calling the **NWSendNCPExtensionRequest** function.

Before a client can access an NCP Extension, call either the **NWGetNCPExtensionInfo (NLM)**, **NWGetNCPExtensionInfoByID**, or **NWScanNCPExtensions (NLM)** function to see if the extension has been registered. If the NCP Extension has been registered, **NWGetNCPExtensionInfoByID** returns SUCCESSFUL; otherwise, it returns ERR_NO_ITEMS_FOUND. The **NWGetNCPExtensionInfo (NLM)** and **NWScanNCPExtensions (NLM)** functions return the same information but they use the name of the NCP Extension, rather than the ID.

The *NCPExtensionID* parameter can be a dynamic ID returned from the **NWGetNCPExtensionInfo (NLM)** or **NWScanNCPExtensions (NLM)** function, or it can be a static ID assigned by Developer Support.

If you are using a static ID, check the name pointed to by the *NCPExtensionName* parameter (on the first call) to verify that the name returned is the same as the name of your NCP Extension.

The *majorVersion*, *minorVersion*, and *revision* parameters are those you provide when you call the **NWRegisterNCPExtension** or **NWRegisterNCPExtensionByID** function. If you have different versions or revisions of the NCP Extension service providers, the client can use these parameters to verify that the service provider is the correct version. If you do not want to use any of these parameters, pass NULL.

There are some cases where **NWGetNCPExtensionInfoByID** can return all of the information your client needs, eliminating the need to call the **NWSendNCPExtensionRequest** function or to have an NCP Extension handler. This information is placed in the client's *queryData* buffer, whose address is passed as a parameter to **NWGetNCPExtensionInfoByID**.

Use this method if the service-providing NLM is periodically updating its *queryData* buffer (with 32 bytes or less of information) and whose address was returned to the NLM when it called **NWRegisterNCPExtension** or **NWRegisterNCPExtensionByID**. If the information you want is in the NLM's *queryData* buffer, you can use **NWGetNCPExtensionInfoByID** to copy the contents of the *queryData* buffer for the service-providing NLM into the *queryData* buffer for the client. This method is useful only if a one-way server-to-client message is sufficient.

If you are using the *queryData* buffer, pass NULL to the *queryData* parameter.

NOTE: If an NLM is unloaded, all NCP Extensions associated with it are deregistered. If the NLM is reloaded, and it registers its NCP Extensions by calling **NWRegisterNCPExtensionByID**, the IDs for the extensions are the same.

If the NLM is reloaded, and it registers its NCP Extensions by name by calling the **NWRegisterNCPExtension** function, the NCP Extensions do not have the same NCP Extension IDs, even though they have the same names.

If any of the client (NLM or workstation) NCP Extension functions return ERR_NO_ITEMS_FOUND (or ERR_NCPEXT_NO_HANDLER after previously working properly), call the **NWGetNCPExtensionInfo (NLM)** function. The **NWGetNCPExtensionInfo (NLM)** function will return the new *NCPExtensionID* parameter if the NCP Extension has been deregistered and then reregistered.

See Also

NWDeRegisterNCPExtension, **NWGetNCPExtensionInfo (NLM)**, **NWRegisterNCPExtension**, **NWScanNCPExtensions (NLM)**, **NWSendNCPExtensionRequest**

NWGetNCPExtensionInfoByName

Returns information for the specified NCP extension handler
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11 and above, 4.0 and above
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetNCPExtensionInfoByName(
   NWCONN_HANDLE   conn,
   pnstr8          NCPExtensionName,
   pnuint32        NCPExtensionID,
   pnuint8         majorVersion,
   pnuint8         minorVersion,
   pnuint8         revision,
   pnuint8         queryData);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWGetNCPExtensionInfoByName
  (conn : NWCONN_HANDLE;
   NCPExtensionName : pnstr8;
   NCPExtensionID : pnuint32;
   majorVersion : pnuint8;
   minorVersion : pnuint8;
   revision : pnuint8;
   queryData : pnuint8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

NCPExtensionName

(IN) Points to a buffer containing the NCP extension name (33 bytes) for which to get information (optional).

NCPExtensionID

(OUT) Points to the ID of the NCP extension handler.

majorVersion

(OUT) Points to the major version number of the NCP extension handler (optional).

minorVersion

(OUT) Points to the minor version number of the NCP extension handler (optional).

revision

(OUT) Points to the revision number of the NCP extension handler (optional).

queryData

(OUT) Points to a 32-byte buffer of custom information the NCP extension handler can optionally use (optional).

Return Values

None

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 36 00 Scan NCP Extensions
0x2222 36 02 Scan Currently Loaded NCP Extensions By Name

See Also

NWGetNCPExtensionInfo, **NWNCPExtensionRequest**, **NWFragNCPExtensionRequest**, **NWScanNCPExtensions**, **NWGetNCPExtensionsList**, **NWGetNumberNCPExtensions**

NWGetNCPExtensionsList

Returns a list of NCP extension handlers loaded on the server
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11 and above, 4.0 and above
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetNCPExtensionsList(
   NWCONN_HANDLE   conn,
   pnuint32        startNCPExtensionID,
   pnuint16        itemsInList,
   pnuint32        NCPExtensionIDList);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWGetNCPExtensionsList
  (conn : NWCONN_HANDLE;
   startNCPExtensionID : pnuint32;
   itemsInList : pnuint16;
   NCPExtensionIDList : pnuint32
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

startNCPExtensionID

(IN) Points to the next extension ID to use to obtain a list.

itemsInList

(OUT) Points to the number of NCP extension handler IDs.

NCPExtensionIDList

(OUT) Points to a buffer to receive list of NCP extension handler IDs (512 bytes or 4 times the number of NCP extension IDs, whichever is less).

Return Values

None

Remarks

If there are more than 128 extension handlers loaded, call **NWGetNCPExtensionsList** multiple times.

Set *startNCPExtensionID* to 0 for the first iteration. **NWGetNCPExtensionsList** returns the next value to use.

NCP Calls

0x2222 23 17 Get File Server Information
0x2222 36 0 Scan Loaded NCP Extensions
0x2222 36 04 Get NCP Extension Loaded List

See Also

NWGetNCPExtensionInfo, **NWNCPExtensionRequest**, **NWFragNCPExtensionRequest**, **NWScanNCPExtensions**, **NWGetNCPExtensionInfoByName**, **NWGetNumberNCPExtensions**

NWGetNumberNCPExtensions

Returns the number of NCP extension handlers loaded on the specified server

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11 and above, 4.0 and above

Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT

Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWGetNumberNCPExtensions(
   NWCONN_HANDLE    conn,
   pnuint32         numNCPExtensions);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWGetNumberNCPExtensions
  (conn : NWCONN_HANDLE;
   numNCPExtensions : pnuint32
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

numNCPExtensions

(OUT) Points to the number of NCP extension handlers installed on the server.

Return Values

None

NCP Calls

0x2222 23 17 Get Server Info
0x2222 36 0 Scan Loaded NCP Extensions
0x2222 36 3 Get Number Of Loaded NCP Extensions

0x2222 36 3 Get Number Of Loaded NCP Extensions

See Also

NWGetNCPExtensionInfo, **NWNCPExtensionRequest**, **NWFragNCPExtensionRequest**, **NWScanNCPExtensions**, **NWGetNCPExtensionInfoByName**, **NWGetNCPExtensionsList**

NWNCPExtensionRequest

Sends and receives small data from an NCP extension handler
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11 and above, 4.0 and above
Platform: DOS, NLM, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWNCPExtensionRequest(
   NWCONN_HANDLE   conn,
   nuint32         NCPExtensionID,
   nptr            requestData,
   nuint16         requestDataLen,
   nptr            replyData,
   pnuint16        replyDataLen);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWNCPExtensionRequest
  (conn : NWCONN_HANDLE;
   NCPExtensionID : nuint32;
   requestData : nptr;
   requestDataLen : nuint16;
   replyData : nptr;
   replyDataLen : pnuint16
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

NCPExtensionID

(IN) Specifies the ID of the NCP extension handler to use for the request.

requestData

(IN) Points to a buffer containing request data.

requestDataLen

(IN) Specifies the length of request data.

replyData

(OUT) Points to a buffer to receive reply data (can be the same buffer as request data; optional if no reply data is expected).

replyDataLen

(IN/OUT) Points to amount of data expected and how much data was returned (optional if no reply data is expected).

Return Values

None

Remarks

NWNCPExtensionRequest should be used only if the send size is 511 bytes or less, and the receive size is 100 bytes or less. On 4.0, **NWNCPExtensionRequest** should be used only if the receive size is 523 bytes or less.

If either of these limits is exceeded, **NWNCPExtensionRequest** will send the packets via **NWFragNCPExtensionRequest**.

NCP Calls

0x2222 23 17 Get File Server Information

See Also

NWGetNCPExtensionInfo, **NWFragNCPExtensionRequest**, **NWScanNCPExtensions**, **NWGetNCPExtensionInfoByName**, **NWGetNCPExtensionsList**, **NWGetNumberNCPExtensions**

NWNCPSend

Sends an NCP request to a currently connected server
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <nwconn.h>

int NWNCPSend (
   BYTE   functionCode,
   char  *sendPacket,
   WORD   sendLen,
   char  *replyBuf,
   WORD   replyLen);
```

Parameters

functionCode

(IN) Specifies the NCP function code.

sendPacket

(IN) Points to the input buffer for the NCP.

sendLen

(IN) Specifies the length of the *sendPacket* parameter.

replyBuf

(IN/OUT) Points to the reply buffer for the NCP.

replyLen

(IN/OUT) Specifies the length of the *replyBuf* parameter.

Return Values

ESUCCESS or NetWare errors.

Remarks

An NCP request consists of function code and a request buffer that contains input information needed to process the request.

NWRegisterNCPExtension

Registers a service to be provided as an NCP extension
Local Servers: blocking
Remote Servers: N/A
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>
int NWRegisterNCPExtension (
   const char    *NCPExtensionName,
   BYTE         (*NCPExtensionHandler)(
      NCPExtensionClient  *client,
      void                *requestData,
      LONG                 requestDataLen,
      void                *replyData,
      LONG                *replyDataLen),
   void         (*ConnectionEventHandler)(
      LONG   connection,
      LONG   eventType)
   void         (*ReplyBufferManager)(
      NCPExtensionClient  *client,
      void                *replyBuffer),
   BYTE          majorVersion,
   BYTE          minorVersion,
   BYTE          revision,
   void        **queryData);
```

Parameters

NCPExtensionName

(IN) Points to the name of an NCP Extension.

NCPExtensionHandler

(IN) Points to the function to be called when the NCP Extension calls the **NWSendNCPExtensionRequest** function (optional).

ConnectionEventHandler

(IN) Points to the function to be called and action to follow when a connection is freed, killed, logged out, or restarted (optional).

ReplyBufferManager

(IN) Points to a buffer manager function used to reply to NCP Extension requests (optional).

majorVersion

(IN) Specifies the major version number of the service provider.

minorVersion

(IN) Specifies the minor version number of the service provider.

revision

(IN) Specifies the revision number of the service provider.

queryData

(OUT) Points to a 32-byte area that NetWare has allocated.

Return Values

0	0x00	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
5	0x05	ENOMEM Not enough memory was available on the server to register the service.
166	0xA6	ERR_ALREADY_IN_USE The NCP Extension name is already registered. Your service is not registered.
255	0xFF	ERR_BAD_PARAMETER The *NCPExtensionName* parameter is longer than the 32-byte limit.

Remarks

NWRegisterNCPExtension is called by the service-providing NLM applications in conjunction with the**NWDeRegisterNCPExtension** function.

NCP extension names are case sensitive and must be unique. They have a maximum length of 32 bytes plus a NULL terminator.

The *queryData* parameter can be used by the service provider to return up to 32 bytes of information to the client and is aligned on a DWORD (32-bit) boundary. This information can then be retrieved by calling the **NWGetNCPExtensionInfo (NLM)** or **NWScanNCPExtensions (NLM)** function. The *queryData* parameter is also used by the registering NLM as the NCP extension handle when the **NWDeRegisterNCPExtension** function is called.

NOTE: The *NCPExtensionHandler* parameter returns a BYTE representing the value returned when the **NWSendNCPExtensionRequest** function is called. The extension handler can return any value other than those used by the lower-level NCP-transport services (see niterror.h). However, information is placed into the *replyData* parameter after the *NCPExtensionHandler* parameter returns SUCCESSFUL.

Other status information can be returned to the client with the extension

handler. However, do not return any values (other than SUCCESSFUL) that **NWRegisterNCPExtension** can return. Otherwise, future versions of the OS might return values you have defined and confuse their meaning. If the extension handler always returns SUCCESSFUL and then uses a "status" field in the *replyData* parameter to return status information, the meaning of each return value will be clear.

If you can provide all needed information by updating the 32-byte *queryData* buffer, pass NULL to the *NCPExtensionHandler* parameter. Then call either the **NWGetNCPExtensionInfo (NLM)** or **NWScanNCPExtensions (NLM)** function to obtain information in the *queryData* buffer. This is a passive method of passing information. The NCP extension will not be notified that the *queryData* parameter was accessed.

NOTE: The *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager* parameters are function callbacks that run as OS threads. They need to have CLIB context if they are going to call NetWare functions that need context.

The function pointed to by the *NCPExtensionHandler* parameter has the following parameters:

client

(IN) Points to the NCPExtensionClient structure containing the connection and task of the calling client (also used by the *ReplyBufferManager* parameter to associate the request with the reply notification it receives).

requestData

(IN) Points to a buffer, which might be DWORD aligned, to hold the request information.

requestDataLen

(IN) Specifies the size (in bytes) of the data in the *requestData* parameter.

replyData

(OUT) Points to a buffer to store the response data from the service routine if the *ReplyBufferManager* parameter is NULL. Otherwise, points to the address of a valid buffer, which might be DWORD aligned, that the NCP extension handler created.

replyDataLen

(IN/OUT) Inputs the maximum size (in bytes) of information that can be stored in the reply buffer. Outputs the actual number of bytes that the *NCPExtensionHandler* parameter stored in the reply buffer.

The function pointed to by the *ConnectionEventHandler* parameter has the following parameters:

connection

(IN) Specifies the connection number for any connection (NCP extension clients and others) that was logged out or cleared (optional).

eventType

(IN) Specifies the type of event that is being reported for NetWare 3.12 and higher (optional):

CONNECTION_BEING_FREED
CONNECTION_BEING_KILLED
CONNECTION_BEING_LOGGED_OUT
CONNECTION_BEING_FREED

You must decide if it is important for your service to be aware of when clients (particularly the NCP extension clients) log out or terminate a connection.

The *ConnectionEventHandler* parameter does not return a value.

The function pointed to by the *ReplyBufferManager* parameter has the following parameters:

client

(IN) Points to the NCPExtensionClient structure containing the connection and task of the calling client.

replyBuffer

(IN) Points to a buffer whose information has been returned to the client (optional).

See Also

GetThreadContextSpecifier, **NWDeRegisterNCPExtension**, **NWGetNCPExtensionInfo (NLM)**, **NWScanNCPExtensions (NLM)**, **NWSendNCPExtensionRequest**, **NWRegisterNCPExtensionByID**, **SetThreadContextSpecifier**

NWRegisterNCPExtensionByID

Registers a service to be provided as an NCP Extension and assigns the NCP Extension a specific ID

Local Servers: blocking

Remote Servers: N/A

NetWare Server: 3.12, 4.x

Platform: NLM

Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWRegisterNCPExtensionByID (
   LONG          NCPExtensionID,
   const char  *NCPExtensionName,
   BYTE       (*NCPExtensionHandler)(
      NCPExtensionClient  *NCPExtensionClient,
      void                *requestData,
      LONG                 requestDataLen,
      void                *replyData,
      LONG                *replyDataLen),
   void       (*ConnectionEventHandler)(
      LONG   connection,
      LONG   eventType)
   void        (*ReplyBufferManager)(
      NCPExtensionClient  *NCPExtensionClient,
      void                *replyBuffer),
   BYTE        majorVersion,
   BYTE        minorVersion,
   BYTE        revision,
   void      **queryData);
```

Parameters

NCPExtensionID

(IN) Specifies the unique ID to be associated with your service for the NCP Extension (assigned by Developer Support).

NCPExtensionName

(IN) Points to the name to identify the NCP Extension.

NCPExtensionHandler

(IN) Points to the function to be called when the NCP Extension calls the **NWSendNCPExtensionRequest** or **NWSendNCPExtensionFraggedRequest** function (optional).

ConnectionEventHandler

(IN) Points to the function to be called and steps to follow when a connection is freed, killed, logged out, or restarted (optional).

ReplyBufferManager

(IN) Points to a buffer manager function used to reply to NCP Extension requests (optional).

majorVersion

(IN) Specifies the major version number of the service provider.

minorVersion

(IN) Specifies the minor version number of the service provider.

revision

(IN) Specifies the revision number of the service provider.

queryData

(OUT) Points to a 32-byte area that NetWare has allocated.

Return Values

0	0x00	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
5	0x05	ENOMEM Not enough memory was available on the server to register the service.
166	0xA6	ERR_ALREADY_IN_USE The NCP Extension name is already registered. Your service is not registered.
251	0xFB	ERR_UNKNOWN_REQUEST The server version does not support this request.
255	0xFF	ERR_BAD_PARAMETER The *NCPExtensionName* parameter is longer than the 32-byte limit.

Remarks

NWRegisterNCPExtensionByID is called by the service-providing NLM applications in conjunction with **NWDeRegisterNCPExtension** and **NWRegisterNCPExtension**.

NCP extension names are case sensitive and must be unique. They have a maximum length of 32 bytes plus a NULL terminator.

For an explanation of the *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager* parameters, see the Remarks section for **NWRegisterNCPExtension**.

The *queryData* parameter can be used by the service provider to return up to 32 bytes of information to the client and is aligned on a DWORD

(32-bit) boundary. This information can then be retrieved by calling **NWGetNCPExtensionInfo (NLM)**, **NWGetNCPExtensionInfoByID**, or **NWScanNCPExtensions (NLM)**. The *queryData* parameter is also used by the registering NLM as the NCP extension handle when **NWDeRegisterNCPExtension** is called.

NOTE: The *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager* parameters are function callbacks that run as OS threads. They need to have CLIB context if they are going to call NetWare functions that need context.

See Also

GetThreadContextSpecifier, **NWDeRegisterNCPExtension**, **NWRegisterNCPExtension**, **SetThreadContextSpecifier**

NWSendNCPExtensionFraggedRequest

Sends a request to the specified NCP extension and allows data to be retrieved from and stored in noncontiguous memory locations

Local Servers: blocking

Remote Servers: blocking

NetWare Server: 3.11, 3.12, 4.x

Platform: NLM

Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWSendNCPExtensionFraggedRequest (
   LONG                            NCPExtensionID,
   const struct NCPExtensionMessageFrag
                                   *requestFrag,
   struct NCPExtensionMessageFrag
                                   *replyFrag);
```

Parameters

NCPExtensionID

(IN) Specifies the ID of the NCP Extension to process the request.

requestFrag

(IN) Points to the NCPExtensionMessageFrag structure containing information about the lengths and locations of the fragmented data for the NCP Extension handler to process (optional).

replyFrag

(IN/OUT) Points to the NCPExtensionMessageFrag structure. Inputs the maximum length of the data to return and where to place the data. Outputs the length of all returned data and where the data is stored (optional).

Return Values

0	0x00	SUCCESSFUL The extension was found, and the non-null output parameters were filled.
126	0x7E	ERR_NCPEXT_TRANSPORT_PROTOCOL_VIOLATION The message transport mechanism entered a bad state in the protocol.
150	0x96	ERR_NO_ALLOC_SPACE There was not enough memory available on the server to allocate space for the

0	6	memory available on the server to allocate space for the message.
252	0xFC	ERR_NCPEXT_SERVICE_PROTOCOL_VIOLATION The service provider tried to return more data than the reply buffer could hold.
254	0xFE	ERR_NCPEXT_NO_HANDLER The NCP exception handler could not be found.
1-16		A communications error has occurred. (See niterror.h.)

Remarks

Call **NWSendNCPExtensionFraggedRequest** when you want to send your NCP Extension handler information stored at various locations which will avoid copying the information into a single buffer before sending it to your NCP Extension.

NWSendNCPExtensionFraggedRequest can also place the reply data into up to four specific locations, eliminating the need for you to copy the data from a reply buffer.

If your NCP Extension uses a single input buffer and/or a single output buffer, call **NWSendNCPExtensionRequest** instead of **NWSendNCPExtensionFraggedRequest**.

If your NLM registers its NCP Extension by a specific ID, use that ID when calling **NWSendNCPExtensionFraggedRequest**. If your NLM registers its NCP Extension by name, call **NWGetNCPExtensionInfo (NLM)** or **NWScanNCPExtensions (NLM)** to obtain the ID before calling **NWSendNCPExtensionFraggedRequest**.

NWSendNCPExtensionFraggedRequest copies the number of bytes from the server (indicated in the *totalMessageSize* field of the NCPExtensionMessageFrag structure), places them into memory locations (specified in the *fragList* field of the NCPExtensionMessageFrag structure), and sets a value to reflect the actual number of bytes transferred (indicated by the *totalMessageSize* field of the NCPExtensionMessageFrag structure).

NOTE: The information in the *replyFrag* parameter is valid only if **NWSendNCPExtensionFraggedRequest** returns SUCCESSFUL.

The request and reply buffers of the client must be reproduced on the server, so the maximum size of the buffers depends upon the memory available on the server that registers the NCP Extension. When **NWSendNCPExtensionFraggedRequest** is called, it attempts to allocate server memory for two message buffers. If it cannot allocate enough space, ERR_NO_ALLOC_SPACE will be returned. However, the request should be retried several times since server memory use is dynamic.

See Also

NWSendNCPExtensionRequest

NWSendNCPExtensionRequest

Sends a request to the specified NCP extension
Local Servers: blocking
Remote Servers: blocking
NetWare Server: 3.11, 3.12, 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWSendNCPExtensionRequest (
   LONG         NCPExtensionID,
   const void  *requestData,
   LONG         requestDataLen,
   void        *replyData,
   LONG        *replyDataLen);
```

Parameters

NCPExtensionID

(IN) Specifies the ID of the NCP Extension to process your request (obtained by calling the **NWGetNCPExtensionInfo (NLM)** or **NWScanNCPExtensions (NLM)** function.

requestData

(IN) Points to information for the NCP Extension handler to process (optional).

requestDataLen

(IN) Specifies the length (in bytes) of the input request buffer that is being sent to the NCP Extension (optional).

replyData

(OUT) Points to the information returned by the NCP Extension (optional).

replyDataLen

(IN/OUT) Inputs a pointer to the length (in bytes) of the *replyData* parameter. Outputs a pointer to the actual number of bytes placed into the buffer (optional).

Return Values

0	0x0	SUCCESSFUL The extension was found, and the

	0	non-null output parameters were filled.
126	0x7E	ERR_NCPEXT_TRANSPORT_PROTOCOL_VIOLATION The message transport mechanism entered a bad state in the protocol.
150	0x96	ERR_NO_ALLOC_SPACE There was not enough memory available on the server to allocate space for the message.
252	0xFC	ERR_NCPEXT_SERVICE_PROTOCOL_VIOLATION The service provider tried to return more data than the reply buffer could hold.
254	0xFE	ERR_NCPEXT_NO_HANDLER The NCP exception handler could not be found.
1-16		A communications error has occurred. (See niterror.h.)

Remarks

NWSendNCPExtensionRequest sends the number of bytes specified by the *requestDataLen* parameter to the server.

If the *requestData* or *requestDataLen* parameter is set to NULL or zero respectively, no request data is sent.

NWSendNCPExtensionFraggedRequest copies the number of bytes from the server (indicated in the *totalMessageSize* field of the NCPExtensionMessageFrag structure), places them into memory locations (specified in the *fragList* field of the NCPExtensionMessageFrag structure), and sets a value to reflect the actual number of bytes transferred (indicated by the *totalMessageSize* field of the NCPExtensionMessageFrag structure).

NWSendNCPExtensionRequest copies the number of bytes from the server (specified in the *replyDataLen* parameter), places them into memory (specified in the *replyData* parameter), and sets a value to reflect the actual number of bytes transferred (specified by the *replyDataLen* parameter.

If the *replyData* or *replyDataLen* parameter is set to NULL or zero respectively, no reply data is returned.

NOTE: The information in the *replyData* parameter is valid only if **NWSendNCPExtensionRequest** returns SUCCESSFUL.

The request and reply buffers of the client must be reproduced on the server, so the maximum size of the buffers depends upon the memory available on the server that registers the NCP Extension. When **NWSendNCPExtension** is called, it attempts to allocate server memory for two message buffers. If it cannot allocate enough space, ERR_NO_ALLOC_SPACE will be returned. However, the request should

be retried several times since server memory use is dynamic.

See Also

NWDeRegisterNCPExtension, **NWGetNCPExtensionInfo (NLM)**, **NWScanNCPExtensions (NLM)**, **NWSendNCPExtensionFraggedRequest**, **NWRegisterNCPExtension**

NWScanNCPExtensions

Scans the server for NCP extension handlers
NetWare Server: 4.x
Platform: DOS, OS/2, Windows 3.1, Windows NT
Service: NCP Extension

Syntax

```
#include <nwncpext.h>
or
#include <nwcalls.h>

NWCCODE N_API NWScanNCPExtensions(
   NWCONN_HANDLE   conn,
   pnuint32        NCPExtensionID,
   pnstr8          NCPExtensionName,
   pnuint8         majorVersion,
   pnuint8         minorVersion,
   pnuint8         revision,
   pnuint8         queryData);
```

Pascal Syntax

```
#include <nwncpext.inc>

Function NWScanNCPExtensions
  (conn : NWCONN_HANDLE;
   NCPExtensionID : pnuint32;
   NCPExtensionName : pnstr8;
   majorVersion : pnuint8;
   minorVersion : pnuint8;
   revision : pnuint8;
   queryData : pnuint8
) : NWCCODE;
```

Parameters

conn

(IN) Specifies the NetWare server connection handle.

NCPExtensionID

(IN) Points to the ID of the NCP extension handler for which to get information. Should be set to -1 for the first iteration.

NCPExtensionName

(OUT) Points to the 32-byte buffer to receive the NCP extension name (optional).

majorVersion

(OUT) Points to the major version number of the NCP extension handler (optional).

minorVersion

(OUT) Points to the minor version number of the NCP extension handler (optional).

revision

(OUT) Points to the revision number of the NCP extension handler (optional).

queryData

(OUT) Points to the 32-byte buffer of custom information the NCP extension handler can use (optional).

Return Values

None

NCP Calls

0x2222 36 00 Scan Currently Loaded NCP Extensions

See Also

NWGetNCPExtensionInfo, **NWNCPExtensionRequest**, **NWFragNCPExtensionRequest**, **NWGetNCPExtensionInfoByName**, **NWGetNCPExtensionsList**, **NWGetNumberNCPExtensions**

NWScanNCPExtensions (NLM)

Iteratively returns information about all registered NCP extensions
Local Servers: nonblocking
Remote Servers: blocking
NetWare Server: 4.x
Platform: NLM
Service: NCP Extension

Syntax

```
#include <ncpext.h>

int NWScanNCPExtensions (
   LONG  *NCPExtensionID,
   char  *NCPExtensionName,
   BYTE  *majorVersion,
   BYTE  *minorVersion,
   BYTE  *revision,
   void  *queryData);
```

Parameters

NCPExtensionID

(IN/OUT) Points to the ID of the desired NCP Extension.

NCPExtensionName

(OUT) Points to the name of the NCP Extension that was found.

majorVersion

(OUT) Points to the major version number of the NCP Extension provider (optional).

minorVersion

(OUT) Points to the minor version number of the NCP Extension provider (optional).

revision

(OUT) Points to the revision number of the NCP Extension provider (optional).

queryData

(OUT) Points to 32 bytes of information from the NCP Extension service provider and allocated when the NLM calls the **NWRegisterNCPExtension** function (optional).

Return Values

0	SUCCESSFUL: Extension was found and non-NULL output parameters were filled
-1	No more NCP Extensions were found
1-16	An NCP error occurred (see niterror.h)

Remarks

NWScanNCPExtensions (NLM) can be used iteratively to return the names of all the NCP Extensions registered on the server being queried. To scan the complete list of NCP Extensions, set the *NCPExtensionID* parameter to BEGIN_SCAN_NCP_EXTENSIONS. When **NWScanNCPExtensions (NLM)** returns, the *NCPExtensionID* parameter will be set to the ID of the first NCP Extension in the list and will return SUCCESSFUL. Use the ID in the *NCPExtensionID* parameter as a seed value to find the next NCP Extension ID. Continue calling **NWScanNCPExtensions (NLM)**, using the new IDs returned in the *NCPExtensionID* parameter, until you find the information you want or until -1 is returned.

Call **NWScanNCPExtensions (NLM)** when you want to list the names of the NCP Extensions but are not looking for the name of a specific extension. If you know the name of your NCP Extension, such as "My NCP Extension," you should call **NWGetNCPExtensionInfo (NLM)** to see if the extension is registered because **NWGetNCPExtensionInfo (NLM)** needs to be called only once.

Call **NWScanNCPExtensions (NLM)** to do the following:

- See if the NCP Extension is registered.
- Check the version of the NCP Extension.
- Get the NCP Extension ID number to call **NWSendNCPExtensionRequest**.
- Receive 32 bytes of information from the NCP Extension without calling the **NWSendNCPExtensionRequest** function.

The *NCPExtensionName* parameter should be set to a buffer that is MAX_NCP_EXTENSION_NAME_BYTES (33) bytes long. The returned name is case sensitive and unique for each NCP Extension.

NOTE: The IDs of NCP Extensions are not always consecutive numbers. Therefore, you should not assume that if you increment the value in the *NCPExtensionID* parameter by one that it is a valid NCP Extension ID.

The *majorVersion*, *minorVersion*, and *revision* parameters are assigned by

the registering NLM when it calls **NWRegisterNCPExtension**. If you have different versions or revisions of the NCP Extension, use these fields to verify that the extension is of the correct version. If you do not want any of this information, pass NULL.

NWScanNCPExtensionInfo can return all of the information you need, eliminating the need to call **NWSendNCPExtensionRequest**. If all your information can be returned in the *queryData* parameter, obtain the buffer contents by calling **NWGetNCPExtensionInfo (NLM)**. Receiving information this way does not call the NCP Extension handler and is useful only if a one-way server-to-client message is sufficient. If you do not need the information that is returned in the buffer, pass NULL.

See Also

NWDeRegisterNCPExtension, **NWGetNCPExtensionInfo (NLM)**, **NWRegisterNCPExtension**, **NWSendNCPExtensionRequest**

Communication Service Group

NCP Extension: Structures

FragElement

Defines a fragment of a fragmented NCP extension request
Service: NCP Extension
Defined In: ncpext.h

Structure

```
struct FragElement {
   void  *ptr;
   LONG   size;
};
```

Fields

ptr

Points to the fragment data.

size

Specifies the number of bytes that can be placed in the *ptr* field.

NCPExtensionClient

Defines an NCP extension client
Service: NCP Extension
Defined In: ncpext.h

Structure

```
struct NCPExtensionClient {
   LONG   connection;
   LONG   task;
};
```

Fields

connection
Specifies the connection number of the client.

task
Specifies the task number of the client.

NCPExtensionMessageFrag

Defines a fragmented NCP extension request
Service: NCP Extension
Defined In: ncpext.h

Structure

```
struct NCPExtensionMessageFrag {
   LONG                 totalMessageSize;
   LONG                 fragCount;
   struct FragElement   fragList[4];
};
```

Fields

totalMessageSize

Specifies the limit (in bytes) for the returned data.

fragCount

Specifies the number of FragElement structures stored in the*fragList* field.

fragList

Specifies an array of up to four FragElement structures.

NWSIPX

NWSIPX: Guides

NWSIPX: General Guide

NWSIPX: Concepts

Connection Endpoint Management in NWSIPX

The functions associated with connection-oriented services are used within the context of a connection endpoint. By definition, IPX sockets are the network-addressable service access points for the IPX/SPX protocols. Connection endpoints act as extensions to the IPX socket for connection-oriented services. When you open a connection endpoint, you receive a connection endpoint handle that uniquely identifies the connection endpoint. The connection endpoint handle is a required parameter for all connection-oriented functions.

A connection endpoint is opened by calling **NWSipxOpenConnectionEndpoint**. Because a connection endpoint is always associated with an IPX socket, the API automatically opens a socket for every connection endpoint that is allocated. The socket number can be specified or dynamically selected by the API.

NWSIPX Services maintains a set of connection element parameters for each socket. These parameters are retrieved by calling **NWSipxGetInformation** with the *infoType* parameter set to SIPX_CONN_INFORMATION.

The connection element parameters are returned in an SIPX_CONN_INFO structure.

Connection-endpoint parameters can be examined and modified using **NWSipxGetInformation** and **NWSipxSetInformation**.

The following functions are used for connection endpoint management:

Function	Header	Purpose
NWSipxCloseConnectionEndpoint	nwsipx32.h	Closes an open connection endpoint.
NWSipxOpenConnectionEndpoint	nwsipx32.h	Opens a connection endpoint.

Connection-Oriented Message Management and Data Transfer

Data segments making up a complete or partial message are defined for the API using a fragment list. The fragment list is an array of fragment descriptors that identify the data fragments constituting the data segment. The number of fragment descriptors contained in the fragment list and the pointer to the fragment list are stored in the *TCBFragmentCount* and *TCBFragmentList* fields of the NWTCB respectively.

For a definition of the fragment descriptor, see FRAGMENT.

To send data messages, call **NWSipxSendMessage**. Before calling **NWSipxSendMessage**, you must initialize the following fields in the NWTCB:

TCBConnHandle	Handle of the connection endpoint associated with the transport connection.
TCBFragmentCount	Number of fragment descriptors in the fragment list (maximum of 15).
TCBFragmentList	Pointer to the fragment list defining the data segment to send.
TCBFlags	The SIPX_SNDMSG_PARTIAL flag is set or cleared to indicate whether or not this data segment is the end of a message.

NOTE: If you want to send an SPX header without data, set the *TCBFragmentCount* field to zero. The API will then ignore the *TCBFragmentList* field.

You must segment data messages larger than the maximum transport service data unit (TSDU) size supported by the underlying SPX protocol stack. The maximum packet size is determined by calling **NWSipxGetMaxTsduSize** or referencing the *CIMaxTsduSize* connection endpoint parameter. If the maximum TSDU value is equal to SIPX_CONN_UNLIMITED_TSDU, you can send any size message to the peer system and the API will automatically fragment the message according to the output mode selected in the *CIOutput* connection endpoint parameter. Otherwise, the TSDU size must be honored and you must segment the data message as described above. You must also indicate message boundaries to the API. This is done by setting the SIPX_SNDMSG_PARTIAL flag in the *TCBFlags* field of the NWTCB passed to **NWSipxSendMessage** if it is **not** the last segment of a message. Otherwise, the SIPX_SNDMSG_PARTIAL flag should be clear.

When sending data, you can select between two output modes: fragmentation mode and transparent mode. In fragmentation mode (the default), the API automatically fragments data messages for delivery across the network to the peer system as needed. In transparent mode, the API does not fragment data messages, but sends the messages as received from you to the peer system. In transparent mode, it is your responsibility to determine valid message sizes. Transparent mode is provided for your

applications that negotiate message sizes with peer applications and have a means for determining the optimal message size. The output mode is contained in the *CIOutputMode* connection endpoint parameter, which can be read and set using the **NWSipxGetInformation** and **NWSipxSetInformation** functions.

You are not required to build the SPX header as part of the data message. The API adds any necessary protocol headers. You can optionally specify the SPX data stream type that will be stored in SPX protocol header by either setting the SPX Datastream Type connection endpoint parameter or setting the SIPX_SNDMSG_DSTRM_TYPE flag in the *TCBFlags* field and storing the data stream value in the *TCBDataStreamType* field in the NWTCB passed to **NWSipxSendMessage**. You can request that the attention flag be set in the SPX header by setting the SIPX_SNDMSG_ATTN flag in the *TCBFlags* field of the NWTCB passed to **NWSipxSendMessage**.

> **NOTE:** The NWSIPX API for Windows NT* does not support the setting of the attention flag in the SPX header.

When the message has been successfully sent, you are notified according to the event notification method specified in the NWTCB. The *TCBBytesTransferred* and *TCBFinalStatus* fields of the NWTCB are updated to reflect the number of data bytes actually sent and the final status of the service request.

To receive data messages, call **NWSipxReceiveMessage**. You are responsible for allocating the data space in which the input message is stored. Before calling **NWSipxRecieveMessage**, you must initialize the following fields in the NWTCB:

TCBConnHandle	Handle of the connection endpoint associated with the transport connection.
TCBFragmentCount	Number of fragment descriptors in the fragment list.
TCBFragmentList	Pointer to the fragment list defining the allocated data space.

When a message has been received, you are notified according to the event notification method specified in the NWTCB. Before notification, the NWSIPX API updates the *TCBFinalStatus*, *TCBBytesTranferred*, *TCBDatastreamType*, *TCBMsgSequenceNumber*, and *TCBFlags* fields of the NWTCB. The SIPX_RCVMSG_PARTIAL flag is set in the *TCBFlags* field of the NWTCB if a partial message was received. If the whole message was received, the SIPX_RCVMSG_PARTIAL flag is clear. If the attention flag was set in the SPX header, the SIPX_RCVMSG_ATTN flag is set in the *TCBFlags* field of the NWTCB.

You can select between streaming mode and message mode when receiving data. In streaming mode, you receive message fragments as they arrive from the network and you are responsible for putting them into a meaningful

context. The API indicates which fragment is the final fragment of a message by clearing the SIPX_RCVMSG_PARTIAL flag in the *TCBFlags* field of the NWTCB. In message mode, you will only receive whole data messages unless the input message is larger than the data space you allocated for input. If the API is able to buffer the excess data, the API sets the SIPX_RCVMSG_PARTIAL flag in the *TCBFlags* field of the NWTCB and it returns a final status of SIPX_SUCCESS, indicating that **NWSipxReceiveMessage** must be called again to receive the remainder of the message. If no more internal resources exist to hold the excess data, the input message is truncated and a final status of SIPX_DATA_OVERFLOW is returned. The connection's input mode is contained in the *CIInputMode* connection endpoint parameter, which can be read and set using the **NWSipxGetInformation** and **NWSipxSetInformation** functions.

To enable you to determine the order in which message segments are received from a peer, the sequence number of the message segment is stored in the *TCBMsgSequenceNumber* field of the NWTCB. The message sequence number is maintained as part of the connection endpoint and is set to zero when a transport connection is established. It is incremented each time a message segment is received and its value is stored in the *TMCBMsgSequenceNumber* field of the NWTCB of the message segment. The message sequence number is **not** the SPX protocol sequence number.

Parent Topic: NWSIPX Connection-Oriented Service

Datagram Message Management and Data Transfer

Datagrams are defined for the API with a fragment list. The fragment list is an array of fragment descriptors that define the data fragments constituting a complete datagram. The number of fragment descriptors contained in the fragment list and the pointer to the fragment list are stored in the *TCBFragmentCount* and *TCBFragmentList* fields of the NWTCB respectively.

For a definition of the fragment descriptor, see FRAGMENT.

You send datagrams by calling **NWSipxSendDatagram**. Before calling **NWSipxSendDatagram**, you must initialize the following fields in the NWTCB:

TCBSockHandle	Handle of the socket the datagram is to be sent on.
TCBFragmentCount	Number of fragment descriptors in the fragment list (maximum of 15).
TCBFragmentList	Pointer to the fragment list defining the datagram.
TCBRemoteAddress	The network address of the remote system.

NOTE: If you want to send an IPX header without data, set the *TCBFragmentCount* field to zero. The API will then ignore the *TCBFragmentList* field.

You can optionally specify the IPX packet type to be stored in the IPX protocol header by either setting the *SIPacketType* parameter of the socket (see SIPX_SOCKET_INFO) or by setting the SIPX_SNDDG_PACKET_TYPE flag and storing the packet type value in the *TCBFlags* and *TCBPacketTyp*e fields of the NWTCB passed to **NWSipxSendDatagram**.

You are not required to determine and maintain the address of the next hop in the network path to a remote system (also known as the local target). Whenever datagrams are sent to a remote system, the NWSIPX API determines and maintains the address of the local target. If the next hop in the network path to the remote system cannot be ascertained by the API when you call **NWSipxSendDatagram**, the SIPX_NO_ROUTE_TO_TARGET status is returned. Even though you are not required to initially determine the local target address, you must request the API to verify the local target address whenever it appears that the path to the remote system is no longer valid. This is done by setting the SIPX_SNDDG_VERIFY_ROUTE flag in the *TCBFlags* field of the NWTCB before calling **NWSipxSendDatagram** to send the next datagram. The SIPX_SNDDG_VERIFY_ROUTE flag should not be set every time **NWSIPXSendDatagram** is called, as it causes the API to flush the destination address from its route cache and query the network for a new route, resulting in higher processing overhead and degraded performance.

You can also request certain routing options when sending a datagram. The following table shows the options and the methods used to select the options.

Select a specific subnetwork	Set the SIPX_SNDDG_SPECIFIC_ROUTE flag in the *TCBFlags* field and store the handle of the subnetwork to use in the *TCBSubnetworkHandle* field of the NWTCB.
Select the best route over any subnetwork	Set the SIPX_SNDDG_BEST_ROUTE flag in the *TCBFlags* field of the NWTCB.

When the message has been successfully sent, you are notified using the event notification method specified in the NWTCB. The *TCBBytesTransferred*, *TCBFinalStatus*, *TCBPacketType*, and *TCBBoardNumber* fields of the NWTCB are updated to reflect the number of data bytes actually sent, the final status of the service request, the packet type value stored in the IPX header, and the board number of the network interface card used to send the datagram.

You receive datagrams by calling **NWSipxReceiveDatagram**. You are responsible for allocating the data space in which to store the input

datagram. Before calling **NWSipxRecieveDatagram**, the you must initialize the following fields in the NWTCB:

TCBSockHandle	Handle of the socket to monitor for input.
TCBFragmentCount	Number of fragment descriptors in the fragment list (maximum of 15).
TCBFragmentList	Pointer to the fragment list defining the allocated space.

When a datagram has been received, you are notified using the method specified in the NWTCB. Before you are notified, the NWSIPX API updates the *TCBFinalStatus*, *TCBBytesTranferred*, *TCBPacketType*, *TCBRemoteAddress*, and *TCBSubnetworkHandle* fields of the NWTCB.

You must segment outgoing messages according to the maximum packet size supported by the underlying network layer. Conversely, you must also reassemble any data fragments received as input into a single datagram. The maximum packet size is determined for a specific socket by calling **NWSipxGetMaxNsduSize** or by referencing the *SIMaxNsduSize* socket parameter. (See SIPX_SOCKET_INFO.)

If checksum processing is desired, you can request it when calling **NWSipxSendDatagram** by setting the SIPX_SNDDG_GENERATE_CHKSUM flag in the *TCBFlags* field of the NWTCB. The API will generate a checksum for that message only. Datagrams received with checksums will be automatically validated before being passed to you.

Parent Topic: NWSIPX Datagram Service

Event Notification in NWSIPX

When allocating an NWTCB, you select the method of event notification to be employed for any function using the NWTCB. The following event notification methods are available:

Blocking

Polling

Callback functions

User-managed events

API-managed events

Multiplexed API-managed events

The event notification method of a request is determined by the NWTCB that a function uses. When you allocate an NWTCB, you specify an event notification method that is to be assigned to the NWTCB. Whenever that NWTCB is used, the event notification method associated with the NWTCB is invoked to indicate the completion of the service request.

You can change the event notification method associated with an NWTCB by using one of the following methods:

Call **NWSipxChangeControlBlock**.

Retrieve the NWTCB element parameters by calling **NWSipxGetInformation**, change the *NWIEventType* and *NWIEventInfo* element parameters, and then call **NWSipxSetInformation** to make the change.

If an NWSIPX function requires an NWTCB, the return value indicates whether or not the request was **initiated** successfully, and the API stores the final status of the request in the *TCBFinalStatus* field of the NWTCB when the request completes. If a service request requiring an NWTCB cannot be initiated successfully, (that is, the return value indicates an error), the return value is the final status of the request and the *TCBFinalStatus* field is not updated.

The **SIPX_SUCCESS** and **SIPX_ERROR** macros are provided to help applications easily determine whether an error has occurred during the execution of an NWSIPX service request. These macros take a function's return value or the contents of an NWTCB's *TCBFinalStatus* field, and return a boolean value of TRUE or FALSE, indicating the success or failure of the service request.

The event notification methods are described in the following table:

Event Notification Method	Description
Blocking (SIPX_BLOCKING)	The function is blocked; that is, control is not returned on that execution thread until the request is complete. You must then check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request.
Polling (SIPX_POLLING)	The function returns immediately. If the request was initiated successfully, you must call **NWPoll** repetitively until a status other than SIPX_PENDING is returned, indicating that the request has completed. You must then check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request.
Callback function (SIPX_CALLBACK)	The function returns immediately. If the request was initiated successfully, your callback function is called when the request

	completes. When the callback function receives control, you must check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request. **You must not block on a callback function execution thread.** The address of the callback function is passed as a parameter to **NWSipxAllocControlBlock** when the associated NWTCB is allocated. The callback address can be changed by calling **NWSipxChangeControlBlock** or by storing the address of the new callback function in the *NWIEventInfo* NWTCB element parameter by calling **NWSIpxSetInformation**. Every callback function has the prototype of : `void (N_CDECL Callback_Function)(*NWTCB)`
User-managed events (SIPX_USER_EVENT)	The function returns immediately. If the request was initiated successfully, your application's event object is signaled when the request completes. When you receive control, you must check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request. A handle to the event object that you allocated is passed as a parameter to **NWSipxAllocControlBlock**. The event object handle can be changed by calling **NWSipxChangeControlBlock**, or by storing a new event object handle in the *NWIEventInfo* NWTCB event parameter and calling **NWSipxSetInformation**. Any type of event object supported by the OS platform can be used, and it is your responsibility to coordinate any execution thread synchronization associated with the signaled event object.
API-managed events (SIPX_API_EVENT)	The function returns immediately. If the request was initiated successfully, you must call **NWSipxWaitForSingleEvent**, which blocks until the request completes or the specified timeout period has elapsed. You must then check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request. The use of event objects for thread synchronization is completely managed by the API.
Multiplexed	This method of event notification allows you to

Multiplexed API-managed events (SIPX_API_MUX_EVENT)	This method of event notification allows you to monitor multiple NWTCBs for event completion with a single call to **NWSipxWaitForMultipleEvents**. To group NWTCBs for use by **NWSipxWaitForMultipleEvents**, an NWTCB mux group must be created and the mux group handle specified when allocating the NWTCB. A new NWTCB mux group is automatically created by the NWSIPX API whenever a mux group handle value of SIPX_ALLOC_MUX_GROUP is passed to **NWSIPXAllocControlBlock**. If successfully allocated, the handle of the new NWTCB mux group is returned for use with subsequent calls to **NWSipxAllocControlBlock**. The mux group handle can be changed by calling **NWSipxChangeControlBlock** or by storing the new mux group handle into the *NWIEventInfo* NWTCB element parameter and calling **NWSipxSetInformation**. To monitor multiple NWTCBs, you must call **NWSipxWaitForMultipleEvents**, which blocks until an event completes for one of the multiplexed NWTCBs or the specified timeout period has elapsed. A pointer to the NWTCB with the completed event is returned. you must then check the *TCBFinalStatus* field of the NWTCB to determine the result of the service request. The use of event objects for thread synchronization is completely managed by the API.

The following functions are used for event notification management:

Function	Header	Purpose
NWSipxAllocControlBlock	nwsipx32.h	Allocates an NWTCB.
NWSipxPoll	nwsipx32.h	Checks if a pending request has completed.
NWSipxWaitForMultipleEvents	nwsipx32.h	Monitors multiple NWTCBs for a transport event.
NWSipxWaitForSingleEvent	nwsipx32.h	Monitors a single NWTCB for a transport event.

Migrating to NWSIPX from Previous IPX/SPX APIs

Client applications written to use previous versions of IPX/SPX APIs can be converted to use the NWSIPX API in a direct manner. In general, the overall logic flow of an existing application does not have to change when using the NWSIPX API and in many areas can be greatly simplified. Existing applications must be modified in the following areas to use the NWSIPX API:

- Use 32-bit memory addresses
- Use new NWSIPX functions
- Use NWTCBs instead of ECBs
- Investigate and possibly use new event notification methods
- Remove code that builds IPX and SPX headers
- Remove code that determines the next hop in network path (local target)
- Query the API for the maximum size datagram or message that can be sent
- Open and use connection endpoints for connection-oriented services
- Investigate use of operational parameters

The NWSIPX API supports 32-bit addressing. It is assumed that your application is written to operate in a 32-bit environment.

The NWSIPX API supports a set of functions that are similar to the functions supported by previous IPX/SPX APIs but have been adapted to the new capabilities of the NWSIPX API. In most cases, migrating from a previous API to the NWSIPX API requires replacing the old API service requests with the analogous service requests of the NWSIPX API. However, there are a few old API service requests that have either been removed or the function is provided for in a new way. Most often, this is because the API has assumed responsibility for the missing function and your application can be simplified. See NWSIPX: Functions for a complete description of each of the NWSIPX functions.

Instead of ECBs, you use NWTCBs to interface with the NWSIPX API. NWTCBs are used in the same fashion as ECBs, but you must allocate them by calling **NWSipxAllocControlBlock**, instead of allocating memory for the NWTCB yourself. The NWTCB is used to store information needed by the API to properly satisfy the service request. The API stores results of the service request in the NWTCB when the request completes. When allocating an NWTCB, you select the method of event notification to be employed for any function using the NWTCB. See NWTCB Management for a complete description of the NWTCB and its use.

Every event notification method (that is, synchronous, ESR, and so on) supported by previous IPX/SPX APIs is supported by the NWSIPX API. As a result, the overall program structure and logic flow of an application is not required to change. Also, new event notification methods have been added to take advantage of the capabilities of various OS platforms. A key point is that you are not restricted to using a single event notification method, but can choose any event notification method for each function. This gives you greater flexibility in your approach to accessing the network and processing network events. See Event Notification in NWSIPX for complete details.

You are no longer required to build or examine IPX and SPX protocol headers. Protocol headers are built by the API on your behalf. The NWSIPX API allows you to specify the values of user-definable fields in IPX and SPX headers either by setting an operational parameter that causes the selected value to appear in the protocol headers of all messages sent, or by specifying values on a per-message basis. Because you don't have to build protocol headers, client applications can be simplified to concentrate on exchanging data with peers without having to manage the protocols required to transfer the messages across the network. See Datagram Message Management and Data Transfer and Connection-Oriented Message Management and Data Transfer for a complete description of sending and receiving datagrams and messages.

When using datagram services, you are no longer required to determine and maintain the address of the next hop in the network path to a remote system (also known as the local target). Whenever datagrams are sent to a remote system, the API determines and maintains the address of the local target. If the next hop in the network path to the remote system cannot be ascertained by the API, you are notified. Even though you are not required to initially determine the local target address, you must request the API to verify the local target address whenever it appears that the path to the remote system is no longer valid. Local target processing applies to datagram services only. See Datagram Message Management and Data Transfer for a complete description of local target processing.

You are still responsible for segmenting and reassembling data segments that are larger than the message size supported by the underlying transport layer. To determine the maximum size data segment supported by the network, you must query the API on a peer socket or connection endpoint basis depending on whether a datagram or connection oriented-message is being sent. The maximum message size associated with each socket or connection endpoint varies depending on the capabilities of the network layer protocols being used and the type of network media the data message must traverse. See Datagram Message Management and Data Transfer and Connection-Oriented Message Management and Data Transfer for more information on segmenting and reassembling datagrams and messages.

The functions associated with connection-oriented services are used within the context of a connection endpoint. By definition, IPX sockets are the network addressable service access points for the IPX/SPX protocols. Connection endpoints act as extensions to the IPX socket for connection-oriented services. When a connection endpoint is opened, you receive a connection endpoint handle that uniquely identifies the connection

endpoint. The connection endpoint handle is required by all connection-oriented functions and is used as the connection identifier for any transport connection using the connection endpoint (only one transport connection can be open on a connection endpoint at a time). Because a connection endpoint is always associated with an IPX socket, the API automatically opens a socket for every connection endpoint that is allocated. The socket number can be specified or it can be dynamically selected by the API. Each connection endpoint has a set of operational parameters that affect the operation of any transport connection using the connection endpoint. You can read and modify these parameters. See Connection Endpoint Management in NWSIPX for complete information on connections and their use.

Various elements of the NWSIPX API have operational parameters that affect the operation of the API. The API elements supporting parameters are the API environment, sockets, connection endpoints, and NWTCBs. Some parameters are informational (read-only) and others can be modified. All parameters are set with the default values that allow for API operation equivalent to that of the previous IPX/SPX APIs. As a result, existing applications are not required to read or modify parameters to use the NWSIPX API. But it is worthwhile to become familiar with the element parameters and how they can enhance the operation of your application. See NWSIPX Elements for more information on API element parameters and their use.

NWSIPX API Toolbox

The NWSIPX API contains a set of functions and macros that are not included as part the datagram or connection-oriented services, but are useful for other reasons. These services include canceling pending requests, determining the local network address for a socket, timer services, advertising, querying for network services, and determining if an NWSIPX status code indicates an error condition. Because all are extraneous to the datagram and connection oriented services, they have been grouped together as an API Toolbox.

Occasionally, you might want to cancel a request that is in progress. To do so, call **NWSipxCancelPendingRequest** with a pointer to the NWTCB associated with the pending request. The request is immediately terminated and the appropriate event notification method invoked as specified by the NWTCB. The *TCBFinalStatus* field of the NWTCB contains the SIPX_CANCELED status. This service only applies to requests that use an NWTCB.

If you need to know the local network address of a socket, call **NWSipxGetInternetAddress**. The network address is returned in the *TCBRemoteAddress* field of the NWTCB used for the request.

If you require a timing service, you can use **NWSipxRegisterForTransportEvent**. To do so, you must store the

SIPX_SCHEDULE_TIMER_EVENT event type and the timeout value in the *TCBTransportEvent* and *TCBTimeout* fields of the NWTCB used with **NWSipxRegisterForTransportEvent**. The advantage of using this service over native timing services provided by the OS platform is that any of the event notification methods provided by the NWSIPX API can be used to indicate the expiration of the timer.

You can register for notification of subnetwork status changes by storing the SIPX_SUBNET_STATUS_CHANGE event type in the *TCBTransportEventType* field of the NWTCB and calling **NWSipxRegisterForTransportEvent**. When a change in status occurs for a subnetwork, you will be notified according to the event notification methods specified for the NWTCB. You must call **NWSipxGetInformation** to get the subnetwork information. (The subnetwork information is part of the API environment parameters.)

If you provide a service that you want to advertise to the network, you can call **NWSipxAdvertiseService** to initiate the periodic broadcasting of the service to the network. The NWSIPX API broadcasts the service at 60 second intervals for as long as the request is active. As long as the API is advertising the service, queries for the service by other network nodes are answered automatically in your behalf. To cancel the advertising of a service, call **NWSipxCancelAdvertiseService**.

To query for network services, call **NWSipxQueryService**. You pass in an input buffer where the query response is stored. Upon receipt of the query response, the API stores server information into the input buffer until it is full or all of the service information has been stored. To terminate a service query request, call **NWSipxCancelPendingRequest**.

The **SIPX_SUCCESS** and **SIPX_ERROR** macros are provided to help you easily determine whether or not an error has occurred during the execution of an NWSIPX API service request. These macros take an NWSIPX status code (that is, a function return value or the contents of the *TCBFinalStatus* field of the NWTCB) as input and return a value of TRUE or FALSE, indicating the success or failure of the service request. Often the course of action following an error is the same regardless of the error type. These macros provide an easy way for you to determine if an error has occurred. The **SIPX_SUCCESS** macro returns TRUE if the status code does not indicate an error condition. The **SIPX_ERROR** macro returns TRUE if the status code indicates an error condition.

The NWSIPX API Toolbox functions and macros are listed below:

Function	Header	Purpose
NWSipxAdvertiseService	nwsipx32.h	Advertises a service.
NWSipxCancelAdvertiseService	nwsipx32.h	Cancels an active advertise service request.
NWSipxCancelPendingRequest	nwsipx32.h	Cancels a pending request.

NWSipxGetInternetAddress	nwsipx32.h	Returns the local network address of a specified socket.
NWSipxQueryService	nwsipx32.h	Broadcasts a query to discover the identities of all servers of any type, all servers of a specific type, or the nearest server of a specific type and then returns the response.
NWSipxRegisterForTransportEvent	nwsipx32.h	Registers to receive a transport event.
SIPX_ERROR	nwsipx32.h	Indicates whether an NWSIPX return value or status code indicates an error.
SIPX_SUCCESS	nwsipx32.h	Indicates whether an NWSIPX status code indicates success.

NWSIPX Connection Establishment

The establishment of transport connections can be initiated by either local or remote network applications. To establish a transport connection, the local client application calls **NWSipxEstablishConnection**. Before calling **NWSipxEstablishConnection**, you must open a connection endpoint and store the handle of the connection endpoint and the network address of the destination system in the *TCBConnHandle* and *TCBRemoteAddress* fields of the NWTCB passed to **NWSipxEstablishConnection**. The connection endpoint handle is used as the local connection identifier for the transport connection (only one transport connection can be opened on a connection endpoint at a time). When the connection is established, or if an error occurs, you are notified by the event notification method selected in the NWTCB.

To listen for incoming transport connection requests, you open a connection endpoint and then call **NWSipxListenForConnection**. If you have selected immediate acceptance of the transport connection request (the default), the request is immediately accepted by the API and you are notified of the connection establishment. The connection endpoint is put into the connected state and is now usable for data transfer. A new connection endpoint must be used to listen for further incoming connections.

If you select delayed acceptance, the API delays accepting the transport connection request until you call **NWSipxAcceptConnectionEx**. You can accept the incoming connection on the same connection endpoint used for

listening or on a different connection endpoint. In this way, you can selectively accept incoming connections and a single connection endpoint can be used to listen for all incoming connections. If the connection is accepted on a different connection endpoint than the listening connection endpoint, the listening connection endpoint is returned to the SIPX_ALLOCATED state. If you want to reject the incoming connection, call **NWSipxTerminateConnection** with the SIPX_TERM_REJECT flag set in the *TCBFlags* field of the NWTCB.

You are not required to pre-post receive buffers (that is, call **NWSipxReceiveMessage**) to listen for incoming connections.

Parent Topic: NWSIPX Connection-Oriented Service

NWSIPX Connection Release

To terminate a transport connection, call **NWSipxTerminateConnection**. The connection endpoint handle of the transport connection being terminated must be stored in the *TCBConnHandle* field of the NWTCB before calling **NWSipxTerminateConnection**. The NWSIPX API terminates the transport connection and notifies you according to the event notification method selected in the NWTCB. You select whether an acknowledged release or an abortive release occurs by setting the SIPX_TERM_ACKED or SIPX_TERM_ABORT flag in the *TCBFlags* field of the NWTCB. An acknowledged release causes the API to delay notification that the termination is complete until the peer acknowledges the transport connection release. When the abortive release is selected, you are notified of the termination immediately.

You can optionally monitor transport connections for termination by calling **NWSipxRegisterForTransportEvent** with the SIPX_LISTEN_FOR_DISCONNECT event type and the connection endpoint handle stored in the *TCBTransportEvent* and *TCBConnHandle* fields of the NWTCB. If a transport connection is terminated for any reason, you are notified using the event notification method selected for the NWTCB.

Parent Topic: NWSIPX Connection-Oriented Service

NWSIPX Connection-Oriented Service

The NWSIPX API supports connection oriented services. These services include establishing transport connections with peer applications, data transfer, and terminating transport connections. Connection oriented services use the SPX protocol to establish and maintain transport connections and to provide the reliable exchange of data messages with flow control.

When using connection oriented services, you call functions within the

context of connection endpoints. All connection-oriented functions either require a connection endpoint or are used to open or close a connection endpoint.

You are responsible for segmenting and reassembling data messages larger than the maximum message size supported by the underlying transport layer. You must call **NWSipxGetMaxTsduSize** or reference the *CIMaxTsduSize* connection endpoint parameter to get the maximum transport services data unit (TSDU) size supported by the connection endpoint through which the data message is to be sent or received. (See SIPX_CONN_INFO.) By using the maximum TSDU size determined by the API, you can send the largest messages possible according to the capabilities of the underlying network protocols and the requirements of the network path being used, rather than using the smallest maximum message size that fits all networks.You can select between two output modes when sending messages. The fragmentation mode (default) causes the API to automatically fragment data messages that are too large for transfer across the network to the peer system. The transparent mode causes the API to send data messages as received from you without further fragmentation. In this mode, the responsibility of determining valid message size rests with you.

The API adds the SPX header to messages sent to remote applications. You can optionally specify the data stream type value that is included in the SPX header by setting the *CIDataStreamType* connection endpoint parameter or when calling **NWSipxSendMessage**. You can also request the attention flag in the SPX header to be set at the time you call **NWSipxSendMessage**.

You can select between streaming and message mode when receiving data. In streaming mode (the default) you receive message fragments as they arrive from the network and you are responsible for putting them into a meaningful context. In message mode, only whole messages are received unless the receive buffer is too small to contain the whole message. In this case, you must call the API again to receive the rest of the message. If the API runs out of resources to stage an input message, a data overrun error occurs and the remainder of the input is discarded.

The following functions are used for connection management:

Function	Header	Purpose
NWSipxAcceptConnectionEx	nwsipx32.h	Accepts an incoming connection.
NWSipxEstablishConnection	nwsipx32.h	Establishes a transport connection with a peer.
NWSipxGetMaxTsduSize	nwsipx32.h	Returns the maximum data packet size that can be sent as a connection-oriented message for a specific connection endpoint.
NWSipxListenForConnecti	nwsipx32.	Listens for incoming

on	h	connections.
NWSipxReceiveMessage	nwsipx32.h	Prepares you to receive an input message.
NWSipxRegisterForTransportEvent	nwsipx32.h	Registers to receive a transport event.
NWSipxSendMessage	nwsipx32.h	Sends an output message to a peer.
NWSipxTerminateConnection	nwsipx32.h	Terminates a transport connection.

Related Topics

NWSIPX Connection Establishment

Connection-Oriented Message Management and Data Transfer

NWSIPX Connection Release

NWSIPX Datagram Service

The NWSIPX API supports datagram services. These services are limited to data transfer. The data transfer service uses the IPX protocol to provide connectionless exchange of data messages without flow control. Being connectionless in nature, the IPX protocol does not guarantee delivery of messages or that messages will be received in the order they are sent. It is up to you to provide message recovery and to put datagram messages into a meaningful context.

Datagram services are provided within the context of an IPX socket. All datagram service functions either require an open socket or are used to open or close a socket.

When using datagram services, you must be aware of the maximum packet size supported by the underlying network layer. You must segment data messages larger than the datagram packet size and reassemble input messages too large to fit in a single datagram message.

You are not required to build the IPX header when sending datagrams. The API adds the IPX header to any message sent to remote applications. You can optionally specify the value of the IPX packet type passed in the *Packet Type* field of the IPX header.

Checksum processing (generation and validation of checksums) for datagrams is handled completely by the NWSIPX API on your behalf. You can request checksum generation on a per-datagram basis or invoke automatic checksum processing when opening a socket. If automatic checksum processing is requested when opening a socket, every datagram sent over that socket will have a checksum calculated and stored in the IPX header. All datagrams received with a checksum will be automatically

validated by the API before being passed to you.

Your applications can also exercise some control over the routing of a datagram. These options include:

Specifying the subnetwork over which a datagram is sent

Requesting the API to verify and/or refresh the current network route to the destination system

Requesting the API to use the best route over any subnetwork available

The functions comprising the datagram services are listed below:

Function	Header	Purpose
NWSipxGetMaxNsduSize	nwsipx32.h	Returns the maximum data packet size that can be sent as a datagram for a specific socket.
NWSipxReceiveDatagram	nwsipx32.h	Prepares a client application to receive a datagram.
NWSipxSendDatagram	nwsipx32.h	Sends a datagram.

Related Topic: Datagram Message Management and Data Transfer

NWSIPX Elements

The NWSIPX API consists of the following elements:

The API environment that reflects the current state and capabilities of the NWSIPX API on your local system

Service primitive functions that make the services of the IPX and SPX protocols available to API users

Sockets which are the network addressable service access points of the IPX/SPX protocol stack

Connection endpoints that provide the processing context for connection-oriented services

NWTCBs that are used to exchange information between client applications and NWSIPX Services

There are parameters associated with some of the API elements that describe and affect the operation of the NWSIPX API. The following NWSIPX API elements have parameters associated with them:

The API environment

The NWTCB

Sockets

Connection endpoints

A unique data structure is defined for each API element supporting parameters. These structures are used to retrieve and set an element's parameter values. Client applications can retrieve and change these parameters by calling **NWSipxGetInformation** and **NWSipxSetInformation**.

To retrieve a copy of the parameters of an API element, you must take the following steps:

1. **Retrieve the parameter information by calling NWSipxGetInformation.**
2. **When the parameter information is no longer needed, free the data structure holding the parameter information by calling NWSipxFreeInformation.**

To change the parameters of an API element, you must take the following steps:

1. **Retrieve the parameter information by calling NWSipxGetInformation.**
2. **Set the desired fields in the returned structure.**
3. **Make the changes to the API elements parameters by calling NWSipxSetInformation.**
4. **Free the data structure holding the parameter information by calling NWSipxFreeInformation.**

The following table lists the data structures that are associated with each of the API elements:

API Element	**Associated Data Structure**
API Environment	SIPX_API_INFO
NWTCB	SIPX_NWTCB_INFO
Sockets	SIPX_SOCKET_INFO
Connection endpoints	SIPX_CONN_INFO

For a description of the NWSIPX API element parameters, see SIPX_API_INFO, SIPX_NWTCB_INFO, SIPX_SOCKET_INFO, and SIPX_CONN_INFO in the *Structure Reference*.

NWSIPX Environment Management

NWSIPX Services maintains a set of API element parameters that describe the API environment and its operations. These parameters are retrieved by calling **NWSipxGetInformation** with the *infoType* parameter set to SIPX_API_INFORMATION. These parameters are read-only and cannot be set.

The API element parameters are returned in a SIPX_API_INFO structure. See SIPX_API_INFO.

NWSIPX Overview

With the advent and general availability of OS platforms supporting 32-bit addressing, the previous 16-bit IPX/SPX™ APIs are insufficient to take advantage of the new 32-bit computing environment. The NWSIPX API is a 32-bit interface that is rich enough in functionality to operate on all currently known OS platforms, allowing client application developers to implement to a single IPX/SPX interface for all supported client OS platforms. While Novell® will continue to support the other 16-bit IPX/SPX APIs, the NWSIPX API is the standard 32-bit IPX/SPX API for all 32-bit client OS platforms.

The NWSIPX API is a 32-bit, user-mode API that provides access to all of the services available with the IPX and SPX protocols. Although the NWSIPX API is similar in capabilities to its predecessors, it is a departure from the earlier APIs that provided separate interfaces for IPX and SPX. The NWSIPX API provides a single interface to both IPX and SPX, freeing applications from the complexity of correlating and coordinating the events from both the IPX and SPX services.

The NWSIPX API allows you to view IPX/SPX communication as sending messages or datagrams over a connection-oriented or datagram service. The API frees the applications from the details that are specific to the IPX and SPX protocols. Internally, the API uses SPX for connection management and providing reliable data transfer with flow control. The API uses IPX to provide connectionless, unreliable data transfer without flow control.

By removing many of the protocol-specific details, the NWSIPX API also reduces the complexity of client applications. For example, you are no longer required to build the IPX and SPX protocol headers to send with their data messages. Instead, the NWSIPX API adds all necessary protocol headers that the you would normally have to create.

> **NOTE:** While the NWSIPX API provides a simplified interface to IPX/SPX, it is not a protocol-independent interface. Developers who need platform and network protocol independence should look to the

Winsock and TLI/XTI interfaces, which Novell supports.

In providing protocol independence, transport independent APIs sometimes produce inefficiencies and make generalizations that can hide or detract from the specialized features of a given protocol. As a result, some developers prefer to use transport-specific APIs that expose all of the features of a protocol suite and facilitate their use. The NWSIPX API is a 32-bit service interface that provides direct and full access to all of the services available with the IPX and SPX protocols, while exposing the services with a simplified interface.

The new features of the NWSIPX API are listed below:

A new interface control block, called the NetWare® Transport Control Block (NWTCB), is used to transfer information between the your application and the API. NWTCB management (allocating and freeing) is provided through the API. The Event Control Blocks (ECBs) utilized by previous IPX and SPX services are not supported.

You are no longer required to build the IPX and SPX protocol headers to send with their data messages. Instead, the NWSIPX API adds all necessary protocol headers that the you would otherwise have to create.

When using datagram services, you are no longer required to determine and maintain the address of the next hop in the network path to a remote system (also known as the local target). Whenever datagrams are sent to a remote system, the API determines and maintains the address of the local target. If the next hop in the network path to the remote system cannot be ascertained by the API, you are notified. Even though you are not required to initially determine the local target address, you must request the API to verify the local target address whenever it appears that the path to the remote system is no longer valid. Local target processing applies to datagram services only.

When using the NWSIPX connection-oriented service, you can select between streaming and message mode when receiving data. In streaming mode, you receive message fragments as they arrive from the network and are responsible for putting them into a meaningful context. In message mode, only whole messages are received unless the receive buffer is too small to contain the whole message. In this case, you must call the API again to receive the rest of the message. (The API might discard the remainder of the message if it doesn't have the resources to buffer the excess.) You can also select between fragmentation mode and transparent mode when sending data. In fragmentation mode, the API will automatically fragment the data message for delivery across the network to the peer system as needed. In transparent mode, the API will send the data message as received from your application and will not fragment the data message further.

You are responsible for segmenting and reassembling data messages larger than the maximum message size supported by the underlying network or transport layer. You must query the API for the maximum size supported for the socket or connection endpoint through which the

data message will be sent or received. The maximum message size associated with each socket or connection endpoint can vary, depending upon the type of networking media the data message must traverse.

For connection endpoints, the maximum message size can be unlimited, in which case, the API will automatically fragment the message as needed for delivery across the wire.

For both connection-oriented and datagram services, you can select, on a per request basis, one of several methods for event notification. These methods include API-managed events, multiplexed API-managed events, callback functions, polling, blocking, and user-managed events. The desired event notification method is selected when an NWTCB is allocated and can be changed any time the NWTCB is not in use. You are free to employ whatever method of event notification best fits its need for a given task.

The NWSIPX API supports simultaneous attachment to multiple networks and/or multiple attachments to the same physical network using different data encapsulation methods, such as Ethernet 802.2 and Ethernet II. Each network attachment is referred to as a subnetwork. You are not required to select or be aware of which subnetwork must be used for communication with a remote system. The NWSIPX API does this automatically for you. However, the NWSIPX API does allow you to specify the subnetwork to use.

The NWSIPX API supports other services beyond the scope of datagram and connection-oriented services. These additional services include canceling pending requests, timer services, determining local network addresses, advertising and querying for services, and determining if an NWSIPX status code indicates an error. These supplementary services have been grouped together and are referred to as the NWSIPX API toolbox.

The following sections describe how to use the features of the NWSIPX API.

NWTCB Management

The NWTCB is used to transfer information between your application and the NWSIPX API. Client applications use the NWTCB to store information that is needed by the API to properly satisfy a service request. The API stores the results of the service request in the NWTCB when the request completes. Client applications allocate NWTCBs by calling **NWSipxAllocControlBlock**. Applications free NWTCBs by calling **NWSipxFreeControlBlock**.

NWSIPX Services maintains a set of API element parameters for each NWTCB. These parameters are retrieved by calling **NWSipxGetInformation** with the *infoType* parameter set to SIPX_NWTCB_INFORMATION.

The API element parameters are returned in a SIPX_NWTCB_INFO structure. For a description of the NWTCB element parameters and the SIPX_API_INFO structure, see SIPX_API_INFO.

The following functions are used for control block management:

Function	Header	Purpose
NWSipxAllocControlBlock	nwsipx32.h	Allocates an NWTCB.
NWSipxChangeControlBlock	nwsipx32.h	Changes the event notification method of an NWTCB.
NWSipxFreeControlBlock	nwsipx32.h	Frees an NWTCB.

Socket Management in NWSIPX

IPX sockets are the network-addressable service access points of the IPX/SPX protocol stack. All communication with peer applications using the IPX and SPX protocols must be sent to the net.node.socket address of the destination system. As a result, you must open a socket to receive messages from other network applications. Explicit management of sockets is required only when using datagram services. Sockets used by connection endpoints are automatically opened when the connection endpoint is opened.

When opening a socket, you can specify the socket number to open or allow the API to dynamically select an unused socket number. A socket whose number has been specified by a client is known as a static socket, while a socket whose number has been selected by the NWSIPX API is known as a dynamic socket. All sockets are opened as short-lived sockets, meaning they are closed when the application terminates.

You open IPX sockets by calling **NWSipxOpenSocket**, which blocks until the request has completed. The return value indicates success or error in opening the socket. You close IPX sockets by calling **NWSipxCloseSocket**.

NWSIPX Services maintains a set of socket-element parameters for each socket. These parameters are retrieved by calling **NWSipxGetInformation** with the *infoType* parameter set to SIPX_SOCKET_INFORMATION.

The socket element parameters are returned in an SIPX_SOCKET_INFO structure.

The following functions are used for socket management:

Function	Header	Purpose
NWSipxCloseSocket	nwsipx32.h	Closes an open IPX socket.
NWSipxOpenSocket	nwsipx32.h	Opens an IPX socket.

Communication Service Group

NWSIPX: Functions

NWSipxAcceptConnection

NWSipxAcceptConnection is obsolete. See NWSipxAcceptConnectionEx.

Platform: OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxAcceptConnection (
   PNWTCB            pNwtcb,
   SIPXCONN_HANDLE   hAcceptHandle);
```

Parameters

pNwtcb

(IN) Points to the NetWare® Transport Control Block (NWTCB) for this request.

hAcceptHandle

(IN) Specifies the handle of the connection endpoint to be used for this connection.

Return Values

SIPX_ACCESS_VIOLATION	The pointer to the NWTCB is invalid.
SIPX_INVALID_CONNECTION_HANDLE	An invalid *hAcceptHandle* was specified, or the listening connection endpoint handle in the NWTCB is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM™ are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	Data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_STATE	The listening connection endpoint is not in SIPX_WAITING_ACCEPT state.

SIPX_INVALID_SYNC_TYPE	Cannot block within a callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	Request accepted without error and is in progress.
SPX_PRIMITIVE_NOT_SUPPORTED	**NWSipxAcceptConnection** is not supported on this platform.
SIPX_SUCCESSFUL	Request accepted without error.

Final Status Values

SPX_SUCCESSFUL	The request completed without error.
SIPX_CANCELED	The request was canceled.
SIPX_CONNECTION_TERMINATED	The connection indication terminated before it could be accepted.
SIPX_INSUFFICIENT_RESOURCES	System resources required for this request are unavailable.
SIPX_INTERNAL_ERROR	Internal system error occurred and connection was terminated.

Remarks

WARNING: NWSipxAcceptConnection is not supported on all OS platforms. For backwards compatibility, all OS platforms will accept calls to NWSipxAcceptConnection, but those OS platforms that do not support it will return the SIPX_PRIMITIVE_NOT_SUPPORTED error status. NWSipxAcceptConnectionEx is supported by all OS platforms and should be used if cross-platform support is required.

NWSipxAcceptConnection accepts an incoming connection. The user can specify the connection endpoint on which to accept the connection.

NWSipxAcceptConnection is used only if the SIPX_LISTEN_DELAY_ACCEPT flag was set when **NWSipxListenForConnection** was called.

The NWTCB pointed to by *pNwtcb* must be the same as was used with **NWSipxListenForConnection**.

If NULL is specified for *hAcceptHandle* the listening connection endpoint

is used for the connection.

To reject an incoming connection, call **NWSipxTerminateConnection**.

The following NWTCB field must be set before calling **NWSipxTerminateConnection**:

TCBConnHandle must contain the handle of the listening connection endpoint.

The following NWTCB field is set upon completion:

TCBFinalStatus indicates the final status of the request.

See Also

NWSipxAcceptConnectionEx, **NWSipxAllocControlBlock**, **NWSipxListenForConnection**, **NWSipxTerminateConnection**

NWSipxAcceptConnectionEx

Accepts an incoming connection
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxAcceptConnectionEx (
   PNWTCB            pNwtcb,
   PSIPXCONN_HANDLE   pAcceptHandle);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for this request. This must be the same NWTCB used with **NWSipxListenForConnection**.

hAcceptHandle

(OUT) Receives the handle of the connection endpoint on which the connection is accepted. If NULL, the accept connection-endpoint will not be stored.

Return Values

SIPX_ACCESS_VIOLATION	The pointer to the NWTCB is invalid.
SIPX_INVALID_CONNECTION_HANDLE	Listening connection-endpoint handle in the NWTCB is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	Data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_PARAMETER	Invalid *pAcceptHandle* specified.
SIPX_INVALID_STATE	The listening connection endpoint is not in SIPX_WAITING_ACCEPT state.

SIPX_INVALID_SYNC_TYPE	Cannot block within a callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	Request accepted without error and is in progress.
SIPX_SUCCESSFUL	Request accepted without error.

Final Status Values

SPX_SUCCESSFUL	The request completed without error.
SIPX_CANCELED	The request was canceled.
SIPX_CONNECTION_TERMINATED	The connection indication terminated before it could be accepted.
SIPX_INSUFFICIENT_RESOURCES	System resources required for this request are unavailable.
SIPX_INTERNAL_ERROR	Internal system error occurred and connection was terminated.

Remarks

The connection endpoint on which the connection is accepted is opened automatically and its handle returned to the caller.

NWSipxAcceptConnection is used only if the SIPX_LISTEN_DELAY_ACCEPT flag was set when **NWSipxListenForConnection** was called.

To reject an incoming connection, call **NWSipxTerminateConnection**.

It is the responsibility of your application to close the accept connection endpoint.

The following NWTCB flag may be set in the *TCBFlags* field.

SIPX_ACCEPT_ON_LISTEN_ENDPOINT-The connection will be accepted on the listening connection endpoint.

The following NWTCB field must be set before calling **NWSipxTerminateConnection**:

TCBConnHandle must contain the handle of the listening connection

endpoint.

The following NWTCB field is set upon completion:

TCBFinalStatus indicates the final status of the request.

See Also

NWSipxAcceptConnection, **NWSipxAllocControlBlock**, **NWSipxListenForConnection**, **NWSipxTerminateConnection**

NWSipxAdvertiseService

Initiates the advertisement of a client application service
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxAdvertiseService (
   nuint16           serviceType,
   pnstr8            pServiceName,
   SIPXSOCK_HANDLE   hSockHandle);
```

Parameters

serviceType

(IN) Specifies the class of service as assigned by Novell®; for example:

```
4 OT_FILE_SERVER
5 OT_JOB_SERVER
6 OT_PRINT_SERVER
```

pServiceName

(IN) Points to a NULL-terminated string containing the unique name of the server within the network.

hSockHandle

(IN) Specifies the handle of the previously opened socket through which the advertised service is accessible.

Return Values

SIPX_INSUFFICIENT_RESOURCES	System resources required to satisfy this request are unavailable.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_PARAMETER	The *pServiceName* pointer is invalid.
SIPX_INVALID_SERVICE_NAM	The service name did not meet

E	proper criteria. (See service name criteria in the Remarks section.)
SIPX_INVALID_SOCKET_HANDLE	The socket handle is not valid.
SIPX_SERVICE_ALREADY_ACTIVE	A service by this name and type is already active.
SIPX_SOCKET_IN_USE	The socket is already in use by a server.
SIPX_SUCCESSFUL	The request completed without error.
SIPX_UNSUCCESSFUL	Advertiser not available.

Remarks

The *serviceType* parameter value must be formatted in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

The service name specified by *pServiceName* must conform to the following criteria:

Maximum name length is 48 characters, including NULL.

The name can contain any character between 21h (!) and 7Fh (DEL), **excluding** the slash, backslash, colon, semicolon, comma, asterisk, question mark, plus, and minus characters (/\:;,*?+-).

The service type is broadcast on the network every 60 seconds until canceled by **NWSipxCancelAdvertiseService**.

If a service query request is received from the network, the NWSIPX API automatically answers the query as long as the advertise service request is active.

See Also

NWSipxCancelAdvertiseService, **NWSipxQueryServices**

NWSipxAllocControlBlock

Allocates an NWTCB
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxAllocControlBlock (
   nuint32   syncType,
   nptr      pEventInfo,
   PPNWTCB   ppNwtcb);
```

Parameters

syncType

(IN) Indicates the type of event notification associated with this NWTCB.

pEventInfo

(IN) Points to a variable containing supplementary information of differing types depending on the value of *syncType*.

ppNwtcb

(OUT) Points to a variable of type PNWTCB where the pointer to the allocated NWTCB will be stored.

Return Values

SIPX_ACCESS_VIOLATION	An invalid handle was specified in *pEventInfo*.
SIPX_INSUFFICIENT_RESOURCES	Required system resources are unavailable.
SIPX_INVALID_HANDLE	An invalid handle specified in *pEventInfo*.
SIPX_INVALID_MUX_GROUP_HANDLE	An invalid NWTCB mux group handle specified in *pEventInfo*.
SIPX_INVALID_PARAMETER	Either the value of *ppNwtcb* is invalid, or the information in *pEventInfo* is invalid.
SIPX_INVALID_SYNC_TYPE	An invalid event notification type was specified.

SIPX_SUCCESSFUL	The request completed without error.

Remarks

To free the NWTCB, call **NWSipxFreeControlBlock**.

syncType indicates the type of notification to associate with the NWTCB. *pEventInfo* provides supplementary information that is determined by the notification type. The following table lists the valid values for*syncType* and specifies the associated information that must be supplied by *pEventInfo*:

***syncType* Value**	***syncType* Description**	***pEventInfo* Value**
SIPX_USER_EVENT	Your event object will be signaled upon completion of the request.	The handle of the event object
SIPX_CALLBACK	Your callback function will be called upon request completion.	The address of your callback function. Every callback function has the prototype of : `void (N_CDECL Callback_Function) (*NWTCB)`
SIPX_BLOCKING	The client application is blocked until the request completes.	NULL
SIPX_POLLING	You will determine request completion by calling **NWSipxPoll**	NULL
SIPX_API_EVENT	You will determine request completion by calling **NWSipxWaitForSingleEvent**	NULL
SIPX_API_MUX_EVENT	You will determine request completion by calling **NWSipxWaitForMultipleEvents**.	A pointer to a variable of type SIPXMUXGRP_HANDLE that contains the handle of the mux group to which this NWTCB will be assigned. If a value of SIPX_ALLOC_MUX_GROUP is specified, the

		API creates a new mux group and stores its handle into the variable pointed to by this parameter.

The event notification method of an NWTCB can be changed by calling **NWSipxChangeControlBlock**, or by modifying *NIEventType* and *NIEventInfo* element parameters of the NWTCB with **NWSipxSetInformation**.

See Also

NWSipxChangeControlBlock, **NWSipxFreeControlBlock**, **NWSipxPoll**, **NWSipxSetInformation**, **NWSipxWaitForSingleEvent**, **NWSipxWaitForMultipleEvents**

NWSipxCancelAdvertiseService

Cancels an active advertise service request
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxCancelAdvertiseService (
   nuint16           serviceType,
   pnstr8            pServiceName,
   SIPXSOCK_HANDLE   hSockHandle);
```

Parameters

serviceType

(IN) Specifies the class of service as assigned by Novell; for example:

```
4 OT_FILE_SERVER
5 OT_JOB_SERVER
6 OT_PRINT_SERVER
```

pServiceName

(IN) Points to a NULL-terminated string containing the unique name of the server within the network.

hSockHandle

(IN) Specifies the handle of the socket being used by the service to be canceled.

Return Values

SPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_PARAMETER	The *pServiceName* parameter is invalid.
SIPX_INVALID_SOCKET_HANDLE	The socket handle specified by the *hSocketHandle* parameter is invalid.
SIPX_SERVICE_NOT_ACTIVE	The service being canceled is not

	currently active.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

This function deletes a service from the network.

The *serviceType* parameter value must be formatted in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

The service name specified by *pServiceName* must conform to the following criteria:

Maximum name length is 48 characters, including NULL.

The name can contain any character between 21h (!) and 7Fh (DEL), **excluding** the slash, backslash, colon, semicolon, comma, asterisk, question mark, plus, and minus characters (/\:;,*?+-).

See Also

NWSipxAdvertiseService

NWSipxCancelPendingRequest

Cancels a pending request
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxCancelPendingRequest (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB being used by the pending request.

Return Values

SIPX_CANNOT_CANCEL	The NWTCB cannot be canceled.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_REQUEST_NOT_PENDING	No request is pending.
SIPX_SUCCESSFUL	Request completed without error.

Remarks

When **NWSipxCancelPendingRequest** returns, the status SIPX_REQ_CANCELED is stored in the *TCBFinalStatus* field of the NWTCB and the event notification method specified in the NWTCB is invoked.

Do not modify any of the flags or fields in the NWTCB pointed to by *pNwtcb*, since the NWTCB is currently being used for a service request by another function.

See Also

NWSipxAcceptConnectionEx, **NWSipxEstablishConnection**, **NWSipxListenForConnection**, **NWSipxQueryServices**, **NWSipxReceiveDatagram**, **NWSipxReceiveMessage**, **NWSipxRegisterForTransportEvent**, **NWSipxSendDatagram**, **NWSipxSendMessage**

NWSipxChangeControlBlock

Changes the event notification method of an NWTCB
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxChangeControlBlock (
   nuint32   syncType,
   nptr      pEventInfo,
   PNWTCB    pNwtcb);
```

Parameters

syncType

(IN) Specifies the new event notification type for this NWTCB.

pEventInfo

(IN) Specifies supplementary information of differing types, depending on the value of *syncType*.

pNwtcb

(IN) Points to the NWTCB that is to be changed.

Return Values

SIPX_ACCESS_VIOLATION	An invalid pointer was specified by *pEventInfo*.
SIPX_INSUFFICIENT_RESOURCES	Required system resources are unavailable.
SIPX_INVALID_HANDLE	An invalid handle was specified by *pEventInfo*.
SIPX_INVALID_MUX_GROUP_HANDLE	An invalid NWTCB mux group handle was specified by *pEventInfo*.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is invalid.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.

SIPX_INVALID_PARAMETER	There is either invalid *ppNwtcb* parameter or invalid information in *pEventInfo*.
SIPX_INVALID_SYNC_TYPE	An invalid event notification type was specified.
SIPX_NWTCB_IN_USE	The NWTCB is in use by another function.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

The NWTCB specified by *pNwtcb* must not be in use by any other function when **NWSipxCancelPendingRequest** is called. In addition, the *TCBFlags* field of the NWTCB must be set to zero before calling **NWSipxCancelPendingRequest**.

syncType indicates the type of notification to associate with the NWTCB. *pEventInfo* provides supplementary information that is determined by the notification type. The following table lists the valid values for*syncType* and specifies the associated information that must be supplied by *pEventInfo*:

***syncType* Value**	***syncType* Description**	***pEventInfo* Value**
SIPX_USER_EVENT	Your event object will be signaled upon completion of the request.	The handle of the event object
SIPX_CALLBACK	Your callback function will be called upon request completion.	The address of your callback function. Every callback function has the prototype of : `void (N_CDECL Callback_Function)(*NWTCB)`
SIPX_BLOCKING	The client application is blocked until the request completes.	NULL
SIPX_POLLING	You will determine request completion by calling **NWSipxPoll**	NULL
SIPX_API_EVENT	You will determine request completion by calling **NWSipxWaitForSingle**	NULL

	Event	
SIPX_API_MUX_EVENT	You will determine request completion by calling **NWSipxWaitForMultipleEvents**.	A pointer to a variable of type SIPXMUXGRP_HANDLE that contains the handle of the mux group to which this NWTCB will be assigned. If a value of SIPX_ALLOC_MUX_GROUP is specified, the API creates a new mux group and stores its handle into the variable pointed to by this parameter.

See Also

NWSipxAllocControlBlock

NWSipxCloseConnectionEndpoint

Closes an open connection endpoint
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxCloseConnectionEndpoint (
   SIPXCONN_HANDLE   hConnHandle);
```

Parameters

hConnHandle

(IN) Specifies the handle of the connection endpoint to be closed.

Return Values

SIPX_INTERNAL_ERROR	An internal system error occurred.
SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

If there is a transport connection active on this connection endpoint, it is aborted before the connection endpoint is closed.

The dynamic socket associated with the connection endpoint is automatically closed as part of closing the connection endpoint.

Any outstanding **NWSipxReceiveMessage** requests are cancelled with a final status of SIPX_CANCELED stored in the *TCBFinalStatus* field of the NWTCB.

See Also

NWSipxOpenConnectionEndpoint

NWSipxCloseSocket

Closes an open IPX socket
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxCloseSocket (
   SIPXSOCK_HANDLE   hSockHandle);
```

Parameters

hSockHandle

(IN) Specifies the handle of the socket being closed.

Return Values

SIPX_INTERNAL_ERROR	An Internal system error occurred.
SIPX_INVALID_SOCKET_HANDLE	The socket handle is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

Any active **NWSipxReceiveDatagram** or **NWSipxQueryServices** requests are terminated with a final status of SIPX_CANCELED stored in the *TCBFinalStatus* field of the NWTCB.

NOTE: Sockets opened by **NWSipxOpenConnectionEndpoint** cannot be closed using **NWSipxCloseSocket**.

See Also

NWSipxOpenSocket

NWSipxEstablishConnection

Establishes a transport connection with a peer client application
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

nuint32 NWSipxEstablishConnection (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

Return Values

SIPX_INSUFFICIENT_RESOURCES	System resources required to satisfy this request are unavailable.
SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in the NWTCB is not valid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NETWORK_ADDRESS	An invalid destination network address is specified in the *TCBRemoteAddress* field of the NWTCB.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_STATE	The connection endpoint is not in the SIPX_ALLOCATED state.
SIPX_INVALID_SYNC_TYPE	You cannot block within a callback function.

SIPX_MEMORY_LOCK_ERROR	System error: could not lock memory.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	Request accepted without error and is in progress.
SIPX_SUCCESSFUL	The request completed without error.

Final Status Values

SIPX_CANCELLED	Request canceled.
SIPX_INSUFFICIENT_RESOURCES	System resources required to satisfy this request are unavailable.
SIPX_INTERNAL_ERROR	Internal system error.
SIPX_NO_RESPONSE_FROM_TARGET	The destination system did not respond.
SIPX_NO_ROUTE_TO_TARGET	A network route to the remote system identified by the *TCBRemoteAddress* field of the NWTCB does not exist.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

Before calling **NWSipxEstablishConnection**, the following NWTCB fields must be set:

TCBConnHandle, indicating the handle of the connection endpoint for this connection.

NOTE: When establishing a connection, the NWSIPX API for Windows NT* ignores the subnetwork specified by the connection endpoint and uses the subnetwork, providing the best route to the destination node.

TCBRemoteAddress, indicating the network address of the destination node (in network byte order).

In addition, the following NWTCB flag can be set as follows:

SIPX_CONNECT_NO_WATCHDOG, disabling the watchdog function.

NOTE: The NWSIPX API for Windows NT does not allow the

NOTE: The NWSIPX API for Windows NT does not allow the watchdog function to be disabled.

The following NWTCB fields are set upon completion:

TCBFinalStatus, indicating the final status of the request.

TCBRemoteAddress, indicating the actual network address of the destination node (in network byte order).

NOTE: The network address components must be specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order. When specifying an IPX network address, you must set the *NAType* field to TA_IPX_SPX and the *NALength* field to the value of **sizeof**(IPXADDR).

See Also

NWSipxCloseConnectionEndpoint, **NWSipxOpenConnectionEndpoint**, **NWSipxSetInformation**

NWSipxFreeControlBlock

Frees an NWTCB
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxFreeControlBlock (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB that is to be freed.

Return Values

SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_NWTCB_IN_USE	The NWTCB is in use by another function.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

The NWTCB must not be in use by another function when the NWTCB is freed.

If you use multiple threads that access the same NWTCB, you must ensure that none of the threads try to access the NWTCB after it has been freed.

The TCBFlags field of the NWTCB must be zero.

Communication Service Group

See Also

NWSipxAllocControlBlock

NWSipxFreeInformation

Frees a previously allocated API element information structure
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

nuint32 NWSipxFreeInformation (
   nptr   pInfoStruct);
```

Parameters

pInfoStruct
(IN) Points to the information structure to be freed.

Return Values

SIPX_INVALID_PARAMETER	*pInfoStruct* does not point to an API element information structure.
SIPX_SUCCESSFUL	The request completed without failure.

Remarks

The information structure being freed must have been allocated by **NWSipxGetInformation**.

If you use multiple threads that access the same information structure, you must ensure that none of the threads try to access the information structure after it has been freed.

See Also

NWSipxGetInformation, **NWSipxSetInformation**

NWSipxGetInformation

Returns current information about an NWSIPX API element
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxGetInformation (
   nuint32    infoType,
   nptr       hHandle,
   pnptr      ppInfoStruct,
   pnuint32   pInfoStructLen);
```

Parameters

infoType

(IN) Specifies the type of API element information being requested.

hHandle

(IN) Specifies different types of handles, depending upon the information type specified by *infoType*.

ppInfoStruct

(OUT) Points to a variable of type nptr where a pointer to the information structure of the API element is to be stored. The information structure returned is initialized with the current parameter settings for the requested API element.

pInfoStructLen

(OUT) Points to the location where the length of the information structure is to be stored.

Return Values

SIPX_INSUFFICIENT_RESOURCES	Insufficient system resources are available to satisfy this request.
SIPX_INVALID_CONNECTION_HANDLE	An invalid connection endpoint handle specified in *hHandle*.
SIPX_INVALID_INFO_TYPE	An invalid information type specified in *infoType*.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.

SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_PARAMETER	An invalid pointer specified in *ppInfoStruct* or *pInfoStructLen*.
SIPX_INVALID_NWTCB	An invalid NWTCB specified in *hHandle*.
SIPX_INVALID_SOCKET_HANDLE	An invalid socket handle specified in *hHandle*.
SIPX_SUCCESSFUL	The request completed without failure.

Remarks

The API element information structure must be freed by calling **NWSipxFreeInformation**.

Valid *infoType* values include:

```
SIPX_API_INFORMATION      API environment
SIPX_SOCKET_INFORMATION   Socket
SIPX_CONN_INFORMATION     Connection endpoint
SIPX_NWTCB_INFORMATION    NWTCB
```

Allowed structure types for *ppInfoStruct* include:

```
SIPX_API_INFO      API environment
SIPX_SOCKET_INFO   Socket
SIPX_CONN_INFO     Connection endpoint
SIPX_NWTCB_INFO    NWTCB
```

The following table shows the relationship between the value of the *infoType* parameter and the type of allocated structure whose address is retuned in *ppInfoStruct*.

infoType	**ppInfoStruct Structure**
SIPX_API_INFORMATION	SIPX_API_INFO
SIPX_CONN_INFORMATION	SIPX_CONN_INFO
SIPX_NWTCB_INFORMATION	SIPX_NWTCB_INFO
SIPX_SOCKET_INFORMATION	SIPX_SOCKET_INFO

The type of information specified by the *hHandle* is determined by the

value specified by the *infoType*. The following table shows this relationship.

infoType	hHandle
SIPX_API_INFORMATION	NULL
SIPX_CONN_INFORMATION	Handle of a connection endpoint
SIPX_NWTCB_INFORMATION	Pointer to an NWTCB
SIPX_SOCKET_INFORMATION	Handle of a socket

See Also

NWSipxFreeInformation, **NWSipxSetInformation**

NWSipxGetInternetAddress

Returns the local network address of a socket
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxGetInternetAddress (
   SIPXSOCK_HANDLE   hSockHandle,
   PNETADDR          pNetAddress);
```

Parameters

hSockHandle

(IN) Indicates the handle of the target socket.

pNetAddress

(OUT) Points to a NETADDR structure where the internetwork address is to be stored.

Return Values

SIPX_INSUFFICIENT_RESOURCES	Insufficient system resources are available to satisfy this request.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_PARAMETER	An invalid pointer was specified in *pNetAddress*.
SIPX_INVALID_SOCKET_HANDLE	The socket handle specified in *hSockHandle* is invalid.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

You must allocate space for the structure whose address is pointed to by *pNetAddress*.

Only the net and node portions of the address are returned.

The internetwork address of the subnetwork currently associated with the socket is returned.

NWSipxGetMaxNsduSize

Returns the maximum data packet size that can be sent as a datagram for a specific socket

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxGetMaxNsduSize (
   SIPXSOCK_HANDLE   hSockHandle);
```

Parameters

hSockHandle

(IN) Specifies the handle of the target socket.

Return Values

zero	An invalid socket handle was specified
nonzero	The maximum size of data packet

Remarks

NWSipxGetMaxNsduSize is used with datagram services only.

The maximum NSDU size supported by the subnetwork associated with the socket is returned.

See Also

NWSipxReceiveDatagram, **NWSipxSendDatagram**

NWSipxGetMaxTsduSize

Returns the maximum data packet size that can be sent as a connection-oriented message on a specific connection endpoint

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxGetMaxTsduSize (
   SIPXCONN_HANDLE   hConnHandle);
```

Parameters

hConnHandle

(IN) Specifies the handle of the target connection endpoint.

Return Values

zero	An invalid connection endpoint handle was specified
SIPX_CONN_UNLIMITED_TSDU	An unlimited TSDU size was specified.
nonzero other than the above value	The maximum size of data packet

Remarks

NWSipxGetMaxTsduSize is used only with connection-oriented services.

See Also

NWSipxReceiveMessage, **NWSipxSendMessage**

NWSipxListenForConnection

Listens for incoming connections
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxListenForConnection (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for this request.

Return Values

SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in the NWTCB is not valid.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_STATE	The connection endpoint is not in the SIPX_ALLOCATED state.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	The request was accepted without error and is in progress.
SIPX_SUCCESSFUL	The request completed without

	error.

Final Status Values

SIPX_CANCELED	The request was canceled, or the connection endpoint was closed.
SIPX_INSUFFICIENT_RESOURCES	System resources are not available for this request.
SIPX_INTERNAL_ERROR	Internal system error.
SIPX_SUCCESSFUL	Request completed without error.

Remarks

The incoming connection is acknowledged and accepted immediately on the listening connection endpoint unless the SIPX_LISTEN_DELAY_ACCEPT flag is set.

The following NWTCB field must be set before calling **NWSipxListenForConnection**:

TCBConnHandle indicating the handle of the connection endpoint on which to listen.

The following NWTCB flags can be set:

`SIPX_LISTEN_NO_WATCHDOG`, disabling the watchdog function.

NOTE: The NWSIPX API for Windows NT does not allow the watchdog function to be disabled.

`SIPX_LISTEN_DELAY_ACCEPT`, indicating don't acccept a connection until you call **NWSipxAcceptConnectionEx**.

The following NWTCB fields are set upon completion:

TCBFinalStatus, indicating the final status of the request.

TCBRemoteAddress, indicating the network address of the peer application requesting a connection. (The address is in network byte order.)

TCBSubnetworkHandle, indicating the subnetwork through which the connection request was received.

See Also

NWSipxAcceptConnectionEx, **NWSipxAllocControlBlock**

NWSipxOpenConnectionEndpoint

Opens a connection endpoint
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxOpenConnectionEndpoint (
  SIPXSUBNET_HANDLE   hSubnetworkHandle,
  pnuint16            pSocket,
  PSIPXCONN_HANDLE    pConnHandle);
```

Parameters

hSubnetworkHandle

(IN) Specifies the handle of the subnetwork to use for this connection endpoint. If SIPX_DEFAULT_SUBNETWORK is specified, the default subnetwork is used.

pSocket

(IN/OUT) On input, points to the data location containing the number of the socket to open for the connection endpoint. On output, the data location contains the number of the socket that was actually opened.

pConnHandle

(OUT) Points to the location where the connection handle is to be stored.

Return Values

SIPX_INSUFFICIENT_RESOURCES	System resources are not available for this request.
SIPX_INTERNAL_ERROR	An internal system error has occurred.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_PARAMETER	*pConnHandle* is an invalid pointer.
SIPX_INVALID_SUBNETWORK_HANDLE	An invalid subnetwork handle was specified in

	hSubnetworkHandle.
SIPX_MEMORY_LOCK_ERROR	System cannot lock memory.
SIPX_NO_SUBNETS_BOUND_TO_IPX	A subnetwork is not available for use by IPX.
SIPX_SOCKET_IN_USE	Socket in use.
SIPX_SUCCESSFUL	Request completed without error.

Remarks

This function blocks until the request has completed.

If successful, **NWSipxOpenConnectionEndpoint** allocates and initializes a connection endpoint, assigns a connection endpoint handle, and associates the connection endpoint with an IPX socket. It does not establish a transport connection.

On input, the value pointed to by *pSocket* can be used to specify a specific socket or to request a dynamic socket be opened. If the value pointed to by *pSocket* is a nonzero value that value is used as the number of the socket to open. If the value is 0, a dynamic socket is opened.

NOTE: The socket number must be specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

The handle of the socket opened for the connection endpoint can be obtained from the connection endpoint parameter *CISocketHandle*.

See Also

NWSipxCloseConnectionEndpoint, **NWSipxGetInformation**, **NWSipxSetInformation**

NWSipxOpenSocket

Opens an IPX socket
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxOpenSocket (
   nflag32              SocketAttributes,
   SIPXSUBNET_HANDLE    hSubnetworkHandle,
   pnuint16             pSocket,
   PSIPXSOCK_HANDLE     pSockHandle);
```

Parameters

SocketAttributes

(IN) Specifies flags indicating the desired socket attributes. The following flag can be set:

```
SIPX_SOCK_CHECKSUM
```

hSubnetworkHandle

(IN) Specifies the handle of the subnetwork to use this socket. If SIPX_DEFAULT_SUBNETWORK is specified, the default subnetwork is used.

pSocket

(IN/OUT) On input, points to the location containing the number of the socket to open. If the socket number is 0, a socket is dynamically opened for the user. On output, the data location contains the number of the socket that was actually opened.

pSockHandle

(OUT) Points to the location where the socket handle is to be stored.

Return Values

SIPX_ACCESS_VIOLATION	An invalid pointer is specified in *pSockHandle*.
SIPX_INSUFFICIENT_RESOURCES	Lack of system resources prohibit opening another socket.
SIPX_INTERNAL_ERROR	Internal system error.

SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_SUBNETWORK_HANDLE	An invalid subnetwork handle was specified.
SIPX_MEMORY_LOCK_ERROR	System cannot lock memory.
SIPX_NO_SUBNETS_BOUND_TO_IPX	A subnetwork is not available for use by IPX.
SIPX_SOCKET_IN_USE	The socket is in use.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

NWSipxOpenSocket blocks until the request has completed.

If the SIPX_SOCK_CHECKSUM flag is set in *SocketAttributes*, the API automatically generates a checksum for every packet sent over this socket.

The number of the socket that was opened is stored at the location pointed to by *pSocket*.

NOTE: The socket number must be specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

See Also

NWSipxCloseSocket

NWSipxPoll

Checks if a pending request employing the SIPX_POLLING event notification method has completed

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxPoll (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB whose completion status is pending.

Return Values

SIPX_SUCCESSFUL	The request has completed.
SIPX_PENDING	The request is still pending.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_SYNC_TYPE	The NWTCB pointed to by *pNwtcb* is not using the SPX_POLLING method of event notification.

Remarks

Do not modify any of the flags or fields in the NWTCB pointed to by *pNwtcb*, since the NWTCB might be in use for a service request by another function.

NWSipxQueryServices

Broadcasts a SAP query to discover the identities of all servers of any type, all servers of a specific type, or the nearest server of a specific type and then returns the response

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxQueryServices (
   nuint16   queryType,
   nuint16   serviceType,
   PNWTCB    pNwtcb);
```

Parameters

queryType

(IN) Specifies the type of query to be performed. The following values are valid:

```
SIPX_ALL_SERVERS
SIPX_NEAREST_SERVER
```

serviceType

(IN) Specifies the service type of interest. *serviceType* must match the class of service as assigned by Novell. For example:

```
0x0004   File server
0x0005   Job server
0x0006   Print server
0xFFFF   Wildcard
```

pNwtcb

Points to the NWTCB for the request.

Return Values

SIPX_INSUFFICIENT_RESOURCES	Lack of system resources prohibit opening another socket.
SIPX_INTERNAL_ERROR	An internal system error has occurred.
SIPX_INVALID_IOCTL_BUFFER	Versions of NWSIPX DLL and

_LEN	NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_QUERY_TYPE	An invalid query type was specified.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NO_SUBNETS_BOUND_TO_IPX	A subnetwork is not available for use by IPX.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	The request was accepted without error and is in progress.
SIPX_SUCCESSFUL	The request completed without error.

Final Status Values

SIPX_CANCELED	The request was canceled.
SIPX_PARTIAL_SERVER_INFO	The server information was larger than the allocated data space, so the server information was truncated.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

The following values are valid for *queryType*:

SPX_ALL_SERVERS	All servers of the type indicated by *serviceType*
SIPX_NEAREST_SERVER	Only the nearest server of the type indicated by*serviceType*

The query response consists of one or more SIPX_SERVICE_INFO structures stored into the data space pointed to by the fragment list of the

NWTCB. If more server information is received than fits in the data space allocated, the server information is truncated and *FinalStatus* in NWTCB contains the status of SIPX_PARTIAL_SERVER_INFO.

NOTE: The service type specified by *serviceType* must be stored in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

The following NWTCB fields that must be set before calling **NWSipxQueryServices**:

TCBFragmentCount, indicating the number of entries in the fragment list.

TCBFragmentList, pointing to the fragment list containing information about preallocated data space.

The following NWTCB field and flag can be set:

TCBSubnetworkHandle, indicating the handle of the subnetwork to use.

`SIPX_QUERY_SPECIFIC_ROUTE`, requesting the API to send the service query over the subnetwork identified by the *TCBSubnetworkHandle* NWTCB element parameter.

NOTE: For the value in *TCBSubnetworkHandle* to be used, `SIPX_QUERY_SPECIFIC_ROUTE` must be set.

The following NWTCB fields are set upon completion:

TCBBytesTransferred, indicating the byte count of the stored server information.

TCBFinalStatus, indicating the final status of the request.

TCBSubnetworkHandle, indicating the handle of the subnetwork through which the service query response was received.

The following must be true for this call to complete successfully and for there to be valid data in the return NWTCB packet:

the call returns SPIX_SUCCESSFUL

the *TCBFinalStatus* field contains SIPX_SUCCESSFUL

the *TCBBytesTransferred* field contains a value greater than 0. If this field contains a 0, the requested services were not found and **NWSipxQueryServices** timed out.

See Also

NWSipxAdvertiseService

NWSipxReceiveDatagram

Prepares to receive a datagram
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxReceiveDatagram (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

Return Values

SIPX_INVALID_FRAGMENT_COUNT	The fragment count specified in the NWTCB is invalid.
SIPX_INVALID_FRAGMENT_LIST	The fragment list or one of its fragment descriptors is bad.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVAILD_SOCKET_HANDLE	The socket handle specified by the NWTCB is not valid.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	NWTCB already in use.
SIPX_PENDING	The request was accepted without error and is in progress.

SIPX_SUCCESSFUL	The request was accepted without error.

Final Status Values

SIPX_CANCELED	The request was cancelled or socket was closed.
SIPX_DATA_OVERFLOW	The input (received) message was larger than the preallocated data space. The input message was truncated.
SIPX_INTERNAL_ERROR	An internal system error occurred.
SIPX_INVALID_SOCKET_HANDLE	The socket handle specified in the NWTCB is not valid.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

Checksum processing (checksum validation) is done automatically by the NWSIPX API.

The following NWTCB fields must be set before calling **NWSipxReceiveDatagram**:

TCBFlags, must be set to zero.

TCBFragmentCount, indicating the number of entries in the fragment list.

TCBFragmentList, pointing to the fragment list containing information about preallocated data space.

TCBSockHandle, indicating the handle of the IPX socket to monitor for data.

The following NWTCB fields are set upon completion:

TCBBytesTransferred, indicating the number of bytes received.

TCBFinalStatus, indicating the final status of the request.

TCBPacketType, indicating the value present in the *Packet Type* field of the IPX header.

TCBRemoteAddress, indicating the network address of the peer application sending the datagram.

TCBSubnetworkHandle, indicating the handle of the subnetwork through which the datagram was received.

NOTE: The network address components are specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

See Also

NWSipxSendDatagram

NWSipxReceiveMessage

Prepares to receive an input message
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxReceiveMessage (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

Return Values

SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in NWTCB is invalid.
SIPX_INVALID_FRAGMENT_COUNT	The fragment count specified in the NWTCB is invalid.
SIPX_INVALID_FRAGMENT_LIST	The fragment list or one of its fragment descriptors is bad.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	The request was accepted without error and is in progress.

SIPX_SUCCESSFUL	The request completed without error.

Final Status Values

SIPX_CANCELLED	The request was canceled.
SIPX_CONNECTION_ABORTED	The transport connection was terminated.
SIPX_DATA_OVERFLOW	The input (received) message was larger than the preallocated data space. The input message was truncated.
SIPX_INTERNAL_ERROR	Internal system error.
SIPX_SUCCESSFUL	The request was completed without error.

Remarks

The following NWTCB fields must be set before calling **NWSipxReceiveMessage**:

TCBConnHandle, indicating the handle of the connection endpoint to monitor for incoming messages.

TCBFragmentCount, indicating the number of entries in the fragment list.

TCBFragmentList, pointing to the fragment list containing information about preallocated data space.

The following NWTCB fields are set upon completion:

TCBBytesTransferred, indicating the number of bytes received.

TCBDataStreamType, indicating the value present in the *Data Stream Type* field of the SPX header.

TCBFinalStatus, indicating the final status of the request.

TCBMessageSequenceNumber, indicating message sequence number.

The following NWTCB flags are set upon completion:

SIPX_RCVMSG_ATTN	The attention flag in the SPX header was set.

SIPX_RCVMSG_PARTIAL	**Clear:** Data segment is final fragment of message. **Set:** Data segment is not final fragment of message.

See Also

NWSipxSendMessage

NWSipxRegisterForTransportEvent

Registers to receive a transport event
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxRegisterForTransportEvent (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

Return Values

SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in the NWTCB is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_INVALID_TRANSPORT_EVENT	The transport event specified in the *TransportEvent* field of the NWTCB is not valid.
SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	The request was accepted without error and is in progress.
SIPX_SUCCESSFUL	The request completed without

	error.

Final Status Values

SIPX_CANCELED	The request was canceled.
SIPX_SUCCESSFUL	The request was completed without error.

Remarks

When the transport event occurs, you will be notified according to the event notification method selected for the NWTCB.

The following NWTCB field must be set before calling **NWSIPXRegisterForTransportEvent**:

TCBFlags, must be set to zero.

TCBTransportEvent, indicating the transport event being registered for. The following event types are valid for the NWTCB's *TCBTransportEvent* field:

```
SIPX_LISTEN_FOR_DISCONNECT
SIPX_SCHEDULE_TIMER_EVENT
SIPX_SUBNET_STATUS_CHANGE
```

The following NWTCB fields may need to be set before calling **NWSIPXRegisterForTransportEvent**:

TCBConnHandle indicating the connection endpoint handle of the transport connection being monitored if the SIPX_LISTEN_FOR_DISCONNECT event type has been selected.

TCBEvent.TCBTimeout, indicating the timer value (in milliseconds) if the SIPX_SCHEDULE_TIMER_EVENT event type has been selected.

The following NWTCB field is set upon completion:

TCBFinalStatus, indicating the final status of the request.

NWSipxSendDatagram

Sends a datagram
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxSendDatagram (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

Return Values

SIPX_INVALID_FRAGMENT_COUNT	The fragment count specified in the NWTCB is invalid.
SIPX_INVALID_FRAGMENT_LIST	The fragment list or one of its fragment descriptors is bad.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NETWORK_ADDRESS	The network address specified in the *TCBRemoteAddress* field of the NWTCB is not valid.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_SOCKET_HANDLE	The socket handle specified in the *TCBSocketHandle* field of the NWTCB is not valid.
SIPX_INVALID_SYNC_TYPE	You cannot block within a callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.

SIPX_NWTCB_IN_USE	The NWTCB is already in use.
SIPX_PENDING	The request was accepted without error and is in progress.
SIPX_SUCCESSFUL	The request was completed without error.

Final Status Values

SIPX_CANCELED	The request was canceled or the socket was closed.
SIPX_CHECKSUM_NOT_SUPPORTED	Selected subnetwork does not support checksum processing.
SIPX_INTERNAL_ERROR	Internal system error.
SIPX_INVALID_SUBNETWORK_HANDLE	An invalid subnetwork handle was specified in the *TCBSubnetworkHandle* field of the NWTCB.
SIPX_NO_ROUTE_TO_TARGET	A network route to the remote system identified by the *TCBRemoteAddress* field of the NWTCB does not exist.
SIPX_NO_SUBNETS_BOUND_TO_IPX	A subnetwork is not available for use by IPX.
SIPX_SUCCESSFUL	Request accepted without error.

Remarks

The following NWTCB fields that must be set before calling **NWSipxSendDatagram**:

TCBFragmentCount, indicating the number of entries in the fragment list.

TCBFragmentList, pointing to the fragment list containing information about the data space.

TCBRemoteAddress, indicating the network address of the destination node.

TCBSockHandle, indicating the handle of the IPX socket on which to send the datagram.

If you want to send an IPX header without data, set the *TCBFragmentCount* field to zero. The API will then ignore the

TCBFragmentList field.

NOTE: The network address components must be specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

When specifying an IPX network address, you must set the *NAType* field to TA_IPX_SPX and the *NALength* field to the value of **sizeof** (IPXADDR).

The following NWTCB fields may be set before calling **NWSipxSendDatagram**:

TCBSubnetworkHandle, indicating the handle of the subnetwork to use (the SIPX_SNDDG_SPECIFIC_ROUTE flag must also be set).

TCBPacketType, indicating the value to store in the *Packet Type* field of the IPX header (the SIPX_SNDDG_PACKET_TYPE flag must be set).

The following NWTCB flags may be set before calling **NWSipxSendDatagram**:

SIPX_SNDDG_BEST_ROUTE	Select the best route using any subnetwork available.
SIPX_SNDDG_GENERATE_CHK SUM	Generate and store the checksum in the IPX header. **NOTE:** The NWSIPX API for Windows NT does not support the generation of checksums for datagrams.
SIPX_SNDDG_PACKET_TYPE	Store the value of the *TCBPacketType* NWTCB field in the *Packet Type* field of the IPX header. (*TCBPacketType* must be given a value.)
SIPX_SNDDG_SPECIFIC_ROUTE	Use the subnetwork identified by the *TCBSubnetworkHandle* NWTCB field to send the datagram. (*TCBSubnetworkHandle* must be given a value.)
SIPX_SNDDG_VERIFY_ROUTE	Verify the address of the next hop in the path to the destination node.

The following NWTCB fields are set upon completion:

TCBBytesTransferred, indicating the number of bytes sent.

TCBFinalStatus, indicating the final status of the request.

TCBSubnetworkHandle, indicating the subnetwork over which the datagram was sent.

See Also

NWSipxReceiveDatagram

NWSipxSendMessage

Sends a data message to a peer client application
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxSendMessage (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb
(IN) Points to the NWTCB for the request.

Return Values

SIPX_PENDING	The request was accepted without error and is in progress.
SIPX_SUCCESSFUL	The request was completed without error.

Final Status Values

SIPX_CANCELED	The request was canceled.
SIPX_CONNECTION_ABORTED	The transport connection was terminated abnormally.
SIPX_CONNECTION_TERMINATED	The transport connection was terminated.
SIPX_INSUFFICIENT_RESOURCES	System resources were not available for this request.
SIPX_INTERNAL_ERROR	Internal system error occurred.
SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in the NWTCB is invalid.

Remarks

The following NWTCB fields must be set before calling **NWSipxSendMessage**:

TCBConnHandle, indicating the handle of the connection endpoint to use.

TCBFragmentCount, indicating the number of entries in the fragment list.

TCBFragmentList, pointing to the fragment list containing information about the data message.

If you want to send an SPX header without data, set the TCBFragmentCount field to zero. The API will then ignore the TCBFragmentList field.

The SIPX_SNDMSG_PARTIAL NWTCB flag must be set before calling **NWSipxSendMessage**. If SIPX_SNDMSG_PARTIAL is clear, the data segment is in the final fragment of the message. If SIPX_SNDMSG_PARTIAL is set, the data segment is not in the final fragment of the message. For example, the following indicates that the packet is not the final fragment:

```
sendTCB->TCBFlags = SIPX_SNDMSG_PARTIAL;
```

And the following indicates the packet is the following fragment:

```
sendTCB->TCBFlags = SendTCB->TCBFlags & ~SIPX_SNDMSG_PARTIAL;
```

The *TCBDataStreamType* NWTCB field can be set before calling NWSipxSendMessage. This field indicates the value to store in the *Data Stream Type* field of the SPX header (The SIPX_SNDMSG_DSTRM_TYPE flag must be set).

The following NWTCB flag can be set before calling **NWSipxSendMessage**:

SIPX_SNDMSG_ATTN--Requests the attention flag be set in the SPX header. The NWSIPX API for Windows NT does not support the setting of the attention bit in the SPX header.

SIPX_SNDMSG_DSTRM_TYPE--Stores the value of *TCBDataStreamType* into the *Data Stream Type* field of the SPX header. (*TCBDataStreamType* must be given a value.)

The following NWTCB fields are set upon completion:

TCBBytesTransferred--indicates the number of bytes sent.

TCBFinalStatus--indicates the final status of the request.

NWSipxSetInformation

Sets new information values for a connection endpoint
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxSetInformation (
   nuint32    infoType,
   nptr       hHandle,
   nptr       pInfoStruct);
```

Parameters

infoType

(IN) Specifies the type of API element information structure being modified.

hHandle

(IN) Specifies differing types of handles or pointers depending on the information type specified in *infoType*.

pInfoStruct

(IN) Points to the information structure containing the new API element parameter information. The structure pointed to varies according to the information type specified in *infoType*.

Return Values

SIPX_ACCESS_VIOLATION	An invalid pointer is specified in the *NWIEventInfo* field of the NWTCB information structure.
SIPX_INVALID_CONNECTION_HANDLE	An invalid connection endpoint handle is specified in *hHandle*.
SIPX_INVALID_HANDLE	An invalid handle is specified in the *NWIEventInfo* field of the NWTCB information structure.
SIPX_INVALID_INFO_TYPE	An invalid information type was specified in the *infoType* parameter.
SIPX_INVALID_IOCTL_BUFFER	Versions of NWSIPX DLL and

_LEN	NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_MUX_GROUP_HANDLE	An invalid mux group handle is specified in the *NWIEventInfo* field of the NWTCB information structure.
SIPX_INVALID_NWTCB	An invalid pointer to an NWTCB has been specified in the *hHandle* parameter.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags are specified.
SIPX_INVALID_SOCKET_HANDLE	The socket handle specified in *hHandle* is invalid.
SIPX_INVALID_SYNC_TYPE	An invalid event notification type is specified in the *NWIEventType* field of the NWTCB information structure.
SIPX_NWTCB_IN_USE	The NWTCB is in use by another function.
SIPX_SUCCESSFUL	The request completed without error.

Remarks

The information structure to be modified must be obtained by calling **NWSipxGetInformation** and freed by calling **NWSipxFreeInformation**.

Valid *infoType* values include:

```
SIPX_API_INFORMATION      API environment
SIPX_SOCKET_INFORMATION   Socket
SIPX_CONN_INFORMATION     Connection endpoint
SIPX_NWTCB_INFORMATION    NWTCB
```

Allowed structure types for *pInfoStruct* include:

```
SIPX_API_INFO      API environment
SIPX_SOCKET_INFO   Socket
SIPX_CONN_INFO     Connection endpoint
SIPX_NWTCB_INFO    NWTCB
```

The following table shows the relationship between the value of the *infoType* parameter and the type of allocated structure whose address is retuned in *ppInfoStruct*.

infoType	**ppInfoStruct Structure**

SIPX_API_INFORMATION	SIPX_API_INFO
SIPX_CONN_INFORMATION	SIPX_CONN_INFO
SIPX_NWTCB_INFORMATION	SIPX_NWTCB_INFO
SIPX_SOCKET_INFORMATION	SIPX_SOCKET_INFO

The type of information specified by the *hHandle* is determined by the value specified by the *infoType*. The following table shows this relationship.

infoType	**hHandle**
SIPX_API_INFORMATION	NULL
SIPX_CONN_INFORMATION	Handle of a connection endpoint
SIPX_NWTCB_INFORMATION	Pointer to an NWTCB
SIPX_SOCKET_INFORMATION	Handle of a socket

See Also

NWSipxFreeInformation, NWSipxGetInformation

NWSipxTerminateConnection

Terminates a connection with a peer application
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxTerminateConnection (
   PNWTCB   pNwtcb);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request. If the SPX_TERM_REJECT flag is set in the *TCBFlags* field of the NWTCB, this must be the same NWTCB used with the **NWSipxListenForConnection** function.

Return Values

SIPX_INVALID_CONNECTION_HANDLE	The connection endpoint handle specified in NWTCB is invalid.
SIPX_INVALID_IOCTL_BUFFER_LEN	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_IOCTL_FUNCTION	Versions of NWSIPX DLL and NLM are mismatched.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_NWTCB_FLAGS	Invalid NWTCB flags were specified.
SIPX_INVALID_STATE	The connection endpoint is not in the SIPX_WAITING_ACCEPT or the SIPX_CONNECTED state.
SIPX_INVALID_SYNC_TYPE	You cannot block within the callback function.
SIPX_MEMORY_LOCK_ERROR	System error: cannot lock memory.
SIPX_NWTCB_IN_USE	The NWTCB already in use.
SIPX_PENDING	The request was accepted without error and is in progress.

SIPX_SUCCESSFUL	The request completed without error.

Final Status Values

SIPX_CANCELED	Request canceled.
SIPX_INTERNAL_ERRO R	An internal system error occurred.
SIPX_SUCCESSFUL	The request was accepted without error.

Remarks

If **NWSipxTerminateConnection** is called to reject an incoming connection, the same NWTCB used for **NWSipxListenForConnection** must be used when calling **NWSipxTerminateConnection**.

The following NWTCB field must be set before calling **NWSipxTerminateConnection**:

TCBConnHandle indicating the handle of the connection endpoint for the transport connection to be terminated.

One of the following NWTCB flags must be set before calling **NWSipxTerminateConnection**:

SIPX_TERM_ACKE D	Do not notify the user that termination is complete until the peer acknowledges the termination.
SIPX_TERM_ABOR T	Immediately abort the transport connection and notify the user that termination is complete.
SIPX_TERM_REJEC T	Reject the incoming connection. This flag must be set only in NWTCBs that have been used with **NWSipxListenForConnection**.

The following NWTCB field is set upon completion:

TCBFinalStatus indicating the final status of the request.

See Also

NWSipxEstablishConnection

NWSipxWaitForSingleEvent

Monitors a single NWTCB for a transport event
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxWaitForSingleEvent (
   PNWTCB    pNwtcb,
   nuint32   timeOut);
```

Parameters

pNwtcb

(IN) Points to the NWTCB for the request.

timeOut

(IN) Indicates the time (in milliseconds) to wait for a transport event to occur. Specify SIPX_INFINITE_WAIT if no timeout is desired.

Return Values

SPX_CANCELED	The request was canceled.
SIPX_INVALID_NWTCB	The data space pointed to by *pNwtcb* is not a valid NWTCB.
SIPX_INVALID_SYNC_TYPE	You cannot block within a callback function.
SIPX_PENDING	The timeout value elapsed; request completion is still pending.
SIPX_SUCCESSFUL	The request completed without error.
SIPX_UNSUCCESSFUL	An internal system error occurred.
Other values	Other values according to the API service request initiating the transport event.

Remarks

Because the NWTCB might be is use by another function, the NWTCB flags or fields must not be set before calling

NWSipxWaitForSingleEvent.

The following NWTCB field is set upon completion:

TCBFinalStatus, indicating the final status of the request. The final status values are the same as for the function that initiated the transport event.

See Also

NWSipxAllocControlBlock, **NWSipxWaitForMultipleEvents**

NWSipxWaitForMultipleEvents

Monitors multiple NWTCBs for a transport event
Platform: NT, OS/2, Windows 95
Service: NWSIPX

Syntax

```
#include <ntypes.h>
#include <nwsipx32.h>

N_EXTERN_LIBRARY(nuint32) NWSipxWaitForMultipleEvents (
   SIPXMUXGRP_HANDLE   muxGroupHandle,
   nuint32             timeOut,
   PPNWTCB             ppNwtcb);
```

Parameters

muxGroupHandle

(IN) Specifies the handle of the NWTCB mux group to monitor.

timeOut

(IN) Specifies the time (in milliseconds) to wait for a transport event to occur. Specify SIPX_INFINITE_WAIT if no timeout is desired.

ppNwtcb

(OUT) Points to the location where the pointer to the NWTCB with the transport event will be stored.

Return Values

SIPX_INVALID_MUX_GROUP_HANDLE	The NWTCB mux group handle specified in *muxGroupHandle* is invalid.
SIPX_INVALID_PARAMETER	An invalid pointer was specified by *ppNwtcb*.
SIPX_INVALID_SYNC_TYPE	You cannot block within a callback function.
SIPX_PENDING	The timeout value elapsed; request completion is still pending.
SIPX_SUCCESSFUL	The request completed without error.
SIPX_UNSUCCESSFUL	An internal system error occurred.

Other values	Other values according to the function initiating the transport event.

Remarks

The following NWTCB field is set upon completion:

TCBFinalStatus, indicating the final status of the request. The final status values are the same as for the function that initiated the transport event.

See Also

NWSipxAllocControlBlock,**NWSipxChangeControlBlock**, **NWSipxWaitForSingleEvent**

SIPX_ERROR

(Macro) Determines whether an NWSIPX API service request resulted in an error

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <nwsipx32.h>

SIPX_ERROR (x);
```

Parameters

x

(IN) Specifies the status value to be checked.

Return Values

TRUE	An error occurred.
FALSE	An error did not occur.

Remarks

SIPX_ERROR is a macro that is provided to help client applications determine whether or not an error has occurred during the execution of a NWSIPX service request. The value checked can either be the return value of a function, or it can be the contents of the *TCBFinalStatus* field of the NWTCB.

See Also

SIPX_SUCCESS

SIPX_SUCCESS

(Macro) Determines whether an NWSIPX API service request was successful

Platform: NT, OS/2, Windows 95

Service: NWSIPX

Syntax

```
#include <nwsipx32.h>

SIPX_SUCCESS (x);
```

Parameters

x

(IN) Specifies the status value to be checked.

Return Values

TRUE	The request was successful.
FALSE	The request was not successful.

Remarks

SIPX_SUCCESS is a macro that is provided to help client applications determine whether or not an NWSIPX service request was successful. The value checked can either be the return value of a function, or it can be the contents of the *TCBFinalStatus* field of the NWTCB.

See Also

SIPX_ERROR

Communication Service Group

NWSIPX: Structures

FRAGMENT

Provides information about the location and size of a fragment of data
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_FRAGMENT {
    nptr      FAddress;
    nuint32   FLength;
} FRAGMENT, *PFRAGMENT;
```

Fields

FAddress

Points to a fragment of data.

FLength

Indicates the size (in bytes) of the data fragment.

IPXADDR

Stores an IPX/SPX network address
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_IPXADDR {
   nuint8   NANet[4];
   nuint8   NANode[6];
   nuint8   NASocket[2];
} IPXADDR, *PIPXADDR;
```

Fields

NANet

Indicates a network number, stored as a character array.

NANode

Indicates a node number, stored as a character array.

NASocket

Indicates a socket number, stored as a character array.

Remarks

NOTE: The network address components are specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

NETADDR

Stores multiple types of network addresses
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_NETADDR {
   nuint32   NAType;
   nuint32   NALength;
   union {
      nuint8    NAGenAddress[SIPX_MAX_TRANS_ADDR_LEN];
      IPXADDR   NAIpxAddress;
   } NAAddress;
} NETADDR, *PNETADDR;
```

Fields

NAType

Indicates the type of the network address being stored. Currently, the only address type that is supported is TA_IPX_SPX.

NALength

Indicates the length (in bytes) of the address stored in *NAAddress*.

NAAddress

Indicates a network address.

Remarks

NETADDR can be used to define multiple address types.

The only address type currently supported is TA_IPX_SPX.

NOTE: The network address components are specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

NWTCB

Used to transfer information between the client application and the NWSIPX API

Service: NWSIPX

Defined In: nwsipx32.h

Structure

```
typedef struct TAG_NWTCB {
   TAG_NWTCB           *TCBNext;
   TAG_NWTCB           *TCBPrevious;
   SIPXSOCK_HANDLE     TCBSockHandle;
   SIPXCONN_HANDLE     TCBConnHandle;
   nptr                TCBClientContext;
   nuint32             TCBTransportEvent;
   union {
      nuint32   TCBTimeout;
      nuint8    TCBEventSpace[32];
   } TCBEvent;
   nuint32             TCBFinalStatus;
   nuint32             TCBBytesTransferred;
   nflag32             TCBFlags;
   NETADDR             TCBRemoteAddress;
   SIPXSUBNET_HANDLE   TCBSubnetworkHandle;
   nuint32             TCBMsgSequenceNumber;
   nuint8              TCBPacketType;
   nuint8              TCBDataStreamType;
   nuint8              TCBReserved[2];
   nuint32             TCBFragmentCount;
   PFRAGMENT           TCBFragmentList;
} NWTCB, *PNWTCB, **PPNWTCB;
```

Fields

TCBNext

Points to the next NWTCB in a linked list. This is for use by the client application when it manages multiple NWTCBs. The NWSIPX API does not use this field at any time.

TCBPrevious

Points to the previous NWTCB in a linked list. This is for use by the client application when it manages multiple NWTCBs. The NWSIPX API does not use this field at any time.

TCBSockHandle

Specifies a socket handle. This field is used differently depending upon the function being called.

TCBConnHandle

Specifies a connection-endpoint handle. This field is used differently depending upon the function being called.

TCBClientContext

Points to a location where the client can save its own context-sensitive information. This field is not used by the NWSIPX API at any time.

TCBTransportEvent

Indicates the transport event to be registered for when calling **NWSipxRegisterForTransportEvent**. The currently defined values are:

```
SIPX_LISTEN_FOR_DISCONNECT
SIPX_SCHEDULE_TIMER_EVENT
SIPX_SUBNET_STATUS_CHANGE
```

TCBEvent

Indicates parameters associated with the transport event specified in the *TCBTransportEvent* field. This is an overloaded field that accommodates any parameters that are required for the transport event indicated by the *TCBTransportEvent* field. The currently defined parameter fields are:

```
TCBTimeout    Timer value (in milliseconds) when
              SIPX_SCHEDULE_TIMER_EVENT has been selected.
```

TCBFinalStatus

Indicates the final completion status of a request or event.

TCBBytesTransferred

Indicates the number of bytes sent or received on a data transfer operation.

TCBFlags

Specifies the common flag field of the NWTCB. Any flags required for the operation of an NWSIPX API function are set in this field.

> **NOTE:** Only valid flags must be set in this field. If invalid flags are set, an error status results whenever the NWTCB is used.

TCBRemoteAddress

Indicates a remote address structure that you can store into or examine depending upon the function that is being called.

> **NOTE:** The network address components are specified in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.
>
> When specifying a network address, you must set the *NAType* field to TA_IPX_SPX and the *NALength* field to the value of **sizeof** (IPXADDR).

TCBSubnetworkHandle

Specifies the handle of the subnetwork to be used by the function using this NWTCB.

TCBMsgSequenceNumber

Indicates the sequence number of the message segment associated with the NWTCB (for connection-oriented messages only). The message sequence number is assigned to each message segment as it arrives, allowing you to determine the order in which message segments were received. **This sequence number is not the same as the SPX protocol sequence number.**

TCBPacketType

When sending a datagram, this field indicates a value that is to be stored into the *packetType* field of the IPX header. When receiving a datagram, this field indicates the value that was present in the *packetType* field of the IPX header. *TCBPacketType* is valid only for datagram services.

TCBDataStreamType

When sending a message, this field indicates a value that is to be stored in the *dataStreamType* field of the SPX header. When receiving a message, this field indicates the value that was present in the *dataStreamTyp*e field of the SPX header. *TCBDataStreamType* is valid only for connection-oriented services.

TCBReserved

Is reserved for future use.

TCBFragmentCount

Indicates the number of fragment descriptors in the fragment list pointed to by the *TCBFragmentList* field (maximum of 15).

TCBFragmentList

Points to a fragment list. Each fragment descriptor in the fragment list contains a pointer to the data fragment and its size in bytes.

NOTE: Neither the fragment list nor the data space identified by the fragment list can be modified or freed while the NWTCB is in use by an NWSIPX API request.

SIPX_API_INFO

Holds information describing the NWSIPX API environment
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct {
   nuint32            AIApiVersion;
   nuint32            AIIpxVersion;
   nuint32            AISpxVersion;
   nuint32            Reserved[4];
   nuint32            AISubnetCount;
   SIPX_SUBNET_INFO   AISubnetInfo[1];
} SIPX_API_INFO, *PSIPX_API_INFO;
```

Fields

AIApiVersion

(Read-only) Indicates the current version of the NWSIPX API.

AIIpxVersion

(Read-only) Indicates the current IPX version.

AISpxVersion

(Read-only) Indicates the current SPX version.

Reserved

Is reserved for future use.

AISubnetCount

(Read-only) Indicates the number of subnetworks known to the system.

AISubnetInfo

(Read-only) Indicates an array of SPX_SUBNET_INFO structures describing the attributes of the subnetworks known to the NWSIPX API. The *AISubnetCount* field indicates how many elements are in the array.

Remarks

SIPX_API_INFO structures are allocated by calling **NWSIPXGetInformation** with the *infoType* parameter set to SIPX_API_INFORMATION. They are freed by calling **NWSIPXFreeInformation**.

SIPX_CONN_INFO

Holds information about a connection endpoint
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct {
   nuint32   CISpxVersion;
   nuint32   CIState;
   nuint32   CIDataStreamType;
   nuint32   CIConnectionProfile;
   nuint32   CIInputMode;
   nuint32   CIStreamingTimer;
   nuint32   CIRetryCount;
   nuint32   CISocketHandle;
   nuint32   CIMaxTsduSize;;
   nuint32   CIOutputMode
   nuint32   Reserved[3];
} SIPX_CONN_INFO, *PSIPX_CONN_INFO;
```

Fields

CISpxVersion

(Read-only) Indicates the SPX version currently being used by the connection endpoint.

CIState

(Read-only) Indicates the current state of the connection endpoint. The following states are defined:

```
SIPX_ALLOCATED
SIPX_CONNECTED
SIPX_CONNECTING
SIPX_LISTENING
SIPX_TERMINATING
SIPX_WAITING_ACCEPT
```

CIDataStreamType

Indicates the type of the data stream. The contents of this field are stored in the *dataStreamType* field of the SPX header of every data message that is sent to the peer application through the connection endpoint. This value can be overridden on a per-message basis. The valid ranges of values are 0x00 to 0xFD.

CIConnectionProfile

Indicates the expected duration and the quality of service requirements for a transport connection using this connection

endpoint. The following values are currently defined:

```
SIPX_CONN_LONG_TERM
SIPX_CONN_SHORT_TERM
```

CIInputMode

Indicates the method for receiving data. The defined modes are:

```
SIPX_CONN_MESSAGE
SIPX_CONN_STREAM
```

CIStreamingTimer

Indicates the length of time (in milliseconds) that message fragments should be gathered before delivery. This parameter is used only if *CIInputMode* has a value of SIPX_CONN_STREAM.

CIRetryCount

Indicates the number of times the transfer of a data message should be retried before aborting the transport connection. (This is a read-only parameter for Windows NT*.)

CISocketHandle

(Read-only) Indicates the handle of the socket that the connection endpoint is associated with.

CIMaxTsduSize

(Read-only) Indicates the maximum size of the data segment supported for this connection endpoint. A value of SIPX_CONN_UNLIMITED_TSDU indicates an unlimited TSDU size.

CIOutputMode

Indicates the method for sending data. The defined modes are:

```
SIPX_CONN_FRAGMENT
SIPX_CONN_TRANSPARENT
```

Reserved

Is reserved for future use.

Remarks

NOTE: You must not allocate or free SIPX_CONN_INFO structures yourself. The only way you can allocate valid SIPX_CONN_INFO structures is by calling **NWSIPXGetInformation** with the *infoType* parameter set to SIPX_CONN_INFORMATION. You must free SIPX_CONN_INFO structures by calling **NWSIPXFreeInformation**.

The value of *CIConnectionProfile* allows the NWSIPX API to negotiate different SPX options that most closely match the intended use of the transport connection. This field can only be set when the connection endpoint is in the SIPX_ALLOCATED state.

The modes and descriptions for the *CIInputMode* field are as follows:

SIPX_CONN_MESSAGE	Data is delivered as whole messages. (Partial messages can be delivered if the input buffer is too small to hold the entire message.
SIPX_CONN_STREAM	This behavior depends upon the value of the *CIStreamingTimer* field. If streaming mode is disabled (by *CIStreamingTimer* being set to zero), message fragments are delivered as they arrive from the network. Otherwise, message fragments are gathered until the *CIStreamingTimer* value expires. This field can only be set when the connection endpoint is in the SIPX_ALLOCATED state.

The modes and descriptions for the *CIOutputMode* field are as follows:

SIPX_CONN_FRAGMENT	Data received is fragmented as necessary for delivery to the peer system.
SIPX_CONN_TRANSPARENT	Data received is sent immediately to the peer system without further fragmentation. You are responsible for determining valid message sizes. This field can only be set when the connection endpoint is in the SIPX_ALLOCATED state. You indicate message boundaries by setting or clearing the SIPX_SNDMSG_PARTIAL flag in the *TCBFlags* field of the NWTCB passed to the **NWSipxSendMessage** function. Windows NT cannot guarantee that messages will not be fragmented in Transparent Mode. Implementations of the NWSIPX API Version 2 or earlier do not support this parameter. In these cases, the parameter value will be SIPX_CONN_UNSUPPORTED. The API version can be read from the **APIVersion** API *Environment* parameter.

When a connection is established, its parameters are set to the following default values:

Parameter	**Default Value**
CISpxVersion	Current SPX version

CIState	SIPX_ALLOCATED
CIDataStreamType	0x0
CIConnectionProfile	SIPX_CONN_LONG_TERM
CIOutputMode	SIPX_CONN_FRAGMENT
CIInputMode	SIPX_CONN_STREAM
CIStreamingTimer	0
CIRetryCount	IPX retry count configured for the system
CISockethandle	None
CIMaxTsduSize	Maximum TSDU size supported by the SPX protocol stack.

SIPX_NWTCB_INFO

Contains information about parameters that are associated with an NWTCB
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_SIPX_NWTCB_INFO {
   nuint32   NWIState;
   nuint32   NWIEventType;
   nuint32   NWIEventInfo;
   nuint32   Reserved[4];
} SIPX_NWTCB_INFO, *PSIPX_NWTCB_INFO;
```

Fields

NWIState

(Read-only) Indicates the current state of the associated NWTCB.

NWIEventType

Indicates the event notification type associated with the NWTCB.

NWIEventInfo

Indicates supplementary information pertaining to the event notification type. The supplementary information differs according to the event notification type specified by *NWIEventType*.

Reserved

Is reserved for future use.

Remarks

The following values are valid for the *NWIState* field:

SIPX_TCB_ALLOCATED	The NWTCB is allocated but is not in use.
SIPX_TCB_IN_USE	The NWTCB is in use.

The following values are valid for the *NWIEventType* field:

SIPX_API_EVENT	The function returns immediately. You must call **NWSipxWaitForSingleEvent**, which blocks until the request completes or the specified timeout period completes.

	timeout period completes.
SIPX_API_MUX_EVENT	Allows applications to monitor multiple NWTCBs for event completion with a single call to **NWSipxWaitForMultipleEvents**.
SIPX_BLOCKING	The function blocks until the request is complete.
SIPX_CALLBACK	The function returns immediately and the callback function is called when the request completes.
SIPX_POLLING	The function returns immediately and you application must call **NWSipxPoll** until a status other than SIPX_PENDING is returned.
SIPX_USER_EVENT	The function returns immediately. If the request is initiated successfully, your event object will be signaled when the request completes.

The following table shows the information that the *NWIEventInfo* field provides to supplement the information provided by *NWIState*.

Event Type	Supplementary Information
SPX_API_EVENT	NULL (No additional information is needed.)
SIPX_API_MUX_EVENT	Handle of the mux group to which this NWTCB belongs.
SIPX_BLOCKING	NULL (No additional information is needed.)
SIPX_CALLBACK	Address of the callback function that is to be called when the service request completes.
SIPX_POLLING	NULL (No additional information is needed.)
SIPX_USER_EVENT	Handle of the event object to be signaled upon completion of the service request.

SIPX_NWTCB_INFO structures are used to retrieve and set element parameters associated with an NWTCB. To change the values of NWTCB element parameters, first read the NWTCB's current parameters by calling **NWSIPXGetInformation** to retrieve an SIPX_NWTCB_INFO structure containing the current parameters. Then change the desired values in the SIPX_NWTCB_INFO structure. Then call **NWSIPXSetInformation**, passing in the modified SIPX_NWTCB_INFO structure. Lastly, free the SIPX_NWTCB_INFO structure by calling **NWSIPXFreeInformation**.

NOTE: You must not allocate or free SIPX_NWTCB_INFO structures yourself. The only way you can allocate valid SIPX_NWTCB_INFO structures is by calling **NWSIPXGetInformation** with the *infoType* parameter set to SIPX_NWTCB_INFORMATION. You must free SIPX_NWTCB_INFO

SIPX_NWTCB_INFO structures by calling **NWSIPXFreeInformation**.

SIPX_SERVICE_INFO

Identifies information about a server
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_SIPX_SERVICE_INFO {
   nuint16   SIServerType;
   nstr8     SIServerName[48];
   nuint8    SINetwork[4];
   nuint8    SINode[6];
   nuint16   SISocket;
   nuint16   SIHops;
} SIPX_SERVICE_INFO, *PSIPX_SERVICE_INFO;
```

Fields

SIServerType

Indicates the type of server.

SIServerName

Indicates the name of the server. Maximum name length is 47 bytes plus one for NULL termination.

SINetwork

Indicates the server network number, stored as a character array.

SINode

Indicates the server node number, stored as a character array.

SISocket

Indicates the server socket number, stored as a character array.

SIHops

Indicates the number of hops to the server.

Remarks

NOTE: The byte ordering of all of the fields in this structure is network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

SIPX_SERVER_INFO is used with **NWSipxQueryServices**.

One or more SIPX_SERVER_INFO structures are returned in the data space provided by the client application. These structures contain the requested server information.

SIPX_SOCKET_INFO

Stores parameters associated with a socket
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct TAG_SIPX_SOCKET_INFO {
   nuint32              SIIpxVersion;
   nflag32              SIAttributeFlags;
   nuint16              SISocketNumber;
   nuint16              SIReserved;
   nuint32              SIPacketType;
   SIPX_SUBNET_HANDLE   SISubnetworkHandle;
   nuint32              Reserved[4];
} SIPX_SOCKET_INFO, *PSIPX_SOCKET_INFO;
```

Fields

SIIpxVersion

(Read-only) Indicates the IPX version currently installed.

SIAttributeFlags

(Read-only) Indicates the type of the socket.

SISocketNumber

Indicates the number of the socket.

SIReserved

Is reserved.

SIPacketType

Indicates a value to be stored in the *packetType* field of the IPX header of each message that is sent. This value can be overridden on a per-message basis.

SISubnetworkHandle

Indicates the handle of the subnetwork currently being used by the socket.

Reserved

Is reserved for future use.

Remarks

The following values are valid for the *SIAttributeFlags* field:

SIPX_SOCK_DYNAMIC	The socket number was selected dynamically by the NWSIPX API.
SIPX_SOCK_CHECKSUM	Checksum processing (generation and validation) is done for every datagram sent or received over this socket.
SIPX_SOCK_STATIC	The socket number was specified by the application.

NOTE: The NWSIPX API for Windows NT does not support the generation of checksums for datagrams.

The socket number stored in *SISocketNumber* is stored in network byte order (high-low). See NTYPES.H for macros that facilitate conversion to and from network byte order.

NOTE: You must not allocate or free SIPX_SOCKET_INFO structures yourself. The only way you can allocate valid SIPX_SOCKET_INFO structures is by calling **NWSIPXGetInformation** with the *infoType* parameter set to SIPX_SOCKET_INFORMATION. You must free SIPX_SOCKET_INFO structures by calling **NWSIPXFreeInformation**.

When a socket is opened, its parameters are set to the following default values:

Parameter	Default Value
SIIpxVersion	Current IPX version
SIAttributeFlags	Set according to socket attributes (see *SIAttributeFlags* above)
SISocketNumber	The number of the socket that was opened
SIPacketType	0x4
SISubnetworkHandle	Handle of the default subnetwork or the subnetwork handle specified by the application when it opened the socket

SIPX_SUBNET_INFO

Describes the attributes of a subnetwork
Service: NWSIPX
Defined In: nwsipx32.h

Structure

```
typedef struct  TAG_SIPX_SUBNET_INFO {
   SIPXSUBNET_HANDLE   SNSubnetworkHandle;
   nuint32             SNMaxNsduSize;
   NETADDR             SNNetAddress;
   nflag32             SNFlags;
   nuint32             SNSubnetworkType;
   nuint32             Reserved[4];
} SIPX_SUBNET_INFO, *PSIPX_SUBNET_INFO;
```

Fields

SNSubnetworkHandle

Indicates a handle that uniquely identifies the subnetwork.

SNMaxNsduSize

Indicates the maximum NSDU size supported by the subnetwork.

SNNetAddress

Indicates the local internet address (net.node) associated with the subnetwork.

SNFlags

Indicates flags describing the subnetwork attributes.

SNSubnetworkType

Indicates the type of subnetwork.

Reserved

Is reserved for future use.

Remarks

The NWSIPX API supports simultaneous attachment to multiple networks and/or multiple attachments to the same physical network using different data encapsulation methods, such as ETHERNET_802.2, ETHERNET_II, and ETHERNET_SNAP. Each network attachment is referred to as a subnetwork. The SIPX_SUBNET_INFO describes structure is used to hold information about each of the subnetworks.

The following values are valid for the *SNFlags* field:

SIPX_SN_XSUM_FLAG	Checksum processing is supported.
SPX_SN_ACTIVE_FLAG	Subnetwork is active and usable.
SIPX_SN_DEFAULT_FLAG	The subnetwork is the default subnetwork. It is the subnetwork that will be used when a specific subnetwork is not specified for functions that accept a subnetwork handle as a parameter.

The following values are valid for the *SNSubnetworkType* field:

Subnetwork Type	**Value**	**Description**
SIPX_SN_VIRTUAL_LAN	0x00000000	Frame ID/MAC envelope is not necessary
SIPX_SN_ETHERNET_II	0x00000002	Ethernet using DEC Ethernet II frame type
SIPX_SN_ETHERNET_802_2	0x00000003	Ethernet (802.3) using an 802.2 envelope
SIPX_SN_TOKEN_RING	0x00000004	Token Ring (802.5) using an 802.2 envelope
SIPX_SN_ETHERNET_802.3	0x00000005	IPX 802.3 raw encapsulation
SPX_SN_802_4	0x00000006	Token-passing bus envelope
SIPX_SN_NOVELL_PCN2	0x00000007	Novell's IBM PC Network II envelope
SIPX_SN_GNET	0x00000008	Gateway's GNET frame envelope
SIPX_SN_PRONET_10	0x00000009	Proteon's ProNET I/O frame envelope
SIPX_SN_ETHERNET_SNAP	0x0000000a	Ethernet (802.3) using an 802.2 with SNAP envelope
SIPX_SN_TOKEN_RING_SNAP	0x0000000b	Token Ring (802.5) using 802.2 with SNAP envelope
SIPX_SN_LANPC_II	0x0000000c	Racore frame envelope
SIPX_SN_ISDN	0x0000000d	ISDN
SIPX_SN_NOVELL_RX_NET	0x0000000e	Novell's RX-NET envelope
SIPX_SN_IBM_PCN2_802.2	0x0000000f	IBM PCN2 using 802.2 envelope

SPX_SN_IBM_PCN2_SNAP	0x00000010	IBM PCN2 using 802.2 with SNAP envelope
SIPX_SN_OMNINET_4	0x00000011	Corvus's frame type
SIPX_SN_3270_COAXA	0x00000012	Harris Adcom's frame envelope
SIPX_SN_IP	0x00000013	IP tunnel frame envelope
SIPX_SN_FDDI_802_2	0x00000014	FDDI using 802.2 envelope
SIPX_SN_IVDALN_802_9	0x00000015	Comtex, Inc. frame envelope
SIPX_SN_DATACO_OSI	0x00000016	Dataco's frame envelope
SIPX_SN_FDDI_SNAP	0x00000017	FDDI using 802.2 with SNAP envelope

NOTE: The NWSIPX API for Windows NT cannot determine a subnetwork's type. Therefore, the *SNSubnetworkType* parameter always indicates SIPX_SN_TYPE_UNKNOWN for Windows NT platforms.

SIPX_SUBNET_INFO is associated with SIPX_API_INFO.

SAP

SAP: Guides

SAP: General Guide

Basic Information

- SAP Overview
- SAP Function List

Parts of SAP

- SAP Protocol Requirements
- Service Advertising Packets
- Service Identification Packets
- Service Query Packets

Using SAP

- Broadcasting
- Initializing SAP
- Advertising Services
- Responding to Service Queries
- Terminating Service

Locating Services

- Locating Services with SAP
- Scanning the Bindery
- Making General Service Queries
- Making Nearest Service Queries

Additional Links

- SAP: Functions
- SAP: Structures

SAP: Concepts

Advertising Services

All advertising servers must broadcast their identity once every 60 seconds using a service identification packet. This periodic broadcast keeps bridges and servers (running NetWare 2.x and above) information of the advertising server's identity and presence.

To broadcast a server identification packet, an advertising server sets *response type* to 2. In the packet's IPX header, the server sets *destination node* field to FFFFFFFFFFFFh and *destination socket* field to 0452h.

AdvertiseService sends out an identification packet every 60 seconds. Identification packets are received by all servers and bridges on a network. This function takes the server type, server name, and socket number as input. It opens the socket, prepares the server identification packet, and performs the advertising as described above

Servers add a server to their binderies based on the information in the identification packet. If several minutes pass and a server fails to broadcast, all bridges and servers on the network assume the server has terminated. In that case, servers remove the server from their binderies. The identification packet is automatically forwarded to the entire internet by bridges and servers.

Parent Topic: Broadcasting

Broadcasting

Advertising servers must initialize the service advertising socket, periodically transmit service identification packets, and be prepared to respond to service query packets.

Related Topics

Initializing SAP

Advertising Services

Responding to Service Queries

Terminating Service

Initializing SAP

An advertising server must open the service advertising socket and post one or more ECBs to monitor service queries. If the socket has been opened already, the server must not attempt to monitor the socket. Another process, typically a NetWare server or bridge, is monitoring the socket and responding to service advertising queries on behalf of the advertising server.

If a server is unable to open the socket because the maximum number of sockets are already opened, the server should report an error and terminate.

Parent Topic: Broadcasting

Locating Services with SAP

A service query is sent by an application that wants to find a server that it can establish a client-server relationship with. There are two ways clients can locate services on the network. They can search a server's bindery to detect the presence of the server on the network, or they can use **QueryServices**.

Related Topics

Scanning the Bindery

Sending Service Query Broadcasts

Making General Service Queries

Making Nearest Service Queries

Making General Service Queries

Applications can make a general service query to obtain a list of all eligible servers on the network. General service queries seek a response from every qualified server. The query can specify all servers of a particular type or all servers of any type. Every server on the network that matches the indicated type responds with an identification packet.

Packets for a general service query are of Query Type 1. The server type parameter should be set to a particular service class. The wildcard value (0xFFFF) is also valid.

Parent Topic: Locating Services with SAP

Making Nearest Service Queries

Applications can query for the nearest server of a particular type. In this context, **nearest** refers to the server that is able to respond first. When making this query, the application obtains information about only one server (if any servers offering the requested service are present on the network).

Parent Topic: Locating Services with SAP

Responding to Service Queries

Clients use the service query packet to query advertising servers. Servers receive this query on the service advertising socket and must post at least one ECB on the this socket. In response to the query, the server returns a server identification packet to the client. The response should indicate the type of query the server is responding to. See Advertising Services.)

A service query packet indicates both the query type and server type. The server must check to ensure that it matches the query's server type. The server should also respond to the wildcard server type.

There are two types of queries:

General service query (1)

Nearest service query (2).

The first type seeks all servers of a specified type. The second type searches for the first server that responds of a specified type. Only nearest service queries are currently implemented in the C client library. However, developers can implement general service queries themselves. Although a server may receive an assortment of packets on the service advertising socket, it should respond only to general and nearest service queries.

Applications can call the C library function **QueryServices** to broadcast service queries. This function takes the query type and server type as input and returns the appropriate server information.

Parent Topic: Broadcasting

SAP Function List

SAP functions offered for DOS, MS Windows, and OS/2 workstations are listed below. If you are programming your service to run with DOS, MS Windows, or OS/2 you should use these functions.

Function	Header	Purpose
AdvertiseService	nwsap.h sap.h	Advertises a service on the network.
QueryServices	nwsap.h sap.h	Returns the identities of all servers of all types, all servers of a specific type, or the nearest server of a specific type.
ShutdownSAP (DOS, Win)	nwsap.h sap.h	Stops the advertising of a service on the network.

For NLM applications, the following functions are available

Function	Purpose
AdvertiseService	Informs clients of the presence of a service on the network.
FreeQueryServicesList	Frees the list of SAP response structures after **QueryServices** has been called.
QueryServices	Returns the identities of all servers of all types, all servers of a specific type, or the nearest server of a specific type.
ShutdownAdvertising (NLM)	Stops the advertising of a service on the network.

SAP Overview

Service Advertising Protocol (SAP) defines IPX packet structures and broadcast procedures for servers that want to advertise their name and service on the network. An advertising server can be any application running on a NetWare® server, bridge, or workstation. NetWare servers record in their binderies the advertising server's name, service type, and network address.

Clients of value-added servers can use SAP to query the network for various types of servers. Clients can also use Bindery Services to query the bindery of any NetWare server and discover the names and addresses of various types of servers present on the network.

To use SAP, advertising servers and their clients must operate on nodes running NetWare 2.x or above. Under NetWare 4.x, Directory Services effectively replaces SAP as an advertising medium. However, using Directory Services as an advertising medium isn't possible for bindery-based NetWare servers and workstations. Advertising servers must use SAP to advertise to bindery-based clients.

NOTE: In NetWare 4.x, the maximum number of SAP handles that one NLM™ application can concurrently have open is 20. Otherwise, SAP services have been preserved exactly as they were in NetWare versions before 4.0; the NetWare Directory itself is "SAPing."

SAP Protocol Requirements

SAP governs the server types, server names, and socket assignments available to advertising servers.

Server Types

Novell assigns each type of server a unique *server type*. By specifying a server type, a client application can receive a list of servers providing a particular service. All servers that provide the same service through identical application-level protocols should adopt the same server type. Software developers must contact Novell to obtain a specialized server type (see Reserving Services and Names and How To Get Assistance). The following well-known server object types are defined:

FFFFh (1)	Wild
0000h	Unknown
0003h	Print queue
0004h	NetWare server
0005h	Job server
0007h	Print server
0009h	Archive server
0024h	Remote bridge server
0047h	Advertising print server
8000h	Reserved up to

Server Names

Each advertising server must be given a unique name within its service type. Names are usually assigned by the network administrator as part of the server installation procedure. Servers of different types may share names. For example, a NetWare server named ADMIN and a print server named ADMIN could both reside on the network.

Socket Assignments

SAP designates socket 0452h as the service advertising socket. All SAP operations are performed on this socket. The service advertising socket must be kept separate from working sockets used by an advertising server to provide services. Working sockets are regulated according to IPX conventions. Dynamic socket numbers range from 4000h to 4FFFh. Static socket numbers are assigned by Novell and begin at 8000h.

Scanning the Bindery

All NetWare servers monitor the SAP socket for server broadcasts. When a server broadcasts, each NetWare server adds that server as an object in its bindery. An application can locate a NetWare server by scanning another NetWare server's bindery.

Server objects in the bindery have a name and type that identify them uniquely on the network. No two servers of a particular type can have the same name. Server objects also have a NET_ADDRESS property that contains the network, node, and socket address of the server.

Applications can scan a bindery for a specific NetWare server by name, for all servers of a specific type, or for all servers in general. Once a server is found, the client can use the server's NET_ADDRESS property to deliver IPX packets to the server. For more information about how to scan a bindery, see Bindery.

Parent Topic: Locating Services with SAP

Sending Service Query Broadcasts

In addition to scanning the bindery, applications can call **QueryServices** to send a service query broadcast and receive all the replies. **QueryServices** works if volume SYS: is not mounted, whereas scanning the bindery does not work. It also works to locate services other than NetWare servers.

Parent Topic: Locating Services with SAP

Service Advertising Packets

SAP defines two packet structures: service identification packet and service query packet. Advertising services broadcast Service Indentification Packets. Clients search for services using the service query packet.

NetWare bridges monitor service identification packets to track active advertising servers. Each bridge propagates this information to other

bridges on the network. NetWare servers log the server's name, type, and address in their binderies. If a server fails to make itself known for several minutes, all bridges and NetWare servers assume the server has terminated.

By keeping track of advertising servers, NetWare bridges and servers are able to reply to workstation queries about the servers present on the network. This makes it possible for a workstation to find an appropriate server even if the workstation resides on a different local network.

Related Topics

Service Identification Packets

Service Query Packets

Service Identification Packets

The service identification packet has a length of 96 bytes. In some instances, however, a service identification packet may contain information concerning as many as seven servers. Each additional server adds 64 bytes of data to the packet.

The maximum length of a server identification packet is 480 bytes. Therefore, an application using SAP to locate a specific type of server (or all servers of a given type) must have a listen packet large enough to receive up to 480 bytes. It has the following format:

Offset	Field	Size	Type
0	Header Information	BYTE[30]	high-low
30	Response Type	WORD	high-low
32	Server Type	WORD	high-low
34	Server Name	BYTE[48]	high-low
82	Network	BYTE[4]	high-low
86	Node	BYTE[6]	high-low
92	Socket	WORD	high-low
94	Intermediate Networks	WORD	high-low

Each field in the identification packet is explained in the following list. (The packet is constructed by **AdvertiseService**.)

Header Information

This field contains the IPX header (30 bytes). For a detailed explanation of the IPX header, see IPX Packet Structure.

Response Type

This field identifies the type of SAP packet. If the packet is a response to a services query, the field is set according to the query type. For more information about query types, see Making General Service Queries. For a periodic broadcast, this field is set to 2.

Server Type

This field identifies the type of service the server provides. Server types can be obtained from Novell. For example, a NetWare server advertises itself as type 4. This value becomes the object type for this server in the bindery.

Server Name

This field contains the object name assigned to the server. Server names can be 48 bytes (including a NULL terminator) and must uniquely identify the server on the network.

Network

This field contains the address of the network on which the server resides.

Node

This field contains the address of the node on which the server resides.

Socket

This field contains the socket number on which the server receives service requests.

Intermediate Networks

This field contains the number of hops the identification packet makes traveling from the server to the client. Initially, NetWare sets this field to 0. Each time the packet passes through an intermediate network, the field is incremented by one.

Parent Topic: Service Advertising Packets

Service Query Packets

The structure for a service query packet is shown in the following table. (The packet is built by **QueryServices**.) An explanation of each field follows.

Offset	Field	Size	Type
0	Header Information	BYTE[30]	high-low
30	Packet Type	WORD	high-low
32	Server Type	WORD	high-low

Header Information

This field contains the IPX header (30 bytes). For a detailed explanation of the IPX header, see IPX Packet Structure.

Packet Type

This field identifies the type of service query. For a General Service Query, this field is set to 1. For a Nearest Service Query, this field is set to 3.

Server Type

This field identifies the type of service the server provides. Contact Novell Developer Relations to register a new server type.

Applications can call **QueryServices** to find all servers present or to find only the nearest server of a specific type. Each type of query is explained in the following section.

Parent Topic: Service Advertising Packets

Terminating Service

If a server is preparing to shut down, it can broadcast a shutdown packet that notifies all bridges and servers of its intention. Shutdown advertising is not mandatory but should be performed as a courtesy to the network. The notification allows bridges and servers to purge the server from their tables. A shutdown packet is identical to an identification packet except that **ShutdownAdvertising (NLM)** for NLM applications and **ShutdownSAP (DOS, Win)** for other applications places a value of 0x10 in the intermediate networks field.

Parent Topic: Broadcasting

SAP: Functions

SAP for DOS, Win

AdvertiseService (DOS, Win)

Advertises a server on the internetwork
Server: 2.2, 3.11 and above, 4.0 and above
Platform: DOS, Win
Service: Service Advertising Protocol (SAP)

Syntax

```
#include <nwipxspx.h>

int AdvertiseService(
   WORD    serverType,
   char   *serverName,
   BYTE   *serverSocket);
```

Parameters

serverType

(IN) Indicates the type of server assigned by Novell® for the server's service class. This parameter must appear in low-high order, so call **IntSwap** when using NetWare® defined server types.

serverName

(IN) Indicates the server name (maximum of 48 characters including NULL).

serverSocket

(IN) Points to the server socket number.

Remarks

All servers, regardless of type, must broadcast their identity once every 60 seconds. This periodic identification broadcast informs all bridges and file servers running NetWare 2.0 and above on the local network of a server's identity.

Valid *serverType* parameters follow:

```
OT_WILD                    0xFFFF(only valid for General Service Querie
OT_UNKNOWN                 0x0000
OT_USER                    0x0100
OT_USER_GROUP              0x0200
OT_PRINT_QUEUE             0x0300
OT_FILE_SERVER             0x0400
OT_JOB_SERVER              0x0500
OT_GATEWAY                 0x0600
OT_PRINT_SERVER            0x0700
OT_ARCHIVE_QUEUE           0x0800
```

```
OT_ARCHIVE_SERVER          0x0900
OT_JOB_QUEUE               0x0A00
OT_ADMINISTRATION          0x0B00
OT_NAS_SNA_GATEWAY         0x2100
OT_REMOTE_BRIDGE_SERVER    0x2600
OT_TCPIP_GATEWAY           0x2700
```

In addition, the following extended bindery object types are available:

```
OT_TIME_SYNCHRONIZATION_SERVER   0x2D00
OT_ARCHIVE_SERVER_DYNAMIC_SAP    0x2E00
OT_ADVERTISING_PRINT_SERVER      0x4700
OT_PRINT_QUEUE_USER              0x5300
```

If a zero is passed in as *serverSocket*, a socket is dynamically assigned.

QueryServices (DOS, Win)

Broadcasts a query to discover the identities of all servers of all types, all servers of a specific type, or the nearest server of a specific type

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS, Win

Service: Service Advertising Protocol (SAP)

Syntax

```
#include <nwipxspx.h>

int QueryServices(
   WORD   queryType,
   WORD   serverType,
   WORD   returnSize,
   SAP   *serviceBuffer);
```

Parameters

queryType

(IN) Indicates the type of query to be performed.

serverType

(IN) Indicates the type of server to respond to this query. This parameter must appear in low-high order, so call **IntSwap** when using NetWare defined server types.

returnSize

(IN) Indicates the byte size of how much data to return in *serviceBuffer*.

serviceBuffer

(OUT) Points to a buffer where the server types are returned.

Remarks

Only Nearest Service Queries are currently implemented.

queryType would be set to 3 for Nearest Server Query.

When querying for a downed or absent service, **QueryServices (DOS, Win)** can still return successfully. If *interveningNetworks* of the SAP structure is equal to 16, the service is down and other servers are returning the information.

Valid *serverType* parameters follow:

```
OT_WILD                  0xFFFF (only valid for General Service Queries
OT_UNKNOWN               0x0000
OT_USER                  0x0100
```

```
OT_USER_GROUP             0x0200
OT_PRINT_QUEUE            0x0300
OT_FILE_SERVER            0x0400
OT_JOB_SERVER             0x0500
OT_GATEWAY                0x0600
OT_PRINT_SERVER           0x0700
OT_ARCHIVE_QUEUE          0x0800
OT_ARCHIVE_SERVER         0x0900
OT_JOB_QUEUE              0x0A00
OT_ADMINISTRATION         0x0B00
OT_NAS_SNA_GATEWAY        0x2100
OT_REMOTE_BRIDGE_SERVER   0x2600
OT_TCPIP_GATEWAY          0x2700
```

In addition, the following extended bindery object types are available:

```
OT_TIME_SYNCHRONIZATION_SERVER   0x2D00
OT_ARCHIVE_SERVER_DYNAMIC_SAP    0x2E00
OT_ADVERTISING_PRINT_SERVER      0x4700
OT_PRINT_QUEUE_USER              0x5300
```

ShutdownSAP (DOS, Win)

Broadcasts notification of a SAP server's intention to shut down, thereby allowing bridges and file servers on the network to purge the SAP server from their tables

Server: 2.2, 3.11 and above, 4.0 and above

Platform: DOS, Win

Service: Service Advertising Protocol (SAP)

Syntax

```
#include <nwipxspx.h>

int ShutdownSAP(void);
```

Return Values

0x00	Successful
0x67	SAP ECB not cancelled
0x68	SAP ECB not advertising
0xFF	Failure

Remarks

ShutdownSAP (DOS, Win) issues a Cancel Event request against the ECB being used to schedule the 60-second SAP broadcasts.

SAP for NLM

AdvertiseService (NLM)

Advertises a server on the internetwork
Local Servers: nonblocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: Service Advertising Protocol (SAP)

Syntax

```
#include <sap.h>

LONG AdvertiseService (
   WORD   serverType,
   char  *serverName,
   WORD   serviceSocket);
```

Parameters

serverType

(IN) Specifies the value assigned by Novell for the server's service class.

serverName

(IN) Points to a 48-byte NULL-terminated string that is the server's unique name within the internetwork.

serviceSocket

(IN) Specifies the service socket number that a client wanting to communicate with the server via IPX™ or SPX™ protocols will use.

Return Values

AdvertiseService (NLM) returns a LONG handle associated with this particular call to **AdvertiseService (NLM)**. If **AdvertiseService (NLM)** fails, it returns a value of -1, and NetWareErrno is set to the appropriate error value.

Remarks

AdvertiseService (NLM) is used by servers to advertise the availability of the service they provide.

Servers should post listens (IPX or SPX) on the *serviceSocket* before calling **AdvertiseService (NLM)**. Otherwise, a potential client can attempt to use the advertised service and fail.

Example

AdvertiseService (NLM)

```
#include <sap.h>

#define TAPE_SERVER_TYPE 44
#define TAPE_SERVER_SOCKET 10011
LONG      SAPhandle;

SAPhandle = AdvertiseService (TAPE_SERVER_TYPE, "Tape_Server", TAPE_SER
```

FreeQueryServicesList (NLM)

Frees the list of SAP response structures after **QueryServices** has been called

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x

Service: Service Advertising Protocol (SAP)

Syntax

```
#include <sap.h>

int FreeQueryServicesList (
   SAP_RESPONSE_LIST_ENTRY   *listP);
```

Parameters

listP

(OUT) Points to the list of SAP response structures (returned by **QueryServices (NLM)**).

Return Values

FreeQueryServicesList (NLM) returns a value of 0 if successful.

QueryServices (NLM)

Returns the identities of all servers of all types, all servers of a specific type, or the nearest server of a specific type

Local Servers: blocking

Remote Servers: N/A

Classification: 2.x, 3.x, 4.x

Service: Service Advertising Protocol (SAP)

Syntax

```
#include <sap.h>

SAP_RESPONSE_LIST_ENTRY *QueryServices (
   WORD   queryType,
   WORD   serviceType);
```

Parameters

queryType

(IN) Specifies the type of query to be performed: (1) GENERAL_SERVICE_QUERY (3) NEAREST_SERVER_QUERY. See Service Advertising Packets for more Information.

serviceType

(IN) Indicates the type of server that should respond to this query. A value of 0xFFFF is a wildcard type that causes all server types to respond. The wildcard type is only valid for General Service Queries.

Return Values

QueryServices (NLM) returns a pointer to a singly-linked list of response structures. Each of the elements in the list is a structure defined in SAP.H and is called SAP_RESPONSE_LIST_ENTRY.

Upon failure, **QueryServices (NLM)** returns EFAILURE.

Remarks

The last response structure in the list has a NULL next pointer. Each response structure can contain information on up to eight servers (in the case of a General Service Query). The count field specifies exactly the number of servers for which SAP information is present in a given response structure.

```
#define SAP_RESPONSES_PER_PACKET    8
#define QUERY_LIST_SIGNATURE        0x454C5253

typedef struct SAPResponse
```

```
{
   WORD                 SAPPacketType;      /* 2 or 4 */
   struct
   {
      WORD              serverType;      /* assigned by Novell*/
      BYTE              serverName[48];  /* service name */
      InternetAddress   serverAddress;   /* server
                                            internetwork
                                            address */
      WORD              interveningNetworks;  /* # of networks
                                                 packet must
                                                 traverse */
   }  responses[SAP_RESPONSES_PER_PACKET];
   struct SAPResponse   *next;
   LONG                 signature;
   int                  count;
}  SAP_RESPONSE_LIST_ENTRY;
```

QueryServices (NLM) uses the **malloc** function to allocate the memory for each of the response structures in the list. The **FreeQueryServicesList (NLM)** function should be called when the list is no longer needed to free the memory it occupies.

ShutdownAdvertising (NLM)

Stops advertising a service
Local Servers: blocking
Remote Servers: N/A
Classification: 3.x, 4.x
Service: Service Advertising Protocol (SAP)

Syntax

```
#include <sap.h>

int ShutdownAdvertising (
   LONG   adHandle);
```

Parameters

adHandle

(IN) Specifies the advertising handle to be shut down (returned by **AdvertiseService (NLM)**).

Return Values

ShutdownAdvertising (NLM) returns a value of 0 if successful. Otherwise, it returns an error code (nonzero value).

Remarks

ShutdownAdvertising (NLM) stops the advertising of a service. More than one service can be advertised. The advertising handle identifies which service to shut down. An advertising handle must first be obtained with a call to **AdvertiseService (NLM)**.

Example

ShutdownAdvertising

```
#include <sap.h>

LONG   adHandle;
int    completionCode;
adHandle = AdvertiseService (TAPE_SERVER_TYPE,
   Tape_Server", TAPE_SERVER_SOCKET);
.
.
.
completionCode = ShutdownAdvertising (adHandle);
```

SAP: Structures

SAP

Service: SAP
Defined In: sap.h

Structure

```
typedef struct {
   IPXHeader   Header;
   WORD        ResponseType;          /* HI - LO */
   WORD        ServerType;            /* HI - LO */
   BYTE        ServerName[48];
   BTYE        Network[4];            /* hi - lo */
   BYTE        Node[6];               /* hi - lo */
   BYTE        Socket[2];             /* hi - lo */
   WORD        InterveningNetworks;   /* hi - lo */
}SAP;
```

Fields

Header

ResponseType

ServerType

ServerName

Network

Node

Socket

InterveningNetworks

Remarks

SAP is used by **QueryServices (DOS, Win)**.

SAP_RESPONSE_LIST_ENTRY

Service: SAP
Defined In: nwsap.h

Structure

```
#define SAP_RESPONSES_PER_PACKET   8

typedef struct {
   WORD                 SAPPacketType;       /*2 or 4*/
   struct {
       WORD             serverType;      /*assigned by Novell*/
       BYTE             serverName[48];  /*service name*/
       InternetAddress  serverAddress;   /*server internetwork address
       WORD             interveningNetworks; /*# of networks "hops"*/
   struct {
        SAPResponse   *next;
        LONG                 signature;
        int                  count;
  } SAP_RESPONSE_LIST_ENTRY;
```

Fields

SAPPacketType

serverType

serverName

serverAddress

interveningNetworks

responses

next

signature

count

Remarks

SAP_RESPONSE_LIST_ENTRY is used by **FreeQueryServicesList (NLM)** and **QueryServices (NLM)**.

TCP/IP and TCP/IPX

TCP/IP and TCP/IPX: Guides

TCP/IP and TCP/IPX: Concept Guide

TCP Implementation

TCP

UDP

NetWare Implementation of STREAMS-Based TLI for TCP/IP

TLI Function Notes for TCP/IP

See TCPIP TLI Client: Example and TCPIP TLI Server: Example.

TCP

NetWare Implementation of STREAMS-Based TLI for TCP/IP

TLI Function Notes for TCP/IPX

See TCPIP Socket Client: Example and TCPIP Socket Server: Example
.

TCP/IP and TCP/IPX: Concepts

Modes of Transport Service

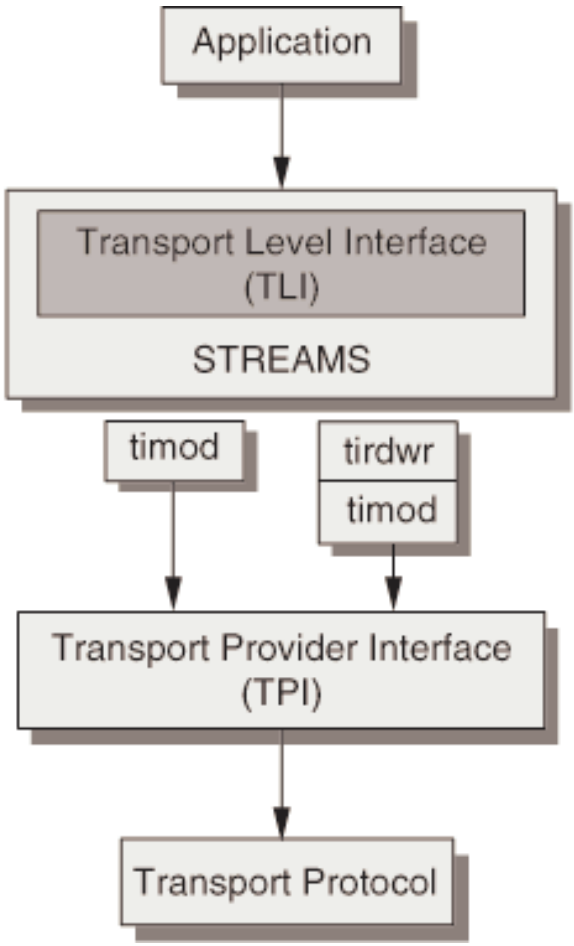
Two modes of transport service are provided:

TCP provides connection-oriented transport service.

UDP provides connectionless transport service.

The figure below illustrates how you can implement these modes.

Figure 13. NetWare TCP/IP Protocol Stack



Most TLI functions initiate an exchange of one or more Transport Provider Interface (TPI) primitives between the TPI and TLI.

The **t_open** function enables a user to choose a specific transport provider and establish an endpoint.

t_bind (Function) binds a transport address to the transport endpoint. Once **t_open** is called, the TLI Library pushes the timod module on top of the TPI driver. The timod module provides the functions that convert M_IOCTL messages, which are coming from the upstream, into M_PROTO messages for the TPI driver. It also converts the corresponding TPI replies into M_IOCACK messages.

When the transport service is in the connectionless mode (UDP), the user can begin the transfer of data using either the **t_sndudata** or **t_rcvudata** functions.

When the transport service is in the connection-oriented mode (TCP), the user can connect to another transport endpoint using **t_connect** and can transfer data using either **t_snd** or **t_rcv**.

The user can disconnect a connection using **t_snddis** or release the connection in an orderly manner using **t_sndrel** and **t_rcvrel**. Then, the user can unbind the transport address by calling **t_unbind** and later close the endpoint with **t_close**.

NetWare Implementation of STREAMS-Based TLI for TCP/IP

The implementation of TLI for TCP/IP includes the following:

Modes of Transport Service (connection-oriented and connectionless)

TCP Implementation

The netbuf and sockaddr_in structures (Network Buffer and Address Structures for TCP/IP)

NetWare Implementation of STREAMS-Based TLI for TCP/IPX

The implementation of TLI for TCP/IPX includes the following:

Modes of Transport Service (connection-oriented)

TCP Implementation

The netbuf and sockaddr_ipx structures (Network Buffer and Address Structures for TCP/IPX)

Network Buffer and Address Structures for TCP/IP

Various functions use the netbuf structure to send and receive data and information. The netbuf structure contains the following members:

unsigned int *maxlen*;

unsigned int *len*;

char **buf*;

The pointer *buf* points to a user input and/or output buffer. The *len* field generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function replaces the user value of *len* on return. Generally, the *maxlen* field is significant only when *buf* is used to receive output from the function.

In this case, *maxlen* specifies the physical size of the buffer, the maximum value of *len* that the function can set. If *maxlen* is not large enough to hold the returned information, a TBUFOVFLW error generally results. However, certain functions may return part of the data and not generate an error.

A sockaddr_in structure is used to represent a network address. This structure is defined as follows:

```
#include <types.h>
#include <netinet/in.h>
struct sockaddr_in {
   short sin_family;
   short sin_port;
   struct in_addr sin_addr;
char sin_zero[8];
};
```

Its members are as follows:

sin_family contains the AF_INET value.

sin_port corresponds to the TCP or UDP port number. It should be specified in network data order (which is not the same as host data order on the 80386-based workstation).

sin_addr contains a 4-byte Internet address, which is also in network data order.

sin_zero must be initialized to zero.

Parent Topic: NetWare Implementation of STREAMS-Based TLI for

Network Buffer and Address Structures for TCP/IPX

Various functions use the netbuf structure to send and receive data and information. The netbuf structure contains the following members:

unsigned int *maxlen*;

unsigned int *len*;

char **buf*;

The pointer *buf* points to a user input and/or output buffer. The *len* field generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function replaces the user value of *len* on return. Generally, the *maxlen* field is significant only when *buf* is used to receive output from the function.

In this case, *maxlen* specifies the physical size of the buffer, the maximum value of *len* that the function can set. If *maxlen* is not large enough to hold the returned information, a TBUFOVFLW error generally results. However, certain functions may return part of the data and not generate an error.

A sockaddr_ipx structure is used to represent a network address. This structure is defined as follows:

```
#include <types.h>
#include <netinet/in.h>
#define sipx_socket   sipx_addr.x_socket
#define sipx_port     sipx_socket
#define sipx_netnum   sipx_addr.x_net.cx_net
#define sipx_nodenum  sipx_addr.x_host.cx_host

struct sockaddr_ipx {
   short           sipx_family;
   struct ipx_addr sipx_addr;
   char            sipx_zero[2];
};
```

Its members are as follows:

sipx_family contains the AF_IPX value.

sipx_netnum contains a 4-character array that holds the network number.

sipx_nodenum contains a 6-character array that holds the node number.

sipx_port contains a 2-character array that holds the socket number.

sipx_zero must be initialized to zero.

Parent Topic: NetWare Implementation of STREAMS-Based TLI for TCP/IPX

TCP

Transmission Control Protocol (TCP) provides a connection-oriented transport service that is circuit-oriented and lets data travel over an established connection in a reliable, sequenced manner. You might use this service if your applications require datastream-oriented interactions.

TCP exhibits the following characteristics:

Provides an identification mechanism that avoids the overhead of transmitting an address and resolution during the data transfer phase.

Provides a context in which successive units of data, transferred between peer users, are logically related.

TCP Implementation

In addition to supporting all the regular TPI primitives, the TCP implementation also supports orderly release of a connection.

The user can use **t_sndrel** and **t_rcvrel** for the orderly release of a connection. Since orderly release of a connection is optional and other transport providers may not support these functions, the developer who is concerned with portability should avoid these functions.

As an alternative to using TLI routines, the user can issue read and write calls on a fully connected transport endpoint by pushing the tirdwr module. This module converts a TLI connection-oriented transport stream into a simple bidirectional pipe.

After pushing the tirdwr module, the user must cease to use TLI routines. The user can end a connection either by closing the file handle associated with the transport endpoint or by popping the tirdwr module off the STREAM.

NOTE: Since this implementation does not support expedited data transfer, it does not support Expedited Transport Service Data Units (ETSDUs). Since TCP does not support data unit boundaries, it does not support Transport Service Data Units (TSDUs), either. The T_MORE and T_EXPEDITED flags are ignored. Currently, calls to **ioctl** are not supported.

The close routine of the TPI adapter is called if either of the following occurs:

The user calls **t_close** explicitly.

The NLM™ application exits with a call to **exit**.

In the NetWare environment, an application can spawn multiple threads, with each thread having the capacity to open its own transport endpoint. The user is advised to close the endpoint by calling **t_close** for each thread; this is advisable since the NetWare API (CLIB) function **ExitThread** does not close the endpoint explicitly.

For a description of standard functions such as **htons** and **htonl** that convert short (16-bit) and long (32-bit) values between host and network data order, see Internet Network Library.

TCP/IP and TCP/IPX Notes: t_accept

The transport endpoint that accepts the connection inherits the following:

The IP address and TCP port number of the endpoint that received the connect indication

The IP address and TCP port number of the endpoint that initiated the connection

TCP considers the connection to be established even before the user issues **t_accept**. If the user rejects the connection with **t_snddis**, TCP aborts the connection.

If the connection is accepted on the endpoint on which the connect indication was received (*resfh==fh*), the endpoint ceases to be a listening endpoint. However, the user can unbind the endpoint with **t_unbind** and bind again with **t_bind (Function)** to make this endpoint a listening endpoint.

If the peer aborts the connection before the user has called **t_accept**, the user is not informed that the connection was aborted with a disconnect indication during **t_accept**. However, the endpoint on which the connection was accepted (that is, *fh*) is informed of the disconnect with a disconnect indication. A listening endpoint is never informed of a disconnect during **t_accept**.

The user must pass the sequence number that **t_listen** returned earlier. Since this implementation does not support the transfer of data during the call to **t_accept**, *udata.len* must be set to zero. Since options are not supported during the **t_accept** call either, *opt.len* must be set to zero.

After a successful call to **t_listen**, the server typically opens and binds a new endpoint. Next, it calls **t_accept**, passing the file handle of the new endpoint. Upon accepting the connection, the server spawns a new thread to send and receive data and then goes back to listen for any other incoming

connections.

TCP/IP and TCP/IPX Notes: t_close

The close routine of the transport provider is called whenever the user calls **t_close**. In the case of TCP, if **t_close** is called from any state other than T_UNBND, TCP aborts the connection; all the data queued to be sent and all the data that had been received are discarded.

The user is advised to close the endpoint explicitly with **t_close** if the application involves more than one thread. This is necessary because **ExitThread** does not close the opened endpoints.

TCP/IP and TCP/IPX Notes: t_open

This implementation supports the following devices:

/dev/tcp for a connection-oriented transport service based on TCP.

/dev/udp for a connectionless transport service based on UDP.

The default characteristics of the underlying transport protocol are described in **t_getinfo**. There is no assigned limit to the number of devices that a user can open.

TCP/IP and TCP/IPX Notes: t_rcv

The maximum amount of data that a single **t_rcv** call can receive is limited by STRMSGZ in STREAM.H. Typically, the maximum is 4,096 bytes. Since this implementation does not support expedited data transfer, it does not support ETSDUs. Since TCP does not support data unit boundaries, it does not support TSDUs, either. T_MORE and T_EXPEDITED flags must be set to zero.

TCP/IP and TCP/IPX Notes: t_rcvdis

If a connection is disconnected, TCP informs the user. The reason field in the t_discon structure has one of the following errors:

ETIMEDOUT: The peer failed to acknowledge repeated attempts to deliver data, window probes, or keep-alive probes.

ECONNRESET: The peer reset the connection.

TCPE_UNLOADING: TCP is either unloading or has been unloaded.

If an active endpoint is disconnected, TCP informs the user immediately. If a passive endpoint is disconnected, the transport provider waits for the user to respond to an earlier successful **t_listen** call and then informs the provider that the connection was disconnected.

The user cannot call **t_connect** again on this endpoint because the protocol address bound to this endpoint is no longer valid. Instead, the user can unbind the endpoint and bind it again.

This implementation does not support any data with the disconnect indication.

TCP/IP and TCP/IPX Notes: t_rcvrel

This implementation supports the orderly closure of a connection. When **t_rcvrel** is successful, the transport provider does not deliver any more data to the user. The user cannot call **t_connect** again on this endpoint because the bound protocol address is no longer valid. Instead, the user can unbind the endpoint with **t_unbind** and bind it again with **t_bind (Function)**.

Typically, a user can call **t_sndrel** for an orderly close if there is no more data to send after a successful call to **t_rcvrel**. A connection is considered to be fully closed if **t_sndrel** follows **t_rcvrel**, or vice versa.

TCP/IP and TCP/IPX Notes: t_snd

The maximum data that can be sent in a single call to **t_snd** is 65,536 bytes. Because this implementation does not support expedited data transfer, it does not support ETSDUs. As a result, T_MORE and T_EXPEDITED flags must be set to zero. Since TCP does not support data unit boundaries, it does not support TSDUs, either.

TCP/IP and TCP/IPX Notes: t_snddis

If the endpoint is active, **t_snddis** causes all the data on the transmitting and receiving queues to be discarded, and the connection is aborted. If the endpoint is passive, TCP considers the connection established when the user is informed of a connect indication. When the user issues **t_snddis**, this connection is aborted. The user cannot call **t_connect** again on this endpoint because the bound address is rendered invalid after **t_snddis**, even though the endpoint is in T_IDLE state after a call to **t_snddis**. To reuse the endpoint for a connection, the user must unbind the address with **t_unbind** and bind it again with **t_bind (Function)**.

Transfer of data is not supported during **t_snddis**; therefore, *udata.len* must

be set to zero.

TCP/IP and TCP/IPX Notes: t_sndrel

This implementation supports the orderly release of a connection. TCP ensures the delivery of data to its destination before initiating a closure.

If the local endpoint initiates the closing of a connection and the remote endpoint responds with a **t_sndrel** call, TCP waits approximately two minutes before considering the connection fully closed. The user cannot call **t_connect** again on this endpoint because the bound address is rendered invalid after **t_sndrel**, even though the endpoint is in T_IDLE state after a **t_sndrel**. To reuse the endpoint for a connection, the user must unbind the address with **t_unbind** and bind it again with **t_bind (Function)**.

Typically, the user can call either **t_rcvrel** after a successful **t_sndrel** call or **t_sndrel** after a successful **t_rcvrel** call.

TCP/IP Notes: t_bind

A STREAM endpoint is bound when it is associated with an IP address and a transport port number. A user can request that the transport provider either bind the endpoint to a specific transport port number and IP address, or allocate a port number and an IP address.

The transport provider returns the port number and IP address in the sockaddr_in structure defined in the SOCKET.H and BSDSKT.H files. The *buf* field in t_bind (Structure) points to the sockaddr_in structure. The sockaddr_in structure represents a network address. Its elements are as follows:

sin_family is AF_INET for IPs.

sin_port specifies the local transport-level port number.

sin_addr specifies the local IP address.

sin_zero must be initialized to zero.

The IP address and the port number must be in network byte order.

If the user specifies a port number as zero, the provider assigns a port number from a list of available port numbers in the nonreserved range (1024 to 5000, inclusive). If the user specifies an IP address of zero, IP assigns the default local IP address.

If the user specifies a nonzero port number and IP address, TCP assigns the port number and IP address if they are available. TCP does not assign the requested port number and IP address if:

The requested port is already being used by another endpoint that is listening.

The requested port number is used by another endpoint that is not yet connected.

In the case of UDP, the requested port number is assigned only if it is available and no other endpoint is using it.

The *len* field in the addr structure contains the length of the protocol address, which is equal to `sizeof(struct sockaddr_in)`.

After a successful call to **t_bind (Function)** in the connection-oriented mode (TCP), the user can listen for a connect indication if the value of qlen is greater than zero or the user can initiate a connection. Currently, there is no assigned limit to the value of *qlen*.

When the transport service is in the connectionless mode (UDP), the user can begin to send and receive data.

Typically, a client does not specify the port and IP address. On the other hand, the server usually specifies a well-known port and IP address.

The user is advised to check the port number and IP address that the transport provider returns to determine whether the endpoint was indeed bound with the requested address.

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_connect

The sockaddr_in structure specifies the destination address. The *buf* field in the t_call structure points to the sockaddr_in structure. The structure's elements are as follows:

sin_family is AF_INET for IPs.

sin_port specifies the destination port.

sin_addr specifies the destination IP address.

sin_zero must be initialized to zero.

Both the destination port number and destination IP address must be in network byte order.

The user can call **t_connect** on either an active or a passive endpoint. The only exceptions, in the case of a passive endpoint, are that the endpoint must be in T_IDLE state and it should not have received a connect indication earlier. Once a **t_connect** call is issued, the passive endpoint ceases to listen; it becomes an active endpoint for the duration of the

connection, until **t_unbind** unbinds the endpoint. To make the endpoint passive again after issuing **t_unbind**, the user can bind again using **t_bind (Function)**.

If the destination address is 0.0.0.0 or 127.0.0.0, the host defaults to the local address. A destination address of 255.255.255.255 is invalid.

The *len* field in the addr structure contains the length of the protocol address, which is equal to `sizeof (struct sockaddr_in)`. Since options are not supported during **t_connect**, the *len* field in the opt structure must be set to zero.

To confirm a connection, the user can specify a non-NULL value for the *rcvcall* argument. The *addr.maxlen* field must be set to `sizeof(struct sockaddr_in)`. On return in *rcvcall*, *addr.buf* points to the sockaddr_in structure, which contains the address of the responding endpoint. Options and transfer of data are not supported during the **t_connect** call; therefore, *opt.len* and *udata* must be set to zero.

Unless the user unbinds and binds a connected endpoint that was either disconnected or released earlier, that endpoint cannot be reused.

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_getinfo

The protocol-dependent information about the transport layer is returned in the info structure. The fields returned have the following values:

addr	size of struct sockaddr_in (16 bytes)
options	-2
tsdu	0 for TCP, 4096 for UDP
etsdu	-2
connect	-2
discon	-2
servtype	T_COTS_ORD (for TCP) T_CLTS (for UDP)

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_listen

On receiving a connect indication from TCP, the transport provider accepts the connection, if possible, and queues the indication. TCP considers the

connection to have been established.

In the t_call structure, the *buf* field points to the sockaddr_in structure, which is used to represent a network address. The structure's elements are as follows:

sin_family is AF_INET for IPs.

sin_port contains the TCP port number of the endpoint initiating the connection.

sin_addr contains the IP address of the node initiating the connection.

sin_zero must be initialized to zero.

The TCP port number and the IP address are in network byte order.

The transport provider assigns a unique sequence number greater than -1 and informs the user of a pending connect indication. When issuing a response to this connection, the user must use the sequence number and information in the sockaddr_in structure.

Since options are not supported during the **t_listen** call and data is not passed to the user, *opt.len* and *udata.len* are set to zero.

The transport provider queues connect indications from TCP up to the maximum specified during the **t_bind (Function)** call in *qlen*. The transport provider rejects further connect indications until the number of queued connect indications is less than *qlen*. The user should respond to a successful **t_listen** call by calling either **t_accept** or **t_snddis**.

Typically, the server listens for connect indications by issuing **t_listen**. After a successful **t_listen** call, the server opens a new endpoint by calling **t_open** and binds that endpoint with **t_bind (Function)**. The server accepts the connection by passing the *fh* of the new endpoint in **t_accept**. The server then listens for any further connect indications. Thus, the server endpoint can be dedicated for listening for any connect indications. If the server does not want to accept a connection, it must call **t_snddis** and continue to listen for further connect indications.

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_rcvconnect

In the addr structure, the *buf* field points to the sockaddr_in structure, which is used to represent a network address. The structure's elements are as follows:

sin_family is AF_INET for IPs.

sin_port contains the TCP port number of the endpoint initiating the connection.

sin_addr contains the IP address of the node initiating the connection.

The TCP port number and the IP address are in network byte order.

This implementation does not support any options or data transfer during **t_rcvconnect**; therefore, opt.len and udata.len are set to zero.

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_rcvudata

The maximum data that a single **t_rcvudata** call can receive is 4,096 bytes. The *buf* field in the addr structure points to a sockaddr_in structure, which is used to represent a network address. The structure's elements are as follows:

sin_family is AF_INET for IPs.

sin_port specifies the UDP port number of the source endpoints.

sin_addr specifies the IP address of the source endpoint.

sin_zero must be initialized to zero.

Both the UDP port number and the IP address are returned in network byte order. Because options are not supported during the call to **t_rcvudata**, *opt.len* is set to zero.

Parent topic: TLI Function Notes for TCP/IP

TCP/IP Notes: t_sndudata

The maximum data that a single call to **t_sndudata** can send is limited by STRMSGZ, as defined in STREAM.H. Typically, the maximum is 4,096 bytes.

The *buf* field in the addr structure points to the sockaddr_in structure, which is used to represent a network address.The structure's elements are as follows:

sin_family is AF_INET for IPs.

sin_port specifies the destination UDP port number.

sin_addr specifies the destination IP address.

sin_zero must be initialized to zero.

Both the UDP port number and the IP address must be in network byte

order.

To broadcast a datagram to all nodes, the user can specify a destination address of 255.255.255.255. To send a datagram to the local node alone, the user can specify a destination address of 127.x.x.x. (the loopback address) or 0.0.0.0.

Because options are not supported during **t_sndudata**, *opt.len* must be set to zero.

Parent topic: TLI Function Notes for TCP/IP

TCP/IPX Notes: t_bind

A STREAM endpoint is bound when it is associated with an IP address and a transport port number. A user can request that the transport provider either bind the endpoint to a specific transport port number and IPX address, or allocate a port number and an IPX address.

The transport provider returns the port number and IPX address in the sockaddr_ipx structure defined in NETIPX/IPX.H. The *buf* field in the t_bind structure points to the sockaddr_ipx structure. The sockaddr_ipx structure represents a network address. Its elements are as follows:

sipx_family is AF_IPX for IPX.

sipx_netnum contains a 4-character array that holds the network number.

sipx_nodenum contains a 6-character array that holds the node number.

sipx_port contains a 2-character array that holds the socket number.

sipx_zero must be initialized to zero.

The IPX address and the port number must be in network byte order.

If the user specifies a port number as zero, the provider assigns a port number from a list of available port numbers in the nonreserved range (1024 to 5000, inclusive). If the user specifies an IPX address of zero, IPX assigns the default local IPX address.

If the user specifies a nonzero port number and IPX address, TCP assigns the port number and IPX address if they are available. TCP does not assign the requested port number and IPX address if:

The requested port is already being used by another endpoint that is listening.

The requested port number is used by another endpoint that is not yet connected.

The *len* field in the addr structure contains the length of the protocol address,

which is equal to `sizeof(struct sockaddr_ipx)`.

After a successful call to **t_bind (Function)** in the connection-oriented mode (TCP), the user can listen for a connect indication if the value of *qlen* is greater than zero or the user can initiate a connection. Currently, there is no assigned limit to the value of *qlen*.

When the transport service is in the connectionless mode (UDP), the user can begin to send and receive data.

Typically, a client does not specify the port and IPX address. On the other hand, the server usually specifies a well-known port and IPX address.

The user is advised to check the port number and IPX address that the transport provider returns to determine whether the endpoint was indeed bound with the requested address.

Parent topic: TLI Function Notes for TCP/IPX

TCP/IPX Notes: t_connect

The sockaddr_ipx structure specifies the destination address. The *buf* field in the t_call structure points to the sockaddr_ipx structure. The structure's elements are as follows:

sipx_family is AF_IPX for IPX.

sipx_netnum contains a 4-character array that holds the network number.

sipx_nodenum contains a 6-character array that holds the node number.

sipx_port contains a 2-character array that holds the socket number.

sipx_zero must be initialized to zero.

Both the destination port number and destination IPX address must be in network byte order.

The user can call **t_connect** on either an active or a passive endpoint. The only exceptions, in the case of a passive endpoint, are that the endpoint must be in T_IDLE state and it should not have received a connect indication earlier. Once a **t_connect** is called, the passive endpoint ceases to listen; it becomes an active endpoint for the duration of the connection, until **t_unbind** unbinds the endpoint. To make the endpoint passive again after issuing **t_unbind**, the user can bind again using **t_bind (Function)**.

The *len* field in the addr structure contains the length of the protocol address, which is equal to `sizeof (struct sockaddr_ipx)`. Because options are not supported during **t_connect**, the *len* field in the opt structure must be set to zero.

To confirm a connection, the user can specify a non-NULL value for the

rcvcall argument. The addr.maxlen field must be set to `sizeof(struct sockaddr_in)`. On return in *rcvcall*, *addr.buf* points to the sockaddr_in structure, which contains the address of the responding endpoint. Options and transfer of data are not supported during the **t_connect** call; therefore, *opt.len* and *udata* must be set to zero.

Unless the user unbinds and binds a connected endpoint that was either disconnected or released earlier, that endpoint cannot be reused.

Parent topic: TLI Function Notes for TCP/IPX

TCP/IPX Notes: t_getinfo

The protocol-dependent information about the transport layer is returned in the info structure. The fields returned have the following values:

addr	size of struct sockaddr_in (16 bytes)
options	-2
tsdu	0 for TCP, 4096 for UDP
etsdu	-2
connect	-2
discon	-2
servtype	T_COTS_ORD (for TCP)

Parent topic: TLI Function Notes for TCP/IPX

TCP/IPX Notes: t_listen

On receiving a connect indication from TCP, the transport provider accepts the connection, if possible, and queues the indication. TCP considers the connection to have been established.

In the t_call structure, the *buf* field points to the sockaddr_ipx structure, which is used to represent a network address. The structure's elements are as follows:

sipx_family is AF_IPX for IPX.

sipx_netnum contains a 4-character array that holds the network number.

sipx_nodenum contains a 6-character array that holds the node number.

sipx_port contains a 2-character array that holds the socket number.

sipx_zero must be initialized to zero.

The TCP port number and the IPX address are in network byte order.

The transport provider assigns a unique sequence number greater than -1 and informs the user of a pending connect indication. When issuing a response to this connection, the user must use the sequence number and information in the sockaddr_ipx structure.

Since options are not supported during the **t_listen** call and data is not passed to the user, *opt.len* and *udata.len* are set to zero.

The transport provider queues connect indications from TCP up to the maximum specified during the **t_bind (Function)** call in *qlen*. The transport provider rejects further connect indications until the number of queued connect indications is less than *qlen*. The user should respond to a successful **t_listen** call by calling either **t_accept** or **t_snddis**.

Typically, the server listens for connect indications by calling **t_listen**. After a successful **t_listen** call, the server opens a new endpoint by calling **t_open** and binds that endpoint with **t_bind (Function)**. The server accepts the connection by passing the *fh* of the new endpoint in **t_accept**. The server then listens for any further connect indications. Thus, the server endpoint can be dedicated for listening for any connect indications. If the server does not want to accept a connection, it must call **t_snddis** and continue to listen for further connect indications.

Parent topic: TLI Function Notes for TCP/IPX

TCP/IPX Notes: t_rcvconnect

In the addr structure, the *buf* field points to the sockaddr_ipx structure, which is used to represent an IPX network address. The structure's elements are as follows:

sipx_family is AF_IPX for IPX.

sipx_netnum contains a 4-character array that holds the network number.

sipx_nodenum contains a 6-character array that holds the node number.

sipx_port contains a 2-character array that holds the socket number.

sipx_zero must be initialized to zero.

The TCP port number and the IP address are in network byte order.

This implementation does not support any options or data transfer during **t_rcvconnect**; therefore, *opt.len* and *udata.len* are set to zero.

Parent topic: TLI Function Notes for TCP/IPX

TLI Function Notes for TCP/IP

The implementation notes in this section provide information specific to use of TLI functions for accessing the transport protocol TCP/IP in NetWare 3.11 and later. These implementation notes supplement the function notes in Overview of TLI Functions.

Notes for some TLI functions are the same for both TCP/IP and TCP/IPX. See the following information for using TLI with TCP/IP:

TCP/IP and TCP/IPX Notes: t_accept

TCP/IP Notes: t_bind

TCP/IP and TCP/IPX Notes: t_close

TCP/IP Notes: t_connect

TCP/IP Notes: t_getinfo

TCP/IP Notes: t_listen

TCP/IP and TCP/IPX Notes: t_open

t_optmgmt (this implementation does not support any options)

TCP/IP and TCP/IPX Notes: t_rcv

TCP/IP Notes: t_rcvconnect

TCP/IP and TCP/IPX Notes: t_rcvdis

TCP/IP and TCP/IPX Notes: t_rcvrel

TCP/IP Notes: t_rcvudata

TCP/IP and TCP/IPX Notes: t_snd

TCP/IP and TCP/IPX Notes: t_snddis

TCP/IP and TCP/IPX Notes: t_sndrel

TCP/IP Notes: t_sndudata

See TCPIP TLI Client: Example and TCPIP TLI Server: Example.

TLI Function Notes for TCP/IPX

The implementation notes in this section provide information specific to use of TLI functions for accessing the transport protocol TCP/IPX in NetWare

3.11 and later. These implementation notes supplement the function notes in Overview of TLI Functions.

Notes for some TLI functions are the same for both TCP/IP and TCP/IPX. See the following information for using TLI with TCP/IPX:

TCP/IP and TCP/IPX Notes: t_accept

TCP/IPX Notes: t_bind

TCP/IP and TCP/IPX Notes: t_close

TCP/IPX Notes: t_connect

TCP/IPX Notes: t_getinfo

TCP/IPX Notes: t_listen

TCP/IP and TCP/IPX Notes: t_open

t_optmgmt (this implementation does not support any options)

TCP/IP and TCP/IPX Notes: t_rcv

TCP/IPX Notes: t_rcvconnect

TCP/IP and TCP/IPX Notes: t_rcvdis

TCP/IP and TCP/IPX Notes: t_rcvrel

t_rcvudata (not supported)

TCP/IP and TCP/IPX Notes: t_snd

TCP/IP and TCP/IPX Notes: t_snddis

TCP/IP and TCP/IPX Notes: t_sndrel

t_sndudata (not supported)

See TCPIP Socket Client: Example and TCPIP Socket Server: Example.

UDP

User Datagram Protocol (UDP) provides a connectionless transport service that is message oriented. It supports transfer of data in self-contained units or in datagrams, with no logical relationship required among multiple datagrams.

This service requires only a preexisting association between the peer users involved. This association determines the characteristics of the data to be transmitted. You may use this service if your applications

Need short-term request/response interactions

Exhibit a high level of redundancy

Are dynamically reconfigurable

Do not require guaranteed, sequenced delivery of data

UDP has the following characteristics:

It does not support the dynamic negotiation of parameters and options.

It features single-service access (SSA), which does not necessarily relate to any other service access. SSA presents the transport provider with all the information required to deliver a datagram (for example, a destination address), together with the data to be transmitted.

Each transmitted datagram is self-contained, and the transport provider can independently route it.

See UDP/IP Client: Example and UDP/IP Server: Example.

See UDP/IP TLI Client: Example and UDP/IP TLI Server: Example.

TLI

TLI: Guides

TLI: Concept Guide

Overview of Transport Protocols

This provides an introduction to the Transport Level Interface (TLI) for the NetWare® OS. This interface provides a standard means of gaining direct access to transport services. The discussion in the following topics assumes a working knowledge of C language programming and data communication concepts.

TLI Connection Mode Services

This chapter describes the connection-mode service of TLI. As discussed in Overview of Transport Protocols, the connection mode service can be illustrated using a client-server paradigm.

The concepts of connection mode service can be presented using two programming examples. The first example illustrates how a client establishes a connection to a server and then communicates with it; the second example shows the server's side of the interaction.

In these examples, the client establishes a connection with a server and then the server transfers a file to the client. The client, in turn, receives the data from the server and writes it to its standard output file.

TLI Connectionless Mode Services

Connectionless mode service is appropriate for short-term request/response interactions, such as transaction processing applications. Data is transferred in self-contained units with no logical relationship required among multiple units.

This discussion describes the connectionless mode services using a transaction server as an example. This server waits for incoming transaction queries and then processes and responds to each query.

Datagram Errors

A TLI Read/Write Interface: Guide

A TLI Read/Write Interface

Write

Read

Close

Overview of TLI Functions: Guide

Overview of TLI Functions

TLI Terms

TLI Error Handling

Synchronous and Asynchronous Execution Modes

Overview of Connection-Oriented Service

Overview of Connectionless Service

Transport Provider States

NetWare IPX/SPX/SPX II: Guide

NetWare IPX/SPX/SPX II

SPX Protocol

SPX II Protocol

IPX Protocol

NetWare Implementation of STREAMS-Based IPX/SPX/SPX II

Environment Enhancement for TLI

TLI Function Implementation Notes for IPX/SPX/SPX II

NetWare OSI: Guide

NetWare OSI

OSI Address Format

Options Management Structures

TLI Function Implementation Notes for OSI

TLI State Transitions

These tables describe the state transitions, events, and service types associated with TLI.

TLI States

Outgoing Events

Incoming Events

Transport User Actions

TLI: Concepts

A TLI Read/Write Interface

A user might want to establish a transport connection and use the read and write functions for I/O needs. TLI does not directly support a read/write interface to a transport provider, but one is available with NetWare® 4.x. This interface enables a user to issue read and write calls over a transport connection that is in the data transfer phase.

This interface is not available with the connectionless mode service.

The read/write interface is presented using the client example in TLI Connection Mode Services with some minor modifications. The clients are identical until the data transfer phase is reached.

At that point, this client uses the read/write interface and process incoming data.

The following shows only the differences between this client and that of the example in TLI Connection Mode Services:

TLI Read/Write Interface

```
#include <io.h>          /* read, write, ioctl                */
#include <sys/stropts.h>/* I_PUSH                             */
   .
   .
   .
   if (t_connect(fh, sndcall, NULL) == -1)
   {
      t_error("t_connect");
      exit(1);
   }

   /* ioctl push tirdwr module */
   if ( ioctl(fh, I_PUSH, "tirdwr") == -1 )
   {
      perror("ioctl");
      exit(1);
   }

   /* Receive line at a time and print it on the screen */
   while( read(fh, iobuf, (LONG)sizeof(iobuf)) > 0 )
   {
      printf("%s",iobuf);
   }
```

```
    }

/* Free allocated structures */
    if( t_free((char *) sndcall, T_CALL) == -1 )
    {
       t_error("t_free");
       exit(1);
    }

    /* Close the endpoint */
    if( close(fh) == -1 )
    {
       perror("close");
       exit(1);
    }
    exit(0);
}
```

The client invokes the read/write interface by pushing the **tirdwr** module onto the STREAM associated with the transport endpoint where the connection was established. This module converts TLI above the transport provider into a pure read/write interface. Because the transport endpoint identifier is a file handle, the **read** and **close** functions can be executed.

Because TLI uses the STREAMS service, the facilities of this character I/O mechanism can be used to provide enhanced user services. By pushing the **tirdwr** module above the transport provider, the user's interface is effectively changed. The semantics of read and write must be followed, and message boundaries are not preserved.

> **NOTE:** The tirdwr module can be pushed onto a STREAM only when the transport endpoint is in the data transfer phase. Once the module is pushed, the user must not call any TLI routines. If a TLI routine is invoked, **tirdwr** generates a fatal protocol error, EPROTO, on that STREAM, rendering it unusable. Furthermore, if the user pops the **tirdwr** module off the STREAM, the transport connection is aborted.

Subsequent sections describe the exact semantics of **write**, **read**, and **close** using **tirdwr**. To summarize, the **tirdwr** module enables a user to send and receive data over a transport connection using **read** and **write**. This module translates all TLI indications into the appropriate actions. The connection can be released with the **close** system function.

Related Topics

Write

Read

Close

Abortive Connection Release

At any point during data transfer, the user can release the transport connection and end the conversation.

In the case of abortive release, the connection terminates immediately and can result in the loss of any data that has not yet reached the destination user.

Either user can call **t_snddis** to generate an abortive release. Also, the transport provider can abort a connection if a problem occurs below TLI.

The **t_snddis** function enables a user to send data to the remote user when aborting a connection. Although all transport providers support the abortive release, not all of them support the ability to send data when aborting a connection.

When the remote user is notified of the aborted connection, **t_rcvdis** must be called to retrieve the disconnect indication. This function returns a reason code that identifies why the connection was aborted. It also returns any user data that might have accompanied the disconnect indication (if the abortive release was initiated by the remote user). This reason code is specific to the underlying transport protocol and should not be interpreted by protocol-independent software.

Related Topics

Abortive Connection Release: the Server

Abortive Connection Release: the Client

Parent Topic: TLI Connection Mode Services

Abortive Connection Release: the Client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until **t_rcv** fails. If the server releases the connection (using **t_snddis**), **t_rcv** fails and sets *t_errno* to TLOOK. The client then processes the connection release as follows:

TLI Abortive Connection Release---the Client

```
/* Receive a connection closure */
   if( t_rcvdis(fh, NULL) == -1 ) {
      t_error("t_rcvdis");
      exit(1);
   }

   /* Unbind the address from the endpoint */
   if( t_unbind(fh) == -1 ) {
      t_error("t_unbind");
      exit(1);
```

```
    }
    /* Free allocated structures */
    if( t_free((char *) sndcall, T_CALL) == -1 ) {
       t_error("t_free");
       exit(1);
    }

    /* Close the endpoint */
    if( t_close(fh) == -1 ) {
       t_error("t_close");
       exit(1);
    }
    exit(0);
}
```

After receiving and processing the disconnection, the client unbinds the transport endpoint, closes it, and frees used resources.

Each user must take steps to prevent data loss. For example, the user can insert a special byte pattern in the data stream to indicate the end of a conversation. There are many possible routines for preventing data loss. Each application and high-level protocol must choose an appropriate routine, given the target protocol environment and requirements.

Parent Topic: Abortive Connection Release

Abortive Connection Release: the Server

The following client-server example assumes that the transport provider supports the abortive release of a connection. When the server has transferred all the data, the connection can be disconnected as follows:

TLI Abortive Connection Release---the Server

```
/* Send disconnect to the client */
   if( t_snddis(*fh, NULL) == -1 ) {
      t_error("t_snddis");
      t_close(*fh);
      free(fh);
      return;
   }
/* Unbind TEP */
   if( t_unbind(*fh) == -1 ) {
      t_error("t_unbind");
      t_close(*fh);
      free(fh);
      return;
   }

   t_close(*fh);
   free(fh);
```

```
}
```

Parent Topic: Abortive Connection Release

Abortive Release Summary

An abortive release is the only release supported by the SPX™ protocol. Call **t_snddis** to perform an abortive release. This function breaks the connection without preparing the partner, who receives only an indication that the connection was broken. As input, this function takes the local endpoint and t_call:

```
int t_snddis(int fh, struct t_call *call);
```

call is optional. You can use it to provide information to the partner about the release. No more data can be sent or received on the connection after you call **t_snddis**.

If the partner sends an abortive release, you receive an error value while processing the connection. The release is reported in *t_errno* as T_DISCONNECT. Call **t_look** to confirm the release, then call **t_rcvdis** to close your side of the connection. As input, this function takes the local endpoint and t_discon to receive any information explaining the reason for releasing the connection:

```
int t_rcvdis(int fh, struct t_discon *discon);
```

The disconnect information is protocol dependent. Examples of reasons for disconnecting might be a failed connection or a malformed packet.

Parent Topic: Releasing a Connection in TLI

Asynchronous Events

A function that attempts to receive data in synchronous mode waits until data arrives before returning control to the user. This is the default mode of execution. It is useful for user processes that want to wait for events to occur, or for user processes that have no other useful work to perform.

The asynchronous mode of operation, on the other hand, can notify a user of some event without forcing the user to wait for the event. This enables users to work while waiting for a particular event.

For example, a function that attempts to receive data in asynchronous mode returns control to the user immediately if no data is available. The user can then periodically poll for incoming data until it arrives. The asynchronous mode is intended for those applications that expect long delays between events and have other tasks that they can perform in the meantime.

The two execution modes are not provided through separate interfaces or different functions. Instead, functions that process incoming events have two modes of operation: synchronous and asynchronous. The asynchronous mode is specified through the O_NDELAY flag. This flag can be set when the transport provider is initially opened or before any specific function or group of functions is executed by calling **t_nonblocking**. The effect of this flag is specified in the description of each function.

A process that calls functions in synchronous mode must still be able to recognize certain asynchronous events immediately and act on them, if necessary. This is handled through a the transport error TLOOK, which is returned by a function when an asynchronous event occurs. The **t_look** function is then called to identify the specific event that has occurred.

You can accomplish asynchronous processing through polling. The polling capability enables processes to do useful work and periodically poll for one of the asynchronous events listed in the table inSynchronous and Asynchronous Execution Modes Set the O_NDELAY flag for the appropriate functions and use **t_look** to poll.

Parent Topic: Synchronous and Asynchronous Execution Modes

Asynchronous Mode for Some TLI Functions

Many TLI library routines might block while waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, you might want to perform local processing while waiting for some asynchronous transport interface event to occur.

Support for asynchronous processing of TLI events is available to applications using a combination of the STREAMS asynchronous features and the nonblocking mode of TLI functions. Examples in previous chapters have illustrated the use of **poll** for processing events asynchronously.

In addition, each TLI function that might block while waiting for some event can be run in a special nonblocking mode using the NetWare® specific function **t_nonblocking**.

t_nonblocking puts the transport endpoint into a nonblocking mode. It forces the stream head associated with the transport endpoint to return an error code to calling processes if their requests cannot be completed immediately.

For example, **t_listen** normally blocks, waiting for a connect indication. However, a server can periodically poll a transport endpoint for existing connect indications by calling **t_listen** in the nonblocking (or asynchronous) mode. The asynchronous mode is enabled by setting O_NDELAY or O_NONBLOCK on the file handle. These can be set as flags on **t_open** or by calling **fcntl** before calling the TLI function. The **fcntl** function can be used to

enable or disable this mode at any time.

The NetWare specific function **t_blocking** puts the transport into a blocking mode. The blocking mode forces NetWare to make the user's thread sleep until the I/O is available. This function can be useful in a multitasking environment because it is not always desirable to do "busy waiting." This is the situation where the user consumes processor cycles checking for work, only to find that there is no work. Because NetWare is a nonpreemptive OS, this means that no other threads can work if the user is checking for work.

O_NDELAY or O_NONBLOCK affect each TLI function differently. To determine the exact semantics of O_NDELAY or O_NONBLOCK for a particular function, see the function description in TLI: Functions.

See TLI Asynchronous Mode: Advanced Programming Example for a programming example.

Parent Topic: TLI Asynchronous Execution Mode

Blocking and Nonblocking Modes

Most TLI functions can execute in blocking or nonblocking modes (also called synchronous and asynchronous modes). The endpoint that the operation is performed on controls the blocking mode. In blocking mode, a function doesn't return until it has completed the operation. In nonblocking mode, the function returns as quickly as possible but in many cases continues processing the operation in the background.

TLI determines an endpoint's blocking mode at the time you open the endpoint. **t_open** opens an endpoint. You can request an endpoint be opened in nonblocking mode by passing **t_open** O_NDELAY. Otherwise, **t_open** opens the endpoint in blocking mode. Use **t_getinfo** to find an endpoint's blocking status after it is opened.

t_rcv is a good example of how a function operates differently under blocking and nonblocking. In blocking mode, **t_rcv** blocks until it receives data from the transport provider. That means if you call **t_rcv** in blocking mode, you lose control until data arrives at the endpoint.

In nonblocking mode, **t_rcv** receives any data that is available and returns. If no data is available, **t_rcv** fails and returns TNODATA in *t_errno*. So if you need more control over the operation, you can open an endpoint in nonblocking mode and use **t_rcv** to poll for incoming data. For details about how the blocking status affects particular functions, look up the function in TLI: Functions.

Parent Topic: TLI Local Management Issues

Close

With **tirdwr** on a STREAM, the user can send and receive data over a transport connection for the duration of that connection. Either user can terminate the connection by closing the file handle associated with the transport endpoint or by popping the **tirdwr** module off the STREAM. In either case, **tirdwr** takes the following actions:

If an orderly release indication was previously received by **tirdwr**, an orderly release request is passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected indication when data transfer completes.

If a disconnect indication was previously received by **tirdwr**, no special action is taken.

If neither an orderly release indication nor a disconnect indication was previously received by **tirdwr**, a disconnect request is passed to the transport provider to abort the connection.

If an error previously occurred on the STREAM and a disconnect indication has not been received by **tirdwr**, a disconnect request is passed to the transport provider.

A process must not initiate an orderly release after **tirdwr** is pushed onto a STREAM, but **tirdwr** handles an orderly release properly if it is initiated by the user on the other side of a transport connection.

If the client in this section is communicating with a server program that supports orderly release, that server terminates the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. When the file handle is closed, **tirdwr** initiates the orderly release request from the client's side of the connection. This generates the indication that the server is expecting, and the connection is released properly.

Parent Topic: A TLI Read/Write Interface

Connection Establishment Phase

In this phase, two transport users establish a transport connection between them. One user is considered active and initiates the conversation, whereas the second user is passive and waits for a transport user to request a connection.

The active user requests a connection and then receives a response from the passive user. The passive user waits for connect indications (indications of a connect request) and then either accepts or rejects the request.

The functions that support these operations are listed as follows.

Table auto. Connection Establishment Functions

Function	Task
t_connect	This function requests a connection to the transport user at a specified destination and waits for the passive user's response. This function can be executed in either synchronous or asynchronous mode. In synchronous mode, the function waits for the passive user's response before returning control to the active user. In asynchronous mode, the function initiates connection establishment but returns control to the active user before a response arrives.
t_rcvconnect	This function enables an active transport user to determine the status of a previously sent connect request. If the request was accepted, the connection establishment phase is complete on return from this function. This function is used with **t_connect** to establish an asynchronous connection.
t_listen	This function enables the passive transport user to receive connect indications from other transport users.
t_accept	The passive user calls this function to accept a particular connect request, after an indication has been received.

Parent Topic: Overview of Connection-Oriented Service

Connection Establishment: the Client

Continuing with the client-server example, the steps needed by the client to establish a connection are as follows:

TLI Connection Mode Connection Establishment---the Client

```
/* Allocate space for server's address */
   sndcall = (struct t_call *) t_alloc(fh, T_CALL, T_ADDR);
   if (sndcall == NULL) {
      t_error("sndcall");
      exit(1);
   }

   /* Copy address of the server into t_call structure */
   sndcall->addr.len = sizeof (IPX_ADDR);
   memcpy(sndcall->addr.buf, pv, sndcall->addr.len);

   /* Initiate connect with a server */
```

```
    if (t_connect(fh, sndcall, NULL) == -1) {
       t_error("t_connect");
       exit(1);
    }
```

The **t_connect** function establishes the connection with the server. The first argument to **t_connect**, *fh*, identifies the transport endpoint through which the connection is established, and the second argument, *sndcall*, identifies the destination server. This argument is a pointer to a t_call structure with the following format:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
}
```

The *addr* field identifies the address of the server. The *opt* field can be used to specify protocol-specific options that the client would like to associate with the connection. The *udata* field identifies user data that can be sent with the connect request to the server. The *sequence* field has no meaning for **t_connect**.

The **t_alloc** function is called to allocate the t_call structure dynamically. Once t_call is allocated, the appropriate values are assigned. In this example, no options or user data are associated with **t_connect**, but the server's address must be set. The third argument to **t_alloc** is set to T_ADDR to specify that an appropriate netbuf buffer should be allocated for the address. The server's address is then assigned to *buf*, and *len* is set accordingly.

The third argument to **t_connect** can be used to return information about the newly-established connection to the user. It can also retrieve any user data sent by the server in its response to the connect request. Here, the client sets it to NULL to indicate that this information is not needed. The connection is established on successful return of **t_connect**. If the server rejects the connect request, **t_connect** fails and sets *t_errno* to TLOOK.

Parent Topic: Connection Mode: Connection Establishment

Connection Mode: Connection Establishment

The connection establishment procedures highlight the distinction between clients and servers. TLI imposes a different set of procedures in this phase for each type of transport user.

The client starts the connection establishment procedure by requesting a connection to a particular server using **t_connect**. The server is then notified of the client's request by calling **t_listen**. The server can accept or reject the client's request. It calls **t_accept** to establish the connection or **t_snddis** to

reject the request. The client is notified of the server's decision when **t_connect** completes.

During connection establishment, TLI supports two facilities that might not be supported by all transport providers:

The ability to transfer data between the client and server when establishing the connection. The client can send data to the server when it requests a connection. This data is passed to the server by **t_listen**. Similarly, the server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by **t_open** determines how much data, if any, two users can transfer during connect establishment.

The negotiation of protocol options. The client can specify protocol options that it would like the transport provider and/or the remote user to support. TLI supports both local and remote option negotiation.

See Connection Establishment: the Client for an example.

Parent Topic: TLI Connection Mode Services

Connection Mode: Data Transfer

Once the connection has been established, both the client and server can begin transferring data over the connection using **t_snd** and **t_rcv**. TLI does not differentiate the client from the server from this point on. Either user can send and receive data, or disconnect the connection. TLI guarantees reliable, sequenced delivery of data over an existing connection. Two classes of data can be transferred over a transport connection:

Normal data

Expedited data

Expedited data is typically associated with urgent information. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of an expedited data class (see **t_open**).

All transport protocols support the transfer of data in byte stream mode, where "byte stream" implies no concept of message boundaries on data that is transferred over a connection. However, some transport protocols support the preservation of message boundaries over a transport connection. This service is supported by TLI, but protocol-independent software must not rely on its existence. SPX II, ADSP, and OSI™ connection transports support message services.

The message interface for data transfer is supported by the T_MORE flag of **t_snd** and **t_rcv**. The messages, called Transport Service Data Units (TSDUs), can be transferred between two transport users as distinct units.

The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from **t_open** and **t_getinfo** . Because the maximum TSDU size can be large (possibly unlimited), TLI allows a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the T_MORE flag on every **t_snd** call except the last. This flag specifies that the user will send more data associated with the message in a subsequent call to **t_snd**. The last message unit should be transmitted with T_MORE turned off to specify that this is the end of the TSDU.

Similarly, a TSDU can be passed in multiple units to the receiving user. Again, if **t_rcv** returns with the T_MORE flag set, the user should continue calling **t_rcv** to retrieve the remainder of the message. The last unit in the message is identified by a call to **t_rcv** that does not set T_MORE.

> **NOTE:** The T_MORE flag implies nothing about how the data iscan packaged below TLI or how the data is delivered to the remote user. Each transport protocol, and each implementation of that protocol, can package and deliver the data differently.
>
> For example, if a user sends a complete message in a single call to**t_snd**, there is no guarantee that the transport provider will deliver the data in a single unit to the remote transport user. Similarly, a TSDU transmitted in two message units can be delivered in a single unit to the remote transport user. The message boundaries can be preserved only by noting the value of the T_MORE flag on **t_snd** and **t_rcv**. This guarantees that the receiving user sees a message with the same contents and message boundaries as those sent by the remote user.

Related Topics

Connection Mode Data Transfer: the Client

Connection Mode Data Transfer: the Server

Parent Topic: TLI Connection Mode Services

Connection Mode Data Transfer: the Client

Continuing with the client-server example, the server transfers a log file to the client over the transport connection. The client receives this data and writes it to the screen. A byte stream interface is used by the client and server, in which message boundaries (that is, the T_MORE flag) are ignored. The client receives data using the following instructions:

```
/* Receive line at a time and print it on the screen */
while( t_rcv(fh, iobuf, sizeof(iobuf), &flags) != -1 ) {
   printf("%s",iobuf);
}
```

The client continuously calls **t_rcv** to process incoming data. If no data is

currently available, **t_rcv** blocks until data arrives. The **t_rcv** function retrieves the available data up to 132 bytes, which is the size of the client's input buffer, and returns the number of bytes received. The client then writes this data to standard output and continues.

The data transfer phase is complete when **t_rcv** fails. The **t_rcv** function fails if a disconnect indication arrives (see Abortive Connection Release). If the transport endpoint is closed (either by **exit** or **t_close**) during the data transfer phase, the connection is aborted, and the remote user receives a disconnect indication.

Parent Topic: Connection Mode: Data Transfer

Connection Mode Data Transfer: the Server

Looking now at the other side of the connection, the server manages its data transfer by beginning a new thread to send the data to the client. The parent thread then loops back to listen for further connect indications. The server calls **BeginThread** to begin a new thread. Once this occurs, the **ProcessConnection** function is executed on the new thread. The first parameter for the **ProcessConnection** function is *fh*, which is a pointer to a newly-established connection.

TLI Connection Mode Data Transfer Server Example

```
void ProcessConnection(int *fh) {
   FILE *fp;               /* file pointer      */
   char iobuf[132];        /* line buffer       */

   if( (fp = fopen( LOGFILE, "r" )) == NULL ) {
      printf("Couldn't open file: %s\n", LOGFILE);
      t_close(*fh);
      free(fh);
      return;
   }

   /* Read lines from a file and send to the client */
   while( fgets( iobuf, sizeof(iobuf), fp ) != NULL ) {
      delay(100);    /* timeout and thread switch */
      if( t_look(*fh) == T_DISCONNECT ) {
      printf("Received disconnect, EXIT.\n");
      if( t_rcvdis(*fh, NULL) < 0 )
      terror("t_rcvdis");
      fclose(fp);
      t_close(*fh);
      free(fh);
      return;
      }
if( t_snd( *fh, iobuf, sizeof(iobuf), NULL) == -1 ) {
      t_error("t_snd");
```

```
            fclose(fp);
            t_close(*fh);
            free(fh);
            return;
            }
        }
        fclose(fp);
    }
```

After beginning a new thread, the parent thread returns to the main processing loop and listens for further connect indications. Meanwhile, the child thread manages the newly-established transport connection. If **BeginThread** fails, the transport endpoint is closed, sending disconnect indications to the client. The client's call to **t_rcv** fails.

The server process reads one line of the log file at a time and sends that data to the client using **t_snd**. The *iobuf* argument points to the start of the data buffer, and `sizeof(iobuf)` specifies the number of bytes to be transmitted. The fourth argument can contain one of the following two optional flags:

T_EXPEDITED specifies that the data is expedited.

T_MORE defines message boundaries when transmitting messages over a connection.

Neither flag is set by the server in this example.

If the user floods the transport provider with data, the provider can exert blocking pressure to provide flow control. In such cases, **t_snd** blocks until the flow control is relieved and then resumes its operation. The **t_snd** function does not complete until a whole buffer has been passed to the transport provider. If the endpoint is in nonblocking mode, it is possible that only part of the data is accepted by the transport provider. In this case, **t_snd** sets T_MORE and returns a value less than `sizeof(iobuf)`.

In order to check for incoming events, **t_look** is called before each **t_snd** call. If **t_look** returns T_DISCONNECT, which means that a disconnect indication has arrived, the thread closes the transport endpoint and terminates.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated **t_snd** and **t_rcv** calls, it could rely on **t_rcv** to recognize an incoming disconnect indication.

Parent Topic: Connection Mode: Data Transfer

Connection Mode: Event Handling

The TLOOK error has special significance in TLI. TLOOK notifies the user

when a TLI routine is interrupted by an unexpected asynchronous transport event on the given transport endpoint. As such, TLOOK does not report an error with a TLI routine, but the normal processing of that routine does not proceed because of the pending event. The events defined by TLI are described in the following table.

Table auto. TLI Events

Event	Description
T_LISTEN	A request for a connection, called a connect indication, has arrived at the transport endpoint.
T_CONNECT	The confirmation of a previously sent connect request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connect request.
T_DATA	User data has arrived at the transport endpoint.
T_EXDATA	Expedited user data (discussed in Connection Mode: Data Transfer) has arrived at the transport endpoint.
T_DISCONNECT	A notification that the connection was aborted or that the server rejected a connect request, called a **disconnect indication**, has arrived at the transport endpoint.
T_UDERR	The notification of an error in a previously-sent datagram, called a unitdata error indication, has arrived at the transport endpoint. (See TLI Connectionless Mode Services.)
T_ORDREL	A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.
T_GODATA	It is possible to send normal data is again.
T_GOEXDATA	It is possible to send expedited data again.

It is possible in some states to receive one of several asynchronous events, as described in the state tables in TLI State Transitions. The **t_look** function enables a user to determine what event has occurred if a TLOOK error is returned. The user can then process that event accordingly.

In the example, if a connect request is rejected, the event passed to the client is a disconnect indication. The client exits if its request is rejected.

See Event Handling: the Server for an example.

Parent Topic: TLI Connection Mode Services

Connection Mode: Local Management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using **t_open**, and establish its identity (or address) using **t_bind (Function)**.

The set of services supported by TLI might not be implemented by all transport protocols. Each transport provider has a set of characteristics that determines the services it offers and the limits associated with those services. This information is returned to the user by **t_open** and consists of the characteristics summarized in the following table.

Table auto. Transport Provider Characteristics

Characteristic	Description
addr	Maximum size of a transport address.
options	Maximum bytes of protocol-specific options that can be passed between the transport user and transport provider.
tsdu	Maximum message size that can be transmitted in either connection mode or connectionless mode.
etsdu	Maximum expedited data message size that can be sent over a transport connection.
connect	Maximum bytes of user data that can be passed between users during connection establishment.
discon	Maximum bytes of user data that can be passed between users during the abortive release of a connection.
servtype	The type of service supported by the transport provider.

The three service types defined by TLI are described in the following table.

Table auto. TLI Service Types

Service Type	Description
T_COTS	The transport provider supports connection mode service but does not provide the optional orderly release facility.
T_COTS_ORD	The transport provider supports connection mode service with the optional orderly release facility.
T_CLTS	The transport provider supports connectionless mode service. Only one such service can be associated with the transport provider identified by **t_open**.

The **t_open** function returns the default provider characteristics associated with a transport endpoint. However, some characteristics can change after an endpoint has been opened. This occurs if the characteristics are associated with negotiated options. (Option negotiation is described later in this chapter.)

For example, if the support of expedited data transfer is a negotiated option, the value of this characteristic can change. Call **t_getinfo** function to retrieve the current characteristics of a transport endpoint.

After establishing a transport endpoint with the chosen transport provider, the user must establish its identity. **t_bind (Function)** does this by binding a transport address to the transport endpoint. In addition, for servers, this routine informs the transport provider that the endpoint will be used to listen for incoming connect indications, also called connect requests.

An optional facility, **t_optmgmt (Function)**, is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider.

Each transport protocol is expected to define its own set of negotiable protocol options, which can include such information as*QualityofService* parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The following topics discuss the local management requirements of the example client and server.

Connection Mode Local Management: the Client

Connection Mode Local Management: the Server

Parent Topic: TLI Connection Mode Services

Connection Mode Local Management: the Server

The server in this example must take similar local management steps before communication can begin. The server must establish a transport endpoint (TEP) through which it listens for connect indications. The necessary definitions and local management steps are as follows:

TLI Connection Mode Local Management---the Server

```
#define SRV_SOCK 0x4800       /* Server socket (dynamic sock*/
#define TLI_TYPE 0x9000       /* Server type (dynamic area) */
#define LOGFILE  "README.TXT" /* File name                  */

/* Global variables */
int fh;                       /* TEP handle                 */
LONG SAPhandle;               /* SAP handle                 */
```

```
struct t_call  *call;         /* call structure ptr         */
struct t_bind  *bind;         /* bind structure ptr         */

void TerminateNLM(void)
{
   if( SAPhandle )
      ShutdownAdvertising( SAPhandle );
   if( bind )
      t_free((char *)bind, T_BIND);
   if( call )
      t_free((char *)call, T_CALL);
   t_unbind( fh );
   t_close( fh );
}

main(int argc, char **argv)
{
   int *fh_new;            /* new transport endpoint pointer*/
   IPX_ADDR *addr;         /* address structure for SPX     */

   if( argc != 2 ) {
      printf("Usage: %s <server's name>\n", argv[0]);
      exit(1);
   }

   /* Register function that will be executed when NLM
      exits or is unloaded */
   if( atexit( TerminateNLM ) != NULL ) {
      printf("atexit failed");
      exit(1);
   }

/* Open server's endpoint, no info needed */
   if((fh = t_open("/dev/nspx", O_RDWR, NULL)) == -1) {
      t_error("t_open");
      exit(1);
   }

   /* Allocate space for structures */
   bind = (struct t_bind *) t_alloc(fh, T_BIND, T_ALL);
   call = (struct t_call *) t_alloc(fh, T_CALL, T_ADDR);
   if( bind == NULL || call == NULL ) {
      t_error("t_alloc");
      exit(1);
   }
   /* Prepare bind structure and then call t_bind */
   bind->qlen = 1;
   bind->addr.len = sizeof( IPX_ADDR );
   addr = (IPX_ADDR *)bind->addr.buf;
   GetInternetAddress( GetConnectionNumber(),
      addr->ipxa_net, addr->ipxa_node );
   *(WORD *)addr->ipxa_socket = IntSwap(SRV_SOCK);
```

```
*(WORD *)addr->ipxa_socket = IntSwap(SRV_SOCK);

if( t_bind(fh, bind, bind) == -1 ) {
   t_error("t_bind");
   exit(1);
}

/* Check if this is socket you wanted to be bound to */
if( *(WORD *)&bind->addr.buf[10] != IntSwap(SRV_SOCK) ) {
   printf("Bound wrong address: %d != %d\n",
   *(WORD *)&bind->addr.buf[10], IntSwap(SRV_SOCK) );
   exit(1);
}
```

As with the client, the first step is to call **t_open** to establish a transport endpoint with the desired transport provider. This endpoint is used to listen for connect indications. Next, the server must bind its well-known address to the endpoint. Clients use this address to access the server.

The second argument to **t_bind (Function)** requests that a particular address be bound to the transport endpoint. This argument points to a t_bind structure with the following format:

```
struct t_bind {
   struct netbuf addr;
   unsigned int qlen;
}
```

where *addr* describes the address to be bound, and *qlen* specifies the maximum outstanding connect indications that can arrive at this endpoint. All TLI structure and constant definitions are found in TIUSER.H.

The address is specified using a netbuf structure that contains the following members:

```
struct netbuf {
   unsigned int maxlen;
   unsigned int len;
   char *buf;
}
```

where *buf* points to a buffer containing the data, *len* specifies the bytes of data in the buffer, and *maxlen* specifies the maximum bytes the buffer can hold. You need to set *maxlen* only if a TLI routine returns data to the user.

For the t_bind structure, the data pointed to by *buf* identifies a transport address. It is expected that the structure of addresses vary among each protocol implementation under TLI. The netbuf structure is intended to support any address structure.

If the value of *qlen* is greater than 0, the transport endpoint can be used to listen for connect indications. In such cases, t_bind directs the transport provider to begin queuing connect indications destined for the bound address immediately.

Furthermore, the value of *qlen* specifies the maximum outstanding connect indications the server wants to process. The server must respond to each connect indication, either accepting or rejecting the request for connection. An outstanding connect indication is one to which the server has not yet responded.

Often, a server fully processes a single connect indication and responds to it before receiving the next indication. When this occurs, a value of 1 is appropriate for *qlen*.

However, some servers might want to retrieve several connect indications before responding to any of them. In such cases, *qlen* specifies the maximum number of outstanding indications the server will process.

An example of a server that manages multiple outstanding connect indications is presented in TLI Asynchronous Execution Mode.

The **t_alloc** function is called to allocate the t_bind (Structure) structure needed by **t_bind (Function)**. The **t_alloc** function takes three arguments. The first is a file handle that references a transport endpoint. This is used to access the characteristics of the transport provider (see **t_open**). The second argument identifies the appropriate TLI structure to be allocated. The third argument specifies which, if any, netbuf buffers should be allocated for that structure. T_ALL specifies that all netbuf buffers associated with the structure should be allocated and causes the *addr* buffer to be allocated in this example. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size.

The *maxlen* field of this netbuf structure is set to the size of the newly allocated buffer by **t_alloc**. The use of **t_alloc** helps ensure the compatibility of user programs with future releases of TLI.

In this example, the server processes connect indications one at a time, so *qlen* is set to 1. The address information is then assigned to the newly allocated t_bind structure. This t_bind structure passes information to **t_bind (Function)** in the second argument and returns information to the user in the third argument.

On return, the t_bind structure contains the address that was bound to the transport endpoint. If the provider could not bind the requested address (perhaps because it had been bound to another transport endpoint), it chooses another appropriate address.

Each transport provider manages its address space differently. Some transport providers might allow a single transport address to be bound to several transport endpoints, whereas others might require a unique address per endpoint. TLI supports either choice.

Based on its address management rules, a provider determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint.

The server must check the bound address to ensure that it is the one

previously advertised to clients. Otherwise, the clients are unable to reach the server. (To advertise its service in the bindery, the server in this example uses SAP (see SAP).

If **t_bind (Function)** succeeds, the provider begins queuing connect indications, thus entering the next phase of communication, connection establishment.

Parent Topic: Connection Mode: Local Management

Connection Mode Service

The connection mode service is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. It also provides an identification procedure that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connection mode service involves three phases:

Establishing a Connection in TLI

Transferring Data in TLI

Releasing a Connection in TLI

You must perform all three phases in the proper sequence to handle the connection successfully.

Parent Topic: Overview of Transport Protocols

Connection Release Phase

The connection-oriented TLI supports two forms of connection release: abortive and orderly.

An abortive release can be invoked from either the connection establishment phase or the data transfer phase. When in the connection establishment phase, a transport user can use the abortive release to reject or cancel a connect request. In the data transfer phase, either user can abort a connection at any time.

The transport users do not negotiate the abortive release, and it takes effect immediately on request. The user on the other side of the connection is notified when a connection is aborted.

The transport provider can also initiate an abortive release, in which case

both users are informed that the connection no longer exists. There is no guarantee of delivery of user data once an abortive release has been initiated.

The orderly release capability is an optional feature of the connection-oriented service. If supported by the underlying transport provider, an orderly release can be invoked from the data transfer phase to enable two users to gracefully release a connection. The procedure for orderly release prevents the loss of data that can occur during an abortive release.

The functions that support the release of a connection are listed in the following table.

Table auto. Functions for Releasing a Connection

Function	Task
t_snddis	This function can be called by either transport user to initiate the abortive release of a transport connection. It can also be used to reject or cancel a connect request during the connection establishment phase.
t_rcvdis	This function identifies the reason for the abortive release of a connection, when the connection is released by the transport provider or another transport user.
t_sndrel	(Optional) This function can be called by either transport user to initiate an orderly release. The connection remains intact until both users call this function and the **t_rcvrel** function.
t_rcvrel	(Optional) This function is called when a user is notified of an orderly release request, as a means of informing the transport provider that the user is aware of the remote user's actions.

After a connection has been released, the transport user must deinitialize the associated transport endpoint, thereby freeing the resource for future use.

Parent Topic: Overview of Connection-Oriented Service

Connectionless Mode: Data Transfer

Once a user has bound an address to the transport endpoint, datagrams can be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, TLI enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed earlier, each

transport provider defines the set of options, if any, that can accompany a datagram. When the datagram is passed to the destination user, the associated protocol options can be returned as well.

The following sequence of calls illustrates the data transfer phase of the connectionless mode server:

TLI Connectionless Mode Data Transfer for the Server

```
    for (;;)
    {
       printf("Server: %s is ready ...\n", argv[1]);
       /* Receive message from client */
       if( t_rcvudata(fh, udata, &flags) == -1 )
       {
          if( t_errno == TLOOK )
          {
          /* Error on previously sent datagram */
             if( t_rcvuderr(fh, uderr) == -1 )
             {
                t_error("t_rcvuderr");
                exit(1);
             }
             printf("Bad datagram, error=%d\n", uderr->error);
          }
          t_error("t_rcvudata");
          exit(1);
       }
       printf("Received: %s\n", udata->udata.buf);
       query();
       ThreadSwitch();
       if( t_sndudata(fh, udata) == -1 )
       {
          t_error("t_sndudata");
          exit(1);
       }
       printf("Sending: %s\n", udata->udata.buf);
    }
}

query(void){ /* Stub for simplicity */ }
```

The server must first allocate a t_unitdata structure for storing datagrams, using the following format:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
}
```

The *addr* field holds the source address of incoming datagrams and the destination address of outgoing datagrams. The *opt* field identifies any protocol options associated with the transfer of the datagram. The *udata* field

holds the data itself.

The *addr*, *opt*, and *udata* fields must all be allocated with buffers large enough to hold any possible incoming values. The T_ALL argument to **t_alloc** ensures this and sets the *maxlen* field of each netbuf structure accordingly. The server also allocates a t_uderr structure for processing any datagram errors, as discussed in Datagram Errors.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls **t_rcvudata** to receive the next query. The **t_rcvudata** function retrieves the next available incoming datagram. If none is currently available, **t_rcvudata** blocks, waiting for a datagram to arrive. The second argument of **t_rcvudata** identifies the t_unitdata structure in which the datagram should be stored.

The third argument, flags, must point to an integer variable and can be set to T_MORE on return from **t_rcvudata** to specify that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to **t_rcvudata** retrieve the remainder of the datagram. Because **t_alloc** allocates a *udata* buffer large enough to store the maximum datagram size, the transaction server does not have to check the value of flags.

If a datagram is received successfully, the transaction server calls the query routine to process the request. This routine stores the response in the structure pointed to by *udata* and sets *udata->udata.len* to specify the number of bytes in the response. The source address returned by **t_rcvudata** in *ud->addr* is used as the destination address by **t_sndudata**.

When the response is ready, **t_sndudata** is called to return the response to the client. TLI prevents a user from flooding the transport provider with datagrams using the same flow control mechanism described for the connection mode service. In such cases, **t_sndudata** blocks until the flow control is relieved and then resumes its operation.

A connectionless mode client follows the same local management steps as a server does. Because the connectionless mode does not guarantee message delivery once a message is sent, the client waits for a specified period of time; if it does not receive a response from the server, the client exits.

Monitoring for the response is accomplished by using **poll**. The **poll** function waits for a specified period of time for the incoming event. If the event does not arrive, **poll** returns. It is up to the user to check for the received event on a particular transport endpoint and take action accordingly. "Message Received" is displayed if the response from the server is received. Otherwise, the message "Server didn't respond" is displayed.

The following sequence of calls illustrates the data transfer phase of the connectionless mode client:

TLI Connectionless Mode Data Transfer for the Client

.
.

```
        .
        struct pollfd pfh;
        .
        .
        .
        udata->udata.len = strlen(argv[2]) + 1;
        if( udata->udata.maxlen < udata->udata.len ) {
           printf("Buffer to small: %d\n", udata->udata.maxlen);
           exit(1);
        }
        strcpy(udata->udata.buf, argv[2]);

        /* Send message, blocked until send completed */
        printf("Sending: %s\n", udata->udata.buf);
        if( t_sndudata(pfh.fd, udata) == -1 ) {
           t_error("t_sndudata");
           exit(1);
        }

        /* Wait up to 5 seconds for response from server */
        pfh.events = POLLIN;
        if( poll( &pfh, 1, 5000 ) == -1 ) {
           perror("poll");
           exit(1);
        }
        if( pfh.revents == POLLIN ) {
           if( t_rcvudata(pfh.fd, udata, &flags) != -1 ) {
              t_error("t_rcvudata");
              exit(1);
           }
           printf("Received: %s\n", udata->udata.buf);
        }
        else
           printf("Server didn't respond.\n");
     }
```

Parent Topic: TLI Connectionless Mode Services

Connectionless Mode: Local Management

Just as with connection mode service, the transport users must complete the appropriate local management steps before transferring data. A user must choose the appropriate connectionless service provider using **t_open** and establish its identity using **t_bind (Function)**.

t_optmgmt (Function) can be used to negotiate protocol options associated with the transfer of each data unit. As with the connection mode service, each transport provider specifies the options, if any, that it supports. Option negotiation is, therefore, a protocol-specific activity.

In the following example, the definitions and local management calls needed by the transaction server are shown.

TLI Connectionless Mode Local Management for the Server

```
#define SRV_SOCK     0x4800   /* Echo socket (dynamic socket)*/
#define TLI_TYPE     0x9000   /* Server type                 */

int fh;
struct t_bind     *bnd;
struct t_unitdata *udata;
struct t_uderr    *uderr;
LONG SAPhandle;

void TerminateNLM(void)
{
   if( SAPhandle )
      ShutdownAdvertising( SAPhandle );
   if( udata )
      t_free((char *)udata, T_UNITDATA);
   if( uderr )
      t_free((char *)uderr, T_UDERROR);
   if( bnd )
      t_free((char *)bnd, T_BIND);
   t_close( fh );
}

main(int argc, char **argv)
{
   IPX_ADDR *addr;         /* address structure for SPX */
   int flags;

   if( argc != 2 ) {
      printf("Usage: %s <server's name>\n", argv[0]);
      exit(1);
   }

   /* Register function that will be executed when NLM exits
      or is unloaded */
   if( atexit( TerminateNLM ) != NULL ) {
      printf("atexit failed");
      exit(1);
   }

   /* Open server's endpoint, no info needed */
   if((fh = t_open("/dev/nipx", O_RDWR, NULL)) == -1) {
      t_error("t_open");
      exit(1);
   }

   /* Allocate space for structures */
   bnd = (struct t_bind *)t_alloc(fh, T_BIND, T_ALL);
```

```
    udata = (struct t_unitdata *)t_alloc(fh,T_UNITDATA,T_ALL);
    uderr = (struct t_uderr *)t_alloc(fh, T_UDERROR, T_ALL);
    if( !bnd || !udata || !uderr ) {
       t_error("t_alloc");
       exit(1);
    }
/* Prepare bind structure and then call t_bind */
    bnd->addr.len = sizeof( IPX_ADDR );
    addr = (IPX_ADDR *)bnd->addr.buf;
    GetInternetAddress( GetConnectionNumber(),
       addr->ipxa_net, addr->ipxa_node );
    *(WORD *)addr->ipxa_socket = IntSwap(SRV_SOCK);

    if( t_bind(fh, bnd, bnd) == -1 ) {
       t_error("t_bind");
       exit(1);
    }
    /* Check if this is the right socket */
    if( *(WORD *)&bnd->addr.buf[10] != IntSwap(SRV_SOCK) ) {
       printf("Bound wrong address: %d != %d\n",
       *(WORD *)&bnd->addr.buf[10], IntSwap(SRV_SOCK) );
       exit(1);
    }

    /* Advertise server on SRV_SOCK socket */
    SAPhandle = AdvertiseService( TLI_TYPE, argv[1],
       IntSwap(SRV_SOCK) );
    if( SAPhandle == NULL ) {
       printf("AdvertiseService failed for: %s\n", argv[1]);
       exit(1);
    }
```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using **t_open**. Each provider has an associated service type, so the user can choose a particular service by opening the appropriate transport provider file, in this case /dev/nipx.

This connectionless mode server ignores the characteristics of the provider returned by **t_open** in the same way as the users in the connection mode example, by setting the third argument to NULL. For simplicity, the transaction server assumes the transport provider has the following characteristics:

The transport provider supports the T_CLTS service type (connectionless transport service, or datagram).

The transport provider supports protocol-specific options. (None are used in this example.)

The connectionless server also binds a transport address to the endpoint so that potential clients can identify and access the server. A t_bind (Structure) structure is allocated using **t_alloc**, and the *buf* and *len* fields of the address

are set accordingly.

One important difference between the connection mode server and this connectionless mode server is that the*qlen* field of the t_bind structure has no meaning for connectionless mode service. This is because all users are capable of receiving datagrams once they have bound an address. TLI defines an inherent client-server relationship between two users while establishing a transport connection in the connection mode service. However, no such relationship exists in the connectionless mode service. It is the context of this example, not TLI, that defines one user as a server and another as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by **t_bind (Function)** to ensure that it is correct.

Parent Topic: TLI Connectionless Mode Services

Connectionless Service

Because connectionless service doesn't provide the conveniences and safeguards associated with a connection, connectionless service is simpler to arrange. All that a connectionless transfer requires is that you open an endpoint and bind an address to it. You are then ready to send and receive. Messages sent across the network without a connection are known as datagrams.

Sending Connectionless Data

Call **t_sndudata** to send a datagram. This function takes the endpoint and t_unitdata as input:

```
int t_sndudata(int fh, struct t_unitdata *unitdata);
```

t_unitdata includes the destination address, any protocol-specific options, and the message data.

Receiving Connectionless Data

Call **t_rcvudata** to receive a datagram. As input, this function takes the endpoint, t_unitdata, and space for any control flags:

```
int t_rcvudata(int fh,  struct t_unitdata *unitdata,
int *flags);
```

flags will be set to T_MORE if the buffer referenced by *unitdata* isn't large enough to hold the message. In that case, you should continue calling **t_rcvudata** until the entire message has been received.

Transport providers return datagram errors as unit data error events. When a datagram error occurs, the function encountering the error (**t_sndudata** or **t_rcvudata**) fails and returns TLOOK in *t_errno*. The TLOOK event for any unit data error is T_UDERR. (Confirm this event by calling **t_look**.) To process the error, call **t_rcvuderr**. This function clears the error and finds the destination address and protocol options for the associated data unit.

Parent Topic:

Overview of Transport Protocols

Data Transfer Phase of Connection-Oriented Service

Once a transport connection is established between two users, data can be transferred over the connection. Two functions that transfer data in connection mode are listed in the following table.

Table auto. Data Transfer Functions

Function	Task
t_snd	This function enables transport users to send either normal or expedited data over a transport connection.
t_rcv	This function enables transport users toreceive either normal or expedited data on a transport connection.

Parent Topic: Overview of Connection-Oriented Service

Data Transfer Phase of Connectionless Service

Once a transport endpoint has been activated, a user is free to send and receive data units through that endpoint in connectionless mode (see the following table).

Table auto. Datagram Functions

t_sndudata	This function enables transport users to send a datagram to the user at the specified protocol address.
t_rcvudata	This function enables transport users to receive datagrams from other users.
t_rcvuderr	This function enables users to retrieve error information associated with a previously sent

	datagram.

Parent Topic: Overview of Connectionless Service

Datagram Errors

If the transport provider cannot process a datagram that was passed to it by **t_sndudata**, it returns a unit data error event, T_UDERR, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what might be wrong with the datagram.

The reason a datagram could not be processed is protocol-specific. One possible reason is that the transport provider could not interpret the destination address or options. Each transport protocol is expected to specify all the reasons why it cannot process a datagram.

The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication is used. Remember, the connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of this error event when it attempts to receive another datagram. In this case, **t_rcvudata** fails, setting *t_errno* to TLOOK. If TLOOK is set, the only possible event is T_UDERR, so the server calls **t_rcvuderr** to retrieve the event. The second argument to **t_rcvuderr** is the t_uderr structure that was allocated earlier. This structure is filled in by **t_rcvuderr** and has the following format:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;
}
```

The *addr* and *opt* fields identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that specifies why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

Parent Topic: TLI Connectionless Mode Services

dclient.c

dclient.c is the active side of the connection. It calls the server program running on another workstation and asks for the connection. It then initiates

the data transfer and closes the connection after receiving data back.

1. dclient.c requires the user to enter the network node and address of the server on the command line. The parameter takes the form of a network address separated from the node address by a slash. Leading zeros are required:

   ```
   DCLIENT 00001234/00001b02362e
   ```

   There are several ways to obtain the network and node address. If the server program is attached to a NetWare server or is advertising with SAP, the user can find the server's address by scanning the server's bindery. (The utility USERLIST performs this operation.) Under NDS, the user can look up the server address if the server takes the appropriate steps to store it there.

   The client uses a pair of subroutines, **ParseAddress** and **PutAddress**, to parse the parameter entered by the user. Although somewhat incidental to the purposes of this example, the routines are included below to demonstrate one way to solve the problem of translating the address's character representation into byte values. The resulting address is stored in IPX_ADDR.

2. After parsing the address, the client begins by opening an endpoint for calling the server. It binds the endpoint to a socket. Although the client needs to know the socket on which the server is listening at the other end, the client doesn't need to worry about which local socket it uses. **t_bind (Function)** assigns a socket.

3. Next, the client prepares t_call to pass to **t_connect**. The client sets up this structure to reference IPX_ADDR containing the destination address in the same way the server sets up t_call for listening.

4. The client calls **t_connect** to attempt the connection.

5. If the server accepts the connection, the client enters a for loop in which the client uses the same endpoint to carry on the connection. The for loop repeatedly sends a message to the server with **t_snd** and receives the same message back with **t_rcv**. If at any point an error occurs, **SPXDisconReason** is called to report the reason for the error.

6. After sending and receiving the messages, the client disconnects from the server with **t_snddis** and closes the endpoint.

dclient.c example

```
#include <stdio.h>
#include <ctype.h>
#include <process.h>
#include <fcntl.h>
#include <string.h>
#include <tispxipx.h>
#include <tiuser.h>
```

```
#define     SPX_SOCKET      31   /* arbitrary socket */

int    ParseAddress( char *address, IPX_ADDR *destination );
int    PutAddress( char *string, char *buf, int hexBytes );
void   SPXDisconReason( int fd );

int ParseAddress( char *addr, IPX_ADDR *destination )
{
   if ( strlen( addr ) == 21 && addr[ 8 ] == '/' )
   if ( PutAddress( addr, destination->ipxa_net, 4 ) )
      if ( PutAddress( &addr[ 9 ], destination->ipxa_node, 6 ) )
         return 1;
   return 0;
}

int PutAddress( char *string, char *buf, int hexBytes )
{
   int    i, j, value;
   char   c;

   for ( i = 0; i < hexBytes; i++ )
   {
      value = 0;               /* build a byte from two nibbles */
      for ( j = 0; j < 2; j++ )
      {
         value <<= 4;
         if ( ( c = (char)toupper( *string ) ) >= '0' && c <= '9')
            value += c - '0';
         else if ( c >= 'A' && c <= 'F' )
            value += c - 'A' + 10;
         else return 0;
         string++;
      }
      *buf++ = (char)value;
   }
   return 1;
}

void main ( int argc, char *argv[] )
{
   IPX_ADDR   spx_addr; /* server address from command line */
   SPX_OPTS   spx_options;
   char       buf[ 100 ], buf2[ 100 ];
   int        fd, flags, i1;
   struct t_call tcall;

   /* Step 1 */

   if ( argc != 2 || !ParseAddress( argv[ 1 ], & spx_addr ) )
   {
      printf( "Usage:\tdclient          ServerAddress\n" );
      printf( "\tServerAddress = Net/Node ( in hex, leading \
```

```
   zero's required )\n" );
   printf( "\tExample:\"dclient 00001234/00001b02362e\"\n" );
   exit( 1 );
}

/* Step 2 */

if ( ( fd = t_open( "/dev/nspx", O_RDWR, ( struct t_info * )0) ) ==
{
   t_error( "open of /dev/nspx failed" );
   exit( 2 );

/* No need to bind to a specific socket */

if ( t_bind( fd, ( struct t_bind * )0, ( struct t_bind * )0 )== -1 )
{
   t_error( "bind failed" );
   exit( 2 );
}

/* Step 3 */

spx_addr.ipxa_socket[ 0 ]        = 0;
spx_addr.ipxa_socket[ 1 ]        = SPX_SOCKET;
tcall.addr.buf                   = ( char * )&spx_addr;
tcall.addr.len                   = sizeof( spx_addr );
tcall.addr.maxlen                = sizeof( spx_addr );
spx_options.spx_connectionID[ 0 ]       = 0;
spx_options.spx_connectionID[ 1 ]       = 0;
spx_options.spx_allocationNumber[ 0 ]   = 0;
spx_options.spx_allocationNumber[ 1 ]   = 0;
tcall.opt.buf                   = ( char * )&spx_options;
tcall.opt.len                   = sizeof( spx_options );
tcall.opt.maxlen                = sizeof( spx_options );
tcall.udata.buf                 = ( char * )0;
tcall.udata.len                 = 0;
tcall.udata.maxlen              = 0;

/* Step 4 */

if ( t_connect( fd, &tcall, &tcall ) == -1 )
{
   t_error( "t_connect failed" );
   if ( t_errno == TLOOK && t_look( fd ) == T_DISCONNECT )
      SPXDisconReason( fd );
   exit( 2 );
}
printf( "\nt_connect successful, beginning send loop\n" );

/* Step 5 */

for ( i1 = 0; i1 < 10; i1++ )
```

```
    {
       sprintf( buf, "message %d", i1 );
       if ( t_snd( fd, buf, strlen( buf )+1, 0 ) == -1 )
       {
          t_error( "t_snd failed" );
          exit( 2 );
       }
       flags = 0;
       if ( t_rcv( fd, buf2, sizeof( buf2 ), &flags ) == -1 )
       {
          t_error( "t_rcv failed" );
          if ( t_errno == TLOOK && t_look( fd ) == T_DISCONNECT )
             SPXDisconReason( fd );
          exit( 2 );
       }
       if ( strcmp( buf, buf2 ) == 0 )
          printf( "Sent & received message: '%s'\n", buf2 );
       else
       {
          printf( "Received back invalid message %s\n", buf2 );
          t_close( fd );
          exit( 3 );
       }
    }

    /* Step 6 */

    if ( t_snddis( fd, ( struct t_call * )0 ) == -1 )
    {
       t_error( "t_snddis failed" );
       exit( 2 );
    }
    t_close( fd );

} /* End of main */

void SPXDisconReason( int fd )
{
    struct   t_discon discon;
    char   *msg;

    if ( t_rcvdis( fd, &discon ) == -1 )
    {
       t_error( "t_rcvdis failed" );
       exit( 2 );
    }
    switch( discon.reason )
    {
       case TLI_SPX_CONNECTION_FAILED:
          msg = "Connection failed";
          break;
       case TLI_SPX_CONNECTION_TERMINATED:
```

```
            msg = "Connection terminated by client";
            break;
         case TLI_SPX_MALFORMED_PACKET:
            msg = "Internal SPX interface error -- malformed packet";
            break;
         default:
            msg = "Unknown termination reason";
      }
      printf( "SPX Connection terminated: %s\n", msg );
   }
```

Parent Topic: TLI Programming Example

dserver.c

dserver.c is the passive side of the connection. It demonstrates the basic steps required to establish a connection from the listening side. There are no command line parameters:

1. The server begins by opening a read/write endpoint in nonblocking mode with SPX as the transport provider. Nonblocking mode gives the server more control over events when it begins listening for a connection.

2. The next step is to bind the endpoint to an SPX network address.**t_bind (Function)** supplies the local network and node address, but the server must specify a socket. In this case, it is an arbitrary socket value hard-coded in both server and client programs. The server also supplies space for storing the address. It does so by declaring IPX_ADDR, *spx_addr* and assigning the structure to *addr.buf* in t_bind (Structure).

   Even though the server processes only one client at a time, several clients can attempt to connect to the server at once. To handle multiple connection requests, the server sets the endpoint's queue length to 5, allowing it to store up to five requests at a time. Also, note that the data returned by **t_bind** overwrites the data the server passes to the function.

3. Now the server is ready to begin listening for connections. The server must prepare t_call to pass to **t_listen**. **t_listen** fills in t_call with the address of the caller. The server must ensure adequate space is available for the address and the SPX options.

4. The server enters a do-while loop containing most of the server's processing. Inside this loop is another do-while loop. The inner loop polls for a connection with **t_listen**. It continues until **t_listen** receives a request for a connection (*listenResult* is 0) or the user presses a key, at which point the server exits. The outer loop continues until **t_listen** returns an error.

5. When **t_listen** receives a connection request, the server opens a new endpoint and binds it. The server passes this endpoint to **t_accept** to

establish the connection. Switching the connection to a new endpoint allows the server to reserve the original endpoint for listening. Also, *listenResult* is -1 only if **t_listen** returned an error other than T_NODATA.

6. Having established a connection, the server enters a while loop for transferring data. The server receives data from the client with **t_rcv**. Because the endpoint was opened in blocking mode, **t_rcv** waits until it receives data before returning. The server returns the same data to the client with **t_snd**. The while loop breaks when **t_rcv** is no longer receiving data.

7. The server closes the connection with **t_snddis** and closes the endpoint with **t_close**. These procedures conclude the processing within the do-while loop the server entered with step #4.

8. If an error occurs, **SPXDisconReason** reports the reason for the error. Otherwise, the server closes the original endpoint when the user presses a key to interrupt processing.

dserver.c example

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <fcntl.h>
#include <tispxipx.h>
#include <tiuser.h>

#define SPX_SOCKET 31      /* arbitrary socket */

#ifndef t_errno
   int t_errno;
#endif
void SPXDisconReason( int fd );
void main ( )
{
   IPX_ADDR   spx_addr;

   /* Step 1 */

   SPX_OPTS   spx_options;
   struct     t_bind tbind;
   struct     t_call tcall;
   char       buf[ 100 ];
   int        fd, nfd, flags, len, listenResult;

   if ( ( fd = t_open( "/dev/nspx", O_RDWR | O_NDELAY,
                       ( struct t_info * )0 ) ) == -1 )
   {
      t_error( "open of /dev/nspx failed" );
      exit( 1 );
   }
```

```
/* Step 2 */

spx_addr.ipxa_socket[ 0 ]  = 0;
spx_addr.ipxa_socket[ 1 ]  = SPX_SOCKET;
tbind.addr.len             = sizeof( spx_addr );
tbind.addr.maxlen          = sizeof( spx_addr );
tbind.addr.buf             = ( char * ) &spx_addr;
tbind.qlen                 = 5;

if ( t_bind( fd, &tbind, &tbind ) == -1 )
{
   t_error( "bind failed" );
   exit( 1 );
}

/* Step 3 */

tcall.addr.buf      = ( char * ) &spx_addr;
tcall.addr.maxlen   = sizeof( spx_addr );
tcall.addr.len      = sizeof( spx_addr );
spx_options.spx_connectionID[ 0 ]     = 0;
spx_options.spx_connectionID[ 1 ]     = 0;
spx_options.spx_allocationNumber[ 0 ] = 0;
spx_options.spx_allocationNumber[ 1 ] = 0;
tcall.opt.buf        = ( char * )&spx_options;
tcall.opt.len        = sizeof( spx_options );
tcall.opt.maxlen     = sizeof( spx_options );
tcall.udata.buf      = ( char * )0;
tcall.udata.maxlen   = 0;
tcall.udata.len      = 0;

do         /* Step 4 */
{
   do
   {
      if ( kbhit( ) )
      {
         getch( ); /*  clean up keyboard buffer */
         printf( "DSERV terminated\n" );
         t_close( fd );
         exit( 0 );
      }
   } while ( ( listenResult = t_listen( fd, &tcall ) ) == -1 &&
                                                  t_errno == TNODATA )

   /* Step 5 */

   if ( listenResult != -1 )
   {
      if ( ( nfd = t_open( "/dev/nspx", O_RDWR, ( struct t_info * )0
      {
```

```
                t_error( "t_open after listen failed" );
                t_close( fd );
                exit( 1 );
             }
             if ( t_bind( nfd, ( struct t_bind * )0, ( struct t_bind * )0 )
             {
                t_error( "t_bind failed" );
                t_close( nfd );
                t_close( fd );
                exit( 1 );
             }
             if ( t_accept( fd, nfd, &tcall ) == -1 )
             {   t_error( "t_accept failed" );
               t_close( nfd );
               t_close( fd );
               exit( 1 );
             }
             printf( "\nConnection accepted." );
             flags = 0;      /* initialize for t_rcv call */

             /* Step 6 */


             while ( ( len = t_rcv( nfd, buf, sizeof( buf),&flags ) ) != -1
             {
                printf( "Received and returned message'%s'\n", buf );
                if ( t_snd( nfd, buf, len, 0 ) == -1 )
                {
                   t_error( "t_snd failed" );
                   t_close( nfd );
                   t_close( fd );
                   exit( 1 );
                }
             }

             /* Step 7 */

             if ( t_errno == TLOOK  &&  t_look( nfd ) == T_DISCONNECT )
                   SPXDisconReason( nfd );
             else      /* some other error so disconnect */
             {
                if ( t_snddis( nfd, ( struct t_call * )0 ) == -1 )
                t_error( "t_snddis failed " );
             }
             t_close( nfd );
             printf( "listening... " );
          }              /* End of if (listenResult != -1 )  */
       } while ( listenResult != -1 );         /* End of outer do-while loo

       /* Step 8 */

       if ( t_errno == TLOOK && t_look( fd ) == T_DISCONNECT )
```

```
                              SPXDisconReason( fd );
      else
      {
         t_error( "t_listen failed" );
         t_close( fd );
      }
   }
   /* End of main  */

   void SPXDisconReason( int fd )
   {
      struct   t_discon discon;
      char     *msg;

      if ( t_rcvdis( fd, &discon ) == -1 )
      {
         t_error( "t_rcvdis failed" );
         exit( 2 );
      }
      switch( discon.reason )
      {
         case TLI_SPX_CONNECTION_FAILED:
            msg = "Connection failed";
            break;
         case TLI_SPX_CONNECTION_TERMINATED:
            msg = "Connection terminated by client";
            break;
         case TLI_SPX_MALFORMED_PACKET:
            msg = "Internal SPX interface error -- malformed packet";
            break;
         default:
            msg = "Unknown termination reason";
      }
      printf( "SPX Connection terminated: %s\n", msg );
   }
```

Parent Topic: TLI Programming Example

Endpoint Management in TLI

Before attempting to transfer data, both workstations must set up local endpoints to support the connection.

Opening an Endpoint

Call **t_open** to set up an endpoint. As input, this function takes the name of a file identifying the transport provider (SPX,TCP/IP, and so on) and configuration flags. If the endpoint is opened successfully, you receive a

handle to the endpoint for use in subsequent operations:

```
int t_open(char *path, int oflag, struct t_info
*info);
```

The configuration flags specify the endpoint's read/write status and the blocking mode. When you are finished using an endpoint, remove the endpoint with **t_close**.

Endpoint Information

You can choose to receive additional information about the underlying transport at the time you open the endpoint. After you have opened the endpoint, you can read this information by calling **t_getinfo**. Endpoint data is contained by t_info.

Binding an Endpoint

After opening an endpoint, you must assign an address to the endpoint with **t_bind (Function)**. When you call this function, you can either specify the address yourself or TLI will assign one for you:

```
int t_bind(int fh, struct t_bind *req, struct t_bind
*ret);
```

The characteristics of the address depend on the underlying transport. For example, if the transport provider is SPX, you could bind the endpoint to a specific socket or have TLI assign a socket value to the endpoint. Call **t_unbind** to unbind the endpoint. This function disables the endpoint but leaves it available for another operation.

Parent Topic:

TLI Local Management Issues

Environment Enhancement for TLI

To create an environment that allows an NLM to call TLI functions, the following modules must be loaded on the server.

STREAMS.NLM, which provides the STREAMS environment

SPXS.NLM, which couples the STREAMS environment to SPX (if using SPX or SPX II)

CLIB.NLM, which contains the NLM interface

TLI.NLM, which contains the TLI functions

When linking your NLM, you can specify that the above modules be autoloaded, in which case the loader automatically loads them if they are not in memory when your NLM is loaded. (See MODULE).

Parent Topic: NetWare IPX/SPX/SPX II

Error Handling in TLI

If an error occurs during processing, a TLI function places the specific error value in the global *t_errno* variable and returns -1 to your application. To evaluate an error, you should test the value of *t_errno*.

If an asynchronous transport event interrupts a TLI function, the function places TLOOK in *t_errno*. This indicates that an event might be pending that you need to process.

If *t_errno* is set to TLOOK, call **t_look** to find out what kind of event interrupted the operation. Once you have determined the nature of the event, you can process the event accordingly. The following table shows the set of TLOOK events.

You can call **t_error** to output the current value of *t_errno* to the standard output. As input, **t_error** take a description of the problem, which it prints followed by a colon and a standard error message.

Table auto. TLOOK Events

Event	Comment
T_LISTEN	The endpoint received a request for a connection from another endpoint.
T_CONNECT	The endpoint received a confirmation of your own request for a connection on this endpoint.
T_DATA	The endpoint received data from another endpoint.
T_EXDATA	The endpoint received expedited data from another endpoint.
T_DISCONNECT	The endpoint received notice the connection was aborted or rejected.
T_ORDREL	The endpoint received a request for an orderly release of the connection. (Not supported under SPX.)
T_UDERR	The endpoint received notice of an error in a previously sent datagram.

Parent Topic: TLI Local Management Issues

Establishing a Connection in TLI

Once you have opened an endpoint and bound it to an address, you can attempt to establish a connection. The steps to connection establishment require one node to act as the caller and the other node to act as the listener. This arrangement reflects the typical relationship between a client and server where the server continually listens for clients who are attempting to establish connections. However, after the connection is established, TLI doesn't differentiate between the roles assumed by either side.

Listening for a Connection

Call **t_listen** to anticipate and respond to another endpoint's request for a connection. This function takes the endpoint and t_call as input. t_call receives connection data if the connection is successful:

```
int t_listen(int fh, struct t_call *call);
```

As with **t_connect**, you pass protocol-specific information to **t_listen** through netbuf nested in t_call. Using SPX as an example again, *buf.addr* in *call* would point to the IPX address of the calling node and *opt.addr* would point to the SPX connection ID and allocation number if the request is received successfully.

t_listen only receives connection requests, it does not respond to requests. Call **t_accept** to accept a connection request and establish the connection. **t_accept** allows you to transfer the connection from the endpoint that receives the request to another endpoint for handling the connection. As input, this function takes the endpoint that listened for the request, the endpoint that handles the connection, and t_call data returned by **t_listen**:

```
int t_accept(int fh, int resfh, struct t_call
*call);
```

It isn't necessary to specify a new endpoint, but this can be a very useful step, especially for servers that are managing multiple connections. If you are changing endpoints, you must open and bind the new endpoint.

Calling for a Connection

Call **t_connect** to request a connection with another node. As input, this function takes an endpoint and a pair of t_calls. One t_call contains data to initialize the connection. The other structure receives any data relevant to a successful connection.

If the connection is successful, **t_connect** returns zero:

```
int t_connect(int fh, struct t_call *sndcall, struct
t_call *rcvcall);
```

If you don't care about overwriting information, you can use the same t_call

for both *sndcall* and *rcvcall*.

For example, if SPX is the transport provider, addr.buf in *sndcall* would point to the IPX address of the destination and *addr.len* would be the length of this address. Below is an example of the steps that an application might take to set up sndcall. (Assume NET_ADDR, NODE_ADDR , and SPX_SOCKET have been defined and assigned values):

```
IPX_ADDRESS      spx_addr;
struct t_call   sndcall;
.
.
.
memcpy(spx_addr.ipxa_net, NET_ADDR, 4);
memcpy(spx_addr.ipxa_node, NODE_ADDR, 6);
spx_addr.ipxa_socket[0] = 0;
spx_addr.ipxa_socket[1] = SPX_SOCKET;
sndcall.addr.buf        = (char *)&spx_addr;
sndcall.addr.len        = sizeof(spx_addr);
sndcall.addr.maxlen     = sizeof(spx_addr);
.
.
.
```

Likewise, *opt.buf* in *rcvcall* would point to space for receiving the SPX connection ID and SPX allocation number once the connection was established.

Parent Topic:

Connection Mode Service

Event Handling: the Server

Returning to the example, when the client calls **t_connect**, a connect indication is generated on the server's listening transport endpoint. The steps required by the server to process the event are as follows. For each client, the server accepts the connect request and creates a new thread to manage the connection.

TLI Event Handling---Creating Threads to Manage Connections

```
for (;;) {
   /* Listen for incoming connect indication */
   printf("Server: %s is ready ...\n", argv[1]);
   if( t_listen(fh, call) == -1 ) {
      t_error("t_listen");
      exit(1);
   }
   if((fh_new = AcceptConnection()) != NULL)
```

```
            if( BeginThread(ProcessConnection,
               NULL, NULL, fh_new) == -1 ) {
               printf("BeginThread failed\n");
               exit(1);
            }
            ThreadSwitch();
         }
      }
```

The server loops forever, processing each connect indication. The call to **ThreadSwitch** inside the loop ensures that the program relinquishes control and does not hold the system. This is necessary because NetWare is a nonpreemptive OS.

First, the server calls **t_listen** to retrieve the next connect indication. When one arrives, the server calls **AcceptConnection** to accept the connect request. **AcceptConnection** accepts the connection on an alternate transport endpoint (as discussed subsequently) and returns the value of that endpoint.

The *fh_new* variable is a global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server might continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, **BeginThread** starts a new thread to manage the connection.

The first parameter to **BeginThread**, **ProcessConnection**, is a pointer to a function to be executed as a new thread. The second and third parameters are stack parameters, and default values are used in this example. The last parameter, *fh_new*, is a pointer to an argument that is passed to the new thread and is received by **ProcessConnection**.

The server allocates a t_call structure to be used by **t_listen** at the same time the t_bind (Structure) structure is allocated. The third argument to **t_alloc**, T_ALL, specifies that all necessary buffers should be allocated for retrieving the caller's address, options, and user data.

As mentioned earlier, the transport provider in this example does not support the transfer of user data during connection establishment. Therefore, **t_alloc** does not allocate a buffer for the user data. It must, however, allocate a buffer large enough to store the address of the caller. The **t_alloc** function determines the buffer size from the addr characteristic returned by **t_open**. The *maxlen* field of each netbuf structure is set to the size of the newly allocated buffer by **t_alloc**. (For the user data buffer, *maxlen* is 0.)

> **NOTE:** Using the t_call structure, the server calls **t_listen** to retrieve the next connect indication. If one is currently available, it is returned to the server immediately. Otherwise, **t_listen** blocks until a connect indication arrives.

TLI supports an asynchronous mode for these routines, which prevents a

process from blocking. This feature is discussed in TLI Asynchronous Execution Mode.

When a connect indication arrives, the server calls **AcceptConnection** to accept the client's request, as follows:

TLI Event Handling---Accepting Clients' Requests

```
/* Allocate a new TEP, accept and return it on success */
int *AcceptConnection(void) {
   int *fhp;             /* new TEP handle ptr */

   fhp = (int *)malloc(sizeof(int));

   /* Open new TEP */
   if((*fhp = t_open("/dev/nspx", O_RDWR, NULL)) == -1) {
      t_error("t_open");
      free(fhp);
      exit(1);
   }

   /* Bind it, address is not important here */
   if( t_bind(*fhp, NULL, NULL) == -1 ) {
      t_error("t_bind");
      t_close(*fhp);
      free(fhp);
      exit(1);
   }

   /* Accept the call from a client */
   if( t_accept(fh, *fhp, call) == -1 ) {
      t_error("t_accept");
      t_close(*fhp);
      free(fhp);
      exit(1);
   }

/* Check for disconnect */
   ThreadSwitch();
   if( t_look(*fhp) == T_DISCONNECT ) {
      printf("DISCONNECT\n");
      if( t_rcvdis(*fhp, NULL) < 0 )
      terror("t_rcvdis");
      t_close(*fhp);
      free(fhp);
      return NULL;
   }

   printf("ACCEPT\n");
   return fhp;
}
```

The server first establishes another transport endpoint by opening the

/dev/ nspx device node of the transport provider and binding an address. As with the client, a NULL value is passed to **t_bind (Function)** to specify that the user does not care what address is bound by the provider. The newly-established transport endpoint, *fhp*, is used to accept the client's connect request.

The first two arguments of **t_accept** specify the listening transport endpoint and the endpoint where the connection is accepted, respectively. A connection can be accepted on the listening endpoint, but this prevents other clients from accessing the server for the duration of the connection.

The third argument of **t_accept** points to the t_call structure associated with the connect indication. This structure should contain the address of the calling user and the sequence number returned by **t_listen**. The value of sequence is significant if the server manages multiple outstanding connect indications. An example of this situation is presented in TLI Asynchronous Execution Mode.

The t_call structure should identify protocol options the user would like to specify, and user data that might be passed to the client. Because the transport provider in this example does not use protocol options and does not support the transfer of user data during connection establishment, the t_call structure returned by **t_listen** can be passed without change to **t_accept**.

For simplicity in the example, the server exits if either **t_open** or **t_bind (Function)** fails. The **exit** function closes the transport endpoint associated with **fhp*, causing the transport provider to pass a disconnect indication to the client that requested the connection. This disconnect indication notifies the client that the connection was not established: **t_connect** fails, setting *t_errno* to TLOOK.

Upon termination of a program, the **TerminateNLM** function is called to clean up allocated structures and terminate advertising services. This occurs because **TerminateNLM** is registered by calling **atexit** at the beginning of the server program.

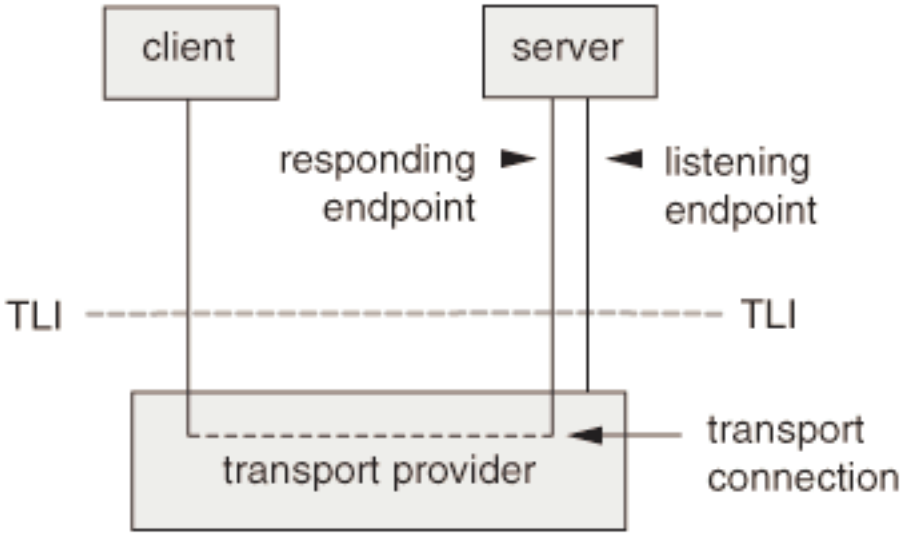
After **t_accept** is executed, an asynchronous event can occur. **ThreadSwitch** is executed to relinquish control, and **t_look** is called to check for an asynchronous event.

The only event that can occur in this state is a disconnect indication. This event can occur if the client decides to undo the connect request it had previously sent. If a disconnect indication arrives, the server must retrieve the disconnect indication using **t_rcvdis**. This function takes a pointer to a t_discon structure as an argument, which is used to retrieve information associated with a disconnect indication. In this example, however, the server does not retrieve this information, so it sets the argument to NULL.

After receiving the disconnect indication, **AcceptConnection** closes the responding transport endpoint and returns NULL, which informs the server that the connection was disconnected by the client. The server then listens for further connect indications. The following figure illustrates how the

server establishes connections.

Figure 14. Listening and Responding Transport Endpoints



The transport connection is established on the newly-created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

Parent Topic: Connection Mode: Event Handling

Guidelines for Writing Protocol-Independent Software

A primary characteristic of TLI is that it is a transport-independent network access method. It enables you to write programs that have no knowledge of the particular transport protocol to which they will interface. This feature enables your networking applications to run in different protocol environments without change.

Use the following guidelines to ensure that the user-level TLI supports protocol independence for applications:

Because the TCP and IPX/SPX protocols do not support data unit boundaries, the concept of a TSDU is not supported. The OSI protocol does support the TSDU.

In the TCP and IPX/SPX protocol environments, the ETSDU feature is not supported. Because both TSDU and ETSDU are not supported, T_MORE and T_EXPEDITED flags are ignored. The OSI protocol does

support the ETSDU.

The protocol-specific service limits returned by **t_open** and **t_getinfo** must not be exceeded. The user is responsible for accessing these limits and adhering to the limits throughout the communication process.

Hide protocol-specific addressing issues from the user program. The user program should not specify any protocol address on **t_bind (Function)** but should allow **t_bind** to assign an address to the user. This hides details concerning protocol-specific addressing from the user.

Similarly, the user must have some way of accessing destination addresses in an invisible manner, such as through a name server.

The reason codes associated with **t_rcvdis** are protocol-dependent. The user should not interpret this information if protocol independence is a concern.

The error codes associated with **t_rcvuderr** are protocol-dependent. The user should not interpret this information if protocol independence is important.

The names of devices should not be hard-coded into the programs. Although you can write software for a particular class of service (for example, the connectionless service), do not write it to depend on any attribute of the underlying protocol.

Programs intended for multiple protocol environments should not use the optional orderly release facility of the connection-oriented service (that is, **t_sndrel** and **t_rcvrel**). This facility is not supported by all connection-based transport protocols.

Parent Topic: TLI States

Incoming Events

The incoming events described in the following table correspond to the successful return of the specified routines, where these routines retrieve data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is pass_conn, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no TLI routine is issued on that endpoint. The pass_conn event is included in the state tables to describe the behavior when a user accepts a connection on another transport endpoint.

Incoming Event	Description	Service Type
listen	Successful return of **t_listen**	T_COTS,

		T_COTS_ORD
rcvconnect	Successful return of **t_rcvconnect**	T_COTS, T_COTS_ORD
rcv	Successful return of **t_rcv**	T_COTS, T_COTS_ORD
rcvdis1	Successful return of **t_rcvdis** with ocnt <= 0	T_COTS, T_COTS_ORD
rcvdis2	Successful return of **t_rcvdis** with ocnt == 1	T_COTS, T_COTS_ORD
rcvdis3	Successful return of **t_rcvdis** with ocnt > 1	T_COTS, T_COTS_ORD
rcvrel	Successful return of **t_rcvrel**	T_COTS_ORD
rcvudata	Successful return of **t_rcvudata**	T_CLTS
rcvuderr	Successful return of **t_rcvuderr**	T_CLTS
pass_conn	Receive a passed connection	T_COTS, T_COTS_ORD

In the above table, the rcvdis events are distinguished by the context in which they occur. The context is based on the value of*ocnt*, which is the count of outstanding connect indications on the transport endpoint.

Parent Topic: TLI State Transitions

Connection Mode Local Management: the Client

The following are the definitions needed by the client program, followed by the required local management steps.

TLI Connection Mode Local Management---the Client

```
#define TLI_TYPE 0x9000       /* Server type (dynamic area) */

main(int argc, char **argv)
{
   int  fh, flags;
   char iobuf[132];
   struct t_call  *sndcall;
   BYTE  pv[128];                /* holds property value */
   BYTE  dc[1];                  /* don't care BYTE      */

   if( argc != 2 ) {
      printf("Usage: %s <server's name>\n", argv[0]);
      exit(1);
   }

   /* Check if the server is active */
```

```
    if(ReadPropertyValue(argv[1], TLI_TYPE, "NET_ADDRESS",
       1, pv, dc, dc) != NULL) {
       printf("Server %s is not available.\n", argv[1]);
       exit(1);
    }

/* Open an endpoint, no info needed */
    if ((fh = t_open("/dev/nspx", O_RDWR, NULL)) == -1) {
       t_error("t_open");
       exit(1);
    }

    /* Request the provider to assign an address */
    if ( t_bind(fh, NULL, NULL) == -1 ) {
       t_error("t_bind");
       exit(1);
    }
```

The first argument to **t_open** is the pathname of a file system node that identifies the transport protocol that will supply the transport service. In this example, /dev/nspx is an SPX™ device node that identifies a connection-oriented transport protocol. It is opened for both reading and writing, as specified by the O_RDWR open flag.

The third argument can be used to return the service characteristics of the transport provider to the user. This information is useful when writing software that is protocol-independent. For simplicity, the client and server in this example ignore this information. NetWare implementation of the STREAMS based TLI for SPX uses an underlying IPX/SPX™ transport protocol. The transport provider for SPX exhibits the following characteristics:

The transport address format is given in the header file TISPXIPX.H (see IPX_ADDR).

The transport provider supports the T_COTS service type.

User data must not be passed between users during either connection establishment or abortive release.

The transport provider supports protocol-specific options. However, the example does not use them.

Because these characteristics are not needed by the user, NULL is specified in the third argument to **t_open**. If the user needed a service other than T_COTS, another transport provider would be opened. An example of the T_CLTS service invocation is presented in TLI Connectionless Mode Services.

The return value of **t_open** is an identifier for the transport endpoint that will be used by all subsequent calls to TLI functions. This identifier is actually a file handle obtained by opening the transport protocol file (see **open**). The significance of this fact is highlighted in A TLI Read/Write

Interface.

After the transport endpoint is created, the client calls **t_bind (Function)** to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address the user wants to bind to the endpoint, and the third argument is set on return from **t_bind** to specify the address that the provider bound.

The address associated with a server's transport endpoint is important because that is the address all clients use to access the server. However, the typical client does not care what its own address is because no other process will try to access it. That is the case in this example, in which the second and third arguments to **t_bind** are set to NULL. A NULL second argument directs the transport provider to choose an address for the user. A NULL third argument specifies that the user does not care what address was assigned to the endpoint.

If either **t_open** or **t_bind** fails, the program calls **t_error** to print an appropriate error message to *stderr*. If any TLI function fails, *t_errno* is assigned a transport error value. A set of error values has been defined (in TIUSER.H) for TLI, and **t_error** prints an error message corresponding to the value in *t_errno*.

The **t_error** function is analogous to **perror**, which prints an error message based on the value of *errno*. If the error associated with a transport function is a system error, *t_errno* is set to TSYSERR, and *errno* is set to the appropriate value.

Parent Topic: Connection Mode: Local Management

Initialization/Deinitialization Phase of Connection-Oriented Service

Before a user can establish a transport connection, the user's environment must be initialized. Specifically, the user must

Create a local communication path to the transport provider (for example, creating the transport endpoint)

Obtain necessary protocol-specific information

Activate the transport endpoint.

A transport endpoint is considered active when the transport provider can accept or request connections associated with the endpoint.

After a connection is released, the transport user must deinitialize the associated transport endpoint, making it available for future use.

The functions listed in the following table support initialization/deinitialization tasks. All these functions provide local

management functions; no information is sent over the network.

Table auto. Functions that Support Initialization/Deinitialization

Function	Task
t_open	Creates a transport endpoint and returns protocol-specific information associated with that endpoint. It also returns a file handle that serves as the local identifier of the endpoint.
t_bind	Associates a protocol address with a given transport endpoint, thereby activating the endpoint. It also directs the transport provider to begin accepting connect indications, if so desired.
t_optmgmt	Enables the user to get or negotiate protocol options with the transport provider.
t_unbind	Disables a transport endpoint so that no further request destined for the given endpoint are accepted by the transport provider.
t_close	Informs the transport provider that the user is finished with the transport endpoint and frees any local resources associated with that endpoint.

The functions listed in the following table are also local management functions, but can be issued during any phase of communication.

Table auto. Local Management Functions That Can Be Called During Any Phase of Communication

Function	Task
t_getinfo	Returns protocol-specific information associated with the specified transport endpoint.
t_getstate	Returns the current state of the transport endpoint.
t_alloc	Allocates storage for the specified library data structure.
t_free	Frees storage for a library data structure that was allocated by **t_alloc.**
t_error	Prints a message describing the last error encountered during a call to a TLI function.
t_look	Returns the current event associated with the given transport endpoint.

Parent Topic: Overview of Connection-Oriented Service

Initialization/Deinitialization Phase of Connectionless Service

Before a user can transfer data in connectionless mode, the environment of the user must be initialized. Specifically, the user must create a local communication path to the transport provider (that is, create the transport endpoint), obtain necessary protocol-specific information, and activate the transport endpoint.

To stop sending or receiving data units through a given transport endpoint, the user must deinitialize the endpoint, thereby freeing the resource for future use.

> **NOTE:** The functions that support the initialization and deinitialization tasks are the same functions used in the connection-mode service.

Parent Topic: Overview of Connectionless Service

IPX Protocol

IPX provides a connectionless transport service that is packet-oriented. It supports transfer of data in self-contained units or datagrams with no logical relationship required among multiple datagrams. IPX requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted.

You can use IPX if your applications need short-term request/response interactions, exhibit a high level of redundancy, are dynamically reconfigurable, or do not require guaranteed, sequenced delivery of data.

IPX has the following characteristics:

Single-service access (SSA), which need not relate to any other service access, presents the transport provider with all the information required to deliver a datagram (for example, a destination address), together with the data to be transmitted.

Each transmitted datagram is self-contained and can be independently routed by the transport provider.

Parent Topic: NetWare IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_accept

(SPX/SPX II) The transport endpoint that accepts the connection contains

the connection ID pertaining to the connection being accepted.

SPX considers the connection established even before you call **t_accept**. If you reject the connection with **t_snddis**, SPX aborts the connection.

If you accept a connection on the same endpoint on which you received the connect indication (*resfd==fd*), the endpoint ceases to be a listening endpoint for the duration of the connection. However, after disconnecting, the endpoint can be used for listening once again. (**t_bind (Function)** address and *qlen* are preserved across the connections.)

You must pass the sequence number, which is the SPX connection ID returned by **t_listen**. This is easily accomplished by calling **t_accept** with the same t_call structure that was used for **t_listen**. Because this implementation does not support the transfer of data during the **t_accept** call, *udata.len* must be set to zero.

After a successful call to **t_listen**, the server typically opens and binds a new endpoint. Next, it calls **t_accept**, passing the new endpoint returned from **t_open**, and a pointer to the same t_call structure that was used for **t_listen**. Upon accepting the connection, the server should spawn a new thread to send and receive data on the new connection, and then go back to listen for other incoming connections.

When processing multiple connections using **t_listen** and **t_accept**, you should take care to ensure that the t_call structure for one **t_listen**/**t_accept** pair is not being used by a second pair at the same time. There is no way to ensure that queued **t_listen**/**t_accept** pairs are processed in order.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_alloc

(IPX/SPX/SPX II) No implementation specific details.

You should use **t_alloc** whenever possible to allocate all TLI structures. This ensures compatibility between different TLI implementations and ensures that all fields are initialized properly.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_bind

(IPX/SPX/SPX II) A STREAM endpoint is bound when it is associated with an SPX socket number. You can either request that the transport provider bind the endpoint to a specific SPX socket number or request that the transport provider allocate and bind a dynamic socket.

If required, the transport provider returns the network, node, and socket

number in the *ret* parameter (a t_bind structure). This represents the address of the workstation and socket number to which the endpoint is bound.

If you pass a NULL (0) for **t_bind (Function)**'s *req* parameter (a t_bind (Structure) structure), the transport provider assigns a dynamic socket number from a list of available socket numbers in the dynamic range (0x4000 to 0x7FFF). Otherwise, the specified socket is opened and the transport endpoint is bound to it.

If the socket to be bound is already open by SPX II, it is reused. (SPX II can multiplex through its sockets.) However, if the socket is opened by a protocol other than SPX II, a new socket is assigned.

The *len* field in the addr structure contains the length of the protocol address, which is set to 12.

After a successful call to **t_bind** in the connection-oriented mode (SPX and SPX II), you can listen for a connect indication if the value of *qlen* is greater than zero, or you can initiate a connection. Currently, there is no assigned limit to the value of *qlen*.

In the connectionless mode (IPX), you can begin to send and receive data after **t_bind** is successfully called.

Typically, you do not specify the socket number to bind. On the other hand, the server usually specifies a well-known socket number or uses SAP to advertise its socket number.

If you request a specific socket number, should check the socket number returned by the transport provider to determine whether the endpoint was indeed bound with the requested socket.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_blocking

(IPX/SPX/SPX II) This function is specific to the NetWare implementation of TLI and is not a part of TLI specifications. It sets the STREAM to blocking mode.

TLI is asynchronous by nature; events occur independently of the application that is using TLI. For example, an application might be sending data over a connection when a disconnect is received. The NLM must be notified of such events and given the opportunity to respond in an asynchronous manner.

TLI responds to these asynchronous events in two modes of operation: synchronous (blocking) mode and asynchronous (nonblocking) mode.

Rather than creating different functions for the different modes of operation, TLI handles incoming events differently, according to the mode of

operation.

In blocking mode, the functions wait for the specific events to complete before returning control to the application. In asynchronous (nonblocking) mode, TLI immediately returns control to the application without requiring the events to actually occur; the application receives notification of the status of the event and can continue to do useful work before the event actually completes.

The mode of operation is normally set through the O_NONBLOCK flag when calling **t_open**.

An example of the differences between the TLI modes of operation can be seen in the connection process. For example, when **t_connect** is called in blocking mode, it returns control to the application only after the connection is established.

In contrast, when **t_connect** is called in nonblocking mode, it does the following:

Returns successfully if the server was set up to properly listen for the incoming connection.

Returns with *t_errno* set to indicate the error found.

In asynchronous (nonblocking) mode, the next step the client must take is to call **t_rcvconnect**, which returns successfully if the server has had time to properly accept the connection, and if no errors are detected.

If the server has not previously accepted the connection, the client can use **t_rcvconnect** to poll the endpoint for existing connect confirmations. If no confirmations exist, **t_rcvconnect** fails, setting *t_errno* to T_NODATA. Control is then returned to the application, giving the application the opportunity to attend to other events during the connection establishment process.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_close

(IPX/SPX/SPX II) The close routine of the transport provider is called whenever you call **t_close**. In the case of SPX, if **t_close** is issued from any state other than T_UNBND, SPX aborts the connection. In this case, any data queued to be sent or the data that had been received is discarded.

If you create more than one thread, each thread should close endpoints explicitly with **t_close** before exiting because **ExitThread** does not close endpoints.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_connect

(SPX/SPX II) Before calling **t_connect**, you must initialize the *sndcall* parameter (a t_call structure) in the following manner:

Set the *addr.buf* field to point to an IPX_ADDR structure that you have initialized. All of the IPX_ADDR fields must be initialized.

Set the *addr.len* and *addr.maxlen* fields to `sizeof (IPX_ADDR)`.

Set the *opt.buf* field to point to an initialized SPX_OPTS structure if you are using SPX, or to an SPX2_OPTIONS structure if you are using SPX II.

Set the *opt.len* and *opt.maxlen* fields to `sizeof (SPX_OPTS)` or `sizeof (SPX2_OPTIONS)`, depending on whether you are using SPX or SPX II.

Set the *udata.len* field to zero because data transfer is not supported by **t_connect**.

Use **t_connect** on active endpoints, **t_listen** for passive endpoints.

After calling **t_unbind**, you can make the endpoint passive again by binding with **t_bind (Function)** (**t_bind** address and *qlen* are preserved across the connections.)

If you want to confirm a connection, you pass in the address of a t_call structure for **t_connect**'s *rcvcall* parameter (instead of passing in NULL). Before calling **t_connect**, initialize the *addr.maxlen* field to `sizeof (IPX_ADDR)`. On return, *addr.buf* points to an IPX_ADDR structure, which contains the address of the responding endpoint.

As long as you unbind and bind a connected endpoint that was either disconnected or released earlier, that endpoint can be reused. (**t_bind (Function)** address and *qlen* are preserved across the connections.)

In blocking mode **t_connect** waits for a connection to complete. In nonblocking mode, it returns immediately and should be followed by a call to **t_rcvconnect**.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_getinfo

(IPX/SPX/SPX II) The protocol-dependent information about the transport layer is returned in the *info* parameter, (a t_info structure). The fields returned have the following values:

addr	`sizeof (IPX_ADDR)`, 12 bytes

options	`sizeof (IPX_OPTS)`, 8 bytes; `sizeof (SPX_OPTS)`, 8 bytes; or `sizeof (SPX2_OPTIONS)`, 52 bytes; depending upon the protocol being used
tsdu	-1 (SPX, SPX II), 546 (IPX)
etsdu	-2
connect	-2
discon	-2
servtype	T_COTS (SPX), T_COTS_ORD (SPX II), T_CLTS (IPX)

For SPX and SPX II, -1 is placed in the *tsdu* field, indicating that the message size is unlimited.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_listen

(SPX/SPX II) Upon receiving a connect indication from SPX, the transport provider accepts the connection, if possible, and queues the indication. SPX and SPX II consider the connection established.

Before calling **t_listen**, you must initialize the *call* parameter (a t_call structure) in the following manner:

Set the *addr.buf* field to point to an IPX_ADDR structure that you have allocated. All of the IPX_ADDR fields must be initialized.

Set the *addr.maxlen* field to `sizeof (IPX_ADDR)`.

Set *opt.buf* field to point to an initialized SPX_OPTS structure if you are using SPX, or to an SPX2_OPTIONS structure if you are using SPX II.

Set the *opt.maxlen* field to `sizeof (SPX_OPTS)` or `sizeof (SPX2_OPTIONS)`, depending on whether you are using SPX or SPX II.

Set the *udata.maxlen* field to zero because data transfer is not supported by **t_listen**.

The transport provider assigns a unique sequence number greater than 0, and informs you of a pending connect indication. When accepting this connection, you must use this sequence number.

The transport provider queues connect indications from SPX up to the maximum specified by **t_bind (Function)** in *qlen*; it rejects further connect indications until the number of queued connect indications are less than *qlen* . You should respond to a successful call to **t_listen** by calling either **t_accept** or a **t_snddis**.

Typically, the server listens for connect indications by calling **t_listen**. After successfully calling **t_listen**, the server opens a new endpoint by calling **t_open** and binds that endpoint with **t_bind**.

The server accepts the connection by passing the *fd* of the new endpoint to **t_accept**. The server then listens for any further connect indications. Thus, the server endpoint can be dedicated for listening for any connect indication. To reject a connection, the server must call **t_snddis** and continue to listen for further connect indications.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_nonblocking

(IPX/SPX/SPX II) This function is specific to the NetWare implementation of TLI and is not a part of TLI specifications. This routine sets the transport stream to nonblocking mode.

For example, in a traditional STREAMS environment, you might enter the following command to open a STREAM in nonblocking mode:

```
fd = t_open("/dev/nspx", O_RDWR | O_NDELAY, (struct t_info *) 0);
```

When using **t_nonblocking**, you must first open a STREAM in blocking mode (that is, do not specify the O_NDELAY flag). Then, change the state using **t_nonblocking**.

If the call to **t_nonblocking** is successful, it returns 0. If the call is unsuccessful, it returns a -1 and sets *t_errno* to TBADF to signify that the file handle is invalid.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_open

(IPX/SPX/SPX II) This implementation supports the following devices:

/dev/nipx for a connectionless transport service based on IPX

/dev/nspx for a connection-oriented transport service based on SPX

/dev/nspx2 for a connection-oriented transport service based on SPX II

The default characteristics of the underlying transport protocol are described in **t_getinfo**. There is no practical limit to the number of endpoints that can be opened by a transport user; however, the underlying SPX protocol has a limit of 2,000 concurrent sessions.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcv

(SPX/SPX II) The **t_rcv** function places no limit on the amount of data that can be received by a single call. In NetWare TLI, if the message size exceeds the maximum transport packet size, it is split into multiple packets and sent across the wire. When it is received, NetWare TLI reassembles the message.

Although the TLI implementations of SPX and SPX II do not limit the size of the transmitted message, the receiving application must ensure that it provides adequate buffers for receiving the message.

Receiving applications should monitor the T_MORE flag closely when using **t_rcv**. If an application's receive buffer is not large enough to hold the entire message, TLI fills the receive buffer and sets the T_MORE flag, meaning a partial message was received. The application must then iteratively call **t_rcv** until the T_MORE flag is not set, meaning the complete message has been received.

Because this implementation does not support expedited data transfer, it does not support ETSDUs.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcvconnect

(SPX/SPX II) In the *call* parameter (a t_call structure), the *addr.buf* field points to the IPX_ADDR structure, which represents the protocol address of the responding endpoint.

Because this implementation does not support data transfer during **t_rcvconnect**, you must set the *udata.len* field to zero.

If **t_connect** was called in nonblocking mode, you can use **t_rcvconnect** to poll for connection confirmations.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcvdis

(SPX/SPX II) SPX informs you if a connection is disconnected. The *discon* parameter (a t_discon structure) to **t_rcvdis** recieves the disconnection information in its *discon.reason* field. The returned reasons are as follows:

```
TLI_SPX_CONNECTION_FAILED
TLI_SPX_CONNECTION_TERMINATED
```

```
TLI_SPX_MALFORMED_PACKET
TLI_SPX_PACKET_OVERFLOW
TLI_SPX_UNRELIABLE_DEST (SPXII only)
```

If an active endpoint is disconnected, SPX immediately informs you. In the case of a passive endpoint, the transport provider waits for you to respond to an earlier successful call to **t_listen** before informing you that the connection was disconnected.

This implementation does not support any data with the disconnect indication.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcvrel

(SPX II) This function is supported by SPX II but not by SPX. **t_rcvrel** can be used to ensure that data is not lost when closing down a connection.

After opening a connection, you should determine if the connection is using SPX II. If it is, you can use **t_rcvrel** to ensure that no data is lost when the connection is closed.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcvudata

(IPX) On return from **t_rcvudata**, the fields of the t_unitdata structure pointed to by *unitdata* contain the following information:

udata.len contains the length of data or length of *udata.buf*, whichever is lower.

udata.buf contains data to the length specified by *udata.len*.

*addr.le*n is set to 12.

addr.buf points to an IPX_ADDR structure containing the address and socket number to which the endpoint is bound.

opt.buf points to *ipx_type* field of an IPX_OPTS structure.

opt.len is set to `sizeof (IPX_OPTS)` or 0, as determined by *opt.maxlen* passed into the function.

The maximum amount of data that can be received by a single call to **t_rcvudata** is 546 bytes.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_rcvuderr

(IPX) This function is called only to receive a unit data error indication. The t_uderr structure pointed to by the *uderr* parameter contains the following fields:

error contains the error code, which can be any of the following:

```
TLI_IPX_MALFORMED ADDRESS
TLI_IPX_PACKET_OVERFLOW
```

addr.buf points to the destination address that caused the error.

opt.buf points to *ipx.packetType*.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_snd

(SPX/SPX II) For SPX and SPX II, there is no limit the size of the message you can send with a single call to **t_snd**.

The sender can also send a message with multiple calls to **t_snd** and have it appear as a single message by the receiver, if the sender sets the T_MORE flag appropriately. If the sender calls **t_snd** with the T_MORE flag set, TLI assumes that the next **t_snd** is part of the same message. When the T_MORE flag is not set, TLI assumes that **t_snd** will not be called again for the current message.

When TLI sends a message with multiple calls to **t_snd**, the receiving side must monitor the T_MORE flag to ensure that the message boundary is preserved.

Because this implementation does not support expedited data transfer, it does not support ETSDUs. Therefore, the T_EXPEDITED flag should be set to zero.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_snddis

(SPX/SPX II) If the endpoint is active, **t_snddis** causes all the data on the transmitting and receiving queues to be discarded, and the connection is aborted. In the case of a passive endpoint, SPX considers the connection established you are informed of a connect indication.

When you call **t_snddis**, this connection is aborted. You can call **t_connect** again on this endpoint because the bound address is still valid.

Because data transfer is not supported during **t_snddis**, set the *udata.len* field of the *call* parameter to zero.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_sndrel

(SPX II) This function is supported by SPX II but not by SPX. It is used for an orderly release of a connection.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

IPX/SPX/SPX II Notes: t_sndudata

(IPX) The maximum data that a single call to **t_sndudata** can send is 546 bytes.

Initialize the fields of the t_unitdata structure pointed to by the unitdata parameter as follows:

Set the *addr.len* field to `sizeof (IPX_ADDR).`

Set the *addr.maxlen*: `sizeof (IPX_ADDR).`

Set the *addr.buf* field to point to the address of the IPX_ADDR structure, which contains the node, network, and socket number to which the endpoint is bound. To broadcast a datagram to all nodes, you can initialize the *ipxa_node* field of the IPX_ADDR structure to -1 (FF).

Set the *opt.maxlen* field to `sizeof (IPX_OPTS).`

Set the *opt.len* field to `sizeof (IPX_OPTS).`

Set the *opt.buf* field to point to the address of the IPX_OPTS structure.

Set the *udata.maxlen* to the length of the data pointed to by the *udata.buf* field.

Set the *udata.len* to the length of the message.

Set the *udata.buf* to point to the address of the buffer containing data to be sent.

Parent Topic: TLI Function Implementation Notes for IPX/SPX/SPX II

NetWare Implementation of STREAMS-Based IPX/SPX/SPX II

Two types of transport service are provided by the NetWare implementation of STREAMS-based TLI:

Connection-oriented transport service, provided by SPX and SPX II

Connectionless transport service, provided by IPX

SPX and SPX II Implementation

SPX supports most of the TLI functions. However, because of the underlying nature of the SPX protocol, it does not support**t_sndrel** and **t_rcvrel**, which provide an orderly release of a connection.

SPX II, on the other hand, supports all of the TLI functions, including **t_sndrel** and **t_rcvrel**.

Because the orderly release of connections is optional and not supported by all transport providers, there are times when your SPX II application is not able to use **t_sndrel** and **t_rcvrel**. For this reason, after your SPX II application completes the connections process, it should check the connection to see if it supports an orderly release. If the connection does not support an orderly release, your application should close the connection using **t_snddis** and **t_rcvdis** instead of **t_sndrel** and **t_rcvrel**.

SPX and SPX II do not support expedited data transfer. Therefore, they do not support ETSDUs.

The close routine is called if one of the following is true:

Your code calls **t_close** explicitly.

The NLM exits by calling **exit**.

The NLM is unloaded by the **UNLOAD** console command.

In the NLM environment, an application can spawn multiple threads, with each thread able to open its own transport endpoint. For each thread that has established an endpoint, you should close the thread's endpoint by calling **t_close** before exiting the thread because **ExitThread** does not close a thread's endpoint explicitly.

SPX relies on IPX to send and receive packets. SPX II communicates directly with the LSL, which makes communication using SPX II faster.

SPX II can only be accessed through TLI. SPX can be accessed through TLI, or through its own interface.

Parent Topic: NetWare IPX/SPX/SPX II

NetWare IPX/SPX/SPX II

This chapter describes TLI for developing applications in NetWare® 3.11 and above using the IPX/SPX™/SPX II transport protocols. SPX II was added for the NetWare 4.0 OS.

The interface provided in the NetWare environment provides virtually all the functionality of the AT&T TLI specification described in Overview of TLI Functions. However, because TLI is not a native interface to the NetWare environment, some changes to the TLI specification have been required.

The following topics are discussed in this chapter:

- The functionality provided by IPX, SPX, and SPX II through TLI
- The NetWare implementation of STREAMS-based TLI
- The environment enhancement needed for an NLM™ application to make TLI calls
- Information specific to using TLI functions for accessing the transport protocol IPX/SPX/SPX II in the NetWare environment

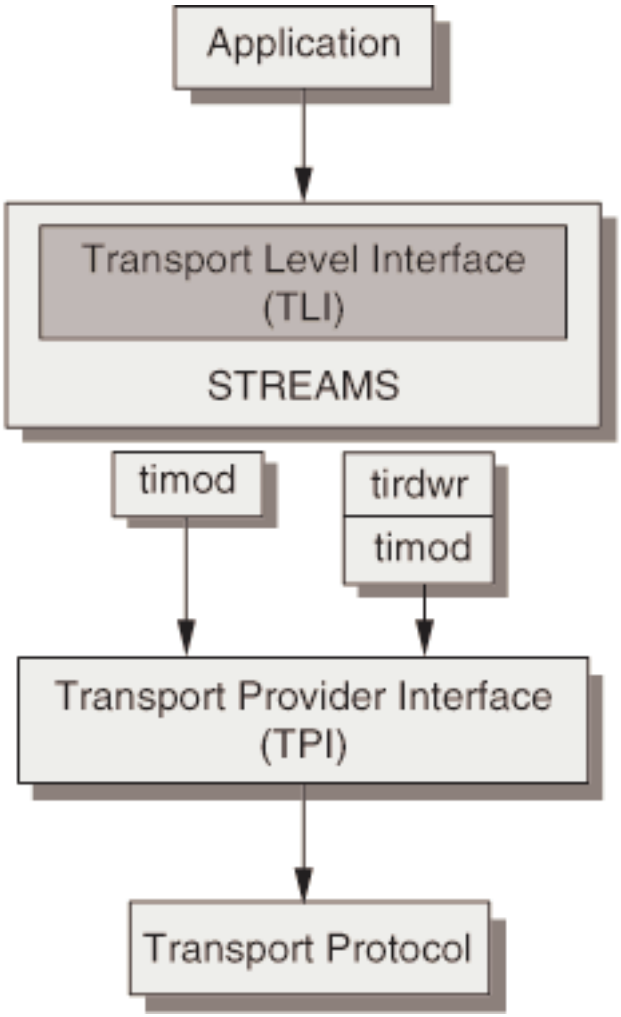
Related Topics

- SPX Protocol
- SPX II Protocol
- IPX Protocol
- NetWare Implementation of STREAMS-Based IPX/SPX/SPX II
- Environment Enhancement for TLI
- TLI Function Implementation Notes for IPX/SPX/SPX II

NetWare OSI

TLI provides a set of library routines that the transport user can use to communicate with the transport layer through the STREAMS-based Transport Provider Interface (TPI). The STREAMS TLI for NetWare 3.11 and later follows this standard closely, with few variations. The following figure illustrates how the STREAMS TLI is associated with the NetWare OSI Lower Layers Protocol Stack.

Figure 15. NetWare OSI Protocol Stack



Two modes of transport service are provided. TP4 provides
connection-oriented transport service, and CLTP provides connectionless

transport service.

Each TLI function initiates an exchange of one or more TPI primitives between the TPI and the TLI. The TPI interfaces the TLI to the transport layer.

1. The **t_open** function enables a user to choose a specific transport provider and establish an endpoint.

2. **t_bind (Function)** binds a transport address to the transport endpoint. Once a call is issued, the TLI library pushes the timod module on top of the TPI driver. Then, it either issues **ioctl** calls to the module or generates TPI primitives, depending on the TLI function that was called.

   Upon receiving the M_IOCTL message, the timod module generates the TPI primitive and returns a response, which the TLI library receives as an M_IOCACK/M_IOCNACK message. The timod module returns any unexpected TPI primitives to the stream head so that the TLI library can handle them.

3. When the transport service is in the connectionless mode (CLTP), the user can begin the transfer of data using either **t_sndudata** or **t_rcvudata**.

   When the transport service is in the connection-oriented mode (TP4), the user can connect to another transport endpoint using **t_connect** and can transfer data using either **t_snd** or **t_rcv**. The user can either disconnect a connection using **t_snddis** or release the connection in an orderly manner using **t_sndrel** and **t_rcvrel**.

4. Then, the user can unbind the transport address by calling **t_unbind**.

5. The user closes the endpoint with **t_close**.

In addition to supporting all the regular TPI primitives, this implementation also supports some option management functions.

The options buffers are modeled after the X/Open standard.

As an alternative to using TLI routines, the user can issue read and write calls on a fully connected transport endpoint by popping the timod module and pushing the tirdwr module. This module converts a TLI connection-oriented transport stream into a simple bidirectional pipe. Plain data issued either from an application to the transport module or from the transport to an application is converted into TPI messages by the tirdwr module, before the data is sent.

After pushing the tirwdr module, the user must stop using TLI routines. The user can end a connection either by closing the file handle associated with the transport endpoint or by popping the tirdwr module off the STREAM.

The close routine of the TPI adapter is called if either of the following occurs:

The user calls **t_close** explicitly.

The NLM™ application exits with a call to **exit**.

In NetWare, the application can spawn multiple threads, with each thread having the capacity to open its own transport endpoint. You should close the endpoint by calling **t_close** for each thread because **ExitThread** does not close the endpoint explicitly.

Various functions use the netbuf structure to send and receive data and information. It contains the following members:

unsigned int *maxlen*;
unsigned int *len*;
char **buf*;

The pointer *buf* points to a user input and/or output buffer. The *len* field generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function replaces the user value of *len* on return.

Generally, the *maxlen* field is significant only when *buf* is used to receive output from the function. In this case, *maxlen* specifies the physical size of the buffer, the maximum value of *len* that the function can set. If *maxlen* is not large enough to hold the returned information, a TBUFOVFLW error generally results. However, certain functions can return part of the data and not generate an error.

Related Topics

OSI Address Format

Options Management Structures

TLI Function Implementation Notes for OSI

Options Management Structures

The options management structures that TLI functions use are defined in OSITLI.H. These structures are passed to the TP4 and CLTP transports to set options.

```
/* * Options Management structures. */

struct rate{
   long targetvalue;                /* target value */
   long minacceptvalue;             /* minimum acceptable value */
};
struct reqvalue {
   struct rate   called;            /* called rate */
   struct rate   calling;           /* calling rate */
```

```
};
struct thrpt {
   struct reqvalue maxthrpt;        /* maximum throughput */
   struct reqvalue avgthrpt;        /* average throughput */
};
struct management {
   short   dflt;         /* T_YES: the following parameters are */
                         /* ignored, default values are used */
                         /* T_NO: the following parameters are used */
   int     ltpdu;        /* maximum length of TPDU (in octets) */
   short   reastime;     /* reassignment time (in seconds) */
   char    class;        /* preferred class; value T_CLASS0 - TCLASS4 *
   char    altclass;     /* alternative class */
   char    extform;      /* extended format T_YES or T_NO */
   char    flowctrl;     /* flow control: T_YES or T_NO */
   char    checksum;     /* checksum (cl. 4) T_YES or T_NO */
   char    netexp;       /* network expedited data: T_YES or T_NO */
   char    netrecptcf;   /* receipt confirmation: T_YES or T_NO */
};

/* * Connection oriented options. */

struct isoco_options {
   struct thrpt      throughput;       /* throughput */
   struct reqvalue   transdel;         /* transit delay */
   struct rate       reserrorrate;     /* residual error rate */
   struct rate       transffailprob;   /* transfer failure prob. */
   struct rate       estfailprob;      /* connection  establ.
                                          failure prob. */
   struct rate       relfailprob;      /* connection release
                                          failure prob. */
   struct rate       estdelay;         /* connection establishment
                                          delay */
   struct rate       reldelay;         /* connection release delay */
   struct netbuf     connresil;        /* connection resilience */
   unsigned short    protection;       /* protection */
   short             priority;         /* priority */
   struct management mngmt;            /* management parameters */
   char              expd;             /* expedited data: T_YES or
                                          T_NO */
};

/* * Connectionless options. */

struct isocl_options {
   struct rate      transdel;          /* transit delay */
   struct rate      reserrorrate;      /* residual error rate */
   unsigned short   protection;        /* protection */
   short               priority;       /* priority */
};

/* * Novell connectionless options. */
```

```
struct novell_isocl_options {
   struct rate      transdel;          /* transit delay */
   struct rate      reserrorrate;      /* residual error rate */
   unsigned short   protection;        /* protection */
   short            priority;          /* priority */
   int              checksum;          /* checksums */
};
```

Related Topics

Setting Options with TP4

Setting Options with CLTP

Parent Topic: NetWare OSI

Orderly Connection Release

During data transfer, the user can generate an orderly connection release. An orderly connection release gracefully terminates a connection and guarantees that no data is lost.

> **NOTE:** All transport providers must support the abortive release procedure. In contrast, orderly release is an optional facility that is not supported by all transport protocols.

The orderly release procedure consists of two steps by each user. The first user to complete data transfer can initiate a release using **t_sndrel**, as illustrated in the example.

Related Topics

Orderly Connection Release: the Server

Orderly Connection Release: the Client

Parent Topic: TLI Connection Mode Services

Orderly Connection Release: the Client

In this example, data is transferred in one direction from the server to the client. When an event occurs on the client's transport endpoint, the client checks whether the expected orderly release indication has arrived. If so, it proceeds with the release procedure by calling **t_rcvrel** to process the indication and **t_sndrel** to inform the server that the client is ready to release the connection. At this point, the client exits, closing its transport endpoint.

The client can respond to an orderly release request as follows:

TLI Orderly Connection Release---the Client

```
/* Process a connection closure */
while ( t_rcvrel(fh) == -1 ) {
   if (t_errno != TNOREL) {
   t_error("t_rcvrel");
   exit(1);
   }
   ThreadSwitch();
}
/* Send disconnection release to the server */
if ( t_sndrel(fh) == -1 ) {
   t_error("t_sndrel");
   exit(1);
}
.
.
.
```

Parent Topic: Orderly Connection Release

Orderly Connection Release: the Server

The **t_sndrel** function informs the client that no more data will be sent by the server. When the client receives this indication, it can continue sending data back to the server, if desired. When all data has been transferred, however, the client must also call **t_sndrel** to indicate that it is ready to release the connection. The connection is released only after both the client and the server have requested an orderly release and received the corresponding indication from the each other.

When the server has transferred all the data, the connection can be released as follows:

TLI Orderly Connection Release---the Server

```
if( t_sndrel(*fh) == -1 ) {
   t_error("t_snddis");
   t_close(*fh);
   free(fh);
   return;
}

while ( t_rcvrel(*fh) == -1 ) {
      if (t_errno != TNOREL) {
      t_error("t_rcvrel");
      t_close(*fh);
      free(fh);
      return;
      }
      ThreadSwitch();
```

```
    }

    if( t_unbind(*fh) == -1 ) {
       t_error("t_unbind");
       t_close(*fh);
       free(fh);
       return;
    }
    t_close(*fh);
    free(fh);
}
```

Parent Topic: Orderly Connection Release

Orderly Release Summary

Call **t_sndrel** to initiate an orderly release. As input, this function takes the local endpoint for the connection to be released. **t_sndrel** returns zero if the request was sent successfully:

```
int t_sndrel(int fh);
```

After calling **t_sndrel**, your application waits for its partner to return an orderly release indication. The indication is reported in *t_errno* as T_ORDREL. You can continue receiving data on the connection after calling **t_sndrel** until you receive the orderly release indication. Call **t_look** to confirm the indication was returned.

If the partner sends a request for an orderly release, you receive notification of the request as an error value while processing the connection. For example, **t_rcv** might receive the error when the partner has no more data to send. The request is reported in *t_errno* as T_ORDREL. Call **t_look** to confirm the request, then call **t_rcvrel** to process the request:

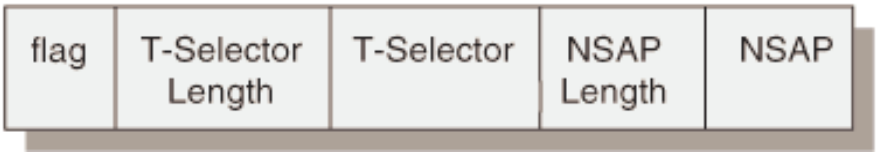
```
int t_rcvrel(int fh);
```

After calling **t_rcvrel**, call **t_sndrel** to send the partner an orderly release indication. Because SPX doesn't support orderly release, TLI doesn't support **t_rcvrel** and **t_sndrel** at the DOS workstation.

Parent Topic: Releasing a Connection in TLI

OSI Address Format

The OSI reference model uses a variable-length format to represent a network address. The following figure shows the OSI address format.



The fields and their contents are as follows:

flag---One-byte field that always contains a zero

TSelector Length---One-byte field that contains the length of the *Tselector* field

TSelector---Variable-length field of 0 to 32 bytes that contains the binary representation of the *Tselector*

NSAP Length---One-byte field that contains the length of the *NSAP* field

NSAP---Variable-length field of 0 to 20 bytes that contains the binary representation of the NSAP address

The address format is identical for CLTP and TP4. However, there are some guidelines that you should follow to ensure that your applications communicate successfully:

When you call **t_bind (Function)**, set the NSAP length to zero to force CLNP to allocate the default local NSAP. When you call **t_sndudata** or **t_connect**, you must specify the NSAP address.

TP4 allows you to specify a zero-length *TSelector*; however, each listener that you establish must be bound to a unique *TSelector*. Therefore, you should avoid using a zero-length *TSelector* with TP4.

TP4 does not allocate server *TSelector* values.

CLTP does not allow a zero-length *TSelector*. If your application calls **t_bind** with a zero-length *TSelector*, CLTP assigns a *TSelector*.

Loopback and multicast are not supported.

Parent Topic: NetWare OSI

OSI Notes: t_accept

The transport endpoint that accepts the connection is set to the following:

The NSAP address and T-Selector of the endpoint that received the connect indication

The NSAP address and T-Selector of the endpoint that initiated the connection

The options negotiated between the initiator and the endpoint that received the connection

If the connecting transport has aborted the connection, a disconnect indication is sent to the *resfh* endpoint.

If the connection is accepted on the same endpoint on which the connect indication was received (*resfh==fh*), the endpoint ceases to be a listening endpoint. However, the user can unbind the endpoint with **t_unbind** and bind it again with **t_bind (Function)** to make this endpoint a listening endpoint.

The user must pass the same sequence number that was passed to **t_listen** earlier.

The isoco_options are not supported during **t_accept**; therefore, *opt_len* should be set to zero.

After a successful **t_listen** call, the server typically opens and binds a new endpoint. Next, it calls **t_accept**, passing the file handle of the new endpoint. Upon accepting the connection, the server spawns a new thread to send and receive data and then goes back to listen for any other incoming connections.

The following sample program uses **t_accept**.

```
call = (struct t_call *) t_alloc (listen_fh, T_CALL, T_ALL);
if (call == NULL)
{
   t_error (t_alloc");
   exit (1);
}

for (;;)
{
   /* Listen for any incoming indications */

   listenresult = t_listen (listen_fh, call);
   if (listenresult == -1)
   {
      t_error (t_listen");
      exit (1);
   }

   /* Open a new endpoint */
   if ((conn_fh = t_open (/dev/tp4", O_RDWR, NULL)) == -1)
   {
      t_error ("t_open");
      exit (1);
```

```
   }

   /* Bind to any address */
   if (t_bind (conn_fh, NULL, NULL) == -1)
   {
      t_error (t_bind");
      exit (1);
   }

   /* Accept the connection */
   if (t_accept (listen_fh, conn_fh, call) == -1)
   {
      t_error (t_accept");
      exit (1);
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_bind

See OSI Address Format.

The following sample programs use **t_bind (Function)**.

```
unsigned char tsel [5] = { 4, 0x00, 0x00, 0x00, 0x01 };

/*
 * Bind to any address. The transport provider will
 * allocate an address
 */

rc = t_bind(fh, NULL, NULL);
if (fh == -1)
{
   t_error("t_bind");
   exit(1);
}

/* Allocate the address request structure */
if ((bind = (struct t_bind *) t_alloc (listen_fh, T_BIND, T_ALL)) ==
          NULL)
{
   t_error (t_alloc");
   exit (1);
}

   bind->qlen = MAX_CIND;      /* Max. no of outstanding
                                * connect indications    */
```

```
bind->addr.len = tsel[0] + 3;     /* Length of the protocol
                                      address */
bind->addr.buf[0] = 0;
memcpy (&bind->addr.buf[1], tsel, tsel[0] + 1);
bind->addr.buf[ 2 + tsel[0]] = 0;
if (t_bind (listen_fh, bind, NULL) == -1)
{
   t_error (t_bind");
   exit (1);
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_close

The close routine of the transport provider is called whenever the user calls **t_close**. In the case of TP4, if **t_close** is called from any state other than T_UNBND, TP4 aborts the connection. Both CLTP and TP4 discard all data queued to be sent, as well as all data that has been received.

The following sample program uses **t_close**.

```
/* close the endpoint */

rc = t_close(fh);
if (rc == -1)
{
   t_error(t_close");
   exit(1);
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_connect

See OSI Address Format.

The user can call **t_connect** on either an active or a passive endpoint. For a passive endpoint, the endpoint must be in T_IDLE state and it should not have received a connect indication earlier. Once the user calls **t_connect**, the passive endpoint ceases to listen and becomes an active endpoint for the duration of the connection, until the endpoint is unbound with **t_unbind**. The user can make the endpoint passive again after calling **t_unbind** and binding again with **t_bind (Function)**.

TP4 supports sending up to 32 bytes of user data with a connect request.

Unless the user unbinds and binds a connected endpoint that was either

disconnected or released earlier, that endpoint cannot be reused.

The following sample program uses **t_connect**.

```
unsigned char nsap[12] = {
11, 0x49, 0x00, 0x01, 0x00, 0x00, 0x1b, 0x02, 0xa5, 0xd3,
0xfe, 0x00
};

unsigned char tsel [5] = { 4, 0x00, 0x00, 0x00, 0x01 };

   /* Synchronous connect mode */

   sndcall = (struct t_call *) t_alloc (fh, T_CALL, T_ADDR);
   if (sndcall == NULL)
   {
      t_error (t_alloc");
      exit (1);
   }

   sndcall->addr.len = nsap[0] + tsel[0] + 3;
   sndcall->addr.buf[0] = 0;
   memcpy (&sndcall->addr.buf[1], tsel, tsel[0] + 1);

   memcpy (&sndcall->addr.buf[2 + tsel[0]], nsap,
      nsap[0] + 1);

   rc = t_connect (fh, sndcall, NULL);
   if (rc == -1)
   {
      t_error (t_connect");
      exit (1);
   }
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_getinfo

The protocol-dependent information about the transport layer is returned in the info structure. The fields returned for TP4 and CLTP have the values listed in the following table.

Table . Fields in the info Structure

Field	TP4	CLTP
addr	54	54
options	128	24
tsdu	1	4,096

etsdu	16	-2
connect	32	-2
discon	64	-2
servtype	T_COTS	T_CLTS

The following sample program uses **t_getinfo**.

```
/* get information about the transport interface */
rc = t_getinfo(fh, info);
if (rc == -1)
{
   t_error("t_getinfo");
   exit(0);
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_listen

On receiving a connect indication from TP4, the transport provider does not accept the connection. The application should call **t_accept** in a timely manner.

See OSI Address Format for information about the OSI addressing format and options management.

The transport provider assigns a unique sequence number greater than -1 and informs the user of a pending connect indication. The t_call structure contains the address of the client and the options settings on the TP4 circuit.

The transport provider queues connect indications from TP4 up to the maximum specified during **t_bind (Function)** in *qlen*; it rejects further connect indications until the number of queued connect indications are less than *qlen*. The user should respond to a successful call to **t_listen** by calling either **t_accept** or **t_snddis**.

Typically, the server listens for connect indications by calling **t_listen**. After a successful call to **t_listen**, the server opens a new endpoint by calling **t_open** and binds that endpoint with **t_bind (Function)**. The server accepts the connection by passing the *fh* of the new endpoint in **t_accept**. The server then listens for any further connect indications. Thus, the server endpoint can be dedicated for listening for any connect indications. If the server does not want to accept a connection, it must call **t_snddis** and continue to listen for further connect indications.

The following is a sample program using **t_listen**.

```
/* Listen for any incoming indication */
while (t_listen(listen_fh, call) == -1)
{
   if (t_errno == TNODATA)
      continue;
   else
   {
      t_error("t_listen");
      exit(1);
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_open

This implementation supports the following devices:

/dev/tp4: For a connection-oriented transport service based on TP4

/dev/cltp: For a connectionless transport service based on CLTP

The default characteristics of the underlying transport protocol are described in **t_getinfo**. There is no assigned limit to the number of devices a user can open.

```
/* Open in synchronous mode */
fh_synch = t_open(/dev/tp4", O_RDWR, NULL);
if (fh_synch == -1)
{
   t_error("t_open");
   exit(1);
}

/* Open in asynchronous mode */
fh_asynch = t_open(/dev/tp4", O_RDWR | O_NDELAY, NULL);
if (fh_asynch == -1)
{
   t_error("t_open");
   exit(1);
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_rcv

Becuase TP4 supports sending expedited data and infinite TSDU size, the user can set the T_EXPEDITED and T_MORE flags upon the successful

return of **t_rcv**.

The following sample program uses **t_rcv**.

```
/*
 * t_rcv() does not necessarily return the full amount
 * requested, so repeat the operation if necessary.
 */
for (nread = 0; nread < sizeof (iobuf); nread += rc)
{
   flags = 0;
   if ((rc = t_rcv(fh_synch, iobuf, sizeof (iobuf),
      &flags)) == -1)
   {
      t_error(t_rcv");
      exit(1);
   }
   nread += rc;
}
printf(Received %d bytes\r\n", nread);

/*
 * t_rcv() in asynchronous mode, if no data is available
 * then t_rcv fails and t_errno is set to TNODATA.
 */
for (nread = 0; nread << sizeof (iobuf); nread += rc)
{
   flags = 0;
   while ((rc = t_rcv(fh_asynch, io_buf, sizeof (iobuf),
      &flags)) == -1)
   {
      if (t_errno == TNODATA)
      flags = 0;
      else
      {
         t_error(t_rcv ");
         exit(1);
      }
   }
 }
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_rcvconnect

The TP4 provider supports the concept of transferring data upon setting up a connection. The maximum amount of data you can receive is 32 bytes.

See OSI Address Format and Options Management Structures.

```
unsigned char nsap[12] = {
```

```
unsigned char nsap[12] = {
11, 0x49, 0x00, 0x01, 0x00, 0x00, 0x1b, 0x02, 0xa5, 0xd3,
0xfe, 0x00};

unsigned char tsel [5] = { 4, 0x00, 0x00, 0x00, 0x01 };

   /* Asynchronous connect */

   sndcall = (struct t_call *) t_alloc (fh, T_CALL, T_ADDR);
   if (sndcall == NULL)
   {
      t_error (t_alloc");
      exit (1);
   }

   sndcall->addr.len = nsap[0] + tsel[0] + 3;
   sndcall->addr.buf[0] = 0;
   memcpy (&sndcall->addr.buf[1], tsel, tsel[0] + 1);

   memcpy (&sndcall->addr.buf[2 + tsel[0]], nsap,
      nsap[0] + 1);

   rc = t_connect (fh_asynch, sndcall, NULL);
   if (rc == -1)
   {
      if (t_errno != TNODATA)
      {
         t_error (t_connect");
         exit (1);
      }
   }

   /* Check whether t_connect succeeded */

   if (t_getstate (fh_asynch) != T_DATAXFER)
   {
      /* loop until t_rcvconnect succeeds */
      while (t_rcvconnect (fh_asynch, NULL) == -1)
      {
         if (t_errno != TNODATA)
         {
            t_error (t_rcvconnect");
            exit (2);
         }
     }
   }
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_rcvdis

If a connection is disconnected, TP4 informs the user. The*reason* field in the t_discon structure has one of the errors listed in the following table.

Table . Disconnection Errors

Error	Meaning
TPDR_NORMAL	Normal disconnect initiated by the session entity
TPDR_CRCONG	Remote transport entity congestion at connect request time
TPDR_CONNEG	Connection negotiation failed
TPDR_DUPSR	Duplicate source reference detected for same pair of NSAPs
TPDR_MMREF	Mismatched references
TPDR_PE	Protocol error
TPDR_REOVFL	Reference overflow
TPDR_NWREF	Connection request refused on this network connection
TPDR_INVHD	Header or parameter length is invalid
TPDR_RNS	Reason not specified
TPDR_CONG	Congestion at the TSAP
TPDR_NOSESS	Session entity not attached to TSAP
TPDR_UNKADDR	Address unknown

If an active endpoint is disconnected, TP4 informs the user immediately. If a passive endpoint is disconnected, the transport provider waits for the user to respond to an earlier successful **t_listen** before informing the user that the connection was disconnected.

The user can call **t_connect** again on this endpoint. TP4 attempts to reconnect to the same protocol address.

TP4 supports the concept of receiving user data while disconnecting a circuit. Up to 64 bytes can be transferred.

```
while (t_rcvdis(fh, NULL) == -1)
{
   if (t_errno != TNODIS)
   {
      t_error("t_rcvdis");
      exit(1);
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_rcvudata

The following sample program uses **t_rcvudata**.

```
/*
   t_rcvudata() in asynchronous mode, if no data is
   available then it fails and t_errno is set to
   TNODATA.
*/
for (nread = 0; nread << sizeof (iobuf); nread += rc)
{
   flags = 0;
   while ((rc = t_rcvudata(fh, unitdata, &flags)) == -1)
   {
     if (t_errno == TNODATA)
         flags = 0;
      else
      {
         t_error("t_rcv ");
         exit(1);
      }
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_snd

TP4 supports expedited data and has an infinite TSDU size. You can set the T_EXPEDITED and T_MORE flags. If *expd* is set to T_NO, an attempt to send expedited data causes the STREAM to terminate. You can determine whether *expd* is on by calling **t_optmgmt (Function)**.

If you send more than 16 bytes of expedited data, the STREAM is terminated. The expedited data channel does not have flow control.

```
/*
 * Assumes that the TP4 quotas add up to more than 1024
 * bytes. Otherwise this strategy invites deadlock.
 */
rc = t_snd(fh, iobuf, sizeof (iobuf), 0);
if (rc == -1)
{
   t_error(t_snd");
   exit(1);
}
```

```
/* asynchronous mode */

while ((rc = t_snd(fh_asynch, iobuf, sizeof (iobuf), 0)) == -1)
{
   if (t_errno != TFLOW)
   {
      t_error(t_snd");
      exit(1);
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_snddis

If the endpoint is active, **t_snddis** causes all the data on the transmitting and receiving queues to be discarded, and the connection is aborted. If the endpoint is passive, **t_snddis** can abort the connection setup (if it is not already aborted). The user can call **t_connect** again on an active endpoint or on a passive endpoint with no pending connect indication.

The maximum *udata* length allowed is 64 bytes.

The following is a sample program using **t_snddis**.

```
/* Aborting an existing connection */

rc = t_snddis(fh, NULL);
if (rc == -1)
{
   t_error("t_snddis");
   exit(0);
}

/* Refuse a connection */

/* Listen for any incoming indication */

while (t_listen(listen_fh, call) == -1)
{
   if (t_errno == TNODATA)
      continue;
   else
   {
      t_error("t_listen");
      exit(1);
   }
}
rc = t_snddis(listen_fh, call);
if (rc == -1)
```

```
if (rc == -1)
{
   t_error("t_snddis");
   exit(0);
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Notes: t_sndudata

See OSI Address Format and Options Management Structures.

The maximum data that a single call to **t_sndudata** can send is limited by STRMSGZ, as defined in STREAM.H. Typically, the maximum is 4,096 bytes.

The following sample program uses **t_sndudata**.

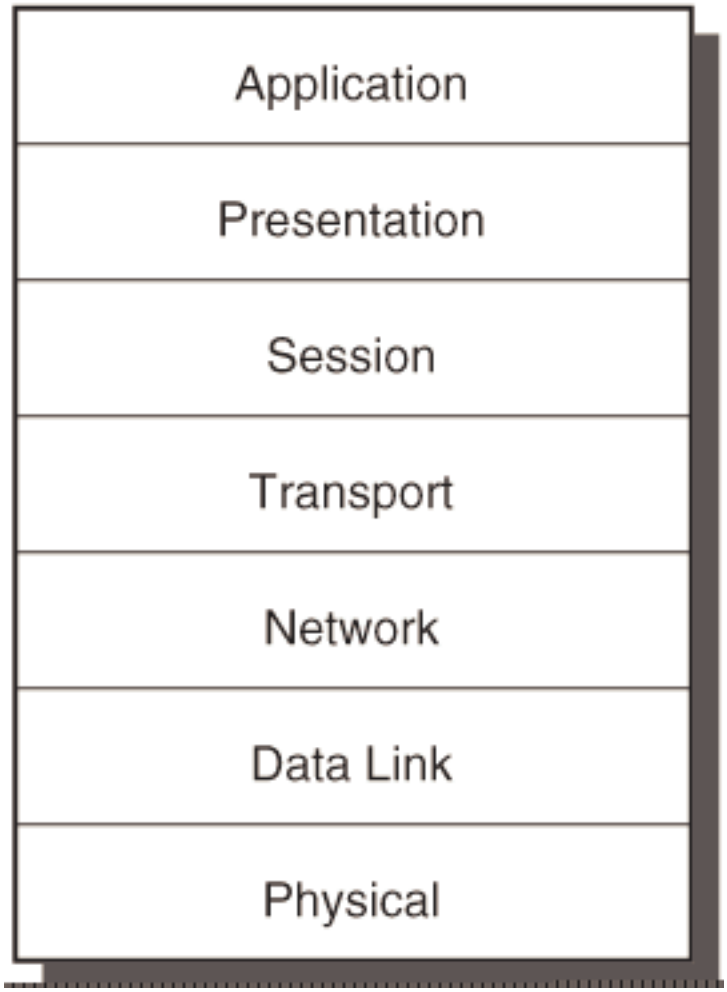
```
while ((rc = t_sndudata(fh, unitdata)) == -1)
{
   if (t_errno != TFLOW)
   {
      t_error("t_snd");
      exit(1);
   }
}
```

Parent Topic: TLI Function Implementation Notes for OSI

OSI Reference Model

To place TLI in perspective, a discussion of the OSI reference model is helpful. The reference model partitions networking functions into seven layers, as shown in the following figure.

Figure 16. Seven-Layer OSI Reference Model



The following table describes the OSI layers.

Table auto. Layers of the OSI Model

OSI Layer	Description
Layer 7	The **application** layer serves as the window between corresponding application processes that are exchanging information.
Layer 6	The **presentation** layer manages the representation of information that application layer entities either communicate or reference in their communication.

Layer 5	The **session** layer provides the services that presentation layer entities need to organize and synchronize their dialog and manage their data exchange.
Layer 4	The **transport** layer provides transparent data transfer services between session layer entities. It relieves those entities from concerns of how to achieve reliable and costeffective data transfer.
Layer 3	The **network** layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layer entities within the network.
Layer 2	The **data link** layer enables the exchange of data between network layer entities. It detects and corrects any errors that can occur in the physical layer transmission.
Layer 1	The **physical** layer is responsible for the transmission of raw data over a communication medium.

A basic principle of the OSI reference model is that each layer provides services needed by the next higher layer, freeing the upper layer from determining how these services are provided. This approach simplifies the design of each layer.

Industry standards have been or are being defined at each layer of the OSI reference model. Two standards are defined at each layer: one that specifies an interface to the services of the layer, and one that defines the protocol by which services are provided. A service interface standard at any layer frees users of the service from details of how that layer's protocol is implemented, or even which protocol provides the service.

The transport layer is important because it provides the basic service of reliable, end-to-end data transfer needed by applications and higher-layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, however, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the following:

ISO protocols

TCP/IP of the ARPAnet

Xerox Network Systems* (XNS*)

Systems Network Architecture* (SNA*)

NetWare IPX/SPX™

In summary, a transport service interface enables applications and higher-layer protocols to be implemented without knowledge of the underlying protocol suite. This is a principal goal of TLI. Also, because the transport layer hides details of the physical medium being used, TLI offers

transport layer hides details of the physical medium being used, TLI offers both protocol and medium independence to networking applications and higher-layer protocols.

TLI is modeled after the industry-standard ISO Transport Service Definition (ISO 8072). As such, it is intended for applications and protocols that require transport services. Because TLI provides reliable data transfer and its services are common to several protocol suites, many networking applications find TLI useful.

NetWare TLI complies with the 1988 XTI specification with 2 minor exceptions: **t_alloc** and UNIX* TLI structure names.

TLI is implemented as a user library using the STREAMS I/O mechanism. Therefore, many services available to STREAMS applications are also available to users of TLI.

Parent Topic: Overview of Transport Protocols

Outgoing and Incoming Events

Outgoing events shown in the following table correspond to the successful return of the specified user-level TLI functions, where these functions send a request or response to the transport provider. Some events in this table (for example, accept) are distinguished by the context in which they occur.

Table auto. Outgoing TLI Events

Outgoing Event	Description	Service Type
connect1	Successful return of **t_connect** in asynchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on **t_connect** in asynchronous mode, or TLOOK error due to a disconnect indication arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	Successful return of **t_accept** with ocnt == 1, fh == resfh	T_COTS, T_COTS_ORD
accept2	Successful return of **t_accept** with ocnt == 1, fh! = resfh	T_COTS, T_COTS_ORD
accept3	Successful return of **t_accept** with ocnt > 1	T_COTS, T_COTS_ORD
snd	Successful return of **t_snd**	T_COTS, T_COTS_ORD
snddis1	Successful return of **t_snddis** with ocnt < = 1	T_COTS, T_COTS_ORD
snddis2	Successful return of **t_snddis** with	T_COTS,

	ocnt > 1	T_COTS_ORD
sndrel	Successful return of **t_sndrel**	T_COTS_ORD
sndudata	Successful return of **t_sndudata**	T_CLTS

The context is based on the values of the following:

ocnt---Number of outstanding connect indications

fh---File handle of the current transport endpoint

resfh---File handle of the transport endpoint where a connection will be accepted

The following table illustrates the allowable sequence of state transitions during connection establishment, data transfer, and release. Given a current state and event, the table illustrates the transition to the next state upon the occurrence of an event.

Table auto. State and Event Combinations

Current State	**Event**	**Next State**
T_IDLE	connect1	T_DATAXFER
T_IDLE	connect2	T_OUTCON
T_IDLE	listen	T_INCON[2]
T_IDLE	pass_conn	T_DATAXFER
T_OUTCON	rcvconnect	T_DATAXFER
T_OUTCON	snddis1	T_IDLE
T_OUTCON	rcvdis1	T_IDLE
T_INCON	listen	T_INCON[2]
T_INCON	accept1	T_DATAXFER[3]
T_INCON	accept2	T_IDLE[3] [4]
T_INCON	accept3	T_INCON[3] [4]
T_INCON	snddis1	T_IDLE[3]
T_INCO	snddis2	T_INCON[3]
T_INCON	rcvdis2	T_IDLE[3]
T_INCON	rcvdis3	T_INCON[3]
T_DATAXFER	snd	T_DATAXFER
T_DATAXFER	rcv	T_DATAXFER
T_DATAXFER	snddis1	T_IDLE
T_DATAXFER	rcvdis1	T_IDLE
T_DATAXFER	sndrel	T_OUTREL
T_DATAXFER	rcvrel	T_INREL
T_OUTREL	rcv	T_OUTREL

T_OUTREL	snddis1	T_IDLE
T_OUTREL	rcvdis1	T_IDLE
T_OUTREL	rcvrel	T_IDLE
T_INREL	snd	T_INREL
T_INREL	snddis1	T_IDLE
T_INREL	rcvdis1	T_IDLE
T_INREL	sndrel	T_IDLE

For example, assume that an endpoint is in T_IDLE state. If a connect1 event occurs, the endpoint enters the T_DATAXFER state. If a snd or rcv event occurs in the T_DATAXFER state, the endpoint remains in the same state. If snddis1 occurs, the endpoint enters the T_IDLE state.

Some of the resulting states in the table include the notation **[n]** as a suffix, where **n** is a number from 1 through 4. These notations indicate the necessary action to be taken by the transport user, as described in the following table.

Table auto. Transport User's Actions

Notation	Transport User's Action
[1]	Set the number of outstanding connect indications to zero
[2]	Decrement the number of outstanding connect indications
[3]	Increment the number of outstanding connect indications
[4]	Pass a connection to another transport endpoint as indicated in **t_accept**

Incoming events, shown in the following table, correspond to the successful return of the specified user-level TLI functions, where these functions retrieve data or event information from the transport provider.

Table auto. Incoming TLI Events

Incoming Event	Description	Service Type
listen	Successful return of **t_listen**	T_COTS, T_COTS_ORD
rcvconnect	Successful return of **t_rcvconnect**	T_COTS, T_COTS_ORD
rcv	Successful return of **t_rcv**	T_COTS, T_COTS_ORD
rcvdis1	Successful return of **t_rcvdis** with ocnt <=0	T_COTS, T_COTS_ORD

rcvdis2	Successful return of **t_rcvdis** with ocnt == 1	T_COTS, T_COTS_ORD
rcvdis3	Successful return of **t_rcvdis** with ocnt > 1	T_COTS, T_COTS_ORD
rcvrel	Successful return of **t_rcvrel**	T_COTS_ORD
rcvudata	Successful return of **t_rcvudata**	T_CLTS
rcvuderr	Successful return of **t_rcvuderr**	T_CLTS
pass_conn	Receive a passed connection	T_COTS, T_COTS_ORD

The only incoming event not associated directly with the return of a function on a given transport endpoint is pass_conn, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint to which the connection is transferred, despite the fact that no function is issued on that endpoint.

In the above table, the rcvdis events are distinguished by the context in which they occur. The context is based on the value of*ocnt*, which is the count of outstanding connect indications on the current transport endpoint.

Parent Topic: TLI Local Management Functions and Events

Outgoing Events

The outgoing events described in the following table correspond to the successful return of the specified TLI functions, where these functions send a request or response to the transport provider.

Event	Description	Service Type
opened	Successful return of **t_open**	T_COTS, T_COTS_ORD, T_CLTS
bind	Successful return of **t_bind**	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	Successful return of **t_optmgmt**	T_COTS, T_COTS_ORD, T_CLTS
unbind	Successful return of **t_unbind**	T_COTS, T_COTS_ORD, T_CLTS
closed	Successful return of **t_close**	T_COTS, T_COTS_ORD,

		T_CLTS
connect1	Successful return of **t_connect** in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on **t_connect** in synchronous mode, or TLOOK error due to a disconnect indication arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	Successful return of **t_accept** with ocnt == 1, fh == resfh	T_COTS, T_COTS_ORD
accept2	Successful return of **t_accept** with ocnt == 1, fh != resfh	T_COTS, T_COTS_ORD
accept3	Successful return of **t_accept** with ocnt > 1	T_COTS, T_COTS_ORD
snd	Successful return of **t_snd**	T_COTS, T_COTS_ORD
snddis1	Successful return of **t_snddis** with ocnt <= 1	T_COTS, T_COTS_ORD
snddis2	Successful return of **t_snddis** with ocnt > 1	T_COTS, T_COTS_ORD
sndrel	Successful return of **t_sndrel**	T_COTS_ORD
sndudata	Successful return of **t_sndudata**	T_CLTS

In the table, some events (such as accept**n**) are distinguished by the context in which they occur. The context is based on the values of the following variables:

ocnt---Count of outstanding connect indications

fh---File handle of the current transport endpoint

resfh---File handle of the transport endpoint where a connection will be accepted

Parent Topic: TLI State Transitions

Overview of Connection-Oriented Service

Connection-oriented transport service consists of four phases of communication:

1. Initialization/deinitialization
2. Connection establishment
3. Data transfer

4. Connection release

Related Topics

Initialization/Deinitialization Phase of Connection-Oriented Service

Connection Establishment Phase

Data Transfer Phase of Connection-Oriented Service

Connection Release Phase

Parent Topic: Overview of Connection-Oriented Service

Parent Topic: Overview of TLI Functions

Overview of Connectionless Service

Connectionless transport service consists of two phases of communication:

1. Initialization/deinitialization
2. Data transfer

Related Topics

Initialization/Deinitialization Phase of Connectionless Service

Data Transfer Phase of Connectionless Service

Parent Topic: Overview of TLI Functions

Overview of TLI Functions

TLI functions provide the services of the transport level. These services provide end-to-end communication, using the services of an underlying network. Your application, which uses TLI functions, is independent of the underlying protocols. By providing media and protocol independence, TLI lets networking applications run in various protocol environments.

This information explains the various phases of communication available through the transport services provided by the transport provider. (Again, a **transport provider** is a transport protocol that provides the services of the transport level.) Two modes of transport service are available: connection-oriented and connectionless.

The SPX™ protocol provides a connection-oriented service, and theIPX™ protocol provides connectionless service. Similarly, TCP provides connection-oriented service, and UDP provides connectionless service. A

single transport endpoint cannot support both modes of service simultaneously.

In TLI environment, an endpoint specifies a communication path between a transport user and a specific transport provider; the endpoint is identified by the local file handle *fh.* A transport endpoint is indicated as an open device, special file. All requests to the transport provider must pass through a transport endpoint.

TLI provides the following library functions:

t_accept

t_alloc

t_bind (Function)

t_blocking

t_close

t_connect

t_error

t_free

t_getinfo

t_getstate

t_listen

t_nonblocking

t_rcvdis

t_rcvrel

t_rcvudata

t_rcvuderr

t_snd

t_snddis

t_sndrel

t_sndudata

t_sync

t_unbind

When using these functions, you must follow certain rules. This overview

shows you how to use these functions and describes their relationships.

Related Topics

TLI Terms

TLI Error Handling

Synchronous and Asynchronous Execution Modes

Overview of Connection-Oriented Service

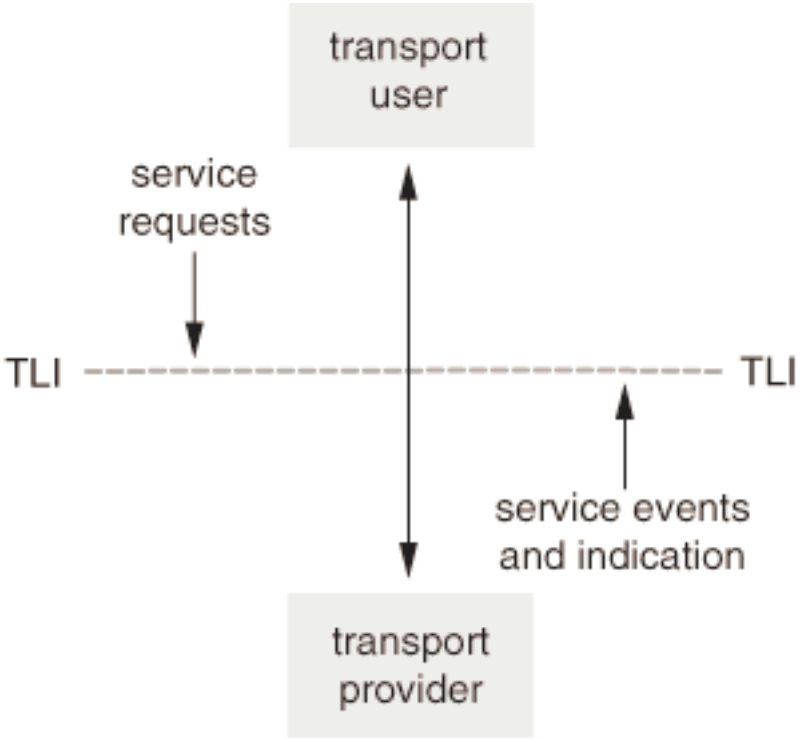
Overview of Connectionless Service

TLI States

Overview of TLI

Here we present a high-level overview of the services of TLI, discussing how it supports the transfer of data between two transport endpoints

Figure 17. TLI Overview



TLI is a transport interface standard designed by USL for the UNIX

environment. Novell has adopted TLI as a standard for applications that need a transport protocol. At the DOS workstation, TLI is implemented as an API library

Advantages in using TLI include the following features:

Multi-protocol support. TLI provides a clean layering between client applications and the implementation issues of network protocols. Applications that use TLI will require only minor modifications to run on top of other protocols, such as TCP/IP, when support for those protocols is made available.

Ease of Use. TLI hides many of the time-consuming details associated with lower-level protocols (such as IPX/SPX). This simplicity makes TLI applications easier to develop and maintain.

Standardization. TLI is an accepted standard within the UNIX environment and defines services that are common to many other networking protocols (such as OSI).

The transport **provider** is the entity that provides the services of TLI, and the transport **user** is the entity that requires these services. An example of a transport provider is the ISO transport protocol, whereas a transport user can be an NLM™ application, a networking application, or a session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

TLI functions support the services of TLI for user processes. These functions enable a user to make requests to the provider and process incoming events.

Related Topics

TLI Local Management Issues

TLI Connectionless Mode Services

TLI State Transitions

TLI Programming Example

Parent Topic: Overview of Transport Protocols

Read

The **read** function can be used to retrieve data that has arrived over the transport connection. The **tirdwr** module passes data to the user from the transport provider. However, any other event or indication passed to the user from the provider is processed by **tirdwr** as follows:

The **read** function cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, **tirdwr** generates a fatal protocol error, EPROTO, on that STREAM. This error causes further system calls to fail. Therefore, when using the read/write interface, you should not communicate with a process that is sending expedited data.

If an abortive disconnect indication is received, tirdwr discards it and generates a STREAMS hangup condition on that STREAM. Subsequent **read** calls retrieve any remaining data, and then **read** returns zero for all further calls (indicating End-of-file (EOF)).

If an orderly release indication is received, **tirdwr** discards the indication and delivers a zero-length STREAMS message to the user. As described in **read**, this notifies the user of EOF by returning 0.

If any other TLI indication is received, **tirdwr** generates a fatal protocol error, EPROTO, on that STREAM. This causes further system calls to fail. If a user pushes **tirdwr** onto a STREAM after the connection has been established, no indication is generated.

Parent Topic: A TLI Read/Write Interface

Releasing a Connection in TLI

There are two ways to release a connection: abortive release and orderly release. With an abortive release, you break the connection without preparing the other endpoint for the release. With an orderly release, both sides signal each other that they are ready to release the connection. An abortive release takes no measures to protect the integrity of any data that the connection might be transmitting.

An orderly release ensures that all data has been processed before the connection is released. TLI requires all transport providers to support an abortive release but not an orderly release. Check the TLI type for support of T_COTS_ORD to determine whether the current version of SPX supports an orderly release.

Related Topics

Orderly Release Summary

Abortive Release Summary

Parent Topic: Connection Mode Service

Rules for Connection-Oriented Transport Service

The following rules apply only to the connection-oriented transport service:

The transport connection release phase can be initiated at any time during the connection establishment phase or data transfer phase.

The only time the state of a transport service interface of a transport endpoint can be transferred to another transport endpoint is when **t_accept** specifies such action. The following rules then apply to the cooperating transport endpoints:

---The endpoint that is to accept the current state of the interface must be bound to an appropriate protocol address and must be in the T_IDLE state.

---The user transferring the current state of an endpoint must have correct permissions for the use of the protocol address and must be in the T_IDLE state.

---The endpoint that transfers the state of TLI is placed into the T_IDLE state by the transport provider after the completion of the transfer if there are no more outstanding connect indications.

Parent Topic: TLI States

Rules for Maintaining the State of TLI

The following are rules for maintaining the state of TLI:

The transport provider is responsible for keeping a record of the state of TLI as seen by the transport user.

The transport provider must never process a function that places TLI out of state.

If the user issues a function out of sequence, the transport provider should indicate this, if possible, through an error return on that function. The state should not change. If any data is passed with the function when not in T_DATAXFER state, the transport provider does not accept or forward that data.

The uninitialized state (T_UNINIT) of a transport endpoint is in the initial state, and the endpoint must be initialized and bound before the transport provider can view it as active.

The uninitialized state is also the final state, and the transport endpoint must be viewed as unused by the transport provider. The **t_close** function closes the transport endpoint and frees TLI library resources for another endpoint.

As shown in Transport Provider States, **t_close** should be called only from the T_UNBND state. If it is called from any other state and no other user has that endpoint open, the action is abortive, the transport endpoint is

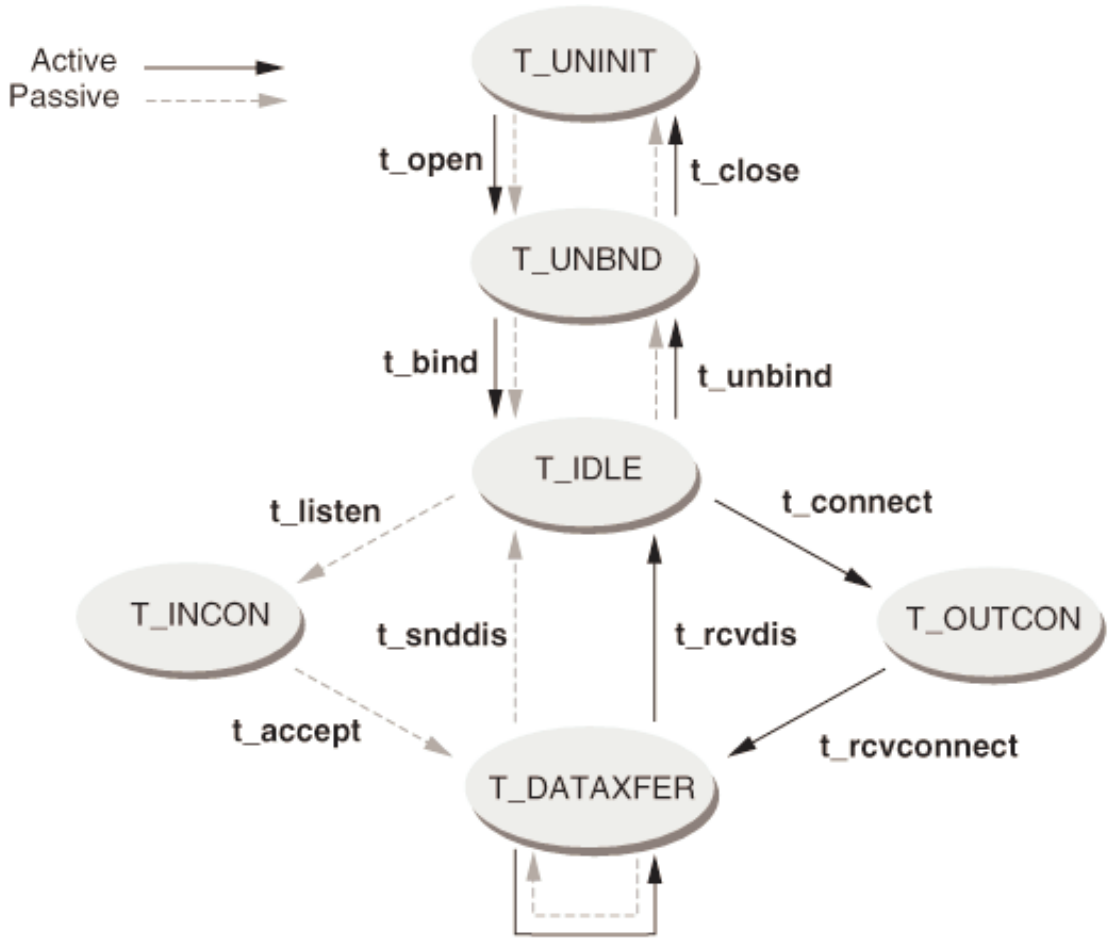
successfully closed, and the library resources are freed for another endpoint. When **t_close** is called, the transport provider must ensure that the address associated with the specified transport endpoint has been unbound from that endpoint. Also, the provider should send appropriate disconnects if **t_close** is not called from the unbound state.

Parent Topic: TLI States

Sequence of TLI Functions

The following figure shows the flow of events through the various states in the connection-oriented service. The broken line represents the passive user, and the solid line represents the active user. This example illustrates the local management as well as the connection establishment, data transfer, and connection release phases, without an orderly release of the connection.

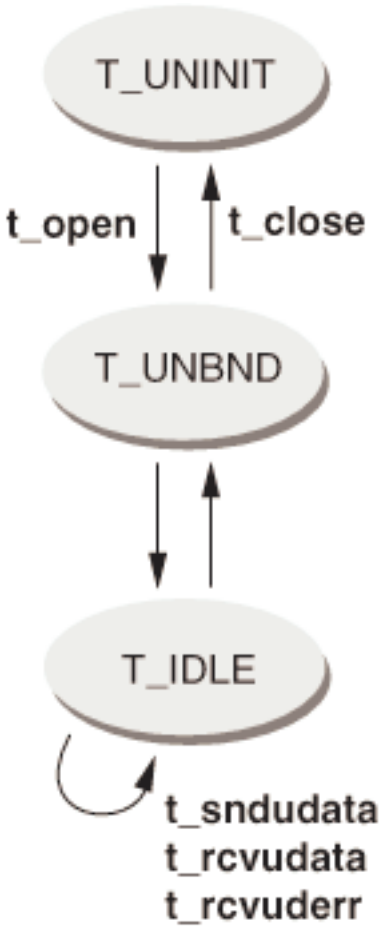
Figure 18. Flow of Events for Connection-Oriented Service



For example, assume that the endpoint is in the T_UNINIT state. If the endpoint is opened with **t_open**, it enters the T_UNBND state. If the user calls **t_bind (Function)**, it enters the T_IDLE state. If the user was bound as a passive user and calls **t_listen**, the endpoint enters the T_INCON state. If an incoming connect indication occurs and the user accepts the connection with **t_accept**, the endpoint enters the T_DATAXFER state. The endpoint remains in the same state if the user calls **t_snd** or **t_rcv**. If the user calls **t_snddis** or if a **t_rcvdis** call is issued and is successful, the endpoint enters the T_IDLE state. If the user calls **t_unbind**, the endpoint enters the T_UNBND state. If the user calls **t_close**, the endpoint is closed and returns to a T_UNINIT state.

The following figure illustrates the flow of events through the various states in a connectionless service.

Figure 19. Flow of Events for Connectionless Service



Parent Topic: TLI States

Setting Options with CLTP

CLTP does not support any of the isocl_options. To remain compatible with the X/Open standard, you can use isocl_options provided that you set all fields to T_UNUSED. Additionally, CLTP supports an extended structure, novell_isocl_options, that includes a *checksum* field. Options negotiated via **t_optmgmt (Function)** affect all packets sent on that STREAM. Options sent with **t_sndudata** override the current option settings for that call only.

Parent Topic: Options Management Structures

Setting Options with TP4

When calling **t_optmgmt (Function)**, **t_connect**, and **t_accept**, you can pass the isoco_options buffer. The following fields are used in TP4:

mngmt.extform---Defines whether TP4 negotiates the use of extended-format sequence numbers

mngmt.checksum---Defines whether TP4 negotiates checksums

expd---Defines whether TP4 negotiates the use of expedited data

All unused fields must be set to T_UNUSED. Calling **t_optmgmt (Function)** to get the defaults sets unused fields to T_UNUSED. Class should be set to T_CLASS4.

Issuing options on a call to **t_accept** has no effect on the options actually negotiated. These options must be negotiated on the listener before receiving a connect indication.

Parent Topic: Options Management Structures

SPX Protocol

SPX provides a connection-oriented transport service and lets data travel over an established connection in a reliable, sequenced manner. You can use SPX II if your applications require message-oriented interactions, meaning you send and receive entire messages. (On the sending side SPX II breaks the message into packets for you and sends the fragments across the wire. At the receiving end, SPX II maintains the message boundary by setting the SPX II end of message bit on the last packet of data associated with that specific message.)

SPX exhibits the following characteristics:

Provides guaranteed delivery by using IPX datagram functions to send packets and receive positive acknowledgments of packet delivery. There is a logical relationship between the packets being exchanged.

After a reasonable number of retransmissions have failed to return a positive acknowledgment, the connection is assumed to have failed.

Related Topics:

NetWare IPX/SPX/SPX II

SPX TLI Multiple Connection Server: Example

SPX TLI Client: Example and SPX TLI Server: Example

SPX II Protocol

SPX II is compatable with the SPX protocol. In addition, it extends the SPX protocol, providing the following additional features:

Uses a sliding window algorithm to minimize packets on the wire and to minimize latency between packets. This decreases the time needed to send a message.

Provides an orderly release of connections so that no data is lost when a connection is released.

Supports end-to-end large data packets, rather than the 534-byte limitation of SPX.

Ensures that the largest packet size is used by allowing packet size negotiation when establishing a connection and packet size renegotation when the delivery route changes.

Allows for receiving of positive and negative acknowledgments of packet delivery.

Communicates directly with the LSL™ facility rather than using IPX datagram functions.

Parent Topic: NetWare IPX/SPX/SPX II

Synchronous and Asynchronous Execution Modes

TLI is inherently asynchronous: various events can occur independent of the actions of a transport user. For example, a user can be sending data over a transport connection when an asynchronous disconnect indication arrives.

The user must somehow be informed that the connection has been broken.

TLI supports two execution modes for handling asynchronous events: synchronous mode and asynchronous mode. In the synchronous mode, the transport functions wait for specific events before returning control to the user. While waiting, the user cannot perform other tasks.

Eight asynchronous events are defined in TLI to cover connection-oriented and connectionless service (see the following table).

Table auto. Asynchronous Events in Connection-Oriented and Connectionless Service

Event	Description
T_LISTEN	Connection-oriented: Occurs when the transport provider receives a connect request from a remote user.
T_CONNECT	Connection-oriented: Occurs when a transport provider receives a connect confirmation.
T_DATA	Connection-oriented: Occurs when a transport provider receives normal data.
T_EXDATA	Connection-oriented: Occurs when a transport provider receives expedited data.
T_DISCONNECT	Connection-oriented: Occurs when a transport provider receives a disconnect indication.
T_UDERR	Connectionless: Occurs when an error is found in a previously sent datagram.
T_ORDREL	Connection-oriented (with orderly release only): Occurs when a provider receives an orderly release indication.
T_ERROR	Both modes: Occurs when the transport provider generates a fatal error, making the transport endpoint inaccessible.
T_GODATA	Connection-oriented: Occurs when it is okay to send normal data again.
T_GOEXDATA	Connection-oriented: Occurs when it is okay to send expedited data again.

Related Topic: Asynchronous Events

Parent Topic: Overview of TLI Functions

TLI Asynchronous Execution Mode

This information presents two important concepts of TLI:

An optional nonblocking (asynchronous) mode for some library functions

An advanced programming example that defines a server supporting multiple outstanding connect indications and operating in an event-driven manner

The example program resides in NWCNLM\SRC\TLI.

Related Topics

Asynchronous Mode for Some TLI Functions

TLI Asynchronous Mode: Advanced Programming Example

TLI Asynchronous Mode: Advanced Programming Example

The following example demonstrates two important capabilities. The first is a server's ability to manage multiple outstanding connect indications. The second is the ability to write event-driven software using TLI and the STREAMS system call interface.

The server example introduced by Connection Mode Service is capable of supporting only one outstanding connect indication, but TLI supports the ability to manage multiple outstanding connect indications.

One reason a server might want to receive several simultaneous connect indications is to impose a priority scheme on each client. A server can retrieve several connect indications and then accept them in an order based on each client's priority.

A second reason for handling several outstanding connect indications is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connect indication, other clients might find it busy. If, however, multiple connect indications can be processed simultaneously, the server is busy only if the maximum number of clients attempt to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming TLI events and then takes the appropriate actions for the current event. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and local management functions needed by this example are similar to those of the server example provided in TLI Connection Mode Services.

TLI Asynchronous Mode---Startup

```
/* Global defines */
#define MAX_CIND 10           /* Max connection indication  */
#define SRV_SOCK 0x4800       /* Server socket (dynamic sock*/
#define TLI_TYPE 0x9000       /* Server type (dynamic area) */
#define LOGFILE  "README.TXT" /* File name                  */

/* Global variables */
int    conn_ind;                /* num connection indicatio*/
struct pollfd pfh;              /* poll structure          */
struct t_bind  *bind;           /* bind structure ptr      */
struct t_call  *calls[MAX_CIND]; /* call structure ptr      */
struct t_discon discon;         /* disconnect structure    */
LONG SAPhandle;                 /* SAP handle              */

/* Function prototypes */
int *AcceptConnection(int i);    /* Assign a new TEP, accept*/
void ProcessConnection(int *);   /* Process new connection  */
void TerminateNLM(void);         /* Cleanup function        */

void TerminateNLM(void)
{
   int i;

   if( SAPhandle )
      ShutdownAdvertising( SAPhandle );
   if( bind )
      t_free((char *)bind, T_BIND);
   for( i=0; i<conn_ind; i++ ) {
      if( calls[i] != NULL ) {
         t_free((char *)calls[i], T_CALL );
         calls[i] = NULL;
      }
   }
   t_unbind( pfh.fd );
   t_close( pfh.fd );
}

main(int argc, char **argv)
{
   int *fh_new;            /* new transport endpoint pointer*/
   BYTE  pv[128];          /* holds property value          */
   BYTE  dc[1];            /* don't care BYTE               */

   if( argc != 3 ) {
      printf("Usage: %s <name> <connections>\n", argv[0]);
      exit(1);
   }
   conn_ind = atoi(argv[2]);
   if( conn_ind > MAX_CIND )
      conn_ind = MAX_CIND;
```

```
   /* Register function that will be executed when NLM
      exits or is unloaded */
   if( atexit( TerminateNLM ) != NULL ) {
      printf("atexit failed");
      exit(1);
   }

   /* Open server's TEP */
   if((pfh.fd = t_open("/dev/nspx", O_RDWR, NULL)) == -1) {
      t_error("t_open");
      exit(1);
   }

   /* Allocate space for structures */
   bind = (struct t_bind *) t_alloc(pfh.fd, T_BIND, T_ALL);
   if( bind == NULL ) {
      t_error("t_alloc");
      exit(1);
   }
   /* Prepare bind structure and then call t_bind */
   bind->qlen = conn_ind;
   bind->addr.len = sizeof( IPX_ADDR );
   addr = (IPX_ADDR *)bind->addr.buf;
   GetInternetAddress( GetConnectionNumber(),
      addr->ipxa_net, addr->ipxa_node );
   *(WORD *)addr->ipxa_socket = IntSwap(SRV_SOCK);

   if( t_bind(pfh.fd, bind, bind) == -1 ) {
      t_error("t_bind");
      exit(1);
   }

/* Check if this is socket you wanted to be bound to */
   if( *(WORD *)&bind->addr.buf[10] != IntSwap(SRV_SOCK) ) {
      printf("Bound wrong address: %d != %d \n",
      *(WORD *)&bind->addr.buf[10], IntSwap(SRV_SOCK) );
      exit(1);
   }

   /* Advertise server on SRV_SOCK socket */
   SAPhandle = AdvertiseService( TLI_TYPE, argv[1],
      IntSwap(SRV_SOCK) );
   if( SAPhandle == NULL ) {
      printf("AdvertiseService failed for: %s\n", argv[1]);
      exit(1);
   }
   printf("Server %s is ready. %d connections.\n",
      argv[1], conn_ind);
```

The file handle returned by **t_open** is stored in a pollfd structure (see **poll**) that polls the transport endpoint for incoming data. Notice that only one transport endpoint is established in this example.

An important aspect of this server is that it sets*qlen* to a value greater than 1 for t_bind. This specifies that the server is willing to handle multiple outstanding connect indications. Remember that the earlier examples single-threaded the connect indications and responses. The server accepts the current connect indication before retrieving additional connect indications. This example, however, can retrieve up to MAX_CIND connect indications at one time before responding to any of them. The transport provider can negotiate the value of *qlen* downward if it cannot support MAX_CIND outstanding connect indications.

Once the server has bound its address and is ready to process incoming connect requests, it does the following:

TLI Asynchronous Mode---Polling for Connect Requests

```
pfh.events = POLLIN;

for (;;)
{
   if( poll( &pfh, 1, -1 ) == -1 )
   {
      perror("poll");
      exit(1);
   }

   if( pfh.revents != POLLIN )
   {
      perror("poll returned error event");
      exit(1);
   }
   if( t_look(pfh.fd) == T_LISTEN )
   {
      for( i=0; i<conn_ind; i++ )
      {
         if( calls[i] == NULL )
            break;
      }
      calls[i]=(struct t_call*)t_alloc(pfh.fd,T_CALL,T_ALL);
      if( calls[i] == NULL )
      {
         t_error("t_alloc");
         exit(1);
      }
      if ( t_listen( pfh.fd, calls[i] ) == -1 )
      {
         t_error("t_listen");
         exit(1);
      }
      if((fh_new = AcceptConnection(i)) != NULL )
         if( BeginThread(ProcessConnection,
                  NULL, NULL, fh_new) == -1 )
         {
            printf("BeginThread failed\n");
```

```
                exit(1);
             }
          }
          else
          {
             t_error("t_look");
             exit(1);
          }
          ThreadSwitch();
       } /* for */
    } /* main */
```

The *events* field of the pollfd structure is set to POLLIN, which notifies the server of any incoming TLI events. The server then enters an infinite loop, in which it polls the transport endpoint for events and then processes those events as they occur.

The **poll** function blocks because the third parameter, a timeout, is set to -1. The call waits for an incoming event. On return, revents is checked for an existing event. If revents is set to POLLIN, **t_look** is called; if **t_look** returns any event other than T_LISTEN, the program exits. If revents is not set to POLLIN, it indicates that an error occurred on the transport endpoint, and the server exits.

The **calls** array contains an entry for each polled transport endpoint, where each entry consists of an array of t_call structures that hold incoming connect indications for that transport endpoint. The array is searched for the next empty slot. Then, struct t_call is allocated. There must always be at least one free entry in the connect indication array; this is because the **calls** array is large enough to hold the maximum number of outstanding connect indications, as negotiated by **t_bind (Function)**.

The **t_listen** function is called to process the incoming connect indication, then the following routine is called to process the connection:

TLI Asynchronous Mode---Processing Connections

```
/* Assign a new TEP, accept and return new TEP on success  */
int *AcceptConnection(int i) {
   int *fhp;             /* new TEP handle ptr */

   fhp = (int *)malloc(sizeof(int));

   /* Open new TEP */
   if((*fhp = t_open("/dev/nspx", O_RDWR, NULL)) == -1) {
      t_error("t_open");
      free(fhp);
      exit(1);
   }

   /* Bind it, address is not important here */
   if( t_bind(*fhp, NULL, NULL) == -1 ) {
      t_error("t_bind");
```

```
            t_close(*fhp);
            free(fhp);
            exit(1);
        }
        /* Accept the call from a client */
        if( t_accept(pfh.fd, *fhp, calls[i]) == -1 ) {
            t_error("t_accept");
            t_close(*fhp);
            free(fhp);
            exit(1);
        }

        /* Check if it was disconnect */
        ThreadSwitch();
        if( t_look(*fhp) == T_DISCONNECT ) {
            printf("DISCONNECT[%d]\n", calls[i]->sequence);
            if( t_rcvdis(*fhp, NULL) < 0 )
            terror("t_rcvdis");
            t_close(*fhp);
            free(fhp);
            t_free((char *)calls[i], T_CALL);
            calls[i] = NULL;
            return NULL;
        }
    /* Success, so free calls[i] slot to be reusable by others */
        printf("ACCEPT[%d]\n", calls[i]->sequence);
        t_free((char *)calls[i], T_CALL);
        calls[i] = NULL;
        return fhp;
    }
```

AcceptConnection is called with the parameter *i*, an index into the **calls** array. The parameter i points to an element of the **calls** array that holds the outstanding connect indication. A t_call structure is allocated for that entry, and the connect indication is retrieved using **t_listen**.

For each indication, the server opens a responding transport endpoint, binds an address to the endpoint, and then accepts the connection on that endpoint. When the connect indication is accepted, its entry in the **calls** array is freed and is set to NULL. **ProcessConnection** is called to process the new connection.

After **t_accept** is called, you need to determine whether the disconnect occurred because this is the earliest you can check for this event.

If a disconnect indication arrives, it must correspond to a previously-received connect indication. This occurs if a client attempts to cancel a previous connect request. In this case, t_discon structure is allocated to retrieve the relevant disconnect information. This structure has the following members:

```
    struct t_discon {
        struct netbuf udata;
```

```
    int reason;
    int sequence;
}
```

The *udata* field identifies any user data that might have been sent with the disconnect indication. The *reason* field contains a protocol-specific disconnect reason code. The *sequence* field identifies the outstanding connect indication that matches this disconnect indication.

The disconnect indication is processed by a call to **t_rcvdis**, the transport endpoint is closed, and the corresponding entry of the calls array is freed.

As mentioned earlier, **ProcessConnection** is called as follows:

TLI Asynchronous Mode---ProcessConnection

```
void ProcessConnection(int *fh) {
   FILE *fp;               /* file pointer      */
   char iobuf[132];        /* line buffer       */

   printf("Client %d\n", *fh);
   if( (fp = fopen( LOGFILE, "r" )) == NULL )
   {
      printf("Couldn't open file: %s\r\n", LOGFILE);
      free(fh);
      return;
   }

   /* Read lines from a file and send to the client */
   while( fgets( iobuf, sizeof(iobuf), fp ) != NULL )
   {
      delay(100);    /* timeout and thread switch */
      if( t_look(*fh) == T_DISCONNECT )
      {
         printf("Client %d received disconnect, EXIT.\n", *fh);
         fclose(fp);
         t_close(*fh);
         free(fh);
         return;
      }
      if( t_snd( *fh, iobuf, sizeof(iobuf), NULL) == -1 )
      {
         t_error("t_snd");
         fclose(fp);
         t_close(*fh);
         free(fh);
         return;
      }
   }
   fclose(fp);

   /* Send disconnect to the client */
   if( t_snddis(*fh, NULL) == -1 )
```

```
    {
       t_error("t_snddis");
       t_close(*fh);
       free(fh);
       return;
    }

/* Unbind TEP */
    if( t_unbind(*fh) == -1 )
    {
       t_error("t_unbind");
       t_close(*fh);
       free(fh);
       return;
    }

    t_close(*fh);
    free(fh);
}
```

The server calls the **BeginThread** routine to begin a new thread. Once this occurs, the **ProcessConnection** function is executed on the new thread. The **ProcessConnection** function is called to manage the data transfer, as shown in TLI Connection Mode Services.

Parent Topic: TLI Asynchronous Execution Mode

TLI Connection-Oriented Functions

Use the following functions to establish a connection, transfer data, and release the connection.

Function	Header	Comment
t_accept	tiuser.h	Accepts a connection request from a calling node.
t_connect	tiuser.h	Requests a connection with the specified listening node.
t_listen	tiuser.h	Listens for a connection request from a calling node.
t_rcv	tiuser.h	Receives either normal or expedited data sent from another endpoint with **t_snd**.
t_rcvconnect	tiuser.h	Determines the status of a previously sent connection request.
t_rcvdis	tiuser.h	Identifies another endpoint's reason for disconnecting and retrieves user data sent along with the disconnect.

t_rcvrel	tiuser.h	Acknowledges receipt of an orderly release (not supported by SPX).
t_snd	tiuser.h	Sends normal or expedited data to another endpoint in connection mode.
t_snddis	tiuser.h	Initiates an abortive release on a connection.
t_sndrel	tiuser.h	Initiates an orderly release on a connection (not supported by SPX).
t_sync	tiuser.h	Synchronizes the TLI data structures with information from the transport provider. (Under SPX, **t_sync** returns only the current state of the transport provider.)

Parent Topic: Overview of Transport Protocols

TLI Connectionless Function List

Connectionless Services include functions for sending and receiving data without the support of a connection.

Function	Header	Comment
t_rcvudata	tiuser.h	Receives data sent from another endpoint by **t_sndudata**.
t_rcvuderr	tiuser.h	Receives information concerning an error of data sent previously from another endpoint with **t_rcvudata**.
t_sndudata	tiuser.h	Sends data to another endpoint in connectionless mode.

TLI Data Management

TLI defines the following structure types to handle parameters for TLI operations:

t_bind (Structure)
t_call
t_optmgmt (Structure)
t_discon
t_unitdata
t_uderr

t_info

netbuf is a generic type for buffering transport data.

netbuf is one of the keys to TLI's multiprotocol support. Most TLI structure types contain pointers netbuf. For example, input to **t_bind (Function)** includes t_bind (Structure) containing an address a local endpoint will be bound to.

Nesting netbuf within t_bind (Structure) allows the address to be expressed in whatever format the local transport provider requires.netbuf helps separate TLI from the protocol-specific issues related to addressing data.

Parent Topic: TLI Local Management Issues

TLI Error Handling

Two levels of errors are defined in TLI:

Library errors

OS service routine errors

On the library error level, each library function has one or more error return values. Failures are indicated by a return value of -1. An external integer, *t_errno*, holds the specific error number when such a failure occurs. This value is set when an error occurs but is not cleared on successful library calls, so it should be tested only after an error has been indicated.

A diagnostic function, **t_error**, prints information about the current transport error. The state of the transport provider can change if a transport error occurs.

The second level of error is the OS service routine level. TSYSERR is a special library-level error number that is generated by each library function when an OS service routine fails or some general error occurs. When a function sets t_errno to TSYSERR, it stores a specific error code in the external variable errno.

When a protocol error occurs, the transport provider generates the system error EPROTO. If the error is severe, it can cause the file handle and transport endpoint to be unusable. To continue, all users of the file must close it and then reopen and initialize it.

Parent Topic: Overview of TLI Functions

TLI Function Implementation Notes for IPX/SPX/SPX II

The implementation notes in this section provide information specific to the use of TLI functions for accessing the IPX, SPX, and SPX II transport protocols in the NetWare environment.

Some of the TLI functions are implemented with IPX, SPX and SPX II. These functions are listed in the following table.

Table auto. TLI Functions Implemented for IPX, SPX and SPX II

t_alloc	**t_event**	**t_open**
t_bind	**t_free**	**t_optmgmt**
t_blocking	**t_getinfo**	**t_sync**
t_close	**t_getstate**	**t_unbind**
t_error	**t_look**	**t_nonblocking**

Some of the TLI functions are for connectionless, or datagram services only. The NetWare implementation of TLI used IPX to implement these functions. These connectionless functions are listed in the following table.

Table auto. TLI Functions Implemented for IPX

t_rcvudata	**t_rcvuderr**	**t_sndudata**

Some of the TLI functions are for connection-oriented services. The NetWare implementation of TLI uses SPX and SPX II, but not IPX, to implement these functions. These connection-oriented functions are listed in the following table.

Table auto. TLI Functions Implemented for SPX and SPX II

t_accept	**t_rcvconnect**	**t_snddis**
t_connect	**t_rcvdis**	**t_sndrel**
t_listen	**t_rcvrel**	
t_rcv	**t_snd**	

The following sections discuss the implementation specific details of the TLI functions for IPX, SPX, and SPX II. These implementation notes supplement the function descriptions in IPX: Functions and SPX: Functions.

In each of the following function descriptions, the protocols that use the functions are listed, within parenthesis, at the first of the description. For example, IPX, SPX, and SPX II use **t_alloc**.

See the following topics for notes about each TLI function:

IPX/SPX/SPX II Notes: t_accept

IPX/SPX/SPX II Notes: t_alloc

IPX/SPX/SPX II Notes: t_bind

IPX/SPX/SPX II Notes: t_blocking

IPX/SPX/SPX II Notes: t_close

IPX/SPX/SPX II Notes: t_connect

t_error---no implementation-specific details

t_event---no implementation-specific details

t_free---no implementation-specific details

IPX/SPX/SPX II Notes: t_getinfo

t_getstate---no implementation-specific details

IPX/SPX/SPX II Notes: t_listen

t_look---no implementation-specific details

IPX/SPX/SPX II Notes: t_nonblocking

IPX/SPX/SPX II Notes: t_open

t_optmgmt---no implementation-specific details

IPX/SPX/SPX II Notes: t_rcv

IPX/SPX/SPX II Notes: t_rcvconnect

IPX/SPX/SPX II Notes: t_rcvdis

IPX/SPX/SPX II Notes: t_rcvrel

IPX/SPX/SPX II Notes: t_rcvudata

IPX/SPX/SPX II Notes: t_rcvuderr

IPX/SPX/SPX II Notes: t_snd

IPX/SPX/SPX II Notes: t_snddis

IPX/SPX/SPX II Notes: t_sndrel

IPX/SPX/SPX II Notes: t_sndudata

t_sync---no implementation-specific details

t_unbind---no implementation-specific details

Parent Topic: NetWare IPX/SPX/SPX II

TLI Function Implementation Notes for OSI

The implementation notes provide information specific to the use of TLI functions for accessing the OSI transport protocol in NetWare 3.11 and above. These implementation notes supplement the complete function descriptions in TLI: Functions. See the following topics:

OSI Notes: t_accept

OSI Notes: t_bind

OSI Notes: t_close

OSI Notes: t_connect

OSI Notes: t_getinfo

OSI Notes: t_listen

OSI Notes: t_open

t_optmgmt---see Options Management Structures

OSI Notes: t_rcv

OSI Notes: t_rcvconnect

OSI Notes: t_rcvdis

t_rcvrel---TP4 does not support orderly release of a connection.

OSI Notes: t_rcvudata

OSI Notes: t_snd

OSI Notes: t_snddis

t_sndrel---TP4 does not support orderly release of a connection.

OSI Notes: t_sndudata

Parent Topic: NetWare OSI

TLI Local Management Function List

Use the following functions to set up and tear down endpoints and to

maintain the local TLI environment:

Function	Header	Comment
t_alloc	tiuser.h	Dynamically allocates memory for TLI structure parameters.
t_bind (Function)	tiuser.h	Activates an endpoint and associates a protocol address with it.
t_blocking	tiuser.h	Puts an endpoint into blocking mode. (This function is NetWare specific.)
t_nonblockin g	tiuser.h	Puts an endpoint into nonblocking mode. (This function is NetWare specific.)
t_close	tiuser.h	Closes an endpoint and frees any associated local resources.
t_error	tiuser.h	Produces a message on the standard error output describing the most recent TLI error.
t_free	tiuser.h	Frees memory previously allocated by **t_alloc**.
t_getinfo	tiuser.h	Returns the current characteristics of the underlying transport provider for the specified endpoint.
t_getstate	tiuser.h	Returns the current state of the transport provider for the specified endpoint.
t_look	tiuser.h	Returns the current event on the specified endpoint.
t_open	tiuser.h	Opens and initializes an endpoint.
t_optmgmt (Function)	tiuser.h	Retrieves, verifies, and negotiates protocol options with the transport provider.
t_unbind	tiuser.h	Disables the specified endpoint previously bound by **t_bind**.

Parent Topic: Overview of Transport Protocols

TLI Local Management Functions and Events

The local management functions help in managing the endpoint locally and do not involve the transmission of information across the network. The following table lists the local management functions and the event that occurs for each upon successful return. Each function supports all three service types: connectionless, connection-oriented, or connection-oriented with orderly release.

Table auto. Local Management Functions and Events

Function	Event
t_open	opened
t_bind	bind
t_optmgmt	optmgmt
t_unbind	unbind
t_close	closed

The following table illustrates the state transitions that can occur during the initialization/de-initialization phase. Given a particular state of the endpoint and a particular event, the table shows that endpoint's next state. For example, in the same table, assume that the endpoint is in T_UNINIT state. If the endpoint is opened, the endpoint enters the T_UNBND state. If the endpoint is bound, it enters the T_IDLE state. Some of the resulting states include the notation **[n]** as a suffix, where **n** is a number from 1 through 4. These notations indicate the necessary action to be taken by the transport user, as described in next table.

This table also illustrates the state transitions that are not allowed. For example, the endpoint cannot be opened during the T_UNBND state. Therefore, the corresponding entry for opened /T_UNBND is marked Invalid in the table.

Table auto. Initialization/Deinitialization Tasks and State Transitions

Event	T_UNINIT State	T_UNBND State	T_IDLE State
opened	T_UNBND	Invalid	Invalid
bind	Invalid	T_IDLE[1]	Invalid
optmgmt	Invalid	Invalid	T_IDLE
unbind	Invalid	Invalid	T_UNBND
closed	Invalid	T_UNINIT	Invalid

Related Topic: Outgoing and Incoming Events

Parent Topic: TLI States

TLI Local Management Issues

Local management issues include the following:

TLI Data Management

TLI Memory Allocation

Blocking and Nonblocking Modes

Error Handling in TLI

Endpoint Management in TLI

Parent Topic: Overview of Transport Protocols

TLI Memory Allocation

TLI includes **t_alloc** and **t_free** for dynamically allocating and releasing space for TLI structure variables. TLI does not require you to call these functions, but they can reduce the amount of work it takes to allocate and initialize TLI's complex structure variables.

Returning to the t_bind (Structure) example, you must allocate sufficient space to store the address referenced by *addr.buf*. The amount of space required to store the address depends on the local transport provider. By using **t_alloc** to dynamically allocate t_bind and other TLI structures, you can assure such allocations are managed correctly.

Call **t_alloc** to allocate a structure. As input, **t_alloc** takes the endpoint, the structure type, and *fields*.

The returned structure can be used as a parameter to any number of TLI functions. The size of all buffers related to TLI structure parameters is based on the endpoint's transport information. **t_alloc** can allocate none, some, or all of the associated buffers.

Parent Topic: TLI Local Management Issues

TLI Programming Example

This section looks at a simple example of how to use TLI to exchange data between DOS workstations. The example consists of two programs: a DOS server, dserver.c, and a DOS client, dclient.c. The dserver.c program opens an endpoint and listens for a connection. The dclient.c program contacts the server side and asks for the connection. Once the connection is established, the programs pass a message back and forth and then close the connection.

dserver.c

dclient.c

Parent Topic: Overview of Transport Protocols

TLI State Transitioning

Obviously the sequence of TLI operations is very important. You can't receive packets on a connection until the connection is established. You can't establish a connection without opening and binding an endpoint. The relationship between TLI operations can be viewed as a set of finite states, that is to say, there is a finite set of states that a TLI program can assume.

Finite states are a conceptual tool that can help you understand the relationships among TLI functions. At any given moment a TLI program is in a particular state that defines the actions the program must or can perform. Events move the program from one state to another. The following table lists the possible TLI states.

Table auto. Table of TLI States

State	Comment
T_UNINIT	T_UNINIT is the uninitialized state of a TLI program. At T_UNINIT the program hasn't initialized or has completed all TLI operations. This is the program's initial and final state.
T_UNBND	At T_UNBND the endpoint is open but not yet bound. The open operation moves the program from T_UNINIT to T_UNBND. Closing the endpoint moves the program back to T_UNINIT.
T_IDLE	At T_IDLE the program has bound the endpoint to an address. A connection mode program can move to T_INCON or T_OUTCON. A connectionless program can move to T_DATAXFER by sending or receiving data. Unbinding the endpoint moves the program back to T_UNBND.
T_OUTCON	T_OUTCON is a connection mode state. At T_OUTCON the program is attempting to establish a connection but hasn't received confirmation from the other side. From T_OUTCON the program can move to T_DATAXFER by receiving a connection confirmation. Receiving a refusal from the other side can move the program back to T_IDLE.
T_INCON	T_INCON is a connection mode state. At T_INCON the program is listening for a connection but hasn't yet received the connection request from the other side. From T_ICON the program can move to T_DATAXFER by receiving and accepting the connection request. Refusing the connection can move the program back to T_IDLE.

T_DATAXFER	At T_DATAXFER the program is sending or receiving data. A connection mode program can move back to T_OUTREL by requesting an orderly release and to T_INREL by receiving a request for an orderly release. It can move to T_IDLE by sending or receiving an abortive release. A connectionless program returns to T_IDLE when it has completed sending or receiving.
T_OUTREL	T_OUTREL is a connection mode state. At T_OUTREL the program has requested an orderly release and is awaiting a response. Receiving the response moves the program back to T_IDLE. T_OUTREL isn't supported by SPX.
T_INREL	T_INREL is a connection mode state. At T_INREL the program has received a request for an orderly release. Processing the request moves the program back to T_IDLE. T_INREL isn't support by SPX.

Parent Topic: Overview of Transport Protocols

TLI States

The following table defines the states used to describe TLI state transitions.

Table auto. States Describing TLI State Transitions

State	Description	Service Type
T_UNINIT	Uninitialized initial and final state of interface	T_COTS, T_COTS_ORD , T_CLTS
T_UNBND	Initialized but not bound	T_COTS, T_COTS_ORD , T_CLTS
T_IDLE	No connection established	T_COTS, T_COTS_ORD , T_CLTS
T_OUTCON	Outgoing connection pending for client	T_COTS, T_COTS_ORD
T_INCON	Incoming connection pending for server	T_COTS, T_COTS_ORD
T_DATAXFER	Data transfer	T_COTS, T_COTS_ORD
T_OUTREL	Outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	Incoming orderly release (waiting to	T_COTS_ORD

	send orderly release request)	

Parent Topic: TLI State Transitions

TLI State Tables

The state tables describe TLI state transitions. Given a current state and an event, the transition to the next state is shown, as well as any actions that the user must take (indicated by [**n**]). The state is that of the transport provider as seen by the user.

The contents of each box represent the next state, given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state and event combination that is invalid. Along with the next state, each box may include an action list (as specified in the previous section). The transport user must take the specific actions in the order specified in the state table.

Keep in mind the following when studying state tables:

The **t_close** routine is referenced in the state tables (see closed event in the Outgoing Events table) but can be called from any state to close a transport endpoint. If **t_close** is called when a transport address is bound to an endpoint, the address is unbound. Also, if **t_close** is called when the transport connection is still active, the connection is aborted.

If a transport user issues a routine out of sequence, the transport provider recognizes this and the routine fails, setting t_errno to TOUTSTATE. The state does not change.

If any other transport error occurs, the state does not change unless otherwise noted. The exception to this is a TLOOK or TNODATA error on **t_connect**, as described in the Incoming Events table. The state tables assume correct use of TLI.

The state tables exclude the following support routines because they do not affect the state: **t_getinfo**, **t_getstate**, **t_alloc**, **t_free**, **t_sync**, **t_look**, and **t_error**.

The following state tables are provided.

Table auto. Common Local Management State Table

Event	T_UNINIT State	T_UNBND State	T_IDLE State
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt			T_IDLE

unbind			T_UNBND
closed		T_UNINIT	

Table auto. Connectionless-Mode State Table

Event	T_IDLE State
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table auto. Connection-Mode State Table

Current State	Event	Next State
T_IDLE	connect1	T_DATAXFER
T_IDLE	connect2	T_OUTCON
T_IDLE	listen	T_INCON[2]
T_IDLE	pass_conn	T_DATAXFER
T_OUTCON	rcvconnect	T_DATAXFER
T_OUTCON	snddis1	T_IDLE
T_OUTCON	rcvdis1	T_IDLE
T_INCON	listen	T_INCON[2]
T_INCON	accept1	T_DATAXFER[3]
T_INCON	accept2	T_IDLE[3] [4]
T_INCON	accept3	T_INCON[3] [4]
T_INCON	snddis1	T_IDLE[3]
T_INCON	snddis2	T_INCON[3]
T_INCON	rcvdis2	T_IDLE[3]
T_INCON	rcvdis3	T_INCON[3]
T_DATAXFER	snd	T_DATAXFER
T_DATAXFER	rcv	T_DATAXFER
T_DATAXFER	snddis1	T_IDLE
T_DATAXFER	rcvdis1	T_IDLE
T_DATAXFER	sndrel	T_OUTREL

T_DATAXFER	rcvrel	T_INREL
T_OUTREL	rcv	T_OUTREL
T_OUTREL	snddis1	T_IDLE
T_OUTREL	rcvdis1	T_IDLE
T_OUTREL	rcvrel	T_IDLE
T_INREL	snd	T_INREL
T_INREL	snddis1	T_IDLE
T_INREL	rcvdis1	T_IDLE
T_INREL	sndrel	T_IDLE

Parent Topic: Transport User Actions

TLI Terms

The following defines terms that are frequently used in the TLI environment:

active transport user

The transport user that initiates a connection.

asynchronous execution

The mode of execution in which transport service functions do not wait for specific asynchronous events to occur before returning control to the user, but instead return immediately if the event is not pending.

ETSDU

Expedited Transport Service Data Unit. The expedited data that is transmitted over a transport connection. Its identity is preserved from one end of a transport connection to the other.

passive transport user

The transport user that listens for an incoming connect indication.

protocol address

The address, also known as the Transport Service Access Point (TSAP) address, that identifies the transport user. This interface places no particular structure or semantics on an address.

synchronous execution

The mode of execution in which transport service functions wait for specific asynchronous events to occur before returning control to the user.

transport endpoint

The communication path identified by a file handle, between a

transport user and a specific transport provider.

transport provider

The transport protocol that provides the services of the TLI.

transport user

The user-level application or protocol that is accessing the services of TLI.

TSDU

Transport Service Data Unit. The user data that is transmitted over a transport connection. Its identity is preserved from one end of a transport connection to the other.

Parent Topic: Overview of TLI Functions

Transferring Data in TLI

Once you have established a connection, you send and receive data with **t_snd** and **t_rcv**. Both functions can execute in synchronous or asynchronous mode according to the configuration of the endpoint.

Receiving Data

Call **t_rcv** to receive data. As input, this function takes the endpoint, a buffer for receiving data, the buffer size, and space to receive control flags.**t_rcv** returns the number of bytes received:

```
int t_rcv(int fh, char *buf, unsigned nbytes, int
*flags);
```

If the data has been fragmented, the T_MORE flag is set. You should continue calling **t_rcv** until the flag is cleared.

A typical way to set up a data transfer is for the sending side to loop on **t_snd**, checking for an error value that signals a disconnection:

```
while ((bytes=t_snd(fh, buf, sizeof(buf), 0)) == -1) {
  if (t_errno == TLOOK && t_look(fh) == T_DISCONNECT)
    /* disconnect */
    .
    .
}
```

At the same time, the receiving side loops on**t_rcv**, also checking for a disconnection. Either function will return a TLOOK value if the other side closes or loses the connection:

```
while ((bytes=t_rcv(fh, buf, sizeof(buf), &flags)) == -1) {
  if (t_errno == TLOOK && t_look(fh) == T_DISCONNECT)
```

```
    if (t_errno == TLOOK && t_look(fh) == T_DISCONNECT)
      /* disconnect */
      .
      .
    else
      /* move the data out of the buffer */
      if (fwrite(buf, 1, sizeof(buf), stdout) < 0)
        exit(1);
}
```

Sending Data

Call **t_snd** to send data. As input, this function takes the endpoint, a buffer containing the data to transfer, the buffer size, and any control flags. **t_snd** returns the number of bytes sent successfully:

```
int t_snd(int fh, char *buf, unsigned nbytes, int
flags);
```

If the amount of data is too large for the buffer you have defined, you can set the T_MORE control in *flags*. Setting T_MORE tells the transport provider you need to make several calls to **t_snd** to complete the send operation. Be sure to clear T_MORE the last time you call **t_snd**.

t_snd itself can set the T_MORE flag if it returns before it is able to send the number of bytes you specify. This would only happen if you had opened the endpoint in nonblocking mode.

Parent Topic:

Connection Mode Service

Transport Provider States

The transport provider exists in a particular state at any given time. For example, when a transport provider is not yet initialized, it is in a T_UNINIT state. Once it is initialized, the provider is in a T_UNBND state. The following table, Transport Provider States, shows all the possible states of the transport provider as seen by the transport user. The service type can be connectionless, connection-oriented, or connection-oriented with orderly release.

State	Description	Service Type
T_UNINIT	Uninitialized. This is the initial and final state of TLI.	T_COTS, T_CLTS, T_COTS_ORD
T_UNBND	Unbound.	T_COTS,

		T_CLTS, T_COTS_ORD
T_IDLE	No connection established. The sndudata, rcvudata, and rcvuderr events cause the endpoint to remain in the T_IDLE state.)	T_COTS, T_CLTS, T_COTS_ORD
T_OUTCON	Outgoing connection pending for the active user.	T_COTS T_COTS_ORD
T_INCON	Incoming connection pending for the passive user.	T_COTS, T_COTS_ORD
T_DATAXFER	Data transfer.	T_COTS, T_COTS_ORD
T_OUTREL	Outgoing orderly release (waiting for orderly release indication).	T_COTS_ORD
T_INREL	Incoming orderly release (waiting to send an orderly release request).	T_COTS_ORD

Related Topics

TLI Local Management Functions and Events

Sequence of TLI Functions

Rules for Maintaining the State of TLI

Rules for Connection-Oriented Transport Service

Guidelines for Writing Protocol-Independent Software

Parent Topic: Overview of TLI Functions

Transport User Actions

In the state tables (see TLI State Tables), some state transitions are accompanied by a list of actions the transport user must take. The notation [**n**] represents these actions, where **n** is the number of the specific action, as listed in the following table.

Table auto. List of Actions for Values of n

Value of n	**Corresponding Action**
1	Set the count of outstanding connect indications to zero
2	Increment the count of outstanding connect indications
3	Decrement the count of outstanding connect indications

4	Pass a connection to another transport endpoint as indicated in **t_accept**

Parent Topic: TLI State Transitions

WinSock 2

WinSock 2 is replacing TLI as the transport of choice since WinSock 2 supports Internet Protocols (IP).

See the http://www.stardust.com URL for more information.

Write

The user can transmit data over the transport connection using **write**. The **tirdwr** module passes data to the transport provider. However, if a user attempts to send a zero-length data packet (which the STREAMS mechanism allows), **tirdwr** discards the message.

If the transport connection is aborted (for example, because the remote user aborts the connection using **t_snddis**), a STREAMS hangup condition is generated on that STREAM, and further **write** calls fail and set *errno* to ENXIO. The user can still retrieve any available data after a hangup.

Parent Topic: A TLI Read/Write Interface

TLI: Functions

poll

Monitors input and output on a set of file descriptors that reference open transport endpoints to allow multiplexing on those endpoints

Local Servers: blocking

Remote Servers: N/A

Platforms: DOS, NLM™, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <poll.h>

int poll (
   struct pollfd   fds[ ],
   unsigned long   nfds,
   int             timeout);
```

Parameters

fds

(OUT) Points to an array of file descriptors and events to be polled.

nfds

(IN) Specifies the number of file descriptors to be polled.

timeout

(IN) Specifies the number of milliseconds **poll** should wait for an event if no events are pending (-1 specifies a wait forever).

Return Values

Upon successful completion, **poll** returns a non-negative value indicating the number of file descriptors that have been selected. A value of 0 indicates that the call timed out and that no file discriptors have been selected. Upon failure, **poll** returns -1 and sets *errno* to indicate the error (see remarks).

Remarks

NOTE: While **poll** is not defined in the TLI standard, the use of **poll** is common in TLI applications. Therefore, the documentation for **poll** is grouped with TLI Services as a matter of convenience for the developer.

NLM applications can also use **poll** with NetWare STREAMS Services.

Upon failure, **poll** sets *errno* to one of the following values:

EAGAIN	The allocation of internal data structures failed, but the request should be attempted again.
EINTR	A signal was caught during the **poll** system call.
EINVAL	The argument *nfds* is less than zero or greater than [OPEN_MAX].

poll examines each file descriptor for the requested event, and on return, indicates which events have occurred for each file descriptor.

poll identifies those transport endpoints over which the user can send or receive data, or on which certain events have occurred. For each endpoint of interest, the application fills out a pollfd structure specifying the file descriptor to examine and the events to monitor.

The pollfd structure is defined in POLL.H

An application can specify which events to query by setting the bits in the *events* field of the pollfd structure that is associated with the file descriptor. *events* is a bitmask that is set to the bitwise inclusive OR of events to be monitored on the associated file descriptor. This bitmap can include the following events:

POLLIN	Is input data is available on the transport endpoint associated with the given file descriptor?
POLLPRI	Is a priority message is available on the transport endpoint associated with the given file descriptor?
POLLOUT	Is the transport endpoint associated with the given file is writable? That is, has the endpoint relieved the flow control that would prevent an application from sending data over that endpoint?

When **poll** is called, it examines each element in the *fds* array. For each element in the array, it examines the file descriptor, specified by *fd*, for the events specified in *events* field. It then sets *revents* to indicate which of the requested events has occurred. (If *fd* is set to a value less than zero, **poll** ignores *events* and sets *revents* to zero.)

If **poll** succeeds, the calling application examines the *revents* field of each element in the *fds* array. If *revents* is set to 0, no event has occurred on that file descriptor. If *revents* is set to a value other than 0, an event has occurred on that file descriptor. *revents* can also contain notification of error conditions. The following values are valid for *revents*:

POLLIN	Input data is available on the transport endpoint associated with the given file descriptor.

POLLPRI	A priority message is available on the transport endpoint associated with the given file descriptor.
POLLOUT	The transport endpoint associated with the given file is writable. That is, the endpoint has relieved the flow control that would prevent a user from sending data over that endpoint.
POLLERR	A fatal error has occurred in some module or driver on the transport endpoint associated with the specified file descriptor. Further function calls will fail.
POLLHUP	A hangup condition exists on the transport endpoint associated with the specified file descriptor.
POLLNVAL	The specified file descriptor is not associated with an open transport endpoint.

An application uses the value of *revents* to determine what actions it can take on the associated file descriptor. For example, if *revents* is set to POLLIN, incoming data is available and can be read from the device associated with the file descriptor.

Besides identifying the events that have occurred on a file descriptor, the *revents* field can also specify errors that have occurred on the file descriptor. For example, if POLLIN is requested in *events*, and **poll** sets the associated *revents* field to a value other than 0 or POLLIN, an error event must have occurred on the associated transport endpoint, because the only requested event was POLLIN.

The POLLERR, POLLHUP, and POLLNVAL events cannot be polled for by placing them in the *events* field before calling **poll**. Instead, these events are reported in *revents* whenever they occur.

If no event has occurred on any of the polled file descriptors, the behavior of **poll** depends upon the value of the *timeout* parameter. If *timeout* is a positive value, **poll** waits at least that many milliseconds before returning. If *timeout* is 0, **poll** returns immediately after examining the file descriptors. If *timeout* is -1, **poll** blocks until an event has occurred on at least one of the specified file descriptors.

poll is not affected by the O_NDELAY and O_NONBLOCK flags.

See Also

getmsg, **putmsg**

t_accept

Accepts a connect request
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_accept(
   int               fd,
   int               resfd,
   struct t_call   *call);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the connect indication arrived.

resfd

(OUT) Indicates the local transport endpoint where connection is to be established.

call

(IN/OUT) Points to the information required by the transport provider to complete the connection.

Return Values

	Successful
-1	Failure (sets *t_errno* Code)

t_errno Values

Upon successful completion, **t_accept** returns a value of 0. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport

	endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived.
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in T_IDLE.
TACCESS	User cannot access specified option.
TBADOPT	Specified protocol options were in an incorrect format or contained illegal information.
TBADDATA	Amount of user data specified was not within the bounds allowed by the transport provider.
TBADADDR	Specified protocol address was in an incorrect format or contained illegal information.
TBADSEQ	Invalid sequence number was specified.
TLOOK	Asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate. attention
TNOTSUPPORT	Underlying transport provider does not support this function.
TSYSERR	System error occurred during execution of this function.

Remarks

t_accept is issued by a transport user to accept a connect request.

Blocking Information**t_accept** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

call points to a t_call structure, which is defined in TIUSER.H.

A transport user can accept a connection on the same local transport endpoint as the one on which the connect indication arrived, or on a different one.

If the same endpoint is specified (that is, resfd = fd), the connection can be accepted unless the user has received other indications on that endpoint but has not responded to them (with **t_accept** or **t_snddis**). For this condition, **t_accept** fails and sets *t_errno* to TBADF.

If a different transport endpoint is specified (resfd != fd), the endpoint must be bound to a protocol address and must be in T_IDLE (see **t_getstate**) before **t_accept** is called.

For both types of endpoints, **t_accept** fails and sets *t_errno* to TLOOK if indications (that is, a connect or disconnect) are waiting to be received on

the endpoint.

The values of *opt* and the syntax of those values are protocol-specific.

udata enables the called transport user to send user data to the caller. The amount of user data must not exceed the limits supported by the transport provider, as returned in *connect* of *info* of **t_open** or **t_getinfo**. If *len* of *udata* is zero, no data will be sent to the caller.

See Also

t_connect, **t_getstate**, **t_listen**, **t_open**, **t_optmgmt (Function)**, **t_rcvconnect**, AppleTalk Notes: t_accept, IPX/SPX/SPX II Notes: t_accept , OSI Notes: t_accept, TCP/IP and TCP/IPX Notes: t_accept

Example

t_accept

```
int listen_fd, conn_fd;
struct t_call *call;
      .
      .
      .
/* Bind to any address */
if (t_bind(conn_fd, NULL, NULL) == -1)
{
   t_error(t_bind ");
   exit(1);
}
/* Accept connection */
if (t_accept(listen_fd, conn_fd, call) == -1)
{
   if ((t_errno == TLOOK) && (t_look(listen_fd) == T_DISCONNECT))
   {
      if (t_rcvdis(listen_fd, NULL) == -1)
      {
         t_error("t_rcvdis");
         exit(1);
      }
      t_error("t_accept");
      exit(1);
   }
}
```

t_alloc

Dynamically allocates memory for various TLI function argument structures and buffers referenced by the structures

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

char t_alloc(
   int   fd,
   int   struct_type,
   int   fields);
```

Parameters

fd

(IN) Indicates the local transport endpoint through which to pass the newly allocated structure.

struct_type

(IN) Indicates the structure to which to allocate memory.

fields

(IN) Indicates the memory allocated to the buffers associated with the specified fields.

Return Values

NULL	Failure (sets *t_errno* Code)
Pointer	Successful

t_errno Values

On successful completion, **t_alloc** returns a pointer to the newly allocated structure. On failure, it returns NULL and sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TSYSERR	System error occurred during execution
TNOTSUPPORT	Underlying transport provider does not support this function
TNOSTRUCTYPE	Specified *struct_type* is not supported

Remarks

t_alloc allocates memory for TLI structures.

Blocking Information In the NLM platform, **t_alloc** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The structure to allocate is specified by *struct_type* and can be one of the following:

Structure Type	Structure
T_BIND	t_bind
T_CALL	t_call
T_OPTMGMT	t_optmgmt
T_DIS	t_discon
T_UNITDATA	t_unitdata
T_UDERROR	t_uderr
T_INFO	t_info

Each one of these structures can subsequently be used as an argument to one or more TLI functions.

Each of the *struct_type* structures except T_INFO contains at least one field of type struct netbuf, which is defined in TIUSER.H.

For each field that is a netbuf structure, the user can specify that the buffer for that field should be allocated as well. (netbuf is defined in TIUSER.H.)

The *fields* argument specifies this option, where the argument is the bitwise-**or** of any of the following:

T_ADDR	The *addr* field of the t_bind, t_call, t_unitdata, or t_uderr structures.
T_OPT	The *opt* field of the t_optmgmt, t_call, t_unitdata, or t_uderr structures.
T_UDATA	The *udata* field of the t_call, t_discon, or t_unitdata structures.
T_ALL	All relevant fields of the given structure.

Each one of these structures can subsequently be used as an argument to one or more TLI functions.

Each *struct_type* except T_INFO contains at least one field of *struct netbuf*. For each *structnetbuf*, the user may specify that the buffer for that field should be allocated as well.

fields specifies this option, where the argument is the bitwise OR of any of the following:

For each field specified in *fields*, **t_alloc** allocates memory for the buffer associated with the field and initializes *buf* and *maxlen*, accordingly.

The length of the buffer allocated is based on the same size information that is returned to the user on **t_open** and **t_getinfo**. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure is passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see **t_open** or **t_getinfo**), **t_alloc** cannot determine the size of the buffer to allocate and fails, setting *t_errno* to TSYSERR and *errno* to EINVAL. For any field not specified in fields, *buf* is set to NULL and *maxlen* is set to zero.

Calling **t_alloc** to allocate structures helps ensure the compatibility of user programs with future releases of the TLI.

See Also

t_free, **t_getinfo**, **t_open**, IPX/SPX/SPX II Notes: t_alloc

Example

t_alloc

```
/*
   Allocate a structure of type struct t_bind and
   allocate all the relevant fields of that structure
*/
if ((bind = (struct t_bind *) t_alloc(listen_fd, T_BIND, T_ALL)) == NUL
{
   t_error("t_alloc");
   exit(0);
```

```
}
```

t_bind (Function)

Associates a protocol address with the specified transport endpoint and activates that transport endpoint

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

*int t_bind(
   int               fd,
   struct t_bind   *req,
   struct t_bind   *ret);
```

Parameters

fd

(IN) Indicates the local transport endpoint to be activated.

req

(IN) Points to the request that an address, represented by *netbuf*, be bound to the given transport endpoint.

ret

(OUT) Points to the address the transport provider acutally bound to the transport endpoint; this may be different from the address specified by the user in *req*.

Return Values

0	Successful
-1	Failure (sets *t_errno* Code)

t_bind returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

t_errno Values

TBADF	Specified file handle does not refer to a transport endpoint
TOUTSTATE	Function was issued in the wrong sequence
TBADADDR	Specified protocol address was in an incorrect format or contained illegal information
TACCESS	User cannot access specified option
TNOADDR	Transport provider could not allocate an address
TBUFOVFLW	Number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state changes to T_IDLE and the information to be returned in *ret* is discarded
TSYSERR	System error occurred during execution of this function
TADDRBUSY	Address the transport provider bound to the transport endpoint is busy.

Remarks

Blocking Information t_bind blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

req and *ret* point to t_bind (Structure), which is defined in TIUSER.H.

Fields	req	ret
maxlen	Has no meaning for *req*	Specifies the maximum size of the address buffer. If *maxlen* is not large enough to hold the returned address, an error results.
len	Specifies the number of bytes in the address.	Specifies the number of bytes in the bound address.
buf	Points to the address buffer.	Points to the buffer where the address is to be placed.

If the requested address is not available or if no address is specified in *req* (*len* of *addr* in *req* is zero), the transport provider can assign an appropriate address to be bound and return that address in *addr* of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

req can be NULL if the user does not want to specify an address to be bound. Here, *qlen* is assumed to be zero, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be NULL

if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to NULL for the same function, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

qlen has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that the transport provider has passed to the transport user. A value of *qlen* greater than zero is meaningful only when issued by a passive transport user that expects other users to call it. *qlen* is negotiated by the transport provider and can be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, *qlen* in *ret* contains the negotiated value.

t_bind allows more than one transport endpoint to be bound to the same protocol address (if the transport provider supports this capability). However, it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address.

Only one **t_bind** for a given protocol address can specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, the transport provider can assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. For a given protocol address, there can be only one listening endpoint at any time. This prevents more than one transport endpoint bound to the same protocol address from accepting connect indications.

See Also

t_alloc, **t_close**, **t_open**, **t_optmgmt (Function)**, **t_unbind**, AppleTalk Notes: t_bind, IPX/SPX/SPX II Notes: t_bind, OSI Notes: t_bind, TCP/IP Notes: t_bind, TCP/IPX Notes: t_bind

Example

t_bind

The following example illustrates calling the **t_bind** function to bind to any address, which is typically done by a client. The transport server allocates an address.

```
rc = t_bind(fd, NULL, NULL);
if (rc == -1)
{
   t_error("t_bind");
   exit(1);}
```

The following example for the TCP/IP protocol illustrates using the**t_bind** function to bind to a specific address, which is typically done by a server.

```
#define SRV_PORT   7
#define SRV_ADDR   "130.57.7.77"
#define MAX_CIND   5

   /* Allocate the address request structure */
   if ((bind = (struct t_bind *) t_alloc(listen_fd, T_BIND, T_ALL)) ==
   {
      t_error("t_alloc");
      exit(1);
   }
   /*
      Allocate a structure to check the returned
      information
   */
   if ((bound = (struct t_bind *) t_alloc(listen_fd, T_BIND, T_ALL)) ==
   {
      t_error("t_alloc");
      exit(1);
   }
   /* Specify the socket you want to bind to */
   /*
      Specify the max. no. of outstanding connect
      indications
   */
   bind->qlen = MAX_CIND;
   /* Length of the protocol address */
   bind->addr.len = sizeof (struct sockaddr_in);
   sinpt = (struct sockaddr_in *) bind->addr.buf;
   /* Family for internet protocols */
   sinpt->sin_family = AF_INET;
   /* Wellknown port */
   sinpt->sin_port = SRV_PORT;
   /* Local address */
   sinpt->sin_addr.s_addr = inet_addr(SRV_ADDR);
   if (t_bind(listen_fd, bind, bound) == -1)
   {
      t_error("t_bind");
      exit(1);
   }
   /* Check if this port is the one you wanted to bind to */
   if (((struct sockaddr_in *) (bound->addr.buf))->sin_port != SRV_PORT
   {
      printf("Bound wrong address\n");
```

```
            exit(1);
        }
```

t_blocking

Puts the transport into a blocking mode (NetWare® specific)
Local Servers: nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_blocking(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the transport is to be put into blocking mode.

Return Values

0	Successful
-1	Failure (sets *t_errno* Code)

t_errno Values

t_blocking returns a value of 0 if it is successful. If an error occurs, it returns a value of -1 and sets *t_errno* to the following:

TBADF	Specified file handle does not refer to a transport endpoint

Remarks

t_blocking can be useful in a multitasking environment, because it is not always desirable to do "busy waiting." In the case of busy waiting, the user consumes processor cycles checking for work, sometimes only to find that there is no work. Since the NetWare operating system is a nonpreemptive operating system, busy waiting would mean no other

threads could work while the user is checking for work.

The blocking mode forces the NetWare operating system to make the user's thread sleep until I/O is available.

See Also

t_nonblocking, IPX/SPX/SPX II Notes: t_blocking

t_close

Informs the transport provider the user is finished with the specified transport endpoint, frees any local library resources associated with the endpoint, and closes the file associated with the transport endpoint

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_close(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint for which associated local library resources are to be freed.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_close** function returns 0 if successful. If an error occurs, it returns a value of -1. On failure, it sets *t_errno* to the following:

TBADF	Specified file handle does not refer to a transport endpoint

Remarks

t_close should be called from the T_UNBND state (see **t_getstate**). However, this function does not check state information, so it can be called from any state to close a transport endpoint. If this occurs, the local

library resources associated with the endpoint is freed automatically. In addition, a close is issued for that file handle; the close is abortive if no other process has that file open and breaks any transport connection that might be associated with that endpoint.

Blocking Information**t_close** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

See Also

t_getstate, **t_open**, **t_unbind**, AppleTalk Notes: t_close, IPX/SPX/SPX II Notes: t_close, OSI Notes: t_close, TCP/IP and TCP/IPX Notes: t_close

Example

t_close

```
int rc, fd;
    .
    .
    .
/* Close the endpoint */
rc = t_close(fd);
if (rc == -1) {
   t_error("t_close");
   exit(1);
}
```

t_connect

Enables a transport user to request a connection to the specified destination transport user

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_connect(
   int               fd,
   struct t_call   *sndcall,
   struct t_call   *rcvcall);
```

Parameters

fd

(IN) Indicates the local transport endpoint where communication will be established.

sndcall

(IN) Indicates the information needed by the transport provider to establish a connection.

rcvcall

(OUT) Indicates the information associated with the newly established connection.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_connect** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TOUTSTATE	Function was issued in the wrong sequence
TNODATA	O_NDELAY was set so the function successfully initiated the connection establishment procedure, but it did not wait for a response from the remote user
TBADADDR	Specified protocol address was in an incorrect format or contained illegal information
TACCESS	User cannot access specified option
TBADOPT	Specified protocol options were in an incorrect format or contained illegal information
TBADDATA	Amount of user data specified was not within the bounds allowed by the transport provider
TBUFOVFLW	Number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state (as seen by the user) changes to T_DATAXFER and the connect indication information to be returned in *rcvcall* is discarded
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TSYSERR	System error occurred during execution

Remarks

t_open enables a transport user to request a connection to the destination transport user.

Blocking Informationt_connect blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

sndcall and *rcvcall* point to a t_call structure, which is defined in TIUSER.H. For this function, the fields of t_call have the following meanings:

Fields	sndcall	rcvcall
addr	Specifies the protocol address of the destination transport user.	Returns the protocol address associated with the responding transport endpoint.
opt	Presents any protocol-specific	Presents any

	information that might be needed by the transport provider.	protocol-specific information associated with the connection.
udata	Points to the optional user data that may be passed to the destination transport users during connection establishment. If *len* of *udata* is zero in *sndcall*, no data is sent to the destination transport user.	Points to the optional user data that may be returned by the destination transport user during connection establishment.
sequence	Has no meaning for this function.	Has no meaning for this function.

The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user can choose not to negotiate protocol options by setting *len* of *opt* to zero. Then the provider may use default options.

udata enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider, as returned by **t_open** or **t_getinfo**.

On return, *addr*, *opt*, and *udata* of *rcvcall* are updated to reflect values associated with the connection. Thus, *maxlen* of each argument must be set before calling **t_connect** to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL, in which case no information is given to the user on return from **t_connect**.

The netbuf structure used in the t_call structure is defined in TIUSER.H:

By default, **t_connect** executes in synchronous mode and waits for the destination user's response before returning control to the local user. A successful return (that is, a return value of zero) indicates the requested connection has been established. However, if O_NDELAY is set (by calling **t_open**), **t_connect** executes in asynchronous mode. In this case **t_connect** does not wait for the remote user's response but returns control immediately to the local user. It also returns -1 with t_errno set to TNODATA to indicate the connection has not yet been established. It simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

See Also

t_accept, **t_alloc**, **t_getinfo**, **t_listen**, **t_open**, **t_optmgmt (Function)**, **t_rcvconnect**, AppleTalk Notes: t_connect, IPX/SPX/SPX II Notes: t_connect, OSI Notes: t_connect, TCP/IP Notes: t_connect, TCP/IPX Notes: t_connect

Example

t_connect

```
rc = t_connect(fd,sndcall, NULL);
if (rc == -1)
{
   if (t_errno != TNODATA)
   {
      t_error("t_connect");
      exit(1);
   }
}
```

t_error

Produces a message on the standard error output describing the last error encountered by calling a TLI function

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

void t_error(
   char   *errmsg);
```

Parameters

errmsg

(IN) Indicates the user-supplied error message that gives context to the error.

Return Values

0	Successful
-1	Failure (sets *t_errno* Code)

t_errno Values

On failure, *t_errno* is set to the following:

TNOTSUPPORT	Underlying transport provider does not support this function

Remarks

Blocking Informationt_error blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The **t_error** function prints the user-supplied error message followed by a colon and a standard error message for the current value contained in *t_errno*. If *t_errno* is TSYSERR, **t_error** also prints the standard error message for the current value contained in *errno*.

When an error occurs, *t_errno* is set and is not cleared on subsequent successful calls.

t_errno can be used as an index into the t_errlist array to retrieve the error message string (without a terminating newline).

The error message strings are provided in the t_errlist array. The maximum index value for the t_errlist array is defined by the t_nerr variable.

Example

t_error

If a **t_connect** function fails on transport endpoint *fd* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd");
```

The diagnostic message would print as:

```
t_connect failed on fd: Incorrect transport address format
```

where "**t_connect** failed on *fd*" tells the user which function failed on which transport endpoint, and "Incorrect transport address format" identifies the specific error that occurred.

```
rc = t_connect(fd, sndcall, NULL);
if (rc == -1)
{
   t_error("t_connect failed on fd");
   exit(1);
}
```

t_free

Frees memory previously allocated by **t_alloc**
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_free(
   char   *ptr,
   int     struct_type);
```

Parameters

ptr

(IN) Points to one of the six structure types described for **t_alloc**.

struct_type

(IN) Indicates the structure for which memory will be freed.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_free** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to the following:

TSYSERR	System error occurred during execution of this function
TNOTSUPPORT	Underlying transport provider does not support this function

Remarks

t_free frees memory allocated by **t_alloc**. **t_free** frees memory both for the specified structure and for buffers referenced by the structure.

Blocking Information**t_free** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The netbuf structure used in the t_call structure is defined in TIUSER.H.

If *buf* is NULL, **t_free** does not attempt to free memory. After all buffers are freed, **t_free** frees the memory associated with the structure pointed to by *ptr*.

t_free checks *addr*, *opt*, and *udata* of the given structure (as appropriate) and frees the buffers pointed to by *buf* of *netbuf*.

Undefined results occur if *ptr* or *buf* points to a block of memory not previously allocated by **t_alloc**.

Each of these structures is used as an argument to one or more TLI functions. The structure specified by *struct_type* can be one of the following:

Name	Structure
T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

Services

Transport Level Interface (TLI)

See Also

t_alloc

Example

t_free

```
/* Free the structure allocated with t_alloc */

rc = t_free((char *) bind, T_BIND);
if (rc == -1)
{
   t_error("t_free");
   exit(1);
}
```

t_getinfo

Returns the current characteristics of the underlying transport protocol associated with the specified file handle

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_getinfo(
   int               fd,
   struct t_info   *info);
```

Parameters

fd

(IN) Indicates the file handle for which associated transport protocol characteristics are to be returned.

info

(OUT) Points to the information returned by **t_open**.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_getinfo** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TSYSERR	System error occurred during execution of this function
TNOTSUPPO	Underlying transport provider does not support this

RT	function

Remarks

t_getinfo enables a transport user to access the information returned by *info* during any phase of communication.

Blocking Information**t_getinfo** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

info points to a t_info structure, which is defined in TIUSER.H.

If concerned with protocol independence, a transport user can check*info* to determine how large the buffers must be to hold each piece of information. Alternatively, the user can allocate these buffers with**t_alloc** . An error results if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and **t_getinfo** enables a user to retrieve the current characteristics.

See Also

t_alloc, **t_open**, AppleTalk Notes: t_getinfo, IPX/SPX/SPX II Notes: t_getinfo, OSI Notes: t_getinfo, TCP/IP Notes: t_getinfo, TCP/IPX Notes: t_getinfo

Example

t_getinfo

```
/* Get information about the transport interface */
rc = t_getinfo(fd, info);
if (rc == -1)
{
   t_error("t_getinfo");
   exit(1);
}
```

t_getstate

Returns the current state of the provider associated with the specified transport endpoint

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_getstate(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint for which the current state of the provider is to be returned.

Return Values

Current state	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_getstate** function returns the current state on successful completion and -1 on failure. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TSTATECHNG	Transport provider is undergoing a state change
TSYSERR	System error occurred during execution of this function
TNOTSUPPORT	Underlying transport provider does not support this function

Remarks

t_getstate returns the state of the provider associated with the transport endpoint *fd*.

Blocking Information**t_getstate** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The current state of the provider can be one of the following:

State Name	Description
T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication)
T_INREL	Incoming orderly release (waiting for an orderly release request)

If the provider is undergoing a state transition when **t_getstate** is called, **t_getstate** fails and t_errno is set to TSTATECHNG.

See Also

t_open

Example

t_getstate

```
rc = t_getstate(fd);
if (rc == -1)
{
   t_error("t_getstate");
   exit(1);
}
else
{
   switch (rc)
```

```
        {
            case T_UNBND:
            printf("T_UNBND\n");
            break;

            case T_IDLE:
            printf("T_IDLE\n");
            break;

            case T_OUTCON:
            printf("T_OUTCON\n");
            break;

            case T_INCON:
            printf("T_INCON\n");
            break;

            case T_DATAXFER:
            printf("T_DATAXFER\n");
            break;

            case T_OUTREL:
            printf("T_OUTREL\n");
            break;

            case T_INREL:
            printf("T_INREL\n");
            break;

            default:
            printf("Invalid state\n");
            exit(1);
        }
    }
```

t_listen

Listens for a connect request from a calling transport user
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>

int t_listen(
   int               fd,
   struct t_call   *call);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the connect indications arrive.

call

(OUT) Points to the information describing the connect indication.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_listen** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TBADQLEN	Argument *qlen* of the transport endpoint specified by *fd* is zero
TBUFOVFLW	Number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to

	T_INCON. The connect indication information to be returned in *call* is discarded.
TNODATA	O_NDELAY was set, but no connect indications had been queued
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

t_listen listens for a connect request.

Blocking Information**t_listen** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The *call* parameter points to a t_call structure, which is defined in TIUSER.H. For this function, the fields of t_call have the following meanings:

addr	Returns the protocol address of the calling transport user.
opt	Returns protocol-specific parameters associated with the connect request.
udata	Returns any user data sent by the caller on the connect request.
sequence	Indicates a number uniquely identifying the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since **t_listen** returns values for *addr*, *opt*, and *udata* of *call*, *maxlen* of each must be set before calling **t_listen** to indicate the maximum size of the buffer for each.

If a user issues **t_listen** in synchronous mode on a transport endpoint that was not bound for listening (that is, *qlen* was zero on **t_bind**), the call waits forever because no connect indications arrive on that endpoint.

By default, **t_listen** executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if O_NDELAY is set (using **t_open**), **t_listen** executes asynchronously,

reducing to a poll for existing connect indications. If none are available, it returns -1 and sets *t_errno* to TNODATA.

See Also

t_accept, **t_alloc**, **t_bind (Function)**, **t_connect**, **t_open**, **t_optmgmt (Function)**, **t_rcvconnect**, AppleTalk Notes: t_listen, IPX/SPX/SPX II Notes: t_listen, OSI Notes: t_listen, TCP/IP Notes: t_listen, TCP/IPX Notes: t_listen

Example

t_listen

```
while (t_listen(listen_fd, call) == -1)
{
   if (t_errno != TNODATA)
   {
      t_error("t_listen");
      exit(1);
   }
   ThreadSwitch();
}
```

t_look

Returns the current event on the specified transport endpoint
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_look(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint for which the current event is to be returned.

Return Values

0	Successful (no event occurred) Event Value Successful (the value indicates which of the allowable events has occured)
-1	Failure (sets t_errno Code)

t_errno Values

On failure, **t_look** returns a value of -1 and sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TSYSERR	System error occurred during execution

Upon success, **t_look** returns a value indicating which of the allowable events has occurred or returns a value of 0 if no event occurred. The value for one of the following events is returned:

Allowable Events	Description
T_LISTEN	Connect indication received
T_CONNECT	Connect confirmation received
T_DATA	Normal data received
T_EXDATA	Expedited data received
T_DISCONNECT	Disconnect received
T_UDERR	Datagram error indication
T_ORDREL	Orderly release indication
T_GODATA	Flow control restrictions on normal data flow have been lifted
T_GOEXDATA	Flow control restrictions on expedited data flow have been lifted

Remarks

t_look enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next function to be executed.

t_look also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Blocking Information**t_look** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

See Also

t_open, **t_snd**, **t_sndudata**

t_nonblocking

Puts the transport endpoint into a nonblocking mode (NetWare specific)
Local Servers: nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_nonblocking(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the transport is to be put into nonblocking mode.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

If successful, **t_nonblocking** returns a value of 0. If unsuccessful, it returns a value of -1 and sets *t_errno* to the following:

TBADF	Specified file handle does not refer to a transport endpoint

Remarks

t_nonblocking forces the stream head associated with*fd* to return an error code to calling processes if their requests cannot be completed immediately.

t_nonblocking is implemented with an **ioctl** call with I_SETDELAY and

1 as arguments.

For example, in a STREAMS-based TLI environment, you can open a STREAM in a nonblocking mode as follows:

```
fd = t_open("/dev/tcp",O_RDWR | O_NDELAY,(struct t_info *)0);
```

In NetWare, you begin by opening the transport endpoint in a blocking mode, as follows:

```
fd = t_open("dev/tcp",O_RDWR ,(struct t_info *)0);
```

Then, you can switch to the nonblocking mode with the following command:

```
t_nonblocking(fd);
```

See Also

t_blocking, IPX/SPX/SPX II Notes: t_nonblocking

t_open

Initializes a transport endpoint by opening a file identifying a particular transport provider and returns a file handle identifying the endpoint

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_open(
   char             *path,
   int               oflag,
   struct t_info    *info);
```

Parameters

path

(IN) Points to the pathname of the file to open.

oflag

(IN) Specifies the settings for the flags O_NDELAY and O_RDWR. O_RDWR specifies that the endpoint is to be opened for read/write access. Default is blocking mode. When O_NDELAY is ORed with O_RDWR, the endpoint is opened in nonblocking mode.

info

(OUT) Points to the information about the underlying transport protocol.

Return Values

File handle	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_open** function returns a valid file handle if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADFLAG	Invalid flag was specified
TSYSERR	System error occurred during execution
TBADNAME	Invalid transport provider name was specified

Remarks

t_open establishes a transport endpoint by opening a file that identifies a particular transport provider (that is, transport protocol) and returns a file handle that identifies the endpoint.

t_open must be called to initialize a transport endpoint. For example, opening /dev/iso_cots identifies an OSI connection-oriented transport protocol as the transport provider.

t_open returns a file handle all subsequent functions use to identify the particular local transport endpoint. **t_open** also returns various default characteristics of the underlying transport protocol by setting fields in the t_info structure.

Blocking Information**t_open** blocks if called in blocking mode.

info points to a t_info structure, which is defined in TIUSER.H.

If concerned with protocol independence, a transport user can check*info* to determine how large the buffers must be to hold each piece of information. Alternatively, **t_alloc** can be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any function.

A single transport endpoint can support only one of the preceding services at one time.

If the transport user sets *info* to NULL, **t_open** returns no protocol information.

The protocols and their associated files are as follows:

Protocol	**Protocol File Name**
IPX	/dev/nipx
OSI	/dev/iso_cots
SPX	/dev/nspx
SPX2	/dev/nspx2
TCP	/dev/tcp
UDP	/dev/udp

Services

Transport Level Interface (TLI)

See Also

t_alloc, AppleTalk Notes: t_open, IPX/SPX/SPX II Notes: t_open, OSI Notes: t_open, TCP/IP and TCP/IPX Notes: t_open

Example

t_open

The following example illustrates the use of the **t_open** function in asynchronous mode:

```
fd = t_open("/dev/tcp", O_RDWR | O_NDELAY, NULL);
if (fd == -1)
{
   t_error("t_open");
   exit(1);
}
```

The following example illustrates the use of the **t_open** function in synchronous mode:

```
fd = t_open("/dev/tcp", O_RDWR, NULL);
if (fd == -1)
{
   t_error("t_open");
   exit(1);
}
```

t_optmgmt (Function)

Enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_optmgmt(
   int                  fd,
   struct t_optmgmt    *req,
   struct t_optmgmt    *ret);
```

Parameters

fd

(IN) Indicates the bound transport endpoint.

req

(IN) Indicates a specification of the transport provider. See t_optmgmt (Structure).

ret

(OUT) Indicates the options and flag values. See t_optmgmt (Structure).

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_optmgmt** function returns 0 if successful. Otherwise, it returns a value of - 1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport

	endpoint
TOUTSTATE	Function was issued in the wrong sequence
TBADOPT	Specified protocol options were in an incorrect format or contained illegal information
TACCESS	User cannot access specified option
TBADFLAG	Invalid flag was specified
TBUFOVFLW	Number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* is discarded
TSYSERR	System error occurred during execution
TNOTSUPPORT	Underlying transport provider does not support this function

Remarks

t_optmgmt enables a transport user to retrieve, verify or negotiate protocol options with the transport provider.

Blocking Information t_optmgmt blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

req and *ret* point to a t_optmgmt (Structure), which is defined in TIUSER.H.

req requests a specific action of the provider and sends options to the provider.

Although *maxlen* has no meaning for *req*, it must be set in *ret* to specify the maximum number of bytes the options buffer can hold. The transport provider imposes the actual structure and content of the options.

If issued as part of the connectionless-mode service, **t_optmgmt** can block due to flow control constraints. It does not complete until the transport provider has processed all previously sent data units.

See Also

t_accept, **t_alloc**, **t_connect**, **t_listen**, **t_rcvconnect**, **t_getinfo**, **t_open**, AppleTalk Notes: t_optmgmt

t_rcv

Receives either normal or expedited data
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcv(
   int          fd,
   char        *buf,
   unsigned     nbytes,
   int         *flags);
```

Parameters

fd

(IN) Indicates the local transport endpoint through which data will arrive.

buf

(OUT) Points to a receive buffer where user data will be placed.

nbytes

(IN) Indicates the size of the receive buffer.

flags

(OUT) Points to the optional flags; may be returned from **t_rcv**.

Return Values

Bytes received	Successful
-1	Failure (sets t_errno Code)

t_errno Values

On successful completion, **t_rcv** returns the number of bytes received. On failure, it it returns -1 and sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TNODATA	O_NDELAY was set, but no data is currently available from the transport provider
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

By default, **t_rcv** operates in synchronous mode and waits for data to arrive if none is currently available.

However, if O_NDELAY is set (by calling **t_open**), **t_rcv** executes in asynchronous mode and fails if no data is available. (See TNODATA in the "Return Values" section.)

Blocking Informationt_rcv blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

On return from **t_rcv**, if T_MORE is set in *flags*, there is more data. The current Transport Service Data Unit (TSDU) or Expedited Transport Service Data Unit (ETSDU) must be received in multiple **t_rcv** calls. Each **t_rcv** with T_MORE set indicates another **t_rcv** must follow to get more data for the current TSDU. The end of TSDU is identified by the return of **t_rcv** with T_MORE not set. If the transport provider does not support TSDUs as indicated in *info* on return from **t_open** or **t_getinfo**, T_MORE is not meaningful and can be ignored.

The data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, **t_rcv** sets T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED set on return.

The end of the ETSDU is identified by the return of **t_rcv** with T_MORE not set. If expedited data arrives after part of TSDU has been retrieved, receipt of the remainder of TSDU is suspended until ETSDU has been processed. The remainder of TSDU becomes available to the user only after the full ETSDU has been retrieved (T_MORE not set).

See Also

t_open, **t_snd**, **t_getinfo**, AppleTalk Notes: t_rcv, IPX/SPX/SPX II Notes: t_rcv, OSI Notes: t_rcv, TCP/IP and TCP/IPX Notes: t_rcv

Example

t_rcv

The following example illustrates the use of the **t_rcv** function in synchronous mode:

```
/*
   t_rcv() in synchronous mode. Do a t_rcv() until all
   the necessary data is retrieved.
*/
for (nread = 0; nread < sizeof (iobuf); nread += rc)
{
   flags = 0;
   if ((rc = t_rcv(fd, iobuf,sizeof(iobuf), &flags)) == -1)
   {
      t_error("t_rcv");
      exit(1);
   }
}
```

The following example illustrates the use of the **t_rcv** function in asynchronous mode:

```
/*
   t_rcv() in asynchronous mode. If no data is
   available, then it fails and t_errno is set to
   TNODATA
*/
for (nread = 0; nread < sizeof (iobuf); nread += rc)
{
   flags = 0;
   while ((rc = t_rcv(fd, iobuf,sizeof(iobuf), &flags)) == -1)
   {
      if (t_error != TNODATA)
      {
      t_error("t_rcvudata");
      exit(1);
      }
      flags = 0;
      ThreadSwitch();
   }
}
```

t_rcvconnect

Enables a calling transport user to determine the status of a previously sent connect request

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcvconnect(
   int               fd,
   struct t_call   *call);
```

Parameters

fd

(IN) Indicates the local transport endpoint where communication is established.

call

(OUT) Points to the information associated with the newly established connection.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

t_rcvconnect returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TBUFOVFLW	Number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The connect information to be returned in call is

	discarded. The provider's state, as seen by the user, changes to DATAXFER.
TNODATA	O_NDELAY was set, but a connect confirmation has not yet arrived
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

t_rcvconnect enables a calling transport user to determine the status of a previously sent connect request. You can use **t_rcvconnect** in conjunction with **t_connect** to establish a connection in asynchronous mode. The connection is established on successful completion of this function.

t_rcvconnect can be used in conjunction with **t_connect** to establish a connection in asynchronous mode. The connection is established on successful completion of **t_rcvconnect**.

Blocking Information**t_rcvconnect** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

call points to a t_call structure, which is defined in TIUSER.H. For this function, the fields of t_call have the following meanings:

addr	Returns the protocol address associated with the responding transport endpoint.
opt	Returns protocol-specific information associated with the connect request.
udata	Points to optional user data that may be returned by the destination transport user during connection establishment.
sequence	Has no meaning for **t_rcvconnect**.

maxlen of each argument must be set before issuing **t _rcvconnect** to indicate the maximum size of the buffer for each. However, *call* may be NULL, in which case no information is given to the user on return from **t_rcvconnect**. By default, **t_rcvconnect** executes in synchronous mode and waits for the connection to be established before returning. On return, *addr*, *opt*, and *udata* reflect values associated with the connection.

If O_NDELAY is set (by calling **t_open**), **t_rcvconnect** executes in asynchronous mode and reduces to a poll for existing connect confirmations. If none are available, **t_rcvconnect** fails and returns immediately without waiting for the connection to be established. (See TNODATA in t_errno Values above.) **t_rcvconnect** must be reissued at a later time to complete the connection establishment phase and retrieve the information returned.

See Also

t_accept, **t_alloc**, **t_bind (Function)**, **t_connect**, **t_listen**, **t_open**, **t_optmgmt (Function)**, AppleTalk Notes: t_rcvconnect, IPX/SPX/SPX II Notes: t_rcvconnect, OSI Notes: t_rcvconnect, TCP/IP Notes: t_rcvconnect , TCP/IPX Notes: t_rcvconnect

Example

t_rcvconnect

```
/* Asynchronous Connect */
/* Initialize the sndcall structure as shown in figure(t_connect() firs
rc = t_connect(fd, sndcall, NULL);
if (rc == -1)
{
   if (t_errno != TNODATA)
   {
      t_error("t_connect");
      exit(1);
   }
   /* Check whether t_connect succeeded */
   /* Loop until t_rcvconnect succeeds */
   while (t_rcvconnect(fd, NULL) == -1)
   {
      if (t_errno != TNODATA)
      {
         t_error("t_rcvconnect");
         exit(2);
      }
      ThreadSwitch();
   }
}
```

t_rcvdis

Identifies the cause of a disconnect and retrieves any user data sent with the disconnect

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcvdis(
   int                 fd,
   struct t_discon   *discon);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the connection existed.

discon

(OUT) Indicates the information pertaining to the disconnect.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_rcvdis** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TNODIS	No disconnect indication currently exists on the specified transport Endpoint
TBUFOVFLW	Number of bytes allocated for an incoming argument

	is not sufficient to store the value of that argument. The connect information to be returned in call is discarded. The provider's state, as seen by the user, changes to DATAXFER.
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

discon points to a t_discon structure, which is defined in TIUSER.H.

If a disconnect indication occurs, *sequence* can identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care about incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL; then any user data associated with the disconnect is discarded. However, if a user has retrieved more than one outstanding connect indication (by calling **t_listen**) and *discon* is NULL, the user is unable to identify the connect indication with which the disconnect is associated.

See Also

t_alloc, **t_connect**, **t_listen**, **t_open**, **t_snddis**, AppleTalk Notes: t_rcvdis, IPX/SPX/SPX II Notes: t_rcvdis, OSI Notes: t_rcvdis, TCP/IP and TCP/IPX Notes: t_rcvdis

Example

t_rcvdis

```
while (t_rcvdis(fd, NULL) == -1)
{
   if (t_errno != TNODIS)
   {
      t_error("t_rcvdis");
      exit(1);
   }
   ThreadSwitch();
}
```

t_rcvrel

Acknowledges receipt of an orderly release indication
Local Servers: nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcvrel(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the connection exists.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

t_rcvrel returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TNOREL	No orderly release indication currently exists on the specified transport endpoint
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*

TSYSERR	System error occurred during execution

Remarks

t_sndrel is an optional service of the transport provider and is supported only if the transport provider returned T_COTS_ORD on**t_open** or **t_getinfo**.

After receipt of the orderly release indication, the user cannot attempt to receive more data because such an attempt will block forever. However, if **t_sndrel** has not been issued, the user may continue to send data over the connection.

See Also

t_getinfo, **t_open**, **t_sndrel**, AppleTalk Notes: t_rcvrel, IPX/SPX/SPX II Notes: t_rcvrel, TCP/IP and TCP/IPX Notes: t_rcvrel

Example

t_rcvrel

```
/* Initiate a closure */
rc = t_sndrel(fd);
if (rc == -1)
{
   t_error("t_sndrel");
   exit(1);
}
/* Loop till t_rcvrel is successful */
while (t_rcvrel(fd) == -1)
{
   if (t_errno != TNOREL)
   {
      t_error("t_rcvrel");
      exit(1);
   }
   ThreadSwitch();
}
```

t_rcvudata

Receives a data unit from another transport user in connectionless mode
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcvudata(
   int                  fd,
   struct t_unitdata   *unitdata,
   int                 *flags);
```

Parameters

fd

(IN) Indicates the local transport endpoint through which data will be received.

unitdata

(OUT) Points to the hold information associated with the received data unit.

flags

(OUT) Points to a location that indicates if the complete data unit was not received.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

t_rcvudata returns 0 if successful. Otherwise, it returns a value of - 1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport

	endpoint
TNODATA	O_NDELAY was set, but no data units are currently available from the transport provider
TBUFOVFLW	Number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* is discarded
TLOOK	Asynchronous event has occurred on this transport endpoint and requires immediate attention
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

t_rcvudata is used in connectionless mode to receive a data unit from another transport user.

Blocking Information**t_rcvudata** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

unitdata points to a t_unitdata structure that is defined in TIUSER.H.

maxlen of *addr*, *opt*, and *udata* must be set before issuing **t_rcvudata** to indicate the maximum size of the buffer for each.

The netbuf structure used in the t_unitdata structure is defined in TIUSER.H as follows:

If the buffer defined in *udata* of *unitdata* is not large enough to hold the current data unit, the buffer is filled. Then T_MORE is set in *flags* on return to indicate the user needs to call **t_rcvudata** to retrieve the rest of the data unit.

By default, **t_rcvudata** operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if O_NDELAY is set (by calling **t_open**), **t_rcvudata** executes in asynchronous mode and fails if no data units are available.

Subsequent **t_rcvudata** calls return zero for the length of the address and options until the full data unit has been received.

See Also

t_rcvuderr, **t_sndudata**, AppleTalk Notes: t_rcvudata, IPX/SPX/SPX II

Notes: t_rcvudata, OSI Notes: t_rcvudata, TCP/IP Notes: t_rcvudata

Example

t_rcvudata

```
/*
   t_rcvudata() in asynchronous mode. If no data is
   available then it fails and t_errno is set to
   TNODATA
*/
for (nread = 0; nread < sizeof (iobuf); nread += rc)
{
   flags = 0;
   while ((rc = t_rcvudata(fd, unitdata, &flags)) == -1)
   {
      if (t_errno ! == TNODATA)
      {
         t_error("t_rcvudata");
         exit(0);
      }
      flags = 0;
      ThreadSwitch();
   }
}
```

t_rcvuderr

Receives information in connectionless mode concerning an error on a previously sent data unit

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_rcvuderr(
   int                fd,
   struct t_uderr   *uderr);
```

Parameters

fd

(IN) Indicates the local transport endpoint through which the error report will be received.

uderr

(OUT) Points to the error information.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_rcvuderr** function returns 0 on successful completion and -1 on failure. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TNOUDERR	No unit data error indication currently exists on the specified transport endpoint
TBUFOVFLW	Number of bytes allocated for the incoming protocol

	address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* is discarded
TNOTSUPPORT	Underlying transport provider does not support this function
TSYSERR	System error occurred during execution

Remarks

t_rcvuderr should be issued only after receiving a unit data error indication. It informs the transport user that a data unit, with a specific destination address and protocol options, produced an error.

t_rcvuderr is used in connectionless mode to receive information concerning an error on a previously sent data unit.

Blocking Information**t_rcvuderr** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

uderr points to a t_uderr structure, which is defined in TIUSER.H.

maxlen of *addr* and *opt* must be set before calling **t_rcvuderr** to indicate the maximum size of the buffer for each.

If you don't want to identify the data unit that produced an error, *uderr* can be set to NULL. In that case, **t_rcvuderr** simply clears the error indication without reporting any information to the user.

See Also

t_alloc, **t_open**, **t_rcvudata**, **t_sndudata**, IPX/SPX/SPX II Notes: t_rcvuderr

Example

t_rcvuderr

```
if (t_sndudata(fd, unitdata) == -1)
{
   t_error("t_sndudata");
   exit(1);
}
/* Look for any T_UDERR */
if (t_look == T_UDERR)
{
   if (t_rcvuderr(fd, uderr) == -1)
   {
      t_error("t_rcvuderr");
```

```
                exit(1);
            }
        }
```

t_snd

Sends either normal or expedited data
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_snd(
   int         fd,
   char       *buf,
   unsigned    nbytes,
   int         flags);
```

Parameters

fd

(IN) Indicates the local transport endpoint over which data should be sent.

buf

(IN) Points to the user data.

nbytes

(IN) Indicates the number of bytes of user data to be sent.

flags

(IN) Indicates the optional flags described below.

Return Values

number of bytes	Successful
-1	Failure (sets t_errno Code)

t_errno Values

On successful completion, the **t_snd** function returns the number of bytes

accepted by the transport provider, and it returns -1 on failure. On failure, it sets *t_errno* to one of the following:

TBADDATA	Illegal amount of data. Zero octets is not supported
TBADF	Specified file handle does not refer to a transport endpoint
TBADFLAG	Invalid flag was specified
TFLOW	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time
TLOOK	Asynchronous event occurred on the transport endpoint specified by *fd*
TNOTSUPPORT	Underlying transport provider does not support this function
TSYSERR	System error occurred during execution

Remarks

t_snd sends either normal or expedited data.

Blocking Informationt_snd blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

By default, **t_snd** operates in synchronous mode and can wait if flow control restrictions prevent the local transport provider from accepting the data at the time the call is made. However, if O_NDELAY is set (by calling **t_open**), **t_snd** executes in asynchronous mode and fails immediately if there are flow control restrictions.

Even when there are no flow control restrictions, **t_snd** waits if STREAMS internal resources are not available, regardless of the state of O_NDELAY.

On successful completion, **t_snd** returns the number of bytes accepted by the transport provider. Normally, this equals the number of bytes specified in *nbytes*. However, if O _NDELAY is set, it is possible only part of the data was accepted by the transport provider. In this case, **t_snd** sets T_MORE and returns a value less than *nbytes*. If *nbytes* is zero, no data is passed to the provider and **t_snd** returns a value of 0.

If T_EXPEDITED is set in *flags*, the data is sent as expedited data and is subject to the interpretations of the transport provider.

If T_MORE is set in *flags* or is set as described in the preceding discussion, an indication is sent to the transport provider as notification that a Transport Service Data Unit (TSDU) or an Expedited Transport Service Data Unit (ETSDU) is being sent through multiple **t_snd** functions. Each

t_snd with T_MORE set indicates another **t_snd** will follow with more data for the current TSDU. The end of TSDU (or ETSDU) is identified by **t_snd** with T_MORE not set.

Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. T_MORE implies nothing about how the data is packaged for transfer below the TLI. If the transport provider does not support TSDUs as indicated in *info* on return from **t_open** or **t_getinfo**, T_MORE is not meaningful and the user can ignore it.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by **t_open** or **t_getinfo**. If the size is exceeded, TSYSERR with EPROTO occurs. However, **t_snd** may not fail because EPROTO may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint fails with the associated TSYSERR.

If **t_snd** is called from T_IDLE, the provider can silently discard the data. If **t_snd** is issued from any state other than T_DATAXFER, T_INREL, or T_IDLE, the provider generates TSYSERR with EPROTO (which may be reported and behave in the manner described in the previous paragraph).

See Also

t_getinfo, **t_open**, **t_rcv**, AppleTalk Notes: t_snd, IPX/SPX/SPX II Notes: t_snd, OSI Notes: t_snd, TCP/IP and TCP/IPX Notes: t_snd

Example

t_snd

```
rc = t_snd(fd, iobuf, sizeof (iobuf), 0);
if (rc == -1)
{
   t_error("t_snd");
   exit(1);
}
```

t_snddis

Initiates an abortive release on an already established connection and/or rejects a connect request

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_snddis(
   int               fd,
   struct t_call   *call);
```

Parameters

fd

(IN) Indicates the local transport endpoint of the connection.

call

(IN) Points to the information associated with the abortive release.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_snddis** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TOUTSTATE	Function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost
TBADDATA	Amount of user data specified was not within the

	bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost
TBADSEQ	Invalid sequence number was specified, or a null call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost
TNOTSUPPORT	Underlying transport provider does not support this function
TSYSERR	System error occurred during execution

Remarks

t_snddis initiates an abortive release on an already established connection. It can also reject a connect request.

Blocking Information**t_snddis** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

call points to a t_call structure, which is defined in TIUSER.H.

The values in *call* have different semantics, depending on the context of **t_snddis**.

When **t_snddis** is rejecting a connect request, *call* must not be NULL, and it must contain a valid value of sequence to identify uniquely the rejected connect indication to the transport provider. *addr* and *opt* of *call* are ignored. In all other cases, *call* is used only when data is being sent with the disconnect request. *addr*, *opt*, and *sequence* of t_call are ignored. If the user does not want to send data to the remote user, *call* can be NULL.

udata specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned in *discon* of *info* of **t_open** or **t_getinfo**. If *len* of *udata* is zero, no data is sent to the remote user.

See Also

t_connect, **t_getinfo**, **t_listen**, **t_open**, AppleTalk Notes: t_snddis, IPX/SPX/SPX II Notes: t_snddis, OSI Notes: t_snddis, TCP/IP and TCP/IPX Notes: t_snddis

Example

t_snddis

```
/* Abort an existing connection */
```

```
rc = t_snddis(fd, NULL);
if (rc == -1)
{
   t_error("t_snddis");
   exit(1);
}
```

t_sndrel

Initiates an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_sndrel(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint where the connection exists.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_sndrel** function returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TFLOW	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the function at this time
TLOOK	Asynchronous event occurred on the transport endpoint specified by *fd*
TNOTSUPPORT	Underlying transport provider does not support this function

TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

t_sndrel is an optional service of the transport provider. It is supported only if the transport provider returned T_COTS_ORD on **t_open** or **t_getinfo**. If **t_sndrel** is issued from an invalid state, the provider generates EPROTO; however, this error may not occur until the user makes a subsequent reference to the transport endpoint.

After calling **t_sndrel**, more data cannot be sent over the connection. However, a user can continue to receive data if an orderly release indication has not been received.

Blocking Information**t_sndrel** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

See Also

t_getinfo, **t_open**, **t_rcvrel**, AppleTalk Notes: t_sndrel, IPX/SPX/SPX II Notes: t_sndrel, TCP/IP and TCP/IPX Notes: t_sndrel

Example

t_sndrel

```
rc = t_sndrel(fd);
if (rc == -1)
{
   t_error("t_sndrel");
   exit(1);
}
while ((rc = t_rcvrel(fd)) == -1)
{
   if (t_errno ! = TNOREL)
   {
      t_error("t_rcvrel");
      exit(0);
   }
   ThreadSwitch();
}
```

t_sndudata

Sends a data unit in connectionless mode to another transport user
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_sndudata(
   int                   fd,
   struct t_unitdata    *unitdata);
```

Parameters

fd

(IN) Indicates the local transport endpoint through which data will be sent.

unitdata

(IN) Points to the data unit to be sent.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_sndudata** function returns 0 on successful completion and -1 on failure. On failure, it sets *t_errno* to one of the following:

TBADDATA	Illegal amount of data. Zero octets is not supported
TBADF	Specified file handle does not refer to a transport endpoint
TFLOW	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the function at this time

TLOOK	Asynchronous event occurred on the transport endpoint specified by *fd*
TNOTSUPPORT	Underlying transport provider does not support this function
TOUTSTATE	Function was issued in the wrong sequence on the transport endpoint specified by *fd*
TSYSERR	System error occurred during execution

Remarks

t_sndudata is used in connectionless mode to send a data unit to another transport user.

Blocking Information**t_sndudata** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

unitdata points to a t_unitdata structure, which is defined in TIUSER.H.

You can choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider can use default options.

If *len* of *udata* is 0, no data is passed to the transport provider.**t_sndudata** sends zero-length data units.

By default, **t_sndudata** operates in synchronous mode and can wait if flow control restrictions prevent the local transport provider from accepting data. Under such conditions, when O_NDELAY is set (by calling **t_open** or **fcntl**), **t_sndudata** executes in asynchronous mode and fails.

If **t_sndudata** is issued from an invalid state, or if the amount of data specified in *udata* exceeds TSDU size as returned in *tsdu* of *info* of **t_open** or **t_getinfo**, the provider generates an EPROTO protocol error. (See TSYSERR in the "Return Values" section.)

If the state is invalid, this error may not occur until the user makes a subsequent reference to the transport endpoint.

See Also

t_alloc, **t_open**, **t_rcvudata**, **t_rcvuderr**, AppleTalk Notes: t_sndudata, IPX/SPX/SPX II Notes: t_sndudata, OSI Notes: t_sndudata, TCP/IP Notes: t_sndudata

Example

t_sndudata

```
while ((rc = t_sndudata(fd, unitdata)) == -1)
{
   if (t_errno != TFLOW)
   {
      t_error("t_snd");
      exit(1);
   }
   ThreadSwitch();
}
```

t_sync

Synchronizes the data structures that the TLI library manages with information from the underlying transport provider

Local Servers: either blocking or nonblocking

Remote Servers: N/A

Platforms: DOS, NLM, OS/2, Win

Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_sync(
   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint where data structures are to be synchronized.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The **t_sync** function returns the state of the transport provider on successful completion and -1 on failure. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TSTATECHNG	Transport provider is undergoing a state change
TSYSERR	System error occurred during execution

Remarks

t_sync is a stub function in the NetWare environment. It returns the current state of the transport provider.

Blocking Informationt_sync blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

The states returned by **t_sync** can be one of the following:

State Name	Description
T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication)
T_INREL	Incoming orderly release (waiting for an orderly release request)

Services

Transport Level Interface (TLI)

See Also

t_open

t_unbind

Disables the transport endpoint that *fd* specifies
Local Servers: either blocking or nonblocking
Remote Servers: N/A
Platforms: DOS, NLM, OS/2, Win
Service: Transport Level Interface (TLI)

Syntax

```
#include <tiuser.h>
#include <tispxipx.h>

int t_unbind(

   int   fd);
```

Parameters

fd

(IN) Indicates the local transport endpoint to be disabled.

Return Values

0	Successful
-1	Failure (sets t_errno Code)

t_errno Values

The function **t_unbind** returns 0 if successful. Otherwise, it returns a value of -1. On failure, it sets *t_errno* to one of the following:

TBADF	Specified file handle does not refer to a transport endpoint
TOUTSTATE	Function was issued in the wrong sequence
TLOOK	Asynchronous event has occurred on this transport endpoint
TSYSERR	System error occurred during execution

Remarks

On completion of **t_unbind**, the transport provider accepts no further data or events destined for this transport endpoint. **t_bind** bound the transport endpoint previously.

Blocking Information**t_unbind** blocks if **t_open** was called in blocking mode or if **t_blocking** was called. It is nonblocking if **t_open** was called in nonblocking mode or if **t_nonblocking** was called.

See Also

t_bind (Function), AppleTalk Notes: t_unbind

Example

t_unbind

```
rc = t_unbind(fd);
if (rc == -1)
{
   t_error("t_unbind");
   exit(1);
}
```

TLI: Structures

netbuf

Contains information about a buffer
Service: TLI
Defined In: tiuser.h

Structure

```
struct netbuf {
   unsigned   maxlen;
   unsigned   len;
   char       *buf;
};
```

Fields

maxlen

Specifies the maximum size of the address buffer. If *maxlen* is not large enough to hold the returned address, an error results.

len

Specifies the number of bytes in the address.

buf

Points to the address buffer.

pollfd

Contains information used to poll the status of a transport endpoint
Service: TLI
Defined In: poll.h

Structure

```
struct pollfd {
   int     fd;
   short   events;
   short   revents;
   int   _ifd; /* For internal kernel use. Defined only in the NLM stru
};
```

Fields

fd

Specifies the file descriptor to be polled.

events

Contains a bitmask that contains the bitwise inclusive OR of events to be polled on that file descriptor.

revents

Indicates which of the requested events has occurred.

_ifd

For internal use only. Specifies a file descriptor to be used internally by the kernel. This field is defined only in the poll.h file for NLM applications.

Remarks

NOTE: The only platform for which the *_ifd* field is defined for is the NLM platform. To ensure that you define the correct structure for your applications, use the poll.h file for the platform on which your application will run.

Used by **poll**.

t_bind (Structure)

Contains information about a bound transport endpoint
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_bind {
   struct netbuf   addr;
   unsigned        qlen;
};
```

Fields

addr

Specifies a protocol address, and *qlen* indicates the maximum number of outstanding connect indications.

netbuf

Requests and returns information in netbuf.

t_call

Contains information about a caller
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_call {
   struct netbuf   addr;
   struct netbuf   opt;
   struct netbuf   udata;
   int             sequence;
};
```

Fields

addr

Contains the protocol address of the calling transport user.

opt

Contains protocol-specific information associated with the connection.

udata

Points to any user-defined data.

sequence

Contains the value returned by **t_listen** uniquely associating the response with a previously received connect indication.

This value is a number uniquely identifying the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

t_discon

Contains information about a disconnect
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_discon {
   struct netbuf   udata;
   int             reason;
   int             sequence;
};
```

Fields

udata

Identifies any user data that was sent with the disconnect.

reason

Specifies the reason for the disconnect through a protocol-dependent reason code.

sequence

Identifies an outstanding connect indication with which the disconnect is associated. *sequence* is meaningful only when **t_rcvdis** is issued by a passive transport user who has called **t_listen** one or more times and is processing the resulting connect indications.

Remarks

Used by **t_rcvdis**.

t_info

Contains information about a transport protocol address
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_info {
   long   addr;
   long   options;
   long   tsdu;
   long   etsdu;
   long   connect;
   long   discon;
   long   servtype;
};
```

Fields

addr

Contains information abou the size of a transport protocol address as follows:

> 0	Maximum size of a transport protocol address
-1	No limit on the address size
-2	Transport provider does not provide user access to transport protocol addresses

options

Contains information about the size of protocol-specific options supported by the provider as follows:

> 0	Maximum number of bytes options
-1	No limit on the option size
-2	Transport provider does not support user-settable options

tsdu

Contains information about Transport Service Data Units (TSDUs)

> 0	Maximum size of a TSDU

0	Transport provider does not support TSDUs, although it does support sending a data stream with no logical boundaries preserved across a connection
-1	No limit on the size of a TSDU
-2	Transport provider does not support transferring normal data

etsdu

Contains information about Expedited Transport Service Data Units (ETSDUs)

> 0	Maximum size of an ETSDU
0	Transport provider does not support ETSDUs, although it does support sending an expedited data stream with no logical boundaries preserved across a connection
-1	No limit on the size of an ETSDU
-2	Transport provider does not support transferring expedited data

connect

Contains information about the data sent during connection establishment.

> 0	Maximum amount of data associated with connection establishment functions
-1	No limit on the amount of data sent during connection establishment
-2	Transport provider does not allow data to be sent with connection establishment functions

discon

Contains information about the amount of data that can be associated with **t_snddis** and **t_rcvdis**.

> 0	Maximum amount of data that may be associated with these abortive release functions
-1	No limit on the amount of data sent with these abortive release functions
-2	Transport provider does not allow data to be sent with the abortive release functions

servtype

Contains information about whether the transport provider supports connection mode service and has an orderly release facility.

T_COTS	Transport provider supports a connection mode service but does not support the optional orderly release facility
T_COTS_ORD	Transport provider supports a connection mode service with the optional orderly release facility
T_CLTS	Transport provider supports a connectionless mode service. For this service type, **t_open** returns -2 for *etsdu*, *connect*, and *discon*

t_optmgmt (Structure)

Contains information about protocol options
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_optmgmt {
   struct netbuf   opt;
   long            flags;
};
```

Fields

opt

Identifies protocol options.

The options are represented by *netbuf* in a manner similar to the address in **t_bind**. *netbuf* is used to request and return information as follows (for **t_optmgmt (Function)**):

Fields	req	ret
maxlen	Has no meaning for *req*	Maximum size of the options buffer.
len	Number of bytes in the options.	Number of bytes returned.
buf	Pointer to the options buffer.	Pointer to the buffer where the options are to be placed.

flags

Specifies the action to take with those options, as follows:

Action	Description
T_NEGOTIATE	Negotiate the values of the options specified in *req* with the transport provider. The provider evaluates the requested options and negotiates the values, returning the negotiated values through *ret*.
T_CHECK	Verify whether the options specified in *req* are supported by the transport provider. On return, *flags* of *ret* has either T_SUCCESS or T_FAILURE

	set to indicate to the user whether the options are supported. These flags are only meaningful for T_CHECK.
T_DEFAULT	Retrieve the default options supported by the transport provider into *opt* of *ret*. In *req*, *len* of *opt* must be zero, and *buf* may be NULL.

t_uderr

Contains information about an erroneous data unit
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_uderr {
   struct netbuf   addr;
   struct netbuf   opt;
   long            error;
};
```

Fields

addr

Specifies the destination protocol address of the erroneous data unit.

opt

Identifies protocol-specific options associated with this data unit.

error

Specifies a protocol-dependent error code.

t_unitdata

Contains information about data being sent
Service: TLI
Defined In: tiuser.h

Structure

```
struct t_unitdata {
   struct netbuf   addr;
   struct netbuf   opt;
   struct netbuf   udata;
};
```

Fields

addr

Specifies the protocol address of the user.

opt

Identifies protocol-specific options associated with this data unit.

udata

Specifies the user data.

Communication White Papers

Communication White Papers: Guide

This manual contains white papers that present solutions to various specific NetWare® development challenges.

To use this information effectively, you should already be familiar with NetWare programming. For introductory material, see other manuals in this SDK. To locate the information you want, see the SDK Roadmap.

Currently, this manual contains the following white papers:

NetWare/IP Protocol Packet Definitions

Describes NetWare/IP packets.

Protocol Reference: ODI and STREAMS

Describes the Open Data-Link Interface™ (ODI™) architecture and how it relates to the OSI model.

The RCB Interface to the LAN WorkPlace TCP/IP Protocol Stack

Describes the Request Control Block (RCB) interface to the LAN Workplace® TCP/IP protocol stack.

Using the Low-Level RCB Interface to Support the NetWare/IP DOS/Windows Client v1.0

Describes how to emulate the RCB interface to support NetWare/IP™ clients.

Using the Low-Level RCB Interface to TCP/IP for UDP Communication

Describes how to use the RCB interface for UDP communication.

Chapter 51
NetWare/IP Protocol Packet Definitions

Novell® Engineering

NetWare/IP Protocol Packet Definitions: Guide

NetWare/IP™ (NWIP) software allows total or partial replacement of the IPX™ transport subsystem with the industry standard TCP/IP transport subsystem in a NetWare® network. The following constitute the core components of the NetWare/IP software:

NetWare/IP Server (NWIP) NLM™

Domain SAP/RIP Server (DSS) NLM

NetWare/IP Client executable

This paper documents the protocol, specifically the packet definitions, used for communication between these three components of the NetWare/IP software. The reader is expected to be familiar with the design and architecture of the software. This article does not document the packet structure for standard services used by NetWare/IP software (for example, DNS).

> **NOTE:** All packets described below are UDP packets unless otherwise noted.

See the following areas of information about NWIP.

Get NWIP Common Parameters from DSS

NWIP Server/DSS Communications

Primary DSS/Secondary DSS Communications

NWIP Client/DSS Communications

NWIP Client/NWIP Server Communications

Summary of NWIP Packet Fields

Database Version Number Query Packet (Type = 8)

Offset	Field	Size	Type
0	Packet Type (= 8)	2 bytes	high-low uword

Parent Topic: Database Synchronization Packets

Database Version Number Response Packet (Type = 9)

Offset	Field	Size	Type
0	Packet Type (= 9)	2 bytes	high-low uword
2	Version Number	2 bytes	high-low uword

If the NWIP server finds that the database version has changed, it initiates a TCP connection and requests a transfer. The NetWare/IP servers can request one of the following types of database transfers (D2N stands for transfer between a DSS and a NWIP server):

D2N Transfer Type	Request Packet Type	Response Packet Type
Delta Transfer without Checksum	259 (0x103)	261 (0x105)
Delta Transfer with Checksum	260 (0x104)	261 (0x105)
Full Transfer	257 (0x101)	258 (0x102)

The request and response packet structures for the transfer are shown below.

NOTE: Note that these packets are transferred as TCP data.

Parent Topic: Database Synchronization Packets

D2D Database Transfer Request Packet (Type = 273 or 275)

Offset	Field	Size	Type
0	Packet Type (see above)	2 bytes	high-low uword
2	Local DB Version Number	2 bytes	high-low uword
4	Local IP Address	4 bytes	high-low uword
8	Local IP Subnet Mask	4 bytes	high-low uword

Parent Topic: Primary DSS/Secondary DSS Communications

D2D Database Transfer Response Packet (Type = 274 or 276)

This packet also has three sections---header, SAP records, and RIP records. The packet is ordered so that the header is followed by zero or more SAP records, followed by zero or more RIP records. The 16-byte header contains the number of SAP and RIP records in the packet along with the total length of all three sections. The major difference between this packet and the corresponding D2N packet is that the SAP and RIP records in the D2D packet are no longer of fixed length, since each record can contain more than one IP address.

Header Section:

Offset	Field	Size	Type
0	Packet Type (= 274 or 276)	2 bytes	high-low uword
2	DSS DB Version Number	2 bytes	high-low uword
4	Total Entries in DB (checksum)	4 bytes	high-low ulong
8	Packet Flag	2 bytes	bit map
10	Total Packet Length	2 bytes	high-low uword
12	Number of SAP Records	2 bytes	high-low uword
14	Number of RIP Records	2 bytes	high-low uword

The bit definitions for the Packet Flag field are as follows (bit 0 = least significant bit):

Bit 0:	Set=> Database unchanged since last transfer.
Bit 1:	Set=> More D2D Database Transfer packets to follow. Reset=> End of transfer.
Bits 2 through 15:	Reserved, should be reset.

SAP Record & RIP Record:

The structure of the SAP and RIP record is similar to the corresponding records defined in D2D Database Upload Packet (Type = 277).

Parent Topic: Primary DSS/Secondary DSS Communications

D2D Database Upload Packet (Type = 277)

This packet has three sections---header, SAP records, and RIP records. The packet is ordered so that the header is followed by zero or more SAP records, followed by zero or more RIP records. The 16-byte header contains the number of SAP and RIP records in the packet and the total length of all three sections. The full upload of the database can require the transfer of multiple packets from the secondary to the primary DSS server. The major difference between this packet and the corresponding D2N packet is that the SAP and RIP records in the D2D packet are no longer of fixed length, as each record can contain more than one IP address.

Header Section:

Offset	Field	Size	Type
0	Packet Type (= 277)	2 bytes	high-low uword
2	Reserved Field	6 bytes	undefined
8	Packet Flag	2 bytes	bit map
10	Total Packet Length	2 bytes	high-low uword
12	Number of SAP Records	2 bytes	high-low uword
14	Number of RIP Records	2 bytes	high-low uword

The bit definitions for the Packet Flag field are as follows (bit 0 = least significant bit):

Bit 0:	Reserved, should be reset.
Bit 1:	Set=> More Upload packets to follow. Reset=> End of Upload.

Bits 2 through 15:	Reserved, should be reset.

SAP Record:

Each SAP record in the SAP record section of the D2D Database Transfer response packet is at least 82 bytes, and contains at least one IP address structure. Each additional IP address structure increases the record length by 17 bytes. Note: Some numeric fields in the SAP record are transferred in machine order (that is, the Intel x86 order (low-high)), as indicated below.

Offset	**Field**	**Size**	**Type**
0	Record Type (= 1)	2 bytes	low-high uword
2	Server Name	48 bytes	ASCII string (0 padded)
50	IPX Network	4 bytes	high-low ulong
54	IPX Node	6 bytes	high-low uword
60	IPX Socket	2 bytes	high-low uword
62	Server Type	2 bytes	high-low uword
64	IP Address Count	1 byte	uchar
65	IP Address	4 bytes	high-low ulong
69	IP Subnet	4 bytes	high-low ulong
73	Time to Live	1 byte	uchar
74	Intermediate Networks	1 byte	uchar
75	Reserved Field	2 bytes	undefined
77	Authoritative DSS's IP Addr	4 bytes	high-low ulong
81	Record Flag	1 byte	bit-map
	(Addition IP Addr Structs)		

The bit mappings for the Record Flag field are as follows (bit 0 = least significant bit):

Bit 0:	Set=> The reporting DSS is the authoritative source of the information.
Bit 1:	Reserved.
Bit 2:	Set=> Record is to be deleted. Bit 3 through 6: Reserved.
Bit 7:	Set=> RIP record. Reset=> SAP record.

RIP Record:

Each RIP record in the RIP record section of the D2D Database Transfer response packet is at least 24 bytes, and contains at least one IP address structure. Each additional IP address structure increases the length of the record by 17 bytes. Note: Some numeric fields in the RIP record are transferred in machine order (that is, the Intel x86 order (low-high)) as indicated below.

Offset	Field	Size	Type
0	Record Type (= 2)	2 bytes	low-high uword
2	IPX Network	4 bytes	high-low uword
6	IP Address Count	1 byte	uchar
7	IP Address	4 bytes	high-low ulong
11	IP Subnet	4 bytes	high-low ulong
15	Time to Live	1 byte	uchar
16	Intermediate Networks	1 byte	uchar
17	Ticks	2 bytes	low-high uword
19	Authoritative DSS's IP Addr	4 bytes	high-low ulong
23	Record Flag	1 byte	bit-map
	(Addition IP Addr Structs)		

The record flag bits are similar to those defined in the SAP Record section.

The secondary DSS server can request one of the following types of database transfers (downloads) from the primary DSS server:

D2D Transfer Type	Request Packet Type	Response Packet Type
Delta Transfer	275 (0x113)	276 (0x114)
Full Transfer	273 (0x111)	274 (0x112)

The request and response packet structures for the transfer are shown below. Note that these packets are transferred as TCP data.

Parent Topic: Primary DSS/Secondary DSS Communications

D2N Database Transfer Request Packet (Type = 259, 260, or 257)

Offset	Field	Size	Type
0	Packet Type (see above)	2 bytes	high-low uword
2	Local DB Version Number	2 bytes	high-low uword
4	Local IP Address	4 bytes	high-low uword
8	Local IP Subnet Mask	4 bytes	high-low uword

Parent Topic: Database Synchronization Packets

D2N Database Transfer Response Packet (Type = 261 or 258)

This packet has three sections---header, SAP records, and RIP records. The packet is ordered so that the header is followed by zero or more SAP records, followed by zero or more RIP records. The 16-byte header section contains the number of SAP and RIP records in the packet and the total length of all three sections. Each SAP record is 72 bytes and each RIP record 16 bytes.

Header Section:

Offset	Field	Size	Type
0	Packet Type (= 261 or 258)	2 bytes	high-low uword
2	DSS DB Version Number	2 bytes	high-low uword
4	Total Entries in DB (checksum)	4 bytes	high-low ulong
8	Packet Flag	2 bytes	bit map
10	Total Packet Length	2 bytes	high-low uword
12	Number of SAP Records	2 bytes	high-low uword
14	Number of RIP Records	2 bytes	high-low uword

The bit definitions for the Packet Flag field are as follows (bit 0 = least significant bit):

Bit 0:	Set=> Database unchanged since last transfer.
Bit 1:	Set=> More D2N DB Transfer packets to follow.

	Reset=> End of transfer.
Bits 2 through 15:	Reserved, should be reset.

SAP Record:

Offset	Field	Size	Type
0	Record Type (= 1)	2 bytes	high-low uword
2	Server Type	2 bytes	high-low uword
4	Server Name	48 bytes	ASCII string (0 padded)
52	IPX Network	4 bytes	high-low ulong
56	IPX Node	6 bytes	high-low uword
62	IPX Socket	2 bytes	high-low uword
64	Intermediate Networks	2 bytes	high-low uword
66	Record Flag	2 bytes	bit-map
68	Suitable IP Address	4 bytes	high-low ulong

RIP Record:

Offset	Field	Size	Type
0	Record Type (= 2)	2 bytes	high-low uword
2	Record Flag	2 bytes	bit-map
4	IPX Network	4 bytes	high-low ulong
8	Intermediate Networks	2 bytes	high-low uword
10	Ticks	2 bytes	high-low uword
12	Suitable IP Address	4 bytes	high-low ulong

Parent Topic: Database Synchronization Packets

Get NWIP Common Parameters from DSS

Configuration Information Exchange: The request packet originates from the NWIP server or client and the DSS server answers using the response packet. There are two ways to get NWIP parameters---simple and extended. The request/response packet types are the same, so the DSS server checks

the *Signature* field to identify requests. The NWIP server or client to similarly uses this field to identify responses. The exchange is done using the UDP/IP protocol.

See the following packet types:

NWIP Parameters Request Packet (Type = 12, Simple)

NWIP Parameters Response Packet (Type = 13, Simple)

NWIP Parameters Request Packet (Type = 12, Extended)

NWIP Parameters Response Packet (Type = 13, Extended)

Parent Topic: NetWare/IP Protocol Packet Definitions: Guide

IPX Emulation

For directed IPX packets or broadcast IPX packets that are not addressed to a SAP socket (452h) or RIP socket (453h), the NWIP client encapsulates the entire IPX packet (including 30 bytes of header and data) into a UDP packet. The destination UDP port is the first configured port (default 43981) described in NWIP Client/DSS Communications. A broadcast packet is addressed to the local IP subnet only rather than a network-wide broadcast.

This packet format is for both request and response:

Offset	Field	Size	Type
0	IPX Packet Header	30 bytes	low-high uchar
30	IPX Data Portion	n bytes	low-high uchar

Parent Topic: NWIP Client/NWIP Server Communications

Nearest Server Query

The NWIP client uses the following packet to identify a suitable service provider (for example, a file server) for the service it requires. An NWIP server responds with the response packet (type = 3). Note that the NWIP server returns the matched SAP information only when its Reply To Get Nearest Server flag is on.

See the following packet types:

NWIP Nearest Server Query Packet (Type = 2) (NWIP Client/NWIP

Server)

NWIP Nearest Server Response Packet (Type = 3) (NWIP Client/NWIP Server)

Parent Topic: NWIP Client/NWIP Server Communications

NWIP Client/DSS Communications

The communications between a NetWare/IP client and a DSS serves three functions:

Retrieval of configuration parameters

Querying for Nearest Servers

Registration of SAP Information

The following sections describe packet structures for each function.

NWIP Client/DSS Configuration Information Exchange

The client uses the NWIP Parameters Request Packet (type=12) and the DSS responds with NWIP Parameters Response Packet (type=13). These packets are defined in Get NWIP Common Parameters from DSS.

Nearest Server Query Packets

The NWIP client uses the following packet, if necessary, to determine a suitable service provider (for example, a file server) for the service it requires. The DSS responds with a response packet (type = 3). Note that the DSS returns the matched SAP information regardless of whether the Reply To Get Nearest Server flag is on or not.

See the following packet types:

NWIP Nearest Server Query Packet (Type = 2) (NWIP Client/DSS)

NWIP Nearest Server Response Packet (Type = 3) (NWIP Client/DSS)

SAP Registration

When a service (for example, Print Server) is provided by an NWIP client, the client must register it with the DSS. The SAP Information Update Packet and SAP Information Update Acknowledgment packet are used for this purpose. These packets are defined in SAP/RIP Registration Packets.

Parent Topic:

NetWare/IP Protocol Packet Definitions: Guide

NWIP Client/NWIP Server Communications

The communications between a NetWare/IP client and a NetWare/IP server serve three functions:

Querying for the nearest server

Querying SAP & RIP information

IPX emulation

There are two contiguous TCP/UDP ports (default to 43981, 43982) configured by the primary DSS. These are used for NWIP clients/servers to communicate with an NWIP server. The first port is for service queries (for example, SAP/RIP queries). The second port is for IPX emulation (for example, sending an NCP™ packet to the server).

The following topics describe packet structures for each function.

Nearest Server Query

SAP Information Query

RIP Information Query

IPX Emulation

Parent Topic: NetWare/IP Protocol Packet Definitions: Guide

NWIP Echo Request Packet (Type = 14 or 46)

Offset	Field	Size	Type
0	Packet Type (= 14 or 46)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Start Timer Ticks	2 bytes	low-high uword
8	IPX Network Number	4 bytes	high-low ulong
12	Number of Hops	2 bytes	high-low uword
14	Original Ticks	2 bytes	high-low uword

Parent Topic: RIP Echo with the NetWare/IP Server

NWIP Echo Response Packet (Type = 15 or 46)

Offset	Field	Size	Type
0	Packet Type (= 15 or 46)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Start Timer Ticks	2 bytes	low-high uword
8	IPX Network Number	4 bytes	high-low ulong
12	Number of Hops	2 bytes	high-low uword
14	Original Ticks	2 bytes	high-low uword

Parent Topic: RIP Echo with the NetWare/IP Server

NWIP Nearest Server Query Packet (Type = 2) (NWIP Client/DSS)

Offset	Field	Size	Type
0	Packet Type (= 2)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Server Type	2 bytes	high-low uword
6	Source IP Subnet Mask	4 bytes	high-low ulong

Parent Topic: Nearest Server Query Packets

NWIP Nearest Server Query Packet (Type = 2) (NWIP Client/NWIP Server)

Offset	Field	Size	Type
0	Packet Type (= 2)	2 bytes	high-low uword

2	Source IPX Socket	2 bytes	high-low uword
4	Server Type	2 byte	high-low uword
6	NWIP Domain Length	2 bytes	high-low uword
8	NWIP Domain Name	256 bytes	ASCII String

Parent Topic: Nearest Server Query

NWIP Nearest Server Response Packet (Type = 3) (NWIP Client/DSS)

The response packet from DSS has a fixed length header followed by one or more 64-byte server records. The number of server records is specified in the Record Count field of the header.

Offset	Field	Size	Type
0	Packet Type (= 3)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Record Count	2 bytes	high-low uword
6	Server #1: IP Address	4 bytes	high-low ulong
10	Server #1: Ticks	2 bytes	high-low ushort
12	Server #2: IP Address	4 bytes	high-low ulong
16	Server #2: Ticks	2 bytes	high-low ushort
18	Server #3: IP Address	4 bytes	high-low ulong
22	Server #3: Ticks	2 bytes	high-low ushort
24	Server #4: IP Address	4 bytes	high-low ulong
28	Server #4: Ticks	2 bytes	high-low ushort
30	Server #5: IP Address	4 bytes	high-low ulong
34	Server #5: Ticks	2 bytes	high-low ushort
36	Server Type	2 bytes	high-low uword
38	Server Name	48 bytes	ASCII string (0 padded)
86	IPX Network	4 bytes	high-low ulong
90	IPX Node	6 bytes	high-low uword
96	IPX Socket	2 bytes	high-low uword
98	Intermediate Networks	2 bytes	high-low uword

100	Next Server Record	64 bytes	

Parent Topic: Nearest Server Query Packets

NWIP Nearest Server Response Packet (Type = 3) (NWIP Client/NWIP Server)

The response packet from a NWIP server has a fixed-length header followed by exactly one 64-byte server record. The number of server records is specified in the Record Count field of the header.

Offset	Field	Size	Type
0	Packet Type (= 3)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Record Count (= 1)	2 bytes	high-low uword
6	Server IP Address	4 bytes	high-low ulong
10	Server Ticks	2 bytes	high-low ushort
12	Not Used	24 bytes	N/A
36	Server Type	2 bytes	high-low uword
38	Server Name	48 bytes	ASCII string (0 padded)
76	IPX Network	4 bytes	high-low ulong
80	IPX Node	6 bytes	high-low uword
86	IPX Socket	2 bytes	high-low uword
88	Intermediate Networks	2 bytes	high-low uword

Parent Topic: Nearest Server Query

NWIP Parameters Request Packet (Type = 12, Extended)

Offset	Field	Size	Type
0	Packet Type (= 12)	2 bytes	high-low uword

2	Sequence Number	2 bytes	high-low uword
4	Signature (0x4F574950)	4 bytes	high-low ubyte
8	Number of Additional Params	2 bytes	high-low uword
10	Additional Params IDs	2 bytes	low-high uword

Additional Params IDs at offset 10 is the first additional parameter ID. This can be followed by other IDs depending on the Number of Additional Params at offset 8.

So far there are only two additional parameter ID requests defined:

Get NWIP Domain Name	0001
Get NWIP Protocol Version number	0003

Parent Topic: Get NWIP Common Parameters from DSS

NWIP Parameters Request Packet (Type = 12, Simple)

Offset	Field	Size	Type
0	Packet Type (= 12)	2 bytes	high-low uword
2	Sequence Number	2 bytes	high-low uword

Parent Topic: Get NWIP Common Parameters from DSS

NWIP Parameters Response Packet (Type = 13, Extended)

Offse	Field	Size	Type

t			
0	Packet Type (= 13)	2 bytes	high-low uword
2	Sequence Number	2 bytes	high-low uword
4	IPX Network Number String	8 bytes	string
12	IPX Network Number	4 bytes	high-low ulong
16	NetWare/IP Port Number	2 bytes	low-high uword
18	Checksum Usage Flag	2 bytes	zero = no, else = yes
20	DB Sync. Interval (in secs)	2 bytes	low-high uword
22	Max. UDP Re transmissions	2 bytes	low-high uword
24	Signature (0x4F574950)	4 bytes	high-low ubyte
28	Number of Additional Params	2 bytes	high-low uword
30	Param ID	2 bytes	high-low uword
32	Param Length	2 bytes	high-low uword
34	Param Value	n bytes	string

Additional Params IDs at offset 30 is the first additional parameter value/data structure. This can be followed by other IDs depending on the Number of Additional Params at offset 28.

So far there are only two additional parameter ID responses defined:

Get NWIP Domain Name	0001	Response is a length-preceded string
Get NWIP Protocol Version number	0003	Response is an unsigned long in network order. The most significant 2 bytes indicate the major version number. The least significant 2 bytes indicate the minor version number

Parent Topic:

Parent Topic: Get NWIP Common Parameters from DSS

NWIP Parameters Response Packet (Type = 13, Simple)

Offset	Field	Size	Type
0	Packet Type (= 13)	2 bytes	high-low uword
2	Sequence Number	2 bytes	high-low uword
4	IPX Network Number String	8 bytes	string
12	IPX Network Number	4 bytes	high-low ulong
16	NetWare/IP Port Number	2 bytes	low-high uword
18	Checksum Usage Flag	2 bytes	zero = no, else = yes
20	DB Sync. Interval (in secs)	2 bytes	low-high uword
22	Max. UDP Re transmissions	2 bytes	low-high uword

Parent Topic: Get NWIP Common Parameters from DSS

NWIP RIP Query Packet (Type = 16)

Offset	Field	Size	Type
0	Packet Type (= 16)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	IPX Network Number	4 bytes	high-low ulong

Parent Topic: RIP Information Query

NWIP RIP Response Packet (Type = 17)

Offset	Field	Size	Type
0	Packet Type (= 17)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	IP address	4 bytes	high-low ulong
8	IPX Network Number	4 bytes	high-low ulong
12	Intermediate Networks	2 bytes	high-low uword
14	Ticks	2 bytes	high-low uword

Parent Topic: RIP Information Query

NWIP SAP Query Packet (Type = 18)

Offset	Field	Size	Type
0	Packet Type (= 18)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	SAP Type	2 bytes	high-low uword

Parent Topic: SAP Information Query

NWIP SAP Response Packet (Type = 19)

Offset	Field	Size	Type
0	Packet Type (= 19)	2 bytes	high-low uword
2	Source IPX Socket	2 bytes	high-low uword
4	Record Count	2 bytes	high-low uword
6	SAP Type	2 bytes	high-low uword
8	Server Name	48 bytes	ASCII string (0 padded)
56	IPX Network	4 bytes	high-low ulong

60	IPX Node	6 bytes	high-low uword
66	IPX Socket	2 bytes	high-low uword
68	Intermediate Networks	2 bytes	high-low uword
70	Next Server Record	64 bytes	

Parent Topic: SAP Information Query

NWIP Server/DSS Communications

The communications between a NetWare/IP server and a DSS serves three functions:

Retrieval of configuration parameters

Registration of SAP and RIP information

SAP/RIP database synchronization

NWIP has obtained a unique TCP/UDP port from the Assigned Number Authority. That port, 396 decimal, is the well-known port for DSS to serve other NWIP entities.

The following sections describe packet structures for each function.

SAP/RIP Registration Packets

The update packets originate from the NWIP server and the DSS server acknowledges their receipts using the corresponding acknowledgment packets. The exchange is done using the UDP/IP protocol.

See the following packet types:

SAP Information Update Packet (Type = 1)

SAP Information Update Acknowledgment Packet (Type = 7)

RIP Information Update Packet (Type = 5)

RIP Information Update Acknowledgment Packet (Type = 6)

Database Synchronization Packets

NWIP servers use a UDP packet to ascertain whether their local copy of the SAP/RIP database needs to be updated with respect the database of their DSS. This is done using the Version Number Query packet. The DSS responds with the version number of its database, using the Version

Number Response packet.

See the following packet types:

Database Version Number Query Packet (Type = 8)

Database Version Number Response Packet (Type = 9)

D2N Database Transfer Request Packet (Type = 259, 260, or 257)

D2N Database Transfer Response Packet (Type = 261 or 258)

Parent Topic:

NetWare/IP Protocol Packet Definitions: Guide

Primary DSS/Secondary DSS Communications

The periodic communication between a secondary DSS and its primary DSS is meant to synchronize the SAP/RIP databases on the two servers.

The secondary DSS server initiates a TCP connection with the primary DSS server and uploads any new or changed information which has been reported to it. Following that, the secondary DSS requests a download of new/changed information from the primary. The primary responds with the updated information. Finally, the TCP connection is closed by the secondary DSS server. (Note: D2D stands for transfer between a secondary DSS and the primary DSS servers.)

See the following packet types:

D2D Database Upload Packet (Type = 277)

D2D Database Transfer Request Packet (Type = 273 or 275)

D2D Database Transfer Response Packet (Type = 274 or 276)

Parent Topic: NetWare/IP Protocol Packet Definitions: Guide

RIP Echo with the NetWare/IP Server

The NWIP client uses either an NWIP echo or UDP echo packet to send an echo structure to the NetWare/IP server for two purposes: to validate the target NetWare/IP server is alive, and to calculate the round trip time from the client to the target server.

Depending on whether the NWIP1_1 COMPATIBILITY parameter in net.cfg is OFF (default for v2.1) or ON (v1.1), the client sends the echo

packet to the second dynamic port (for example, 43982) owned by the NetWare/IP server, or the UDP echo port (7) owned by TCP/IP respectively. The echo request type is 14 and response type is 15 when the echo port is 43982. Both the echo request and response types are 46 when the echo port is 7.

See the following packet types:

NWIP Echo Request Packet (Type = 14 or 46)

NWIP Echo Response Packet (Type = 15 or 46)

Parent Topic: NWIP Client/NWIP Server Communications

RIP Information Query

The NWIP client uses the following packets to obtain the RIP information that matches the given IPX Network Number.

NWIP RIP Query Packet (Type = 16)

NWIP RIP Response Packet (Type = 17)

Parent Topic: NWIP Client/NWIP Server Communications

RIP Information Update Acknowledgment Packet (Type = 6)

Offset	Field	Size	Type
0	Packet Type (= 6)	2 bytes	high-low uword
2	Sequence Number	2 bytes	high-low uword

Parent Topic: SAP/RIP Registration Packets

RIP Information Update Packet (Type = 5)

Each of these packet can contain information about multiple routes. The record count field is initialized with the number of routes. The first RIP record starts after the 16-byte packet header, followed by subsequent

records. Each RIP record is 12 bytes.

Offset	Field	Size	Type
0	Packet Type (= 5)	2 bytes	high-low uword
2	Reserved Field	2 bytes	not defined
4	Sequence Number	2 bytes	high-low uword
6	Record Count	2 bytes	high-low uword
8	Source IP Address	4 bytes	high-low ulong
12	Source IP Subnet Mask	4 bytes	high-low ulong
16	IPX Network	4 bytes	high-low ulong
20	Intermediate Networks	2 bytes	high-low uword
22	Ticks	2 bytes	high-low uword
24	Record Flag	2 bytes	high-low bit map
26	Reserved	2 bytes	not defined
28	Next RIP Record	12 bytes	

The bit definitions for the Record Flag are similar to those defined for SAP Info Update packet.

Parent Topic: SAP/RIP Registration Packets

SAP Information Query

The NWIP client uses the following packets to obtain the SAP information that matches the given SAP type.

NWIP SAP Query Packet (Type = 18)

NWIP SAP Response Packet (Type = 19)

Parent Topic: NWIP Client/NWIP Server Communications

SAP Information Update Acknowledgment Packet (Type = 7)

Offset	Field	Size	Type
0	Packet Type (= 7)	2 bytes	high-low uword
2	Sequence Number	2 bytes	high-low uword

Parent Topic: SAP/RIP Registration Packets

SAP Information Update Packet (Type = 1)

Each of these packets can contain information about multiple services. The record count field is initialized with the number of services. The first record starts after the 16-byte packet header, followed by subsequent records. Each service record is 66 bytes.

Offset	Field	Size	Type
0	Packet Type (= 1)	2 bytes	high-low uword
2	Reserved Field	2 bytes	not defined
4	Sequence Number	2 bytes	high-low uword
6	Record Count	2 bytes	high-low uword
8	Source IP Address	4 bytes	high-low ulong
12	Source IP Subnet Mask	4 bytes	high-low ulong
16	Server Type	2 bytes	high-low uword
18	Server Name	48 bytes	ASCII string (0 padded)

66	IPX Network	4 bytes	high-low ulong
70	IPX Node	6 bytes	high-low uword
76	IPX Socket	2 bytes	high-low uword
78	Intermediate Networks	2 bytes	high-low uword
80	Record Flag	2 bytes	high-low bit map
82	Next Service Record	66 bytes	

The bit definitions for the Record Flag field are as follows (bit 0 = least significant bit):

Bit 0:	Reserved, should be reset.
Bit 1:	New Record
Bit 2:	Changed Record
Bit 3:	Changed Record due to retargeting by the NWIP server
Bit 4:	Periodic Identification Information
Bits 5 & 6:	Reserved, should be reset.
Bit 7:	Server/Route Unavailable. Delete Record
Bits 8 through 15:	Reserved, should be reset.

Parent Topic: SAP/RIP Registration Packets

Summary of NWIP Packet Fields

The following table lists all the field types used in the packet definitions. Each field is described in the next column.

Authoritative DSS's IP address	The address identifying the DSS where the service/route was originally registered. (0 => Primary DSS).
Checksum Usage Flag	A boolean field to turn on/off use of UDP checksums.
DB Sync. Interval	The user-tunable database

DB Sync. Interval	The user-tunable database synchronization interval for use by all major components of NetWare/IP.
DSS DB Version Number	Local database's version number maintained by DSS servers.
Intermediate Networks	The number of intermediate nodes reported by the IPX protocol.
IPX Network	The IPX network associated with the SAP service record.
IPX Network Number	A number identifying the virtual IPX network created by NWIP.
IPX Network Number String	ASCII representation of the NWIP virtual IPX network number.
IPX Node	The IPX node number associated with the SAP service record
IPX Socket	The IPX socket number associated with the SAP service record.
Local DB Version Number	Local SAP/RIP database's version number maintained by NWIP Servers.
Max. UDP Retransmissions	The user-tunable parameter that indicates how often to retransmit a UDP packet before assuming the other side is down.
NetWare/IP Port Number	First of two consecutive user defined dynamic UDP port numbers to be used by NWIP.
Number of RIP Records	The count of the RIP records in the current packet.
Number of SAP Records	The count of the SAP records in the current packet.
Packet Type	NetWare/IP Protocol defined value.
Record Count	The count of records included in the variable length part of a packet.
Sequence Number	A identifier used to match responses with requests.
Server Name	ASCII NULL-terminated name.
Server Type	Novell supplied SAP service identifier.
Source IP Address	IP address of the dialog initiating computer.
Source IP Socket	IP socket being used by the sender, where a response should be directed.
Source IP Subnet Mask	IP Subnet Mask of the dialog initiating computer.
Suitable IP Address	IP address of the NWIP server. When an IPX node can be reached through multiple

	NWIP gateways, this field contains the "closest" gateway.
Ticks	The number of ticks as reported by IPX RIP.
Time to Live	An internal parameter in the database, used for timing out stale records.
Total Entries in DB	Total number of SAP and RIP records in the DSS database. Used for double-checking the integrity of the replicated database.

Parent Topic: NetWare/IP Protocol Packet Definitions: Guide

Protocol Reference: ODI and STREAMS

Novell®, Inc.

ODI and STREAMS: Guide

The development of the Open Data-Link Interface™ (ODI™) architecture is an important advance for interoperability. Before you can really take advantage of ODI, you need some background information. This paper presents information relating to the two sides of ODI: server and client.

The Open Data-Link Interface

The Server Side of ODI

Protocol ID Information

The Open Data-Link Interface

ODI is an open data-link architecture that can provide your organization with critical network interoperability. ODI significantly reduces the cost of ownership by providing the ability to protect investments in installed hardware, integrate new resources and technologies into the network, and reduce the expenses associated with support and noninteroperability. ODI integrates the network media of choice with the network transport of choice.

Related Topics

Can We Talk?

Constantly Inconstant

Meet ODI

ODI, a Path to Your Future

ODI on Your Server

ODI on Your Workstation

What ODI Really Means to You

A Case in Point

Parent Topic: ODI and STREAMS: Guide

Can We Talk?

The Open Systems Interconnection (OSI) model is a high-level overview that provides direction on how computers should communicate. This model is composed of seven layers including Application, Presentation, Session, Transport, Network, Data-link, and Physical. Each layer defines the process required for an application on one computer to send a message over a network to an application on another computer.

It is important to note that the OSI model largely explains **what** should happen, not necessarily **how** it should happen. Many vendors use the OSI model as a source of reference in their networking implementations.

Generally, the communication process can be divided into two parts: the process of defining and formatting the message, and the mechanics of actually delivering the message. Layers one and two of the OSI model (Application and Presentation) address the issue of creating, formatting and presenting the message. OSI layers three through seven (Session, Transport, Network, Data-Link, and Physical) deal with the mechanics of delivering the message. Computers can communicate only to the extent that they share the same conventions at all levels of the communication process. In other words, each computer system must have a common interface at each of the seven layers of the OSI reference model.

Since many vendors use the OSI model as a reference, a more standardized way of defining some of the actual processes needs to occur before interoperability between systems can happen. The Institute of Electrical and Electronic Engineering (IEEE) addressed this need and created a body of standards which defined, in part, "how the message is delivered". These standards were authored and formalized by the 802 committee.

The 802 committee created several standards that focused on the Data-Link layer of the OSI reference model and defined the access methods to the media (network cabling). Ethernet (802.3) and token ring (802.5) are two well-known works of the 802 committee. Each standard governs how computers take control of the communication link for transmitting messages. However, keep in mind that the Data-Link layer is only one of seven layers in the OSI reference model. The 802 committee did not address other layers of the OSI model, particularly the layers that govern the actual delivery of the message (that is, OSI layers 3 through 7.)

Many of the communication protocols that govern the transmission of network messages (layer 3 of the OSI model) are not as well-defined. In fact, the industry relies heavily on **de facto** standards (that is, standards that generally arise from sheer popularity). For example, IPX/SPX™ (popularized by Novell®), TCP/IP (popularized by UNIX* workstations), AppleTalk* (popularized by Apple*) and SNA (popularized by IBM*) have become some of today's **de facto** transport standards. It is interesting to note

that some of these transports were developed by assuming a specific access protocol and media; therefore, they are generally not available over other access protocols and media. For example, TCP/IP is traditionally limited to Ethernet and SNA is traditionally limited to token ring.

Parent Topic: The Open Data-Link Interface

Constantly Inconstant

So how can there be so much confusion with so many standards? Keep in mind that the OSI reference is mostly used as a model when networking vendors build their various solutions. Also keep in mind that none of the standards discussed so far define an interface between access protocols and transport protocols. In other words, we know how to get data on and off the wire, but how do we interface with the other protocols (layers 3 through 7 of the OSI model) required for communication?

This is exactly the problem network adapter vendors face when writing driver software that allows their adapters to operate with various systems. The driver is the software that links the hardware with the other software components in the computer. Specifically, the driver implements the functionality defined in both the OSI Data-Link layer (layer 2) and the IEEE 802 specifications.

However, since there is no clear interface defined between the Data-Link layer and the Network layer (layers 2 and 3), network adapter vendors are forced to write drivers for each adapter and for each network transport. The result is a "monolithic" driver, like IPX.COM, that tightly couples a given transport to the driver. This tight coupling reduces flexibility.

In addition, monolithic drivers make it difficult to concurrently support multiple network transports with a single adapter. For example, workstations with monolithic drivers can not easily accessNetWare® servers and IBM hosts at the same time.

What is needed is a clearly defined interface between the driver (Data-Link layer) and the network transport. This interface would make it easier for network adapter vendors to write**generic** drivers and make it easier for users to add flexibility and extensibility to their networks.

Parent Topic: The Open Data-Link Interface

Meet ODI

ODI (Open Data-Link Interface) is a data-link specification jointly developed by Apple Computer and Novell and published to the networking industry in 1989. The strategic goal of ODI is to provide seamless network integration at the network transport level. ODI simplifies

the development, and ultimate availability, of network drivers for a wide variety of network adapters and network transport protocol stacks. The result is easier access to a wide variety of networked resources without requiring multiple network connections or additional investments in hardware and software.

ODI offers several advantages over the earlier Microsoft*/3Com* NDIS (Network Device Interface Specification) architecture. NDIS primarily provided a standard interface between the Physical layer and the Data-Link layer (OSI MAC layer) and also allowed concurrent support of multiple network protocols. While ODI provides these same benefits, ODI also offers true media independence. Unlike ODI, NDIS protocol stacks are tightly coupled to a particular media type much as monolithic drivers. With ODI, software, like network protocols and applications, can now be totally unaware of the media in place. The enormous benefit of this feature is explained later in greater detail.

ODI is also more modular and more focused on performance than NDIS. The modularity of ODI makes it easier to manage and provides a cleaner migration path to the standards of tomorrow. The focus of ODI on performance means access to network resources is fast and transparent.

ODI is fundamental to network integration by allowing the mix and match of popular media and communication protocol suites on the network. This allows the server, in part, to simultaneously support heterogeneous workstations, and provide for the routing of data between network segments. ODI also allows workstations to establish concurrent "any-to-any" connections with available network resources.

Parent Topic: The Open Data-Link Interface

ODI, a Path to Your Future

ODI provides a smooth migration path when integrating either new protocols or new media. ODI was designed to transparently support any network transport regardless of the underlying media. This feature has far-reaching benefits. For example, ODI provides TCP/IP support, which is traditionally limited to Ethernet, over ARCnet* or token ring. This means that you can select any transport like OSI TP4 without regard to present topologies. Conversely, ODI preserves your investment in existing protocols and applications when migrating to new media. ODI gracefully integrates FDDI, wireless technology, or other media types into your present network.

Parent Topic: The Open Data-Link Interface

ODI on Your Server

ODI is the architectural platform that provides support for heterogeneous

workstations and facilitates multiprotocol routing. ODI allows servers to natively support workstations using AppleTalk (Macintosh*), TCP/IP (Sun*, NeXT*, HP*, or UNIX workstations), IPX/SPX (DOS, OS/2, Windows), or SNA (IBM host to terminal) communication protocols. For example, ODI provides transport support so that a Macintosh can use a NetWare server to queue and print documents, save data files that are shareable with other types of workstations, and access larger hosts through communication links. In addition, ODI provides crucial multiprotocol routing functions. This routing allows workstations to access needed resources on network segments other than the one that they are physically on.

Parent Topic: The Open Data-Link Interface

ODI on Your Workstation

ODI allows transparent workstation connections to a wide variety of network resources. This allows for one-to-many or many-to-many connection possibilities. For example, a workstation can concurrently access services from a NetWare server as well as from a UNIX host.

ODI is a modular solution consisting of a hardware driver for each network adapter, a link management layer and one or more protocol stacks like IPX/SPX or TCP/IP. Each piece is dynamically configurable. This eliminates the need to pre-link special drivers using special utilities like Novell's WSGEN program. ODI also makes it easier to use workstation memory more effectively. Any piece of the ODI platform (driver, link manager, or protocol stack) can be unloaded to free valuable memory when needed. The end result is a smaller, more flexible network interface.

Parent Topic: The Open Data-Link Interface

What ODI Really Means to You

Many of the benefits of ODI result from Novell's networking leadership. ODI is a robust architecture that allows over eight million NetWare users to easily interoperate with other resources on the network. ODI provides more flexibility in configuring the network interface, uses resources more effectively and preserves investments in hardware and other network resources. All of these benefits save money.

There are a number of ODI benefits that have more to do with service than with technology. ODI is vital to an integrated networking strategy. Supporting that strategy is a network of people, programs and services that include education, certification and support. ODI developers have access to expert engineering support and development tools. Novell's certification program ensures that NetWare ODI drivers and protocol stacks perform under demanding environments. In addition, there are over 4,500 Certified

Novell Engineers[SM] (CNEs[SM]), strategic NetWare support organizations, and Novell's authorized resellers.

Novell is committed to the ODI standard along with key third-party developers. These vendors---IBM, Compaq*, 3Com, Standard Microsystems, Western Digital*, Racal* Interlan, Tiara, Hewlett-Packard*, Proteon*, Ungermann-Bass*, and others---are currently developing or distributing ODI drivers for their network adapters. Protocol stacks that are currently available include IPX/SPX, TCP/IP, AppleTalk, OSI TP4, DEC* LAT*, and more.

Parent Topic: The Open Data-Link Interface

A Case in Point

Brigham Young University (BYU) has built its information system around various computing platforms that include DEC VAX* machines, IBM mainframes, UNIX workstations, and PCs networked with NetWare. The NetWare networks are interconnected using twisted pair Ethernet and StarLAN. There are approximately 70 NetWare networks tied together along with the Campus hosts by means of a TCP/IP backbone.

BYU has standardized on SMTP-based e-mail and regularly uses FTP and Telnet to access resources from VAX and IBM host systems. To provide this level of functionality, BYU pioneered IPX and TCP/IP coexistence on the PC by using "packet driver" technology and by writing their own version of IPX that operate with these packet drivers. BYU then licensed their IPX implementation to a commercial developer for resale.

Soon after ODI and LAN Workplace® for DOS were released, BYU reevaluated their IPX and TCP/IP coexistence strategy and replaced their packet drivers with ODI and their public domain TCP/IP-based tools with LAN Workplace for DOS. By doing so, BYU now relies on a single vendor, Novell, for timely maintenance and support of their network drivers and protocol stacks. In addition, they are better able to manage their networks themselves.

> **NOTE:** Packet Drivers are public domain NIC drivers that are written to a specification developed by FTP software.

Parent Topic: The Open Data-Link Interface

The Server Side of ODI

The ODI specification allows media-independent communications. Because of this, the specification, along with the STREAMS environment and NLM programs, makes NetWare 3.11 a true server platform on which you can build a customized system. The following items are discussed:

Layers

Packet Transmission

Protocol ID Information

Related Topics

Layers

Packet Transmission

MLID

LSL

Parent Topic: ODI and STREAMS: Guide

Layers

With ODI, NetWare 3.11 supports a large number of LAN adapters (network boards), and can receive or send a variety of communication protocols on the same adapter concurrently (that is, IPX/SPX, TCP/IP, AppleTalk). The implementation of ODI is perhaps best explained as several layers through which communications arriving at or departing from the server, router, or workstation must travel. The layers, as shown in the following graphic, include the LAN adapter and Multiple Link Interface Driver™ (MLID™) layer, the Link Support Layer™ (LSL™), and the Protocol Stack Layer. From the Protocol Stack layer, communication packets can access the NetWare OS.

Parent Topic: The Server Side of ODI

Packet Transmission

The role of ODI can be further illustrated through the process of packet transmission. The following is an explanation of packet transmission on a network, first without ODI, then with it.

Without ODI (for example, on a NetWare 2.x network), a workstation LAN adapter sends an IPX packet to another workstation where it is received by a LAN adapter. The LAN driver passes the packet to IPX, which either passes it to higher levels or routes it down through another LAN adapter to another network. This process permits only one protocol packet per media at a time because the LAN adapter driver is written to identify and accept only a specific type of packet, such as an IPX packet, an AppleTalk packet, or a TCP/IP packet.

The NetWare implementation of ODI, however, supports a number of

The NetWare implementation of ODI, however, supports a number of protocols on each media or LAN adapter. On a NetWare 3.11 network, with ODI, a workstation LAN adapter sends any kind of packet (IPX, AppleTalk, TCP/IP, and so on) to another workstation where it is received by a LAN adapter. The LAN driver for that adapter, unlike the LAN driver for the NetWare 2.x network, can accept any type of packet. Thus, it is called a Multiple Link Interface Driver (MLID). This MLID passes the packet to the next layer in the Open Data-Link Interface, the Link Support Layer (LSL).

The LSL is responsible for identifying the type of packet it receives and passing it to the appropriate protocol in the next higher layer. This next higher layer is called the Protocol Stack layer, and it contains any number of protocol stacks such as IPX, AppleTalk, and TCP/IP. Once a communication packet arrives at its specified protocol stack, it is either passed to higher levels or routed back down through the LSL to an MLID and to a specified LAN adapter which represents another network.

Parent Topic: The Server Side of ODI

MLID

As the section on packet transmission points out, the NetWare MLID is different from a previous NetWare driver. An MLID accepts multiple protocol packets rather than just one packet as previous LAN drivers do. The MLID does not interpret a packet after receiving it, but simply copies identification information from the packet into a receive ECB (Event Control Block) and passes the ECB to the LSL. Likewise, when sending out a packet, the MLID simply copies identification information from a send ECB into the packet and sends it out.

Parent Topic: The Server Side of ODI

LSL

The LSL is a support and a link through which multiple protocol packets travel as they go from the LAN adapter with its accompanying MLID to a designated protocol stack. The LSL acts as a type of switchboard to route packets between LAN adapters with their MLID layers and protocol stacks. As such, it must contain information about these two layers. Thus, the LSL contains a data segment (OSData) that maintains LAN adapter information, Protocol Stack information, binding information, and ECB information.

The LSL assigns each LAN adapter a logical number and maintains information about each LAN adapter. This information is registered with the LSL at load time.

Although an MLID can be configured to send and receive packets for more than one protocol stack, the LSL sees beneath it only LAN adapters,

associating with each adapter a block of data, a send routine, and a control routine. Although LAN adapters may have the same send and control routines (driven by the same driver), this does not matter to the LSL since each LAN adapter always has its own block of data.

The LSL also assigns each protocol stack a logical protocol stack number (up to 16 stacks are supported) and maintains information about each one. As with the LAN adapter information, this protocol stack information is registered with the LSL at load time.

In addition to the information about the LAN adapters below it and the protocol stacks above it, the LSL needs information from the ECBs that come from either direction, depending on whether they are send or receive ECBs. The LSL uses the packet ID information in the ECBs and the other information about LAN adapters below it and protocol stacks above it to route packets from LAN adapters to protocol stacks and from protocol stacks to LAN adapters.

Finally, the LSL must also have a set of routines to support the drivers that are below and a set of routines for the protocol stacks above. Using this information, the LSL is able to act as a switchboard, coordinating protocol stack/MLID interaction and packet movement within the server.

Each network station (server, router, OS/2 workstation, DOS workstation) has its own implementation of an LSL. Novell has created an LSL for NetWare 3.11 as well as for other NetWare versions. Currently, the NetWare 3.11 server LSL supports up to 16 MLID layers and up to 32 protocol stacks.

Parent Topic: The Server Side of ODI

Protocol ID Information

Every packet on a network that supports multiple protocols is made up of the following components:

A communications protocol packet such as IPX, TCP/IP, or AppleTalk.

A media envelope.

A globally administered value (1 to 6 bytes) called a Protocol ID (PID) located inside the media envelope.

As its name indicates, the PID that is located in every packet is a label that identifies two things about the packet:

The packet contains a certain communication protocol header.

The packet contains a certain envelope frame.

For example, an IPX header and an Ethernet II envelope have a PID of 8137h. However, another protocol packet on another medium (for example,

TCP/IP on ARCnet) might have the identical PID of 8137h. PID values are not unique across all media. The LSL uses the PID value to route incoming packets to the proper protocol stack.

Related Topic: STREAMS Interface

Parent Topic: ODI and STREAMS: Guide

STREAMS Interface

STREAMS is a set of tools consisting of system calls, kernel resources, and kernel utility routines. These tools are used to create, use, or dismantle a stream---a full-duplex processing and data transfer path between the driver on the low kernel end of the NetWare OS and the user application on the high end. (The NetWare implementation of STREAMS was developed from STREAMS, available on the UNIX System V™ architecture. For more detailed information on STREAMS, see the *STREAMS Primer* and the *STREAMS Programmers Guide*.)

A stream defines the interface for character I/O within the kernel and between the kernel and the rest of the OS. It is made up of three basic parts:

The stream head that interfaces with user applications

Modules (optional) used to process data traveling between the stream head and the stream end

The stream end or driver providing an external interface by means of a character I/O device

The Transport Level Interface (TLI) is an API that sits between STREAMS and the user application, providing an interface with transport level protocols such as SPX.

NetWare 3.11 implements STREAMS as a number of NLM programs. One NLM includes the STREAMS application interface routines, the utility routines for STREAMS modules, the log device, and an ODI driver. The utility routines in the NLM include all common functions used by STREAMS modules.

Other NLM programs include a number of STREAMS modules to provide communication protocols such as IPX/SPX or TCP/IP.

Parent Topic: Protocol ID Information

The RCB Interface to the LAN WorkPlace TCP/IP Protocol Stack

Novell® Engineering

RCB Interface: Guide

ACCEPT, BIND, CONNECT, GETPEERNAME, GETSOCKNAME, GETMYIPADDR, and GETSUBNETMASK

```
rcb_addr  STRUC
   RCB_addr_cmn   DB   (SIZE rcb_common) DUP (?)
   RCB_port       DW   ?
   RCB_address    DD   ?
rcb_addr  ENDS
```

GETMYIPADDR and GETSUBNETMASK do not require *RCB_socket* to be set.

BIND and CONNECT require input parameters *RCB_port* and *RCB_address*. However, they can be set to zero for BIND. They should be in network byte order.

ACCEPT returns the spawned socket ID in *RCB_socket* and the peer's IP address and port number in *RCB_address* and *RCB_port* in network byte order.

Parent Topic: RCB Structures for Individual Commands

Addendum (6/30/95)

The LAN WorkPlace protocol stack was designed to be lean and mean. As a result, LAN WorkPlace TCP/IP suffers (or enjoys) a less flexible interface with a minimum set of configurable parameters. However, to support our internal development effort and to fulfill customers' requests, we will modify TCP/IP and the socket library to enable reconfiguration of some of the protocol stack's operating parameters during run-time. Currently TCP/IP configuration is set during initialization. This configuration can only be changed by modifying the NET.CFG file, then unloading and reloading the TCP/IP TSR.

Related Topics

New Socket Library Functions

New RCB Definitions

RCB Implementation Notes

Parent Topic: RCB Interface: Guide

Break Handling

For blocking RCB commands, the LAN WorkPlace protocol stack (TCPIP.EXE) blocks the execution thread until processing is completed, then returns the control to the application. To accomplish this, TCPIP.EXE polls the completion status of the RCB. A <Ctrl-Break> interruption mechanism is implemented in TCP/IP with the cooperation of the application so the tight polling loop can be exited. The default blocking RCB request does not allow the use of <Ctrl-Break>.

Break checking is turned on for each individual RCB request by setting *RCB_flags* to RF_BREAKRCB. TCP/IP issues the DOS INT 21h function 0Bh during the polling loop so that DOS can execute an interrupt 23h when <Ctrl-Break> is detected. For the TCP/IP to abort the RCB request, your INT 23h break handler should OR *RCB_flags* with RF_ABORTRCB. The pointer to the blocking RCB can be obtained from a global variable set up when the RCB is submitted to the protocol stack's RCB handler (seeSending an RCB Request).

NOTE: The above <Ctrl-Break> logic is for DOS only.

Parent Topic: Interfacing with the RCB Handler

CLOSE

CLOSE

```
   rcb_close STRUC
      RCB_skt_cmn   DB   (SIZE rcb_common) DUP (?)
rcb_close ENDS
```

rcb_close does not have command specific fields. *RCB_socket* is the only field that must be set.

Parent Topic: RCB Structures for Individual Commands

DOS Host Resident Internet Protocols RCB Format (For Non-I/O Operations)

Offset	Field	Comments
0	RCB_next	Used for link management
4	RCB_prev	Used for link management
8	RCB_ESR	Post address
12	RCB_reserved	For LAN WorkPlace TCP/IP internal use
20	RCB_type	(not used)
21	RCB_command	Command code
22	RCB_socket	Socket ID
23	RCB_status	Status & return code
24	RCB_param (variable)	Content determined by *RCB_command*

Parent Topic: The C Language Version of RCB Structures

Finding the RCB Handler Entry Point

LAN WorkPlace TCPIP.EXE presents a far procedure entry point to its RCB handler for RCB clients to submit requests and receive replies. TCPIP.EXE returns far pointers to its RCB handler in register pairs to the clients during initialization. RCB clients should save the entry point for submitting RCB requests. The entry point discovery mechanism is built on the install check function of the 2F multiplex interrupt handler of TCPIP.EXE. TCPIP.EXE uses multiplex number 7Ah and the function code 40h for install check.

```
[On Entry]
```

```
AX   = 7A40h        {install check and get RCB service entry }

[On Return]
AX   = 7AFFh
BX   = offset of MLID ISR vector in DOS's interrupt vector table,
       for example, BX equal to (5+8)*4=34h for IRQ 5
CX   = version     (0402h for 4.1, 0401h for 4.01, 1 for 4.0)
DX   = 0
ES:DI   = far pointer to the RCB handler
```

The following code demonstrates the function:

```
RCB_service_entry   dd   0   ; stores far pointer to TCPIP's RCB han

  ...
mov  ax, 7A40h           ; TCPIP interrupt 2F install check
int  2Fh
cmp  ax, 7AFFh           ; is TCPIP there?
jnz  not_installed
mov  ax, es
or   ax, di              ; double check the validity of the pointer
jz   not_installed
mov  word ptr RCB_service_entry, di   ; saves far pointer to RCB han
mov  word ptr RCB_service_entry+2, es   ;
ret
not_installed:
   ; handles condition where TCPIP is not loaded before running the RCB
```

Parent Topic: RCB Interface: Guide

GETBOOTPVSA

```
rcb_bootpvsa   STRUC
   RCB_bootp_cmn   DB   (SIZE rcb_common) DUP (?)
   RCB_bootp_vsa   DB   64 DUP (?)
rcb_bootpvsa   ENDS
```

This is a LAN WorkPlace specific RCB command that conveys the vendor-specific area of the bootp reply packet back to application. Only the *RCB_command* field is required. The 64 bytes of vendor-specific information in the bootp reply packet is copied into *RCB_bootp_vsa*. See the (RFC1084) BOOTP Vendor Information Extension for the data format. This information is only valid when the LAN WorkPlace protocol stack used the bootp protocol to retrieve configuration parameters.

Parent Topic: RCB Structures for Individual Commands

GETMYMACADDR

```
rcb_macaddr   STRUC
   RCB_mac_cmn    DB   (SIZE rcb_common) DUP (?)
   RCB_mac_addr   DB   6 DUP (?)
rcb_macaddr   ENDS
```

This is a LAN WorkPlace specific RCB command that returns the MAC (hardware) address of the default network interface. Only the *RCB_command* field is required for this command. Up to six bytes of MAC address is returned in network byte order in *RCB_mac_addr*.

Parent Topic: RCB Structures for Individual Commands

GETPATHINFO

```
rcb_getpathinfo   STRUC
   RCB_pathinfo_cmd   DB   (SIZE rcb_common) DUP (?)
   RCB_pathkey        DB   8 DUP (?)
   RCB_path           DB   128 DUP (?)
   RCB_pathlen        DW   ?
rcb_getpathinfo   ENDS
```

This is a LAN WorkPlace specific RCB command that provides directory paths to LAN WorkPlace system files. The paths are those entered using the PATH keyword in the NET.CFG file for path keys SCRIPT, PROFILE, LWP_CFG, TCP_CFG. Only the *RCB_command* field is required. *RCB_pathkey* holds the path key string (in upper case) for which you want the associated directory path configured in the NET.CFG. The upper case path string from NET.CFG is returned in *RCB_path* (up to 128 bytes). *RCB_pathlen* contains the actual length of the path string returned in *RCB_path* (in bytes).

Parent Topic: RCB Structures for Individual Commands

GETSNMPINFO

```
rcb_getsnmpinfo STRUC
   RCB_snmpinfo_cmn   DB   (SIZE rcb_common) DUP(?)
   RCB_mibvars        DD   ?
rcb_getsnmpinfo ENDS
```

This is an internal LAN WorkPlace specific RCB command that obtains a far pointer to a data area in the protocol stack which contains SNMP statistics and configurations.

Parent Topic: RCB Structures for Individual Commands

GETSOCKOPT, SETSOCKOPT

```
rcb_sockopt   STRUC
   RCB_opt_cmn   DB   (SIZE rcb_common) DUP (?)
   RCB_optname   DW   ?
   RCB_optval    DW   ?
   RCB_linger    DW   ?
rcb_sockopt   ENDS

; valid RCB_optname definitions
SO_REUSEADDR   EQU  4h    ; allow local address reuse
SO_KEEPALIVE   EQU  8h    ; keep connections alive
SO_LINGER      EQU  80h   ; linger on close if data present
```

RCB_socket is required for both commands. *RCB_optname* takes one of the **SO_** definitions below. *RCB_optval* contains the value of the option returned or to set to. A nonzero value in *RCB_optval* for SO_REUSEADDR, SO_KEEPALIVE, and SO_LINGER indicates that the option is set or to needs to be set. For SO_LINGER, *RCB_linger* stores the linger time interval in seconds. Note that LAN WorkPlace currently does not support socket close lingering.

Parent Topic: RCB Structures for Individual Commands

Handling NO_WAIT Commands

ACCEPT, CONNECT, SELECT, CLOSE, RECV, RECVFROM, SEND, SENDTO can be processed by protocol stack asynchronously when *RCB_command* is ANDed with NO_WAIT. Control immediately returns after the request is submitted. If a far notification (post) routine is specified in *RCB_ESR*, the notification routine is called when the request is completed. Unless an error (an *RCB_status* other than PENDING) is returned when submitting the request, the RCB and its associated data buffers should not be altered or reused before the notification routine is called. The data returned is valid only after the post routine is called. *RCB_status* contains the status of the completed request.

The post routine of a NO_WAIT RCB request is called with ES:SI pointing to the original request RCB, DS containing the data segment of the protocol stack, and SS:SP containing the stack's run-time event stack (256 bytes). The post routine must preserve DS and ES:SI, and ensure that the stack does not overflow during post-processing. Because this applies to every post routine, a single common asynchronous notification routine front end can be used with the unique portion of the post routine called afterwards.

Switching to a larger stack in the data segment is also recommended to protect the protocol stack from overflowing, or if the stack segment (SS) must be set to the your local DGROUP for run-time library functions called

from the post routine. A mechanism must be created to associate extra information (for example, the "real" post routine, and so on) with the RCB to accomplish this.

Parent Topic: Interfacing with the RCB Handler

Interfacing with the RCB Handler

Interfacing with the RCB Handler includes the following aspects:

Sending an RCB Request

RCB Completion and Error Handling

Handling NO_WAIT Commands

Break Handling

Parent Topic: RCB Interface: Guide

IOCTL

```
rcb_ioctl STRUC
   RCB_ioc_cmn   DB   (SIZE rcb_common) DUP (?)
   RCB_ioc_arg   DD   ?
   RCB_ioc_cmd   DW   ?
rcb_ioctl ENDS

; values for RCB_ioc_cmd
FIONBIO    EQU  26238
FIONREAD   EQU  26239
```

For the ioctl command FIONBIO, set *RCB_ioc_arg* to 1 to set the socket I/O mode to nonblocking, or to 0 for blocking mode. For FIONREAD, *RCB_ioc_arg* contains the number of bytes available to read on completion.

Parent Topic: RCB Structures for Individual Commands

LISTEN

```
rcb_listen     STRUC
   RCB_lis_cmn     DB   (SIZE rcb_common) DUP (?)
   RCB_lis_bklog   DW   ?    ; listen backlog (maximum is 5)
rcb_listen     ENDS
```

Parent Topic: RCB Structures for Individual Commands

New RCB Definitions

The changes involved for implementing the the new RCB functions include additions to the protocol stack's RCB handler as well as socket libraries for both DOS and Windows. The RCB will be expanded as follows:

```
rcb_common   STRUC
 RCB_next       DD   ?      ; forward link
 RCB_prev       dd   ?      ; back link
 RCB_ESR        DD   ?      ; user completion routine
 RCB_reserved   DB   8 DUP (?)   ; workspace
 RCB_type       DB   ?      ; for de-muxing
 RCB_command    DB   ?      ; command
 RCB_socket     DB   ?      ; socket
 RCB_status     DB   ?      ; status
rcb_common   ENDS

; new RCB commands
;
CMD_GETIFN         EQU   25   ; get interface number
CMD_SETIPINFO      EQU   26   ; set IP info
CMD_GETIPINFO      EQU   27   ; get IP info
CMD_SETDNSINFO     EQU   28   ; set DNS info
CMD_GETDNSINFO     EQU   29   ; get DNS info
CMD_SETROUTES      EQU   30   ; set/modify route entry/entries
CMD_GETROUTES      EQU   31   ; get route entry/entries
CMD_REMOVEROUTES   EQU   32   ; remove route entry/entries
CMD_SETARPS        EQU   33   ; set/modify ARP entry/entries
CMD_GETARPS        EQU   34   ; get ARP entry/entries
CMD_REMOVEARPS     EQU   35   ; remove ARP entry/entries

CMD_MAX            EQU   35


; command structure used by CMD_GETIFN
;
MAXMLIDNAMELENGTH   equ   128
;
rcb_ifn   STRUC
   RCB_ifn_cmn   db   (SIZE rcb_common) dup (?)
   RCB_ifnum     dw   ?   ; interface number (1 - 4) returned
   RCB_mlid_instance dw   ?   ; instance number of the mlid
   RCB_mlid_name     db   MAXMLIDNAMELENGTH dup (?)
               ; this is the file name (short name)
               ; of the mlid and normally should be
               ; less or equal to 8 characters long
rcb_ifn   ENDS
```

```
; command structure used by CMD_GETIPINFO and CMD_SETIPINFO
;
MAXROUTERS   equ   3   ; up to three default routers can be
                       ; specified for each interface
;
rcb_ipinfo   STRUC
   RCB_ipinfo_cmn   db   (SIZE rcb_common) dup (?)
   RCB_ip_ifnum     dw   ?   ; interface number (1 - 4), if 0
                         ; specified the default interface (1)
                         ; will be used
   RCB_ip_address   dd   ?
   RCB_ip_netmask   dd   ?
   RCB_ip_router    dd   MAXROUTERS dup (?)
rcb_ipinfo   ENDS

; command structure used by CMD_GETDNSINFO and CMD_SETDNSINFO
;
MAXDNS      equ   3   ; up to three DNS server can be
                      ; specified
MAXDNAME   equ   128

rcb_dnsinfo   STRUC
   RCB_dnsinfo_cmn   db   (SIZE rcb_common) dup (?)
   RCB_dns_ipaddr   dd   MAXDNS dup (?) ; name server ip address
   RCB_dns_domain   db   MAXDNAME dup (?)
               ; note, this is half of the MAXDNAME
               ; used by the resolver, it is chosen
               ; so the current RCB size is not
               ; increased
rcb_dnsinfo   ENDS

; command structure used by CMD_GETROUTES, CMD_SETROUTES, and
; CMD_REMOVEROUTES
;
MAXROUTES   equ   5   ; up to 5 static routes can be
                      ; manipulated
;
; definitions for RCB_route_type
;
ROUTE_STATIC   equ   1   ; indicate static route
ROUTE_HOST     equ   2   ; host specific route

rcb_route   STRUC
   RCB_dest_ipaddr   dd   ?   ; destination host/net IP address
   RCB_router        dd   ?   ; IP address of the 1st hop router
   RCB_route_type    dw   ?
rcb_route   ENDS

rcb_routeinfo   STRUC
   RCB_routeinfo_cmn db   (SIZE rcb_common) dup (?)
   RCB_rn            dw   ?   ; number of route entries up to
                              ; MAXROUTES specified in the RCB
```

```
   RCB_route_entry   db   (MAXROUTES * SIZE rcb_route) dup (?)
rcb_routeinfo   ENDS

; command structure used by CMD_GETARPE, CMD_SETARPE, and
; CMD_REMOVEARPE
;
MAXARPE   equ   16   ; up to 16 ARP entries can be
                     ; manipulated

rcb_arp   STRUC
   RCB_dest_ip   dd   ?
   RCB_dest_ha   db   6 dup (?)
rcb_arp   ENDS

rcb_arpinfo   STRUC
   RCB_arpinfo_cmn   db   (SIZE rcb_common) dup (?)
   RCB_an            dw   ?   ; number of arp entries up to
                              ; MAXARPE specified in the RCB
   RCB_arp_entry    db   (MAXARPE * SIZE rcb_arp) dup (?)
rcb_arpinfo   ENDS
```

Parent Topic: Addendum (6/30/95)

New Socket Library Functions

Synopses of the new socket library functions follow. (Note that the definitions are for the DOS socket library only, FAR pointers and PASCAL function types are required for Windows.)

```
   define   RCBMAXDNAME   128      /* maximum domain name length */
   define   MAXROUTES       5      /* maximum number of route entries
                                      can be manipulated */
   define   MAXARPE        16      /* maximum number of arp entries can
                                      be manipulated */

struct   ipinfo {  /* used in get/setipinfo() */
   long   addr;    /* IP numbers are in network byte
                      order, only nonzero numbers are
                      valid in SET function */
   long   netmask;
   long   router1;
   long   router2;
   long   router3;
};

struct   dnsinfo {  /* structure used in get/setdnsinfo()
                       to convey relevant DNS information
                       between application and TCPIP */
   long   dns1;     /* name server IP address */
   long   dns2;
```

```
   long   dns3;
   char   dname[RCBMAXDNAME]; /* case sensitive ASCIIZ domain name */
};

struct   route {       /* route entry structure used in
                          get/set/removeroutes() */
   long   dest_addr;   /* destination host/net IP address */
   long   router;      /* first hop router IP address */
   short   route_type;
};

struct   arpe {         /* arp entry structure used in
                           get/set/removearpe() */
   long   dest_ip;      /* IP address of a host */
   char   dest_ha[6];   /* hardware address of the host */
};

typedef char * CHARPTR;
typedef   struct ipinfo    * IPINFO_PTR;
typedef   struct dnsinfo   * DNSINFO_PTR;
typedef   struct route     * ROUTE_PTR;
typedef   struct arpe      * ARPE_PTR;

   include <sys\socket.h>
int getifn               /* interface number returned */
   (CHARPTR   mlidname,  /* case insensitive ASCIIZ mlid short
                            name (file name without suffix) */
    int       mlidinstance);  /* driver instance, 0 indicates 1st */

int getipinfo
   (int   ifn,            /* interface number to get info from */
    IPINFO_PTR addrptr);  /* pointer to address buffer to store
                             returned ipinfo */

int setipinfo
   (int   ifn,           /* interface number to set info to */
    IPINFO_PTR addrptr); /* pointer to address buffer which
                            stores the new address ipinfo */

int getdnsinfo
   (DNSINFO_PTR dnsptr);  /* ptr to dnsinfo buffer to store
                             returned info */

int setdnsinfo
   (DNSINFO_PTR dnsptr);   /* ptr to new dnsinfo buffer to
                              set */

int getroutes              /* number of static routes retrieved */
   (ROUTE_PTR routeptr);   /* ptr to route entry buffer to store
                              returned info, the buffer should
                              be large enough to hold MAXROUTES
                              'route' entries */
```

```
int setroutes
   (int rn,                /* number of routes pointed by routeptr */
    ROUTE_PTR routeptr);   /* ptr to route entries to set/mod. */

int removeroutes
   (int rn,                /* number of routes as specified in
                              routeptr to remove */
    ROUTE_PTR routeptr);   /* ptr to route entry buffer */

int getarps              /* number of ARP entries retrieved */
   (ARPE_PTR arpptr);    /* ptr to arp entry buffer to store
                            returned info, the buffer should
                            be large enough to hole MAXARPE
                            'arpe' entries */
int setarps
   (int an,              /* number of arp entries pointed by arpptr */
    ARPE_PTR arpptr);    /* ptr to arp entries to set/mod. */

int removearps
   (int an,            /* number of arp entries as specified
                          in routeptr to remove */
    ARPE_PTR arpptr); /* ptr to arp entry buffer */

Possible error codes are:
   ENIFNOTFOUND     - nif is not found for the mlid
   EINVALIDIFN      - invalid interface number
   EINVALIDIP       - invalid IP address for local, router, DNS server,
                      target of route/ARP entry to remove
   EINVALIDRN       - invalid number of route entry
   EROUTENOTFOUND   - route entry specified to remove is not found
   EINVALIDAN       - invalid number of ARP entry
   EARPENOTFOUND    - ARP entry specified to remove is not found
   ESETNOTALLOWED   - setting TCPIP operating parameters is not allowed
                      possibly due to open socket
```

Parent Topic: Addendum (6/30/95)

OS Host Resident Internet Protocols RCB Format (For Data-I/O Operations)

Offset	Field	Comments
0	RCB_next	Used for link management
4	RCB_prev	Used for link management
8	RCB_ESR	Post address

12	RCB_reserved	For LAN WorkPlace TCP/IP internal use
20	RCB_type	(not used)
21	RCB_command	Command code
22	RCB_socket	Socket ID
23	RCB_status	Status & return code
24	RCB_flags	Flag of MSG_OOB or MSG_PEEK
26	RCB_port	Port number (ignored for SEND/RECV)
28	RCB_ip_addr	IP address (ignored for SEND/RECV)
32	totallen	Total length of all fragments
34	RCB_frag_cnt	Fragment count
36	RCB_frag [0]	Fragment 0
42	RCB_frag [1]	Fragment 1
48	RCB_frag [2]	Fragment 2
54	RCB_frag [3]	Fragment 3
60	RCB_frag [4]	Fragment 4
66	RCB_frag [5]	Fragment 5
72	RCB_frag [6]	Fragment 6
78	RCB_frag [7]	Fragment 7

Parent Topic: The C Language Version of RCB Structures

Overview of the RCB Interface

The LAN WorkPlace® TCP/IP protocol stack provides a transport module for DOS and Windows* clients. At the upper boundary, it exposes a Request Control Block (RCB) interface for the BSD socket library and other clients (for example, the RFC NetBios driver). At the lower boundary, this interface communicates with the Link Support Layer™ (LSL™) according to the industry-standard Open Data-Link Interface™ (ODI™) specification. Although not directly related to the RCB interface, ODI affects the way how a user's RCB request is handled. LSL exposes an asynchronous interface to the protocol stack at the link layer that surfaces in the RCB interface. However, a blocking (synchronous) RCB request is built into the protocol stack's RCB handler rather than giving this responsibility to RCB clients.

This article describes the RCB interface for developers who want to bypass the LAN WorkPlace SDK socket libraries and use TCP/IP directly. The interface is described in the form of Intel* 8086 assembly language, but this does not prevent its use in applications written in higher-level languages. You can use this interface with higher-level languages by using either an assembly language front end or the assembly language capability built into

some high-level languages. See The C Language Version of RCB Structures for the C language version of RCB structure definitions.

This article describes the interface provided by version 4.1 of LAN WorkPlace. However, this article applies to versions 4.0 and 4.01 of the LAN WorkPlace protocol stack, except for three new RCB commands (GETBOOTPVSA, GETSNMPINFO, and GETPATHINFO) which are not supported.

It is strongly recommended that you have access to LAN WorkPlace SDK documentation (particularly the *LAN WorkPlace for DOS Socket Library Reference*) because LAN WorkPlace socket libraries are implemented on top of the RCB interface. Therefore, information about the LAN WorkPlace socket API and access to function descriptions can help you understand this article.

Parent Topic: RCB Interface: Guide

Protocol Stack RCB Handler

At the top of the LAN WorkPlace protocol stack is the RCB/socket module that interfaces with the application through RCB. The application passes the RCB containing the request context of a BSD socket function (in most cases) or a LAN WorkPlace specific service through the RCB service routine vector it obtained during initialization.

If the request is a blocking request, the completed RCB is returned to the application on the same request processing thread (for example, CMD_RECV only returns when data has been received).

If the request is a NO_WAIT request, the original request is returned immediately and the completed RCB is returned when the post routine specified in the *RCB_ESR* of the request is called (for example, CMD_RECV+NO_WAIT returns immediately and the post routine is called asynchronously on another processing thread---usually an interrupt thread---when data is received). LAN WorkPlace SDK socket libraries use NO_WAIT RCB requests to implement unique ANR socket functions.

The protocol stack RCB handler is a far procedure routine. All application registers except DX are preserved.

Parent Topic: RCB Interface: Guide

RCB Completion and Error Handling

For blocking RCB commands (the NO_WAIT bit is not set), the control does not return to the RCB client until the request is completed. For asynchronous RCB commands (the NO_WAIT bit set and *RCB_ESR* pointing to the post routine---asynchronous notification routine), the request

pointing to the post routine---asynchronous notification routine), the request is initiated and the control is returned immediately. The post routine is called when the request completes. *RCB_status* contains the command processing result. Any codes other than EOK indicate that the request has not successfully processed.

For NO_WAIT commands, *RCB_status* contains the immediate result after submitting the request to the protocol stack RCB handler. Any code other than PENDING indicates an error.

For a nonblocking BSD socket I/O request (set by the FIONBIO option of the IOCTL command for the individual socket), an *RCB_status* of EWOULDBLOCK on return indicates that the request is not fulfilled and would be blocked if submitted as a blocking request. The request is processed if the protocol stack determined that the operation can be completed without delay. As a result, you should retry the request upon receiving the EWOULDBLOCK status.

> **NOTE:** Only asynchronous RCB commands can be issued from the post routine. ESYNNOTSUPP is returned in *RCB_status* if a synchronous command is issued.

Parent Topic: Interfacing with the RCB Handler

RCB Definitions

```
*************************************************************************

   ifndef  _RCB

   define _RCB

   define MAXSG         8         /* Max number of scatter/gather */
   define MAXTOTALLEN   65535          /* Max Total length of data per

   typedef unsigned char    u8bit;
   typedef unsigned short   u16bit;
   typedef unsigned long    u32bit;

  /* RCB defines the static part of both Non IO and IO RCBs */

   typedef struct RCB_static {
      u32bit   RCB_next;           /* RCB management        */
      u32bit   RCB_prev;           /* RCB management        */
      u32bit   RCB_ESR;            /* post address          */
      u8bit    RCB_reserved [8];   /* workspace             */
      u8bit    RCB_type;           /* for RCB de-muxing     */
      u8bit    RCB_command;        /* command code          */
      u8bit    RCB_socket;         /* socket                */
      u8bit    RCB_status;         /* status & return code  */
   };
```

```
   /* frag describes the fragments */

   struct  frag {
      u32bit   fragptr;   /* pointer to the data  */
      u16bit   fraglen;   /* length               */
   };

   /* RCB_io defines the scatter gather arrray for IO RCBs */
   typedef struct   RCB_io {
      u16bit   totallen;
      u16bit   RCB_fragcnt;        /* number of fragments     */
      struct   frag  RCB_frag [8]; /* the fragments. Max = 8  */
   };

/*************************************************************/

   /* values for RCB_type */

   define RCB_BSD43 1         /* BSD 4.3 Internet             */

   /* Values for RCB_command (Non-IO Operations) */

   define ACCEPT           1
   define BIND             2
   define CLOSE            3
   define CONNECT          4
   define GETMYIPADDR      5
   define GETMYMACADDR     6
   define GETPEERNAME      7
   define GETSOCKNAME      8
   define GETSOCKOPT       9
   define GETSUBNETMASK   10
   define IOCTL           11
   define LISTEN          12
   define SELECT          13
   define SETMYIPADDR     14
   define SETSOCKOPT      15
   define SHUTDOWN        16
   define SOCKET          17

/* Data-IO Operations */

   define RECV       18
   define RECVFROM    9
   define SEND       20
   define SENDTO     21

/* New Commands for version 4.02 */

   define GETBOOTPVSA   22
   define GETSNMPINFO   23
```

```
 define GETSNMPINFO   23
 define GETPATHINFO   24

/*************************************************************/

 /* Sockaddr structure has the IPADDR and the PORT */

 struct  RCB_sockaddr {
    u16bit   port;      /* UDP/TCP port */
    u32bit   ip_addr;   /* IP address   */
 };

 /* Define various rcb structures for the socket commands. */
```

Parent Topic: The C Language Version of RCB Structures

RCB Definitions for I/O Operations

```
typedef struct rcb_recv {
   struct RCB_static     rcb;
   u16bit                flags;
   struct RCB_sockaddr   from;
   struct RCB_io         recv_io;
}rcb_recv;

typedef struct rcb_recvfrom {
   struct RCB_static     rcb;
   u16bit                flags;
   struct RCB_sockaddr   from;
   struct RCB_io         recvfrom_io;
}rcb_recvfrom;

typedef struct rcb_send {
   struct RCB_static     rcb;
   u16bit                flags;
   struct RCB_sockaddr   to;
   struct RCB_io         send_io;
} rcb_send;

typedef struct rcb_sendto {
   struct RCB_static     rcb;
   u16bit                flags;
   struct RCB_sockaddr   to;
   struct RCB_io         sendto_io;
}rcb_sendto;

   endif
```

Parent Topic: The C Language Version of RCB Structures

RCB Header Structure

The following structure defines the RCB header:

```
rcb_common      STRUC
   RCB_next        DD   ?          ; forward link
   RCB_prev        DD   ?          ; back link
   RCB_ESR         DD   ?          ; user completion (post) routine
   RCB_reserved    DB   8 DUP (?) ; workspace
   RCB_type        DB   ?          ; for de-muxing
   RCB_command     DB   ?          ; command
   RCB_socket      DB   ?          ; socket
   RCB_status      DB   ?          ; status
rcb_common      ENDS
```

The *RCB_next* and *RCB_prev* fields can be used by clients to manage the allocation of available RCBs and the freeing of a previously allocated RCBs from a linked list of available RCBs. Both fields can be modified by TCP/IP.

RCB_ESR contains a far pointer to the client's event service routine (post routine), which is called when the request is completed for an asynchronous (NO_WAIT) RCB command. This field should be set to NULL for blocking (not NO_WAIT) RCB commands.

The *RCB_reserved* field is reserved for the protocol stack. You must not use this field. *RCB_type* is also reserved for the protocol stack, and its value is modified.

RCB_command contains the command code for the RCB request.

RCB_socket contains the socket ID the request is targeted to. This field should be set to zero for commands not associated with sockets (that is, SOCKET, SELECT, GETMYIPADDR, GETSUBNETMASK, GETMYMACADDR, GETBOOTPVSA, GETSNMPINFO, and GETPATHINFO). The valid ID number ranges from 1 to the configured maximum.

RCB_status is set to the status or error code when the RCB returns on completion of the request.

> **IMPORTANT:** All RCB fields that are not used by the specific RCB command must contain zeroes.

The following defines values for *RCB_flags*:

```
   RCB_flags EQU  RCB_reserved

; RCB_flags values

   RF_INPROG      EQU  01H
   RF_POSTED      EQU  02H
```

```
RF_WINDOWS      EQU  04H
RF_PROTBUF      EQU  08H
RF_ABORTRCB     EQU  10H   ; for break handling
RF_BREAKRCB     EQU  20H   ;
```

The following defines the values for *RCB_command*:

```
; RCB_command code definition

CMD_ACCEPT          EQU  1
CMD_BIND            EQU  2
CMD_CLOSE           EQU  3
CMD_CONNECT         EQU  4
CMD_GETMYIPADDR     EQU  5
CMD_GETMYMACADDR    EQU  6
CMD_GETPEERNAME     EQU  7
CMD_GETSOCKNAME     EQU  8
CMD_GETSOCKOPT      EQU  9
CMD_GETSUBNETMASK   EQU  10
CMD_IOCTL           EQU  11
CMD_LISTEN          EQU  12
CMD_SELECT          EQU  13
CMD_SETMYIPADDR     EQU  14   ; not supported anymore
CMD_SETSOCKOPT      EQU  15
CMD_SHUTDOWN        EQU  16
CMD_SOCKET          EQU  17
CMD_RECV            EQU  18
CMD_RECVFROM        EQU  19
CMD_SEND            EQU  20
CMD_SENDTO          EQU  21
CMD_GETBOOTPVSA     EQU  22
CMD_GETSNMPINFO     EQU  23
CMD_GETPATHINFO     EQU  24

CMD_MAX             EQU  24
```

The following defines the asynchronous command modifier:

```
; asynchronous command modifier - AND with the above command code
; to signify an asynchronous RCB request.  Note, due to the
; semantic nature of the commands,
; only ACCEPT, CONNECT, SELECT, CLOSE, RECV, RECVFROM, SEND
; SENDTO can be NO_WAIT command

  NO_WAIT             EQU  10000000b
```

The following defines values for *RCB_status*:

```
; RCB_status definition

EOK                 EQU  0    ; Operation was successful

EWOULDBLOCK       EQU  35   ; Operation would block
```

```
EINPROGRESS       EQU  36   ; Operation now in progress
EALREADY          EQU  37   ; Operation already in progress
ENOTSOCK          EQU  38   ; Socket operation on non-socket
EDESTADDRREQ      EQU  39   ; Destination address required
EMSGSIZE          EQU  40   ; Message too long
EPROTOTYPE        EQU  41   ; Protocol wrong type for socket
ENOPROTOOPT       EQU  42   ; Protocol not available
EPROTONOSUPPORT   EQU  43   ; Protocol not supported
ESOCKTNOSUPPORT   EQU  44   ; Socket type not supported
EOPNOTSUPP        EQU  45   ; Operation not supported on socket
EPFNOSUPPORT      EQU  46   ; Protocol family not supported
EAFNOSUPPORT      EQU  47   ; Address family not supported by
                            ; protocol family
EADDRINUSE        EQU  48   ; Address already in use
EADDRNOTAVAIL     EQU  49   ; Can't assign requested address
ENETDOWN          EQU  50   ; Network is down
ENETUNREACH       EQU  51   ; Network is unreachable
ENETRESET         EQU  52   ; Network dropped connection on reset
ECONNABORTED      EQU  53   ; Software caused connection abort
ECONNRESET        EQU  54   ; Connection reset by peer
ENOBUFS           EQU  55   ; No buffer space available
EISCONN           EQU  56   ; Socket is already connected
ENOTCONN          EQU  57   ; Socket is not connected
ESHUTDOWN         EQU  58   ; Can't send after socket shutdown
ETOOMANYREFS      EQU  59   ; Too many references: can't splice
ETIMEDOUT         EQU  60   ; Connection timed out
ECONNREFUSED      EQU  61   ; Connection refused
ELOOP             EQU  62   ; Too many levels of symbolic links
ENAMETOOLONG      EQU  63   ; File name too long
EHOSTDOWN         EQU  64   ; Host is down
EHOSTUNREACH      EQU  65   ; No route to host
ENOTINSTLD        EQU  66   ; Protocol stack not installed
EASYNCNOTSUPP     EQU  67   ; Asynchronous operation cannot be
                            ; performed
ESYNCNOTSUPP      EQU  68   ; Synchronous operation cannot be
                            ; performed
ENO_RCB           EQU  69   ; no RCB available

; special RCB_status value indicating that the RCB request is pendin

PENDING           EQU  11111111b
```

Parent Topic: Request Control Block

RCB Implementation Notes

CMD_SETROUTES and CMD_SETARPE imply MODIFY if the route or arp entry for the destination already exists.

Definitions of MAXROUTERS, MAXDNS, MAXROUTES, and

MAXARPE reflect the current (R41-1i and later) implementation of TCP/IP

LAN WorkPlace implements a domain name resolver in the socket library. However, the resolver looks up the protocol stack's bootp reply (if any) and uses the DNS information over the static configured information in resolv.cfg if it exists. Implementation to CMD_SETDNSINFO takes advantage of this fact and fakes building a bootp reply which reflects the DNS information that the user wants the socket library to use. The actual implementation of massaging the bootp reply when there a "real" bootp reply already exists will take into account the existing variable length format of TAG_DOMAIN_SERVER and/or TAG_HOSTNAME records.

The TCP/IP RCB handler will do minimum validity checks on IP numbers passed in **setipinfo**, **setdnsinfo**, **setroutes**, **setarpe**. That is, an address of ff.ff.ff.ff, 0.0.0.0, or 1st octet >= 11100000b is considered invalid.

Error handling by the new socket library functions follows the same convention used in the current DOS/Windows socket library where a return code of -1 indicates an error and the global variable *errno* or function **geterrno** can be used to determine the cause.

The **getifn** function is incapable of identifying the interface when multiple instances of the same Multiple Link Interface Driver™ (MLID™) driver are loaded when only the MLID name is used. The extra "instance number" solves this problem.

The protocol stack does not allow its operating parameters to be set when a socket is opened.

We are considering making set/remove route/ARPe a single entry at the socket library interface level.

CMD_REMOVEROUTES treats NULL (0) for "IP address of the 1st hop router" as a special token that instructs TCP/IP to remove the route based on the "destination IP address" match only.

Parent Topic: Addendum (6/30/95)

RCB Structures for Individual Commands

In addition to the common RCB header, variable fields are defined for specific commands. See the complete definitions of all socket-related parameters for each command in SOCKET.H, supplied with the LAN WorkPlace SDK.

SOCKET

LISTEN

ACCEPT, BIND, CONNECT, GETPEERNAME, GETSOCKNAME, GETMYIPADDR, and GETSUBNETMASK

CLOSE

GETSOCKOPT, SETSOCKOPT

GETMYMACADDR

GETBOOTPVSA

GETSNMPINFO

GETPATHINFO

SELECT

IOCTL

SHUTDOWN

RECV, RECVFROM, SEND, SENDTO

Parent Topic: Request Control Block

RCB Structures for Non-I/O Operations

```
typedef struct    rcb_accept {
   struct RCB_static     rcb;
   atruct RCB_sockaddr   addr;
}rcb_accept;

typedef struct    rcb_bind {
   struct   RCB_static     rcb;
   struct   RCB_sockaddr   addr;
}rcb_bind;

typedef struct    rcb_close {
   struct   RCB_static   rcb;
}rcb_close;

typedef struct rcb_connect {
   struct   RCB_static     rcb;
   struct   RCB_sockaddr   addr;
}rcb_connect;

typedef struct rcb_getmyipaddr {
   struct   RCB_static     rcb;
   struct   RCB_sockaddr   addr;
}rcb_getmyipaddr;
```

```
typedef struct    rcb_getmymacaddr {
   struct RCB_static   rcb;
   u8bit               rcb_macaddr[6];
}rcb_getmymacaddr;

typedef struct rcb_getbootpvsa {
   struct RCB_static   rcb;
   u8bit               rcb_getbootp_vsa[64];
}rcb_getbootpvsa;

typedef struct rcb_getpathinfo {
   struct   RCB_static     rcb;
   u8bit    RCB_key[8];
   u8bit    RCB_path[128];
   u16bit   RCB_pathlen;
}rcb_getpathinfo;

typedef struct rcb_getpeername {
   struct   RCB_static     rcb;
   struct   RCB_sockaddr   addr;
}rcb_getpeername;

typedef struct rcb_getsockname {
   struct   RCB_static     rcb;
   struct   RCB_sockaddr   addr;
}rcb_getsockname;

typedef struct    rcb_getsockopt {
   struct RCB_static   rcb;
   u16bit              optname;
   u16bit              optval;
   u16bit              linger;
}rcb_getsockopt;

typedef struct rcb_ioctl {
   struct RCB_static   rcb;
   u32bit              arg;
   u16bit              ioctl;
}rcb_ioctl;

typedef struct rcb_listen {
   struct RCB_static   rcb;
   u16bit              backlog;
}rcb_listen;

typedef struct rcb_select {
   struct RCB_static   rcb;
   u16bit              socket_count;
   fd_set              readfds;
   fd_set              writefds;
   fd_set              exceptfds;
```

```
   unsigned long       ticks;
}rcb_select;

typedef struct rcb_setsockopt {
   struct RCB_static   rcb;
   u16bit              optname;
   u16bit              optval;
   u16bit              linger;
}rcb_setsockopt;

typedef struct rcb_shutdown {
   struct RCB_static   rcb;
   u16bit              how;
}rcb_shutdown;

typedef struct rcb_socket {
   struct RCB_static   rcb;
   u16bit              protocol;
}rcb_socket;
```

Parent Topic: The C Language Version of RCB Structures

RECV, RECVFROM, SEND, SENDTO

```
rcb_io   STRUC
   RCB_io_cmn      DB   (SIZE rcb_common) DUP (?)
   RCB_io_flags    DW   ?
   RCB_data_port   DW   ?
   RCB_data_addr   DD   ?
   RCB_SendLen     DW   ?
   RCB_FragCnt     DW   ?   ; number of fragments in RCB
   RCB_FragPtr1    DD   ?
   RCB_FragLen1    DW   ?
   RCB_FragPtr2    DD   ?
   RCB_FragLen2    DW   ?
   RCB_FragPtr3    DD   ?
   RCB_FragLen3    DW   ?
   RCB_FragPtr4    DD   ?
   RCB_FragLen4    DW   ?
   RCB_FragPtr5    DD   ?
   RCB_FragLen5    DW   ?
   RCB_FragPtr6    DD   ?
   RCB_FragLen6    DW   ?
   RCB_FragPtr7    DD   ?
   RCB_FragLen7    DW   ?
   RCB_FragPtr8    DD   ?
   RCB_FragLen8    DW   ?
rcb_io   ENDS

rcb_frag  STRUC
```

```
   RCB_FragPtr   DD   ?   ; ptr to the buffer fragment
   RCB_FragLen   DW   ?   ; fragment length in bytes
rcb_frag  ENDS

; RCB_io_flags definition
; MSG_OOB   EQU  1   ; process out-of-band data

; RCB_flags bit definition, note that this flag
; value is present only in rcb_io !
RF_INPROG   EQU  1   ; IO is in progress
```

rcb_io can be used to do network input and output using single buffer or scatter gather buffers. Up to eight application buffer fragments can be specified to receive data or to contain data to be transmitted. *RCB_SendLen* should be set to the total number of bytes of data that you want to send or receive on input. It is set to the actual number of bytes sent or received on completion. *RCB_data_addr* and *RCB_data_port* contain the destination host's IP address and protocol port number in network byte order for SEND and SENDTO on input. They are set to the source IP address and port number for RECV and RECVFROM on completion. On a connection-oriented TCP socket, *RCB_data_addr* and *RCB_data_port* are ignored on input for SEND/SENDTO, but they are still set to the originating host's IP address and port on completion for RECV/RECVFROM.

For connectionless UDP sockets, SEND can be used only if the socket is connected (pseudo connect---a CONNECT has been applied to the socket). *RCB_data_addr* and *RCB_data_port* are ignored in this case.

SENDTO can be used only when the socket is not connected (pseudo). Both *RCB_data_addr* and *RCB_data_port* must be specified. Otherwise an immediate error EDESTADDRREQ is returned.

RCB_io_flags can be set to MSG_OOB to transmit (SEND/SENDTO) out-of-band urgent data.

Parent Topic: RCB Structures for Individual Commands

Request Control Block

The Request Control Block (RCB) is designed in accordance with the BSD version 4.3 socket library API plus several services unique to LAN WorkPlace. You are assumed to have a basic understanding of Berkeley sockets (see *LAN WorkPlace for DOS Socket Library Reference* in the LAN WorkPlace SDK). Each RCB has a common static header followed by a variable format that depends on the command. Note that multiple commands can share an identical structure.

Related Topics

RCB Header Structure

RCB Structures for Individual Commands

Parent Topic: RCB Interface: Guide

SELECT

```
rcb_select   STRUC
   RCB_sel_cmn     DB   (SIZE rcb_common) DUP (?)
   RCB_sel_cnt     DW   ?
   RCB_sel_rd      DD   4 DUP (?)
   RCB_sel_wr      DD   4 DUP (?)
   RCB_sel_ex      DD   4 DUP (?)
   RCB_sel_ticks   DD   ?
rcb_select   ENDS
```

RCB_sel_cnt should be set to one larger than the largest socket ID you want to examine on input because LAN WorkPlace returns socket IDs from 1 up. It is set to indicate the number of sockets that are ready on return.

RCB_sel_rd, *RCB_sel_wr*, and *RCB_sel_ex* are socket ID descriptor sets for (network I/O) read, write, and exception. They should be set to a bit mask of the socket IDs that you are interested in on input, and are set to a subset of the original mask to indicate those sockets ready for reading, ready for writing, or have exception conditions pending on return.

RCB_sel_ticks should be set to the number of DOS timer ticks that you are willing to wait for the event to happen. Set *RCB_sel_ticks* to -1 to wait indefinitely.

LAN WorkPlace currently does not support select on exception conditions.

Parent Topic: RCB Structures for Individual Commands

Sending an RCB Request

After verifying that the LAN WorkPlace protocol stack is loaded and preparing a request RCB, an RCB client can send an RCB request. To send the request, put the far pointer of the request RCB in the register pair ES:SI and invoke the far function pointer of the protocol stack's RCB handler (obtained from the interrupt 2F install check of the protocol stack). The following code shows the process:

```
current_rcb   dd   0    ; stores far pointer to RCB in processing
                        ; used for <Ctrl-Break> handling
...
mov  word ptr current_rcb, si
mov  word ptr current_rcb+2, es
call [RCB_service_entry]      ; es:si -> request RCB
```

```
mov  word ptr current_rcb, 0
mov  word ptr current_rcb+2, 0
  .                      ; es:si -> original request RCB
```

Parent Topic: Interfacing with the RCB Handler

SHUTDOWN

```
rcb_shutdown   STRUC
   RCB_shutdown_cmn    DB   (SIZE rcb_common) DUP (?)
   RCB_how             DW   ?
rcb_shutdown   ENDS
```

This command is not fully implemented in LAN WorkPlace. *RCB_how* can only be set to 1 to disallow further send.

Parent Topic: RCB Structures for Individual Commands

SOCKET

```
rcb_skt   STRUC
   RCB_skt_cmn     DB   (SIZE rcb_common) DUP (?)
   RCB_skt_proto   DW   ?
rcb_skt   ENDS

; RCB_skt_proto values
IPPROTO_ICMP   EQU  1    ; internet control message protocol
IPPROTO_TCP    EQU  6    ; tcp
IPPROTO_UDP    EQU  17   ; udp
```

RCB_socket contains the socket ID allocated on successful completion of the request.

Parent Topic: RCB Structures for Individual Commands

The C Language Version of RCB Structures

This section describes the RCB structure definitions used to convey BSD 4.3 socket requests/replies between drivers and libraries or users.

The RCB can be logically defined as two different substructures: the non-I/O operation RCB and the I/O operation RCB.

Related Topics

DOS Host Resident Internet Protocols RCB Format (For Non-I/O Operations)

OS Host Resident Internet Protocols RCB Format (For Data-I/O Operations)

RCB Definitions

RCB Structures for Non-I/O Operations

RCB Definitions for I/O Operations

Parent Topic: RCB Interface: Guide

Using the Low-Level RCB Interface to Support the NetWare/IP DOS/Windows Client v1.0

Novell® Engineering

Using the Low-Level RCB Interface to Support the NWIP Client: Guide

Prerequisites: A TCP/IP stack vendor must implement the low-level RCB interface as described in Using the Low-Level RCB Interface to TCP/IP for UDP Communication. This must be done to enable the NetWare/IP™ client to send and receive UDP Packets.

Installation and Loading for NWIP

NWIPWIN.EXE and RES_SUPP.DLL should be copied to your Windows SYSTEM directory to be autoloaded by Windows. NWIP.EXE and NWIPINIT.EXE should be located in the NWCLIENT directory that contains ODI files. Make sure that your NET.CFG is in the same directory as LSL and NWIP.EXE.

Load in the following order in DOS (same as NetWare/IP 1.1):

1. LSL
2. LAN driver
3. TCPIP
4. NWIP

You can now load Windows and bring up VxD TCP/IP.

Parent Topic: Using the Low-Level RCB Interface to Support the NWIP Client: Guide

NetWare/IP Operation

The following illustrates how NetWare/IP 2.1 would operate in the described delayed TCP/IP-ready environment.

The NetWare/IP 2.1 workstation operates as follows:

First, NWIP.EXE (shrinked down version) tries to perform initialization (for example, finding a DSS through DNS) when it is loaded and TCP/IP is available (TCP/IP must return 0 as the local IP address if it is not ready).

1. NWIP.EXE loads and executes (through INT 21H, AX=4B). NWIPINIT.EXE (which parses NET.CFG) communicates with the BOOTP/DHCP server for NetWare/IP parameters (if applicable) and finds DSS's through DNS.

2. Upon completing execution, NWIPINIT.EXE terminates and returns control to NWIP.EXE with a code indicating the result of initialization.

3. NWIP.EXE then indicates whether it has loaded successfully or not by displaying a message on the workstation screen.

If TCP/IP is not ready, NWIP.EXE loads without initializing until TCP/IP is ready. The following applies to Windows only.

1. As Windows is loaded, NetWare/IP asks Windows to autoload NWIPWIN.EXE (the WINSOCK-based NetWare/IP Windows application similar to NWIPINIT.EXE for DOS). NWIPWIN.EXE depends on IPCONFIG.DLL and RES_SUPP.DLL.

2. When TCP/IP is ready (for example, TCP/IP VxD is loaded in Windows), the stack should send a message (type WM_USER+0x0050) to NWIPWIN.EXE. Indicate that TCP/IP is ready by setting *wParam* to 0.

3. Upon receiving this message, NWIPWIN.EXE performs the same function in Windows as NWIPINIT.EXE does in DOS.

As a Windows based TCP/IP stack is unloaded (no longer ready), it must let NWIP.EXE know through INT 2FH (AX=7A47h, DI=1000h), so that NWIP.EXE can reset its TCP/IP state to not-ready.

Parent Topic: Using the Low-Level RCB Interface to Support the NWIP Client: Guide

NWIP Software Components

DOS Executables

NetWare/IP contains the following DOS executables:

NWIP.EXE---the TSR.

NWIPINIT.EXE---the DOS initialization program that is loaded and executed by NWIP.EXE in the DOS environment if TCP/IP is ready.

Windows Executables and DLLs

NetWare/IP contains the following Windows executables and DLLs:

NWIPWIN.EXE---the Windows application that NWIP.EXE requests to be autoloaded when Windows is loaded. It performs NetWare/IP initialization similar to that of NWIPINIT.EXE in DOS.

RES_SUPP.DLL---exports functions (such as **wres_send** and **wres_mkqu**) for DNS resolver functions. It is completely based on WINSOCK.

Parent Topic:

Using the Low-Level RCB Interface to Support the NWIP Client: Guide

Requirements for the TCP/IP Stack

NetWare/IP 2.1 operates in a Windows based TCP/IP environment (VxD or TCPIP DLL), provided that the TCP/IP stack supports the following interface:

A TSR emulating the low-level RCB interface that is loaded before NWIP.EXE. NetWare/IP uses RCB to talk to the underlying TCP/IP stack.

WINSOCK.DLL, supporting the following functions:

closesocket
connect
ioctl
recv
recvfrom
select
send
sendto
socket

IPCONFIG.DLL, that exports **get_ip_config** and fills the passed data

structure when called. The function should be defined as follows:

```
WINAPI int get_ip_config(LPSTR ipConfigBuf, int
                         numberOfNameServers);
```

A return value of 1 indicates success; 0 indicates failure.

The *ipConfigBuf* has the following structure:

```
#define MAXNUMDNS   5
#define MAXDNAME   256

typedef struct
{
   unsigned long    icfg_ip;
   unsigned long    icfg_mask;
   unsigned short   icfg_num_dnsaddr; // up to 5
   unsigned long    icfg_dnsaddr[MAXNUMDNS];
   unsigned short   icfg_num_dnsname; // this is the returned number
                                      // name servers which cannot ex
                                      // the numberOfNameServers pass
                                      // by the caller
   unsigned short   icfg_dnslen1;     // length preceded, doesn't inc
                                      // the null terminator
   char icfg_dnsname1[MAXDNAME];
} IPCONFIG;
```

A stack that emulates the LAN WorkPlace® stack BOOTP/DHCP interface with NetWare/IP needs the following:

The ability to send a BOOTP request to an LAN WorkPlace 5.0 BOOTP/DHCP server.

Besides using the IP address, subnet mask, default router, DNS names and addresses from the BOOTP reply (some of those is in the vendor area), the stack must also store the reply buffer pointer in its memory. Real mode seg:offset is expected for the BOOTP reply buffer address.

The BOOTP reply buffer should be retrievable by NetWare/IP by calling INT 2Fh, AX=7A47h, DI=0004. If successful, AX=7AFF, and ES:SI points to the BOOTP reply buffer. Otherwise, the call is unsuccessful.

The BOOTP reply packet is defined as follows:

```
/*
 * UDP port numbers, server and client.
 */
#define IPPORT_BOOTPS 67
#define IPPORT_BOOTPC 68

#define BOOTREPLY     2
#define BOOTREQUEST   1

struct in_addr_list
```

```
struct in_addr_list
{
    unsigned         linkcount, addrcount;
    struct in_addr   addr[1];      /* Dynamically extended */
};

typedef struct bootp {
   unsigned char    bp_op;     /* packet opcode type */
   unsigned char    bp_htype;  /* hardware addr type */
   unsigned char    bp_hlen;   /* hardware addr length */
   unsigned char    bp_hops;   /* gateway hops */
   unsigned long    bp_xid;    /* transaction ID */
   unsigned short   bp_secs;   /* seconds since boot began */
   unsigned short   bp_unused;
   struct in_addr   bp_ciaddr;   /* client IP address */
   struct in_addr   bp_yiaddr;   /* 'your' IP address */
   struct in_addr   bp_siaddr;   /* server IP address */
   struct in_addr   bp_giaddr;   /* gateway IP address */
   unsigned char    bp_chaddr[16]; /* client hardware address */
   unsigned char    bp_sname[64];  /* server host name */
   unsigned char    bp_file[128];  /* boot file name */
   unsigned char    bp_vend[64];   /* vendor-specific area */
} bootp_header;

/*
 * RFC1048 tag values used to specify what information is
 * being supplied in the vendor field of the packet.
 */
#define TAG_PAD               ((unsigned char)   0)
#define TAG_SUBNET_MASK       ((unsigned char)   1)
#define TAG_TIME_OFFSET       ((unsigned char)   2)
#define TAG_GATEWAY           ((unsigned char)   3)
#define TAG_TIME_SERVER       ((unsigned char)   4)
#define TAG_NAME_SERVER       ((unsigned char)   5)
#define TAG_DOMAIN_SERVER     ((unsigned char)   6)
#define TAG_LOG_SERVER        ((unsigned char)   7)
#define TAG_COOKIE_SERVER     ((unsigned char)   8)
#define TAG_LPR_SERVER        ((unsigned char)   9)
#define TAG_IMPRESS_SERVER    ((unsigned char)  10)
#define TAG_RLP_SERVER        ((unsigned char)  11)
#define TAG_HOSTNAME          ((unsigned char)  12)
#define TAG_BOOTSIZE          ((unsigned char)  13)
#define TAG_DNS_DOMAIN_NAME   ((unsigned char)  15)
#define TAG_PARAM_REQUEST     ((unsigned char)  55)
#define TAG_END               ((unsigned char) 255)
```

Parent Topic: Using the Low-Level RCB Interface to Support the NWIP Client: Guide

Using the Low-Level RCB Interface to TCP/IP for UDP Communication

Novell® Engineering

Using the RCB Interface for UDP: Guide

Overview of RCB and UDP

UDP Interface

Finding UDP Entry Points

You must first locate an LSL entry point through LSL, then find a UDP protocol entry point. The following code illustrates this. (For more information, see the *Novell ODI Specification: NetWare® 16-Bit DOS Protocol Stack and MLIDs™* (part number 107-00078-001), available from Novell Labs™, 801-429-5544.)

Finding UDP Entry Points: Example

```
PROTSUP_GET_PROTO_CTL_ENTRY   equ   19

LSLSignature      db   'LINKSUP$'
LSLInitHandler    dd   0
LSLEntryPoints    dd   0
UDPProtoControl   dd   0
ip_protname       db   2, 'IP', 0

FindLSL   proc   near
   sub    ax, ax
   mov    es, ax
   mov    cx, word ptr es:[2Fh*4]
   mov    dx, word ptr es:[2F*4+2]
   mov    ax, cx
   or     ax, dx
   jz     LSLIsNotLoaded
   mov    ax, 0c00h

LookForLSLLoop:
   push   ax
   int    2fh
   cmp    al, 0FFh
```

```
   pop    ax
   je     Int2fSlotUsed

TryNextSlot:
   inc   ah
   jnz   LookForLSLLoop
   jmp   LSLIsNotLoaded

Int2fSlotUsed:
   mov   di, si
   mov   si, offset LSLSignature
   mov   cx, 4
   cld
   rep   cmpsw
      jnz   TryNextSlot

; Found the LSL. DX:BX -> LSL Init Handler.
; Call the handler to obtain the LSL Protocol Stack support
; entry point.

   mov   word ptr LSLInitHandler+0, bx
   mov   word ptr LSLInitEntry+2, dx
   push  ds
   pop   es            ;ES:SI -> parm block
   mov   si, offset LSLEntryPoints
   mov   bx, 2            ;Function #2 request support entry points
   call  LSLInitHandler
   sub   ax, ax
   ret

LSLIsNotLoaded:
   mov   ax, Error
   ret
FindLSL  endp


FindUDPProtEntry   proc   near
;
; Find the ip protocol stack id number
;
   call   ChkProtStkloaded         ;is the protocol stack loaded ?
   jnz   UDPProtoNotLoaded
;
   mov   bx, PROTSUP_GET_PROTO_CTL_ENTRY   ;protocol control function
   lea   di, ip_protnum   ;protocol stack id number
   mov   ax, word ptr [di]
   call  LSLEntryPoints         ;get the control entry from lsl
   jnz   UDPProtoNotLoaded
   mov   word ptr UDPProtoControl, si
   mov   word ptr UDPProtoControl+2, es
   ret
```

```
UDPProtoNotLoaded:
   mov   ax, Error
   ret
FindUDPProtEntry   endp

ChkProtStkLoaded   proc   near
   lea   si, ip_protname         ;IP protocol name
   mov   bx, PROTSUP_GET_PROTNUM_FROM_NAME
   call  LSLEntryPoints
   jnz   cps_exit             ;protocol not registered
   mov   ip_protnum, bx       ;save the protocol number

cps_exit:
   ret
ChkProtStkLoaded   endp
```

Parent Topic: UDP Interface

Overview of RCB and UDP

The LAN WorkPlace® TCP/IP protocol stack provides a transport module for DOS and Windows clients. At the upper boundary, it exposes both a high- and low-level Request Control Block (RCB) interface to applications.

The low-level RCB interface is provided for BSD socket library and other clients such as RFC NetBIOS driver (see The RCB Interface to the LAN WorkPlace TCP/IP Protocol Stack for more information). The high-level interface is provided for the NetWare/IP™ client and other applications that are not socket-based to send and receive using the TCP/IP stack. This article illustrates the low-level interface.

This article describes the low-level RCB interface for developers who want to send and receive UDP packets and for TCP/IP vendors who want to emulate this interface so that applications such as the NetWare/IP client can run on their stacks (see Using the Low-Level RCB Interface to Support the NetWare/IP DOS/Windows Client v1.0). The interface is described in the form of Intel 8086 assembly language, but this does not prevent its use by applications written in higher-level languages. You can use this interface with higher-level languages by using either an assembly language front end or the assembly language capability built into some high-level languages (see The C Language Version of RCB Structures).

Parent Topic: Using the RCB Interface for UDP: Guide

Receiving UDP Packets

If you want to receive incoming UDP packets through the low-level RCB interface, you must register your receive port and handler with UDP by

interface, you must register your receive port and handler with UDP by calling **UDPRegister**. A PECB structure pointer is passed when the receive handler is invoked. The receive handler first points to pe_ecb to get the received ECB, which can contain one or more received fragments. Data pointed to by *Frag1Addr* is the 12-byte ip_addr_info structure followed by 8 bytes of UDP-specific information (source and destination ports, UDP length, and checksum). Thus, UDP data starts at offset 20 (14h). The related data structures and sample code for a receive handler follows.

Resources used to receive (for example, pecb, ecb) are released by the UDP stack when the receive handler does a far return to the stack.

Data Structures and Receive Handler

```
;
; pecb structure passed by UDP stack to receive handler
;
pecb   struc
   pe_pecb  dw   ?   ; singly linked list
   pe_ecb   dd   ?      ; pointer to a received ECB
   pe_flags dw   ?      ; flags
   pe_ws1   dw   ?      ; work space word 1
   pe_ws2   dw   ?      ; work space word 2
pecb   ends

;
; Event Control Block Structure pointed to by pecb
;

ECB   struc
   NextLink     dw   ?,?
   PrevLink     dd   ?
   Status       dw   ?
   ESR          dd   ?
   ProtoNum     dw   ?
   ProtID       db   6 dup (?)
   BoardNum     dw   ?
   ImmAddr      db   6 dup (?)
   DriverWS     db   4 dup (?)
   ProtocolWS   dw   4 dup (?)
   DataLen      dw   ?
   FragCount    dw   ?
   Frag1Addr    dd   ?
   Frag1Len     dw   ?
ECB   ends

;
; cooked ip header
;
ip_addr_info   struc
   iai_srcaddr    db 4 dup (?)   ; source address
   iai_destaddr   db 4 dup (?)   ; destination address
   iai_prec       db ?      ; precedence/security
```

```
   iai_protocol   db ?      ; IP protocol #
   iai_length     dw ?      ; IP data length
ip_addr_info   ends


; aliases for UDP pseudo header prefix. uph_prefix has the same format
; as ip_addr_info passed up from IP.
;
uph_prefix    equ   ip_addr_info
ui_srcaddr    equ   iai_srcaddr
ui_dstaddr    equ   iai_destaddr
ui_zero       equ   iai_prec
ui_protocol   equ   iai_protocol
ui_iplen      equ   iai_length
UPH_PREFIX_LEN   equ   size ip_addr_info


; UDP pseudo header definition.
;
ui_header   struc
ui_prefix    db   UPH_PREFIX_LEN dup ( ? )
ui_srcport   dw   ?
ui_dstport   dw   ?
ui_udplen    dw   ?
ui_cksum     dw   ?
ui_header   ends

; UDP address information passed up to UDP clients
;
udp_addr_info   equ   ui_header
uai_srcaddr     equ   ui_srcaddr
uai_dstaddr     equ   ui_dstaddr
uai_zero        equ   ui_zero
uai_protocol    equ   ui_protocol
uai_iplen       equ   ui_iplen
uai_srcport     equ   ui_srcport
uai_dstport     equ   ui_dstport
uai_udplen      equ   ui_udplen
uai_cksum       equ   ui_cksum

PH_LENGTH   equ   size ui_header
UH_LENGTH   equ   size udp_header

;
; sample code for a receive handler
; get pointer to the physical buffer of the given rECB

   mov   word ptr lkahdr.LMediaHeaderPtr[0], di
   mov   word ptr lkahdr.LMediaHeaderPtr[2], es
   les   bx, es:[di].pe_ecb     ;make es:bx point to recevied ecb
   les   bx, es:[bx].Frag1Addr  ;make es:bx point to first received buf
```

Parent Topic: UDP Interface

Registering UDP

After finding the UDP entry point, you must register with UDP before sending or receiving UDP packets. The following example code shows how to register UDP with the receive handler and obtain a UDP Send entry point.

Registering UDP and Obtaining a UDP Send Entry Point

```
IPSUP_UDP_REGISTER     equ   5
IPSUP_UDP_DEREGISTER   equ   6
MY_PORT_NUMBER         equ   9999

UDPSend   dd   ?

UDPRegister   proc   near
   mov   ax, cs
   mov   es, ax            ; es:si -> application's receive handler
   mov   si, offset MyRecvHandler
   mov   ax, MY_PORT_NUMBER   ; ax has the application's UDP port numbe
   mov   bx, IPSUP_UDP_REGISTER
   call  UDPProtoControl
   jnz   udpreg_exit          ; bad return

;
;save the udp send entry point, it is used by UDPSend routine.
;dx has the UDPSend's code segment, cx has its offset
;

   mov   word ptr UDPSend, cx
   mov   word ptr UDPSend+2, dx

udpreg_exit:
   ret
UDPRegister   endp
```

Parent Topic: UDP Interface

Sending UDP Packets

An RCB is allocated and filled out by a TCP/IP application to pass a send request to the underlying TCP/IP module for processing (for more information about RCB structures, see The RCB Interface to the LAN WorkPlace TCP/IP Protocol Stack). The RCB uses either a single buffer or scatter-gather buffers. Up to 8 application buffer fragments can be specified to contain data to be transmitted. The UDP send entry point is obtained by

calling UDP to register a port for sending and receiving.

Set the fields of the RCB as follows:

RCB_SendLen should be set to the total number of bytes of data you want to send. It returns the actual number of bytes sent on completion.

RCB_data_addr and *RCB_data_port* contain (in network byte order) the destination host's IP address and protocol port number.

RCB_FragCnt is set to the number of fragments to sent (at least 1).

RCB_FragPtr1, *RCB_FragLen1*, and the following fragments are set to the offset/segment pointing to the data to be sent.

RCB_Reserved[0] contains the checksum option (0 = no UDP checksum).

RCB_Reserved[2] contains the local IP address in network byte order.

RCB_ESR is set to the sender's transmit completion handler, which is called by the protocol stack when the UDP packet is sent.

Finally, ES:DI points to the send RCB structure as the input parameter for the UDP send routine.

Parent Topic: UDP Interface

UDP Interface

TCP/IP is implemented as an Open Data-Link Interface™ (ODI™) protocol stack on top of the Link Support Layer™ (LSL™). Before you can send or receive UDP packets, you must get a UDP entry point through LSL, register with UDP to receive, and get a UDP Send entry point. Both sending and receiving are based on an asynchronous model. The following sections describe each step.

Related Topics

Finding UDP Entry Points

Registering UDP

Sending UDP Packets

Receiving UDP Packets

Parent Topic: Using the RCB Interface for UDP: Guide