AIX Version 4.3

Novell Network Transport Services 4.1 for AIX

SC23-4135-00

First Edition (October 1997)

This edition of *Netware Transports* applies to the AIX Version 4.3 Licensed Program, and to all subsequent releases of this product until otherwise indicated in new releases or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: THIS MANUAL IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

It is not warranted that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

It is possible that this publication may contain references to, or information about, products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that such products, programming, or services will be offered in your country. Any reference to a licensed program in this publication is not intended to state or imply that you can use only that licensed program. You can use any functionally equivalent program instead.

The information provided regarding publications by other vendors does not constitute an expressed or implied recommendation or endorsement of any particular product, service, company or technology, but is intended simply as an information guide that will give a better understanding of the options available to you. The fact that a publication or company does not appear in this book does not imply that it is inferior to those listed. The providers of this book take no responsibility whatsoever with regard to the selection, performance, or use of the publications listed herein.

NO WARRANTIES OF ANY KIND ARE MADE WITH RESPECT TO THE CONTENTS, COMPLETENESS, OR ACCURACY OF THE PUBLICATIONS LISTED HEREIN. ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE SPECIFICALLY

DISCLAIMED. This disclaimer does not apply to the United Kingdom or elsewhere if inconsistent with local law.

Novell, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any NetWare software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of NetWare software, at any time, without any obligation to notify any person or entity of such changes.

Notice to U.S. Government Users — Documentation Related to Restricted Rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract.

Trademarks and Acknowledgements

The following trademarks and acknowledgements apply to this information:

AIX is a registered trademark of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Novell, NetWare, the N-Design, and the NetWare Logotype are registered trademarks of Novell, Inc.

IPX (Internetwork Packet Exchange) is a trademark of Novell, Inc.

SPX and Sequenced Packet Exchange are a trademarks of Novell, Inc.

NetWare Directory Services is a trademark of Novell, Inc.

Novell Virtual Terminal is a trademark of Novell, Inc.

UNIX System V is a trademark of Novell, Inc.

386 is a trademark of Intel Corporation.

ARCnet is a registered trademark of Datapoint Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Co. Ltd.

XNS is a trademark of Xerox Corporation.

StarGROUP is a registered trademark of American Telephone and Telegraph.

About This Guide

Novell Network Services 4.1 for AIX (NNS) provides NetWare 4 services on AIX. Any reference in the document to NetWare services refers to NNS.

This guide provides programming information about the communication protocols in the NetWare Protocol Stack. The complete stack consists of daemons, various drivers, and protocol-specific modules and can be run on the host system independent of the NetWare service modules.

Chapter 1 discusses the Internetwork Packet Exchange (IPX), NetWare's datagram protocol, its packet structure and header fields, addressing scheme, and its implementation as a transport module in a UNIX environment.

Chapter 2 discusses the Routing Information Protocol (RIP), which maintains routing information and provides routing services to IPX, its packet flow, packet types, packet structure, and header fields. RIP has no programming interface.

Chapter 3 discusses the Sequenced Packet Exchange (SPX) protocol, NetWare's connection-oriented, reliable, sequenced transport protocol that provides a packet-level service. It provides information on SPX data delivery and acknowledgment, flow control, packet types, packet structure, and header fields.

This chapter is included primarily as a point of comparison with the enhanced protocol, SPXII.

Chapter 4 discusses the enhanced Sequenced Packet Exchange (SPXII) protocol, which is backward compatible with SPX but which offers true windowing, packet size negotiation, and use of larger packets. This chapter provides information on SPXII data delivery and positive and negative acknowledgment, packet types, packet structure, and header fields.

Chapter 5 discusses the Service Advertising Protocol (SAP) which allows service providing entities to be registered in a Server Information Table and advertised on an internetwork via SAP agents. This chapter also provides information on SAP packet structure and header fields, packet flow, and periodic broadcasting.

Chapter 6 provides the protocol-specific information needed to program IPX under TLI/XTI and contains a reference to both functions and structures.

Chapter 7 provides the protocol-specific information needed to program SPX and SPXII under TLI/XTI and contains a reference to the functions and structures used in both server and client applications.

Chapter 8 provides the information needed to program the SAP protocol using SAP Library functions.

Chapter 9 provides the information needed to program IPX directly using ioctls. (It does *not* include information on STREAMS interface **`getmsg/putmsg`**.)

Chapter 10 provides the information needed to access SPX/SPXII data structures using ioctls.

Application developers should be familiar with TLI/XTI programming (including ioctls) as well as with NetWare.

Chapter 11 discusses NCP Extensions, which provide a mechanism by which an application, rather than the NetWare server, can respond to NCP Extensions coming in to the server. This allows NetWare clients that are already authenticated to a Directory tree to access applications running on the AIX application server.

This chapter also provides the information needed to program NCP Extensions and contains a reference to the functions used in server-side NCPX applications.

Contents

3 Sequenced Packet Exchange (SPX) Protocol

4 Enhanced Sequenced Packet Exchange (SPXII) Protocol

chapter 1

Internetwork Packet Exchange (IPX) Protocol

What Is IPX?

IPX is a connectionless, datagram service protocol that does not require an acknowledgment for each packet sent.

No connection between the stations is established when a client uses this transport protocol to communicate with other clients or servers. Because IPX is a datagram service, each packet is routed individually to its destination on a network or internetwork.

IPX is an implementation of Xerox Network Standard (XNS). Other NetWare protocols that provide services, such as guaranteed service and packet sequencing (SPX/SPXII), service advertising (SAP), routing (RIP), and NCP (NetWare Core Protocol) are built on top of IPX.

How IPX Works

IPX performs the Open Systems Interconnection (OSI) network layer tasks of addressing, routing, and switching packets. IPX makes a “best effort” attempt to deliver packets to their destination; there is no guarantee or verification of successful delivery. Packet acknowledgment or connection control must be provided by protocols above IPX.

Guaranteed services, such as SPXII, can be built over IPX. However, IPX is used whenever guaranteed service is not required (for example, in service advertising) and for applications where an occasional lost packet is not critical. The low overhead of IPX means speed and performance.

IPX provides full internetwork addressing within a large address space. It defines network and socket numbers, while using the node

addressing scheme of the network interface hardware for clients. This saves memory, bandwidth, and complexity.

The IPX driver provides routing (RIP) services as well as IPX transport and addressing services. As LAN drivers for the network boards deliver packets to IPX, the IPX driver uses RIP to determine the route for packets outbound to other networks. Packets addressed to a local node are routed by IPX to the applications by using the socket numbers.

The discussion in this chapter covers the following topics:

- IPX addressing
- IPX packet structure
- IPX header
- IPX driver in a UNIX environment
- IPX programming interface

IPX Addressing

IPX addressing defines the internetwork, a collection of LANs connected by routers, bridges, and so forth.

IPX packet headers require both source and destination addresses. These designate the sender and the receiver of the packet respectively.

The IPX internetwork addressing scheme uses three address components shown in Table 1-1 (and which are described in more detail following).

Table 1-1
IPX Internetwork Address Components

Address	Byte Length	Description
Network	4 bytes	Identifies a specific network or LAN on an IPX internetwork.
Node	6 bytes	Identifies individual nodes, or computers, on a network or LAN.

Table 1-1 *continued*
IPX Internetwork Address Components

Address	Byte Length	Description
Socket	2 bytes	Identifies a process or function operating within a node. A socket is identified by a unique number.

Network Address

The network number (not the node address) is used by routers to forward packets to their destination. Each LAN is a configured network in IPX and assigned a unique network number or address.

Each LAN (logical network) must be associated with a physical network and is limited to a single network frame type. When multiple frame types are used on the same networking segment, each frame type is considered a LAN and must have a unique network number. This means that all network devices which are cabled to a network segment and which use a common frame type must also use the same network address.

Where multiple LANs are configured on the same platform, an internal network number (logical address) serves as a common point of connection.

Node Address

The node address identifies a station, node, or individual computer on a network. For clients, the node is defined by networking hardware and is usually factory set. For NetWare servers 3.x and above, NetWare configures a logical node address.

The lower-level Media Access Control (MAC) protocols (such as the token ring, Ethernet, and ARCnet standards) define node addressing for each LAN, which is implemented within the hardware or firmware of each network interface board and is usually factory set.

In addition to the server's node address for the attached LAN, the server's logical node address is associated with the internal network address.

Socket Address

The socket address identifies the process in the destination node and is the ultimate destination for a packet. Socket numbers provide a sort of mail slot that distinguishes each process for IPX.

Sockets are the mechanism that allows multiple applications on the same station to send and receive data, without interfering with each other. Sockets have the same function as ports in other network protocols.

A socket number is assigned to a specific process. A process can use a well-known (static) socket number or can obtain a dynamic (ephemeral) number when the process requests a socket from IPX. Because socket numbers are internal to each node, each node can have its own domain of sockets independent of other nodes.

IPX Packet Structure

The structure of an IPX packet is identical to that of an XNS packet. The packet header consists of 30-bytes. The minimum packet size is 30 bytes (header only), while the maximum size is Maximum Transmission Unit minus the 30-byte header (MTU-30).

Note

In some cases, data packets smaller than MTU-30 must be sent. In general, use the smallest of the maximum packet sizes accepted by the routers on your internetwork.

Some of the fields in the header are byte-order sensitive, and the data must be sent in hi-lo order (network), as illustrated in Figure 1-1 below.

Figure 1-1
Byte Order

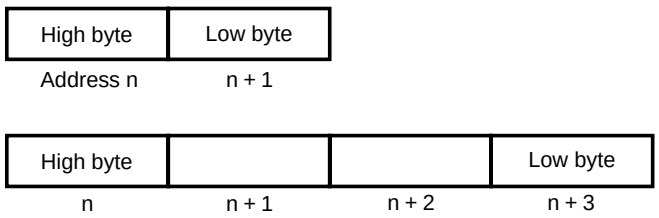
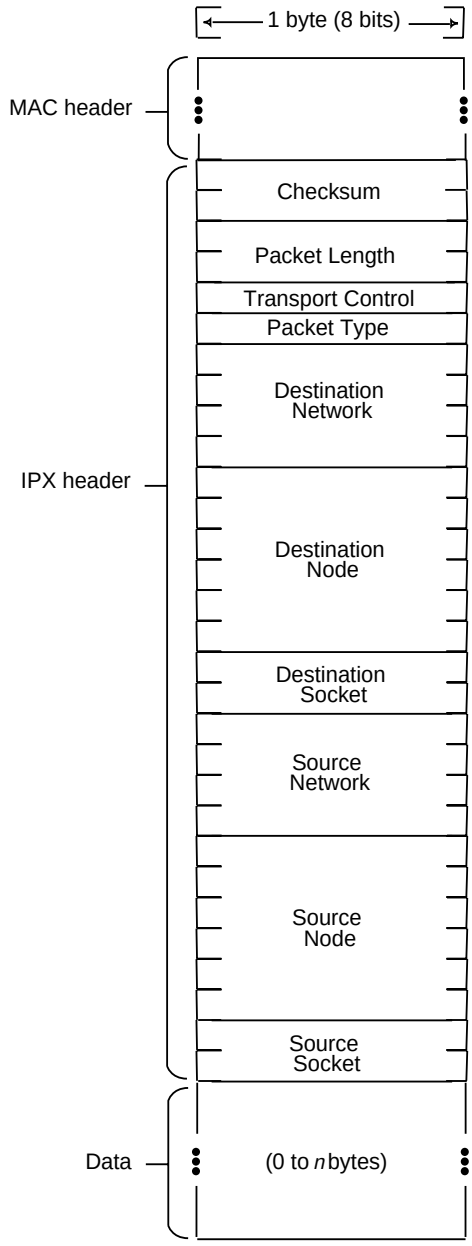


Figure 1-2 illustrates the IPX packet structure.

Figure 1-2
IPX Packet Structure



Although the IPX packet header has the format as shown in Figure 1-2 on page 5, there are frame-specific differences in the placement of the IPX header. The frame type is the format of the MAC (Media Access Control) header. The MAC header is specific to each transport and frame type. For example, IPX recognizes the following Ethernet frame types:

802.3 Raw
IEEE 802.2
Ethernet II
Ethernet SNAP

- If your environment is running with the 802.3 “raw” frame, the IPX header follows immediately after the Length field in the MAC header.
- If your environment is running with the standard 802.3 frame, the IPX header follows immediately after the IEEE 802.2 header (DSAP, SSAP, and Control).
- If your environment is using Ethernet II, then the Type field in the MAC header has a value of 8137, and the IPX header follows immediately after the Type field.
- If your environment is running with the 802.2 SNAP frame, the IPX header follows immediately after the 5-byte protocol identifier field.

Additionally, IPX recognizes the following token ring frame types:

Token ring
Token ring SNAP

The maximum length of the data section of an IPX packet (MTU-30) varies depending on the lower layer MAC protocol (Ethernet or token ring) that is being used.

For example, Ethernet supports 1500-byte packets, where 30 bytes is the IPX header and 1470 bytes is the actual data. In some cases, the maximum data size is limited by routers connecting LANs. Some older Ethernet routers limit the maximum IPX packet size to 576 bytes.

The content and structure of the data portion are entirely the responsibility of the application using IPX and can take any format.

IPX Header Fields

An IPX header has 10 fields and spans 30 bytes. (The data type `uint8` is one unsigned byte.)

When an application sends an IPX packet via the direct interface (see Chapter 9, “IPX Direct Interface”), the application must set the fields marked in Table 1-2 with an asterisk (*). The application can also set the fields marked (**). IPX sets the remaining fields.

Applications programmed via the Transport Layer Interface (TLI / XTI) are handled in a way specific to TLI.

The fields are shown in Table 1-2 and are described below:

Table 1-2
IPX Header Fields

Offset	Field	Type and Size	Byte Order
0	Checksum **	uint8[2]	hi-lo
2	Length *	uint8[2]	hi-lo
4	Transport Control	uint8	
5	Packet Type*	uint8	
6	Destination Network*	uint8[4]	hi-lo
10	Destination Node*	uint8[6]	hi-lo
16	Destination Socket*	uint8[2]	hi-lo
18	Source Network	uint8[4]	hi-lo
22	Source Node	uint8[6]	hi-lo
28	Source Socket**	uint8[2]	hi-lo

Note

* Fields the application must set.

**Fields that are optional for the application to set.

Checksum

The Checksum field is normally set to 0xFFFF, which indicates that no checksum is performed. This field, however, is configurable.

The value IPX_CHKSUM_TRIGGER (defined in the “ipx_app.h” file) directs IPX to generate a checksum of the IPX header (minus the Transport Control field) and the data. Any value which is not IPX_CHKSUM_TRIGGER is treated as 0xFFFF.

Packet Length

The Packet Length field contains the length of the complete IPX packet (header plus the data) and data presented to IPX by the application. The Packet Length field on packets received by the application from IPX indicates the amount of data received.

Transport Control

The Transport Control field is used to monitor the number of routers that a packet has crossed. IPX sets this field to zero before sending the packet. Each router increments the field before sending the packet on. If the packet passes through 16 routers, it is considered undeliverable and the sixteenth router discards it.

Packet Type

The Packet Type field identifies the type of service offered or required by the packet. Some defined values are listed in Table 1-3, but applications typically set this field to zero(0).

Table 1-3
Packet Type Values

Type	Definition
0	Unknown packet type (SAP or RIP)
1	Routing information packet (RIP)
5	Sequenced packet (SPX/SPXII)
17 (11h)	NetWare Core Protocol (NCP)
20 (14h)	NetBIOS broadcast

Destination Address Fields

The three destination fields (Network, Node, and Socket) contain the 12-byte IPX address of the destination.

On incoming packets, the IPX driver first ensures that the packet is intended for its node. Then it looks for socket numbers so it can send packets to their respective processes. If a packet has an unrecognized (unbound) socket number, the packet is discarded.

NetBIOS packets are routed by packet type. NetBIOS uses IPX Packet Type 20 (0x0014) for internet name-to-address resolution packets. In this case, the destination node field is set to 0xFF FF FF FF FF FF FF to indicate that routers should allow the broadcast packet to be propagated throughout an internet. If a process is bound to socket number 0x0455, the packet is also delivered to the process bound to that socket.

Network

This field must be set with the network number to which the destination server is connected. The system administrator assigns a unique network number to each network within an internetwork.

Many NetWare servers are configured with a logical network number, called the internal network. This internal network number provides the server with a single network address even when it is connected to several physical networks.

If zero (0) is used for the network address, the packet is sent to the network to which the source host is connected.

If the source host has an internal network (just as a server), any packets sent with a network address of zero (0) are sent only to the internal network, whereas in the case of a host configured with no internal network (just as a client would be), a packet with a zero (0) network address is sent to the physical network.

Packets with a network address of zero (0) are typically used to query SAP for the address and name of the nearest server.

Node

The destination node address identifies the client on the network. It is typically determined by a factory set address on the network board.

In the case of a server, a configured internal network typically has a node address of 0x00 00 00 00 00 01. Addresses shorter than 6 bytes are left justified and zero filled. A node address of 0xFF FF FF FF FF FF designates a broadcast to all hosts on the network identified by the destination network address.

Socket

The destination socket number directs the packet to a specific process on the destination node. A socket number is assigned to a specific process. On UNIX a process can use multiple sockets, but a socket cannot be shared among multiple processes.

Services written to run over IPX generally have static or well-known socket numbers associated with them. By having static socket numbers, IPX users ensure that their server and client application types match.

The following socket numbers are reserved by the IPX protocol:

0x02	Echo protocol socket
0x03	Error handler packet

In addition, Novell has defined and reserved sockets for specific purposes. Some are listed below.

0x247	Novell VirtualTerminal (NVT) server
0x0451	NetWare Core Protocol
0x0452	NetWare Service Advertising Protocol (SAP)
0x0453	NetWare Routing Protocol (RIP)
0x0456	NetWare Diagnostics Protocol
0x8063	NVT2 Server
0x811E	Print Server

Novell administers a list of sockets that are well-known in all IPX environments. Software developers who are writing IPX or SPX/SPXII based value-added packages that require well-known addresses should contact Novell to obtain socket assignments.

Dynamic sockets are available for use by any application. There are no well-known dynamic socket numbers. Dynamic sockets within the IPX suite begin at 0x4000 and end at 0x7FFF. Well-known sockets assigned by Novell begin at 0x8000. Servers that use dynamic sockets or well-known sockets can make their sockets and services known through SAP.

Source Address Fields

The three source address fields (network, node, and socket) contain the 12-byte IPX address of the sender.

Network

The source network address is filled in by IPX when the packet is sent by the source machine.

Node

The source node address is filled in by IPX when the packet is sent by the source machine.

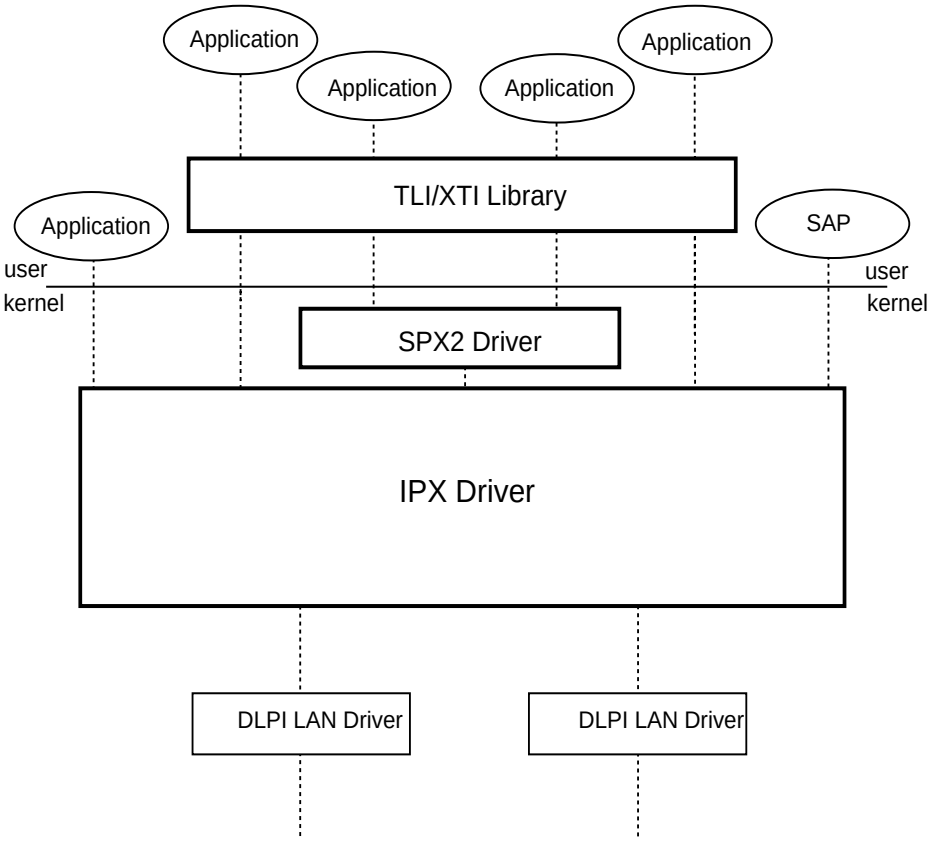
Socket

The source socket address is filled in by IPX when the packet is sent by the source, except when a process has multiple sockets assigned. In this case, the correct socket number must be filled in by the application.

IPX Driver in UNIX Environment

In general, the software for the IPX protocol in the UNIX environment is structured as shown in Figure 1-3:

Figure 1-3
Relationship between IPX, TLI/XTI Library, and Other NetWare Protocols

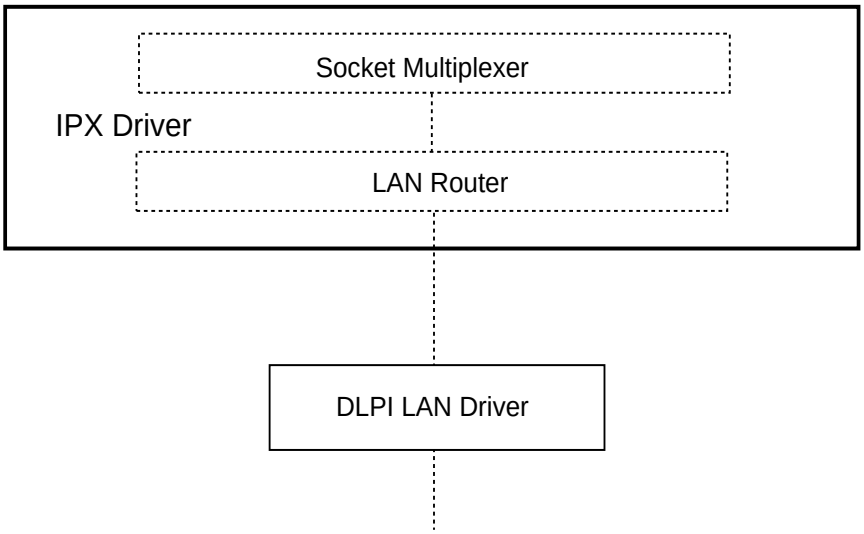


Applications can access IPX via the TLI / XTI library (Chapter 6). They can access SAP information (Chapter 8). Applications that require connection services can be built on SPXII using TLI / XTI (Chapter 7). A direct interface using STREAMS ioctl commands is also supported (Chapter 9).

Single LAN configuration

The computer shown in Figure 1-4 below is configured with a single LAN. That means that only one frame type is recognized. This is the typical configuration for NetWare UNIX client software.

Figure 1-4
Single LAN Configuration



The socket multiplexer is the entity in the IPX software that delivers an IPX packet to the appropriate application (socket).

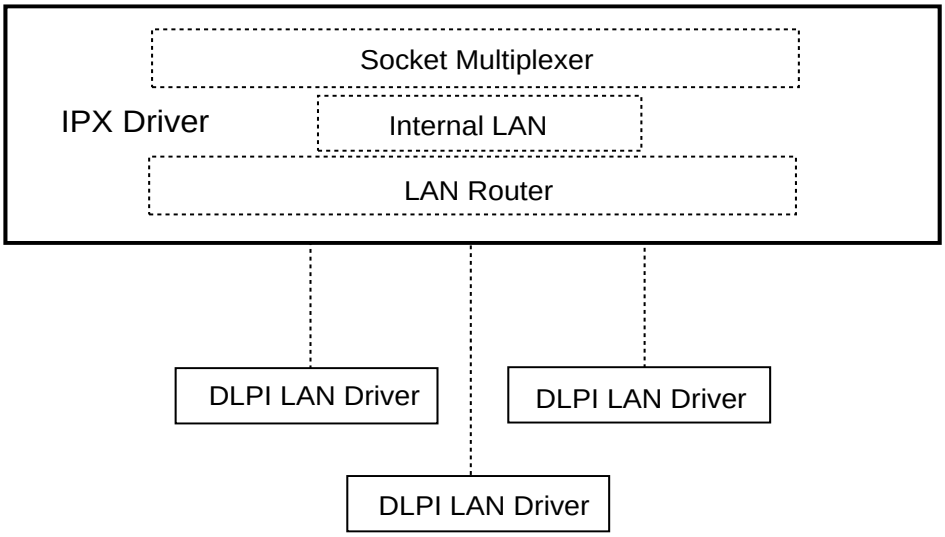
The LAN router is the entity in the IPX software that not only sends and receives data on one or more LANs configured in IPX and but also routes between LANs.

Multiple LAN Configuration

In the case illustrated in Figure 1-5 below, the computer is configured with multiple LANs. The LANs can be connected to one or more physical networks. The LANs can have the same or different frame types.

This configuration requires an internal network to be configured. The network number for the internal network must be unique in the internetwork and uniquely identifies the platform. This configuration also supports routing of IPX packets.

Figure 1-5
Multiple LAN Configuration



A LAN (Local Area Network) is a configured network in IPX. Each LAN (logical network) must be associated with a physical network and is limited to a single network frame type.

A physical network consists of the network boards (installed in each node) that are cabled together. The network boards, cable plant, and hubs must comply with the appropriate MAC standards, such as Ethernet or token ring.

Where the implementations of these standards have resulted in multiple frame types, the LAN driver allows more than one logical network to be configured on the physical network.

Multiple Ethernet frame types can run on the same Ethernet network board if the frame types are different. This means that multiple LANs (logical networks) can be configured on a single physical network, as long as each is a unique frame type on the physical network.

Both token ring frame types can also run on the same token ring network board, which creates two logical networks.

The frame type on different physical networks can be either the same or different.

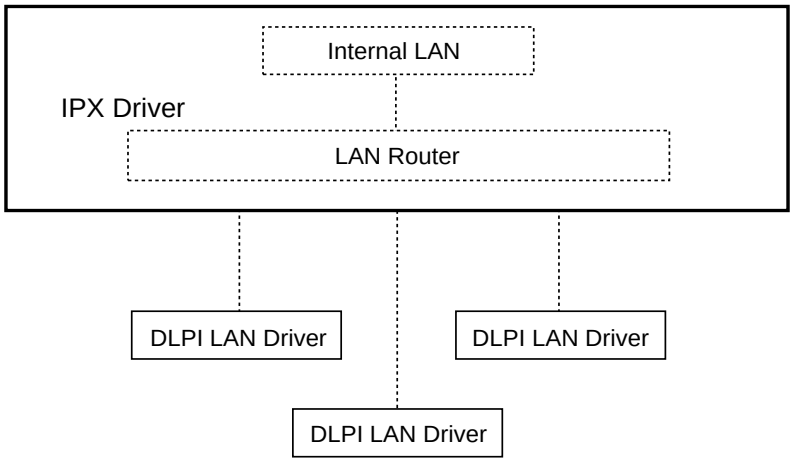
Each LAN is identified by a network number. All devices connected to the same physical LAN which use the same frame type must use the same network number.

The internal network is a logical network that serves as the common point of connection (network address) for a platform when multiple LANs are configured on the platform. The internal network is always required when multiple LANs are configured on a platform.

Router Only Configuration

This configuration shown in Figure 1-6 below supports LAN routing but not a socket multiplexer. No IPX-based applications exist on the platform. This configuration typically is a network backbone that supports only routing functions (clients and servers require the socket multiplexer for applications).

Figure 1-6
Router Only Configuration



IPX Programming Interface

The IPX driver is accessed via the IPX device node */dev/ipx*, which applications open to access IPX. It is readable and writable by everyone.

This device node is accessed with either the UNIX TLI/XTI specification or the user STREAMS I/O interface.

For UNIX TLI/XTI programming information for IPX, see Chapter 6, “TLI/XTI for IPX.”

For direct interface programming information for IPX, see Chapter 9, “IPX Direct Interface.”

chapter 2

Routing Information Protocol (RIP)

What Is RIP?

RIP is a distance vector protocol that is responsible for maintaining a list of distances to each destination network on an internetwork.

RIP was adapted from the Xerox Network Standard (XNS) routing protocol. However, an extra field for time delay (Number of Ticks) was added to the packet structure to improve the decision criteria for selecting the fastest route to a destination. This change prohibits the straight integration of NetWare's RIP with unmodified XNS implementations.

RIP has no application programming interface. In the UNIX environment, routing is implemented as a LAN router within the IPX driver for servers and dedicated routers. The client configuration of the IPX driver does not support routing

All routers keep an internal database of internetwork routing information, called a Routing Information Table (or Router Table). Such tables keep current information on the internetwork's configuration, which they update from RIP broadcast packets over IPX.

RIP allows the following exchanges of information:

- Clients locate the fastest route to a network.
- Routers request routing information from other routers for the purpose of updating their own internal tables.
- Routers respond to route requests from clients and other routers.
- Routers broadcast periodically to ensure that all other routers are aware of the internetwork configuration.
- Routers broadcast whenever they detect a change in the internetwork configuration.

For more information about RIP, refer to *IPX Router Specification* (Part #107-000029-001), which is also available via anonymous ftp at novell.com::/netwire/novlib/11/ipxrtr.zip.

The discussion in this chapter covers the following topics:

- How routing works
- RIP packet structure
- RIP packet fields
- RIP packet types

How Routing Works

To understand RIP packet structure, it is helpful first to understand what is meant by a router and how a route is obtained.

Routers are needed only when the destination and source node reside on different networks.

When that is the case—and where there is redundant cabling—there can also be multiple intermediary routers (hops) to go through in order to reach a router on the destination network.

Routers supply IPX with the router address of the next hop along the fastest route to the destination network. Each intermediary router along the route repeats the process of supplying IPX with the most efficient route until the packet reaches a router on the destination network.

A client needs to obtain a first hop target only the first time the client sends a packet to a node on another network. After that, the first hop address is stored and used until either the connection is terminated or the route becomes dysfunctional (for example, a router along the way shuts down).

Routing Information Tables

A Routing Information Table is a dynamic map of the internetwork's routers. When a router first comes up, it uses the RIP protocol over IPX to broadcast information about itself to the other routers on its network.

The information is passed to other routers until all routers on the internetwork know about the new router.

All routers listen for RIP packets and use the information contained in these packets to build and maintain their Routing Information Tables (which are stored in temporary memory and are never written to disk). These tables store the following information about other routers:

- Number of hops away (the number of routers or intermediate networks a packet crosses to reach the destination node). The number of hops is the distance vector.
- Number of ticks away (a tick equals 1/18.21 of a second or (60 x 60)/0xFFF). To allow a range of time delay, a minimum of one tick per hop is required, but a hop can measure several ticks as a value.
- Internetwork address

From the information provided in a Routing Table, a router could provide more than one route to a destination node. Hops and Ticks provide alternate ways of calculating the cost of each route—distance and time delay—so that the client can use the most economical route.

For more information on the entries in the Routing Information Tables and the method used to determine the most efficient route, including the split horizon algorithm that is used to reduce RIP traffic, see the previously mentioned *IPX Router Specification* (Part #107-000029-001), which is also available via anonymous ftp at novell.com::/netwire/novlib/11/ipxrtr.zip.

Obtaining a Route

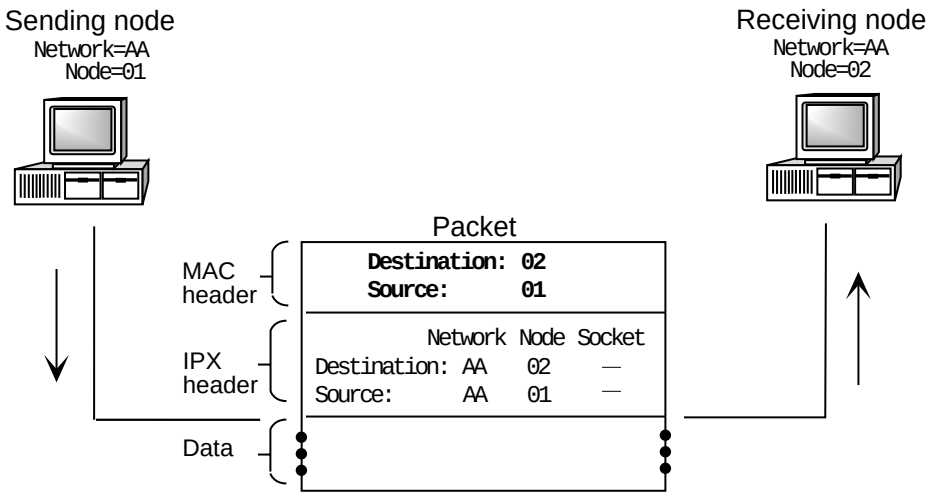
IPX is responsible for routing packets to their destinations. A client obtains an IPX address to a destination node—from the Service Advertising Protocol (SAP) or from the NetWare server's object database—and then sends a packet. IPX compares the network number of the destination node to that of the sending node.

When Routing Is Not Needed

If the network numbers of the destination node and the client are the same, IPX sends the packet directly to the destination node. In this case, no router is required.

This situation is illustrated in Figure 2-1, where the sending node and the receiving node are on the same network.

Figure 2-1
Transmitting a Packet between Two Nodes on the Same Network



When Routing Is Needed

On the other hand, if the network numbers of the destination node and the client are different, routing is required.

In this case, the client's IPX module broadcasts a RIP request to obtain the most efficient route to a router on the destination network. All of the routers on the client's network segment will receive the broadcast and consult their Routing Information Tables for the route with the fewest number of ticks. Only routers that can provide the shortest path to the destination network respond to the request.

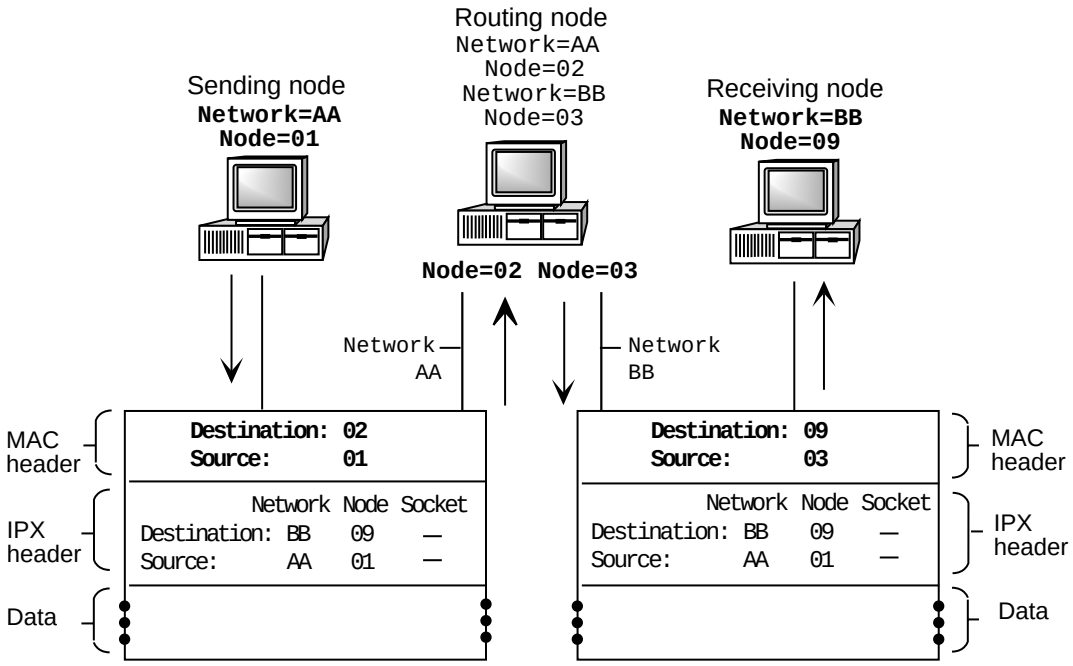
Note

If a router has multiple routes equal to the fewest number of ticks, it selects the route with the lowest hop count. It then sends a response to the client that contains, among other things, its node address and the hop and tick information.

More than one router might respond if several have a route equal to the fewest number of ticks and hops compared to other routers on the network. In this case, IPX accepts only the first router's response and discards any others.

Figure 2-2 illustrates what takes place after the route has been obtained.

Figure 2-2
Transmitting a Packet across Networks



In our example above, the sending node resides on Network AA and the receiving node resides on Network BB. A NetWare server is the router on Network AA with the shortest route to Network BB.

The client software on the sending node places the node address of the NetWare server doing the routing—which it just obtained as a result of a RIP request—in the Media Access Control (MAC) header of its packet and addresses the IPX header with the receiving node's internetwork address.

With the packet addressed in this fashion, the NetWare server doing the routing receives the packet, checks the IPX destination address, consults its Routing Information Table for the shortest route to network BB, and constructs a new MAC header addressed to the receiving node.

The only fields that change as the packet moves from one router to the next are the MAC address fields and the Transport Control field in the IPX header. All other fields are left intact. (The Transport Control field tracks the number of routers a packet passes through to reach the destination network.)

RIP Packet Structure

As with most of the higher-level protocols discussed in this manual, the RIP packet structure is encapsulated within the data area of IPX.

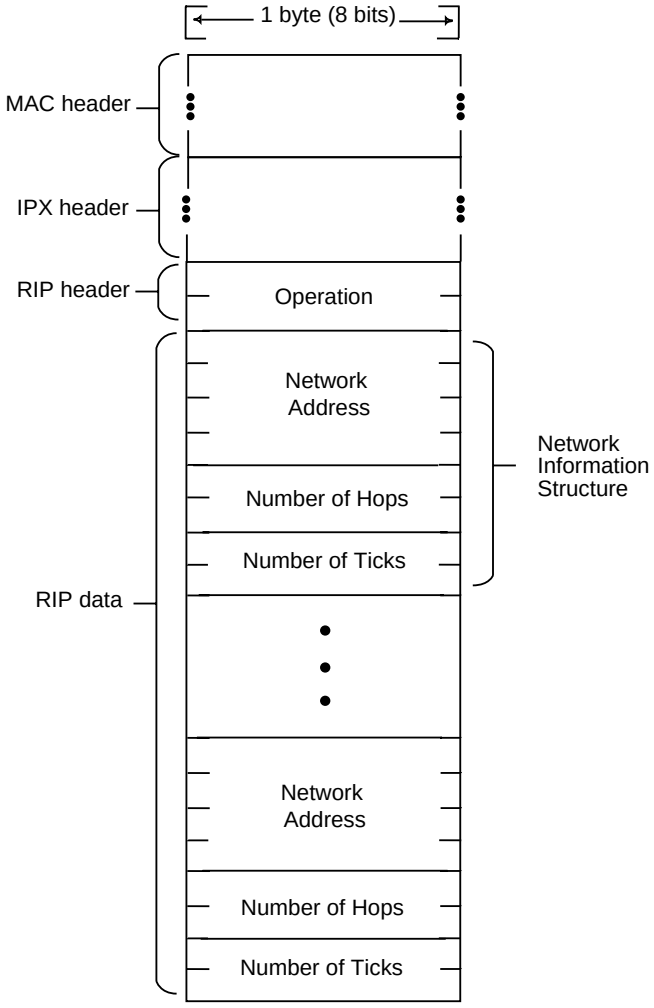
Earlier versions of NetWare did not always set the Packet Type to 1 for RIP packets in the IPX header. Routers should check the Destination Socket fields to determine whether a packet is a RIP packet rather than depend on the Packet Type.

RIP requests may also have the Destination Socket set to 0x0453 but not the Source Socket. In responding to a RIP request whose Source Socket is not 0x0453, the router should set the Destination Socket to the value of the sending node's Source Socket and set the Source Socket to 0x0453.

The RIP packet header has a 2-byte Operation field that indicates the RIP packet type. The Operation field is followed by one or more 8-byte network entries, or Network Information Structures, that contain three fields (Network Number, Number of Hops, and Number of Ticks).

Figure 2-3 illustrates the RIP packet structure.

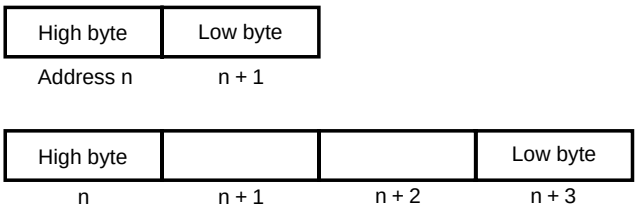
Figure 2-3
RIP Packet Structure



For a minimum size packet (570), up to 50 network entries are placed in the packet. Larger packets can contain more than 50 entries. Thus, the number of network entries contained in the RIP packet can vary anywhere from a minimum size of 40 bytes (IPX header plus one RIP network entry) to a maximum size of MTU-32 bytes/8 (MTU - (IPX header + operator)/8).

The fields in the RIP packet are byte-order sensitive, and the data must be sent in hi-lo order (network), as shown in Figure 2-4.

Figure 2-4
Byte Order



The fields in the RIP header are described in the following section.

RIP Packet Fields

RIP packet fields are listed in Table 2-1 and are described below.

Table 2-1
RIP Packet Fields

Field	Size	Byte Order
Operation (RIP Header)	uint8 [2]*	hi-lo
Network Information Structure	uint8 [8]	hi-lo

Operation

This field indicates whether the packet is a request (query) or a response packet. Request packets have a value of 0x0001, and response packets have a value of 0x0002.

Network Information Structure

This data structure for network entries consists of the three fields shown in Table 2-2:

Table 2-2
Fields in the NetWork Information Structure

Field	Description
Network Number	A 4-byte number that identifies a particular LAN (network) on an internetwork. (See “IPX Addressing” on page 2.)
Number of Hops	Contains the number of routers, or intermediate networks, that must be passed through to reach the LAN specified in the network number field.
Number of Ticks	Contains an estimate of the number of ticks to reach the LAN specified in the network number field. Each tick represents one-eighteenth of a second (1/18.21 of a second or (60 x 60)/0xFFF, the granularity of the original PC’s system time clock.) NetWare routing assumes Ethernet, for which the minimum tick value = 1.

RIP Packet Types

RIP packets use the two Operations types (request and response) to serve the following five routing functions.

General Request (Operation = 1)

Broadcast by nodes or routers to obtain information about all networks that exist on an internetwork. A server or router broadcasts this type of packet when it has just come onto the internet. A request (Operation set to 1) is a general request when the network number field in a network information structure is set to 0xFF FF FF FF.

Specific Request (Operation = 1)

Used to obtain information about a specific network. Clients issue this type of packet to obtain a route to a destination network. In this case, one or more network information structures are filled in with the unique network numbers of those networks being requested.

Periodic Broadcast (Operation = 2)

Used to ensure that all routers are kept abreast of the current internetwork configuration and also to provide routers with a means of aging networks. When a network suddenly becomes inaccessible due to a router going down abnormally, the missing periodic broadcast indicates that the router's services are no longer available and that its information should be removed from the Routing Information Table.

If a router detects new information in a periodic broadcast packet, it updates its Routing Information Table. All routers typically broadcast informational RIP packets of this type every 60 seconds. Each packet can broadcast information about one or more networks.

Response (Operation = 2)

Sent in response to a general or specific request from a router or workstation for a route. (If a RIP response entails providing information for more than the number of networks that will fit in a packet, multiple RIP response packets are required.)

Specific Informational Response (Operation = 2)

Used to advertise a new service on the network and to remove a service that is going down. These packets broadcast a change in the internetwork configuration so that routers can updates their Routing Information Tables.

Specific informational Response packets and Periodic Broadcast packets look identical on the network. They differ only in the context in which they are sent.

chapter 3

Sequenced Packet Exchange (SPX) Protocol

What Is SPX?

SPX is a connection-oriented, reliable, sequenced transport protocol. SPX provides a packet-level service, while enhanced SPXII provides a message-level service.

SPX guarantees packet delivery to the destination endpoint and notifies the user if any errors occur during data transmission. Upon encountering a data transmission error, SPX retries a given number of times before closing the connection and notifying the connection user. SPX also notifies the user if a disconnection indication is received from the remote connection endpoint.

SPX also provides flow control. This regulates the speed with which packets are sent and received by both the sending and receiving processes.

The SPX information in this chapter provides a baseline from which to understand SPXII, the enhanced protocol. The NetWare protocol stack does not include an SPX driver. However, the SPXII driver provides backward compatibility with SPX, both on the wire and in the programming interface.

The discussion in this chapter covers the following topics:

- How SPX works
- SPX packet structure
- SPX packet header fields
- SPX data flow and sequence
- SPX flow control
- SPX connection management

How SPX Works

SPX guarantees the communication between two nodes by using sequenced (numbered) packets and by requiring that the connection endpoints verify that each packet has been received.

The SPX packet numbering system is set up when an SPX connection is established; both sequence and acknowledge numbers are set at zero (0).

As the two nodes continue transmitting packets between them, each node increments the corresponding SPX sequence and acknowledge numbers for each packet. This means that once a connection is established, each subsequent SPX packet is marked with the current sequence and acknowledge numbers.

Using sequence and acknowledge numbers in this way keeps the communication between two endpoints synchronized and allows each endpoint to maintain its unique understanding of the current state of the communication.

Successful transmission is verified because the acknowledge number (ACK) is set to the next number in the packet sequence and hence "requests" the data packet with the corresponding number.

Packet sequencing also makes it possible for SPX to detect lost or dropped packets as follows:

- If an SPX data packet is sent by the source node and no acknowledgment is received, the packet is retransmitted.
- If an acknowledge packet is sent by the destination node and the subsequent data packet has an unexpected sequence number, then the acknowledge packet is retransmitted.

SPX Packet Structure

An SPX packet consists of a 42-byte SPX header followed by zero to 534 bytes of data. The minimum SPX packet size is 42 bytes (the header only) and the maximum size is 576 bytes (header plus 534 bytes of data).

Note

The maximum packet size reflects an Ethernet default. In some cases, larger data packets can be sent. For more information, consult the documentation for the Media Access Control protocol and refer to Chapter 1, “Internetwork Packet Exchange (IPX) Protocol.”

The content and structure of the packet’s data portion are entirely the responsibility of the application using SPX and can take any format.

Some of the fields in the header are byte-order sensitive, and the data must be sent in hi-lo order (network), as illustrated in Figure 3-1.

Figure 3-1
Byte Order

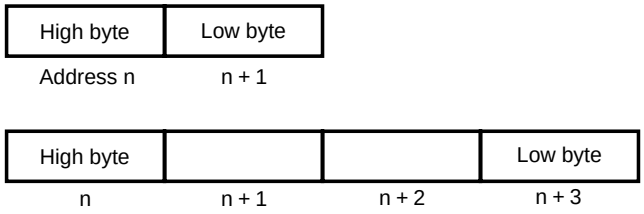
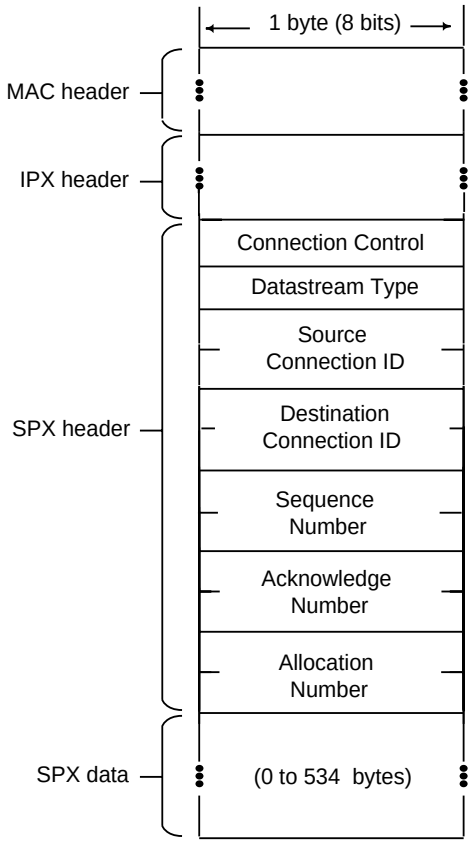


Figure 3-2 illustrates the SPX packet structure.

Figure 3-2
SPX Packet Structure



The header fields are described in the section following.

SPX Header Fields

Table 3-1 shows the field definition for the SPX header, including the IPX encapsulation.

Table 3-1
Packet Header Fields for IPX/SPX

Field	Field Type	Byte Order	Size	Value
Checksum	IPX	hi-lo	uint16 *	0xFFFF
Length	IPX	hi-lo	uint16	42 - 576 bytes
Transport control	IPX		uint8 *	0 - 16 hops
Packet type	IPX		uint8	0 - 6
Destination address.network	IPX	hi-lo	uint8[4]	0 - 0xFFFFFFFF
Destination address.node	IPX	hi-lo	uint8[6]	0 - 0xFFFFFFFFFFFF
Destination address.socket	IPX	hi-lo	uint8[2]	0 - 0xFFFF
Source address.network	IPX	hi-lo	uint8[4]	0 - 0xFFFFFFFF
Source address.node	IPX	hi-lo	uint8[6]	0 - 0xFFFFFFFFFFFF
Source address.socket	IPX	hi-lo	uint8[2]	0 - 0xFFFF
Connection control	SPX		uint8	(See description below)
Datastream type	SPX		uint8	(See description below)
Source connection ID	SPX	hi-lo	uint16	(See description below)
Destination connection ID	SPX	hi-lo	uint16	(See description below)
Sequence number	SPX	hi-lo	uint16	(See description below)
Acknowledge number	SPX	hi-lo	uint16	(See description below)
Allocation number	SPX	hi-lo	uint16	(See description below)

Note * A `uint16` is two unsigned bytes; a `uint8` is one unsigned byte.

The fields in the IPX/SPX packet header are defined as follows:

Checksum

This field is set to 0xFFFF.

Length

This field is set to the length of the complete IPX/SPX packet (42 to 576 bytes).

Transport Control

This field is set to zero before sending the packet. Each router increments the field before sending the packet on. If the packet passes through 16 routers, the sixteenth router discards the packet.

Packet Type

This field is filled in automatically.

Destination Address

When establishing an SPX connection, a packet must have the destination address, which is a 12-byte IPX structure (network, node, and socket). SPX fills in the destination address with the address of the other endpoint (destination). For more information, see “IPX Addressing” on page 2.

Table 3-2 contains a list of the fields.

Table 3-2
Destination Address

Field	Size	Byte Order
Network Address	uint8[4]	hi-lo
Node Address	uint8[6]	hi-lo
Socket Number	uint8[2]	hi-lo

IPX uses the network address to route the packet to the destination network. The node address identifies the client or node that receives the packet. IPX then uses the packet type to deliver the packet to SPX. SPX uses the socket number and the destination connection ID to identify the appropriate process for an incoming packet.

Source Address

SPX fills in the source address of the sender. The source address, like the destination address, is a 12-byte IPX address with the network, node, and socket fields (see “IPX Addressing” on page 2).

Connection Control

This field is in the SPX packet header itself. SPX uses this field to indicate whether the packet is a system or application data packet. System packets are used to acknowledge data packets; data packets are used to send data.

Table 3-3 lists the flags that are valid for SPX.

Table 3-3
Flags for SPX Connection Control

Value	Symbol	Description
0x10	EOM	Set by SPX to indicate end of message. This bit reflects the state of T_MORE from a TLI application.
0x40	ACK	Set to request that the receiving endpoint acknowledge the receipt of this packet.
0x80	SYS	Set to identify a system packet. System packets are internal SPX packets, are not delivered to an SPX application, and do not consume sequence numbers.

Datastream Type

SPX uses this field to indicate either an End-of-Connection (0xFE) or an End-of-Connection Acknowledgment (0xFF). SPX ignores any other values in this field.

Source Connection ID

SPX generates the identification number for the Source Connection Identification field during connection establishment. The source endpoint of the packet assigns the value.

Destination Connection ID

SPX generates the identification number for the Destination Connection Identification field during connection establishment. The destination endpoint of the packet assigns the value.

SPX uses Source and Destination Connection ID numbers to demultiplex packets from multiple connections that arrive on the same socket.

All concurrently active connections on a given node are guaranteed to have unique connection ID numbers.

Sequence Number

SPX uses this field as a counter that tracks the number of data packets sent by the connection and that have been acknowledged. Because each side of the connection keeps its own sequence count, this field keeps a count of packets transmitted in one direction only on a connection. The number wraps to zero after reaching 0xFFFF.

SPX manages this field; applications need not be concerned with it.

Acknowledge Number

SPX uses this field to indicate the sequence number of the next packet expected from the other endpoint. This number remains constant until the next data packet is received. The receiver increments this field to indicate that the data packet was received.

SPX drops any data packet with a sequence number less than the specified acknowledge number because it is a duplicate. When SPX receives a duplicate packet, SPX resends its acknowledgment of the duplicate packet.

Any packet with a sequence number less than the specified acknowledge number has been correctly received by SPX and need not be retransmitted.

SPX manages this field; applications need not be concerned with it.

Allocation Number

SPX uses this field to implement flow control between communicating applications.

The receiving endpoint maintains this number, which is checked by the sending endpoint. SPX sends packets only until the local sequence number equals the allocation number of the remote partner.

SPX has a window of 1. The allocation number minus the acknowledge number plus one ((Alloc# - ACK#) + 1) indicates the number of listen buffers outstanding in one direction on the connection. The sender checks its available buffers and then uses the formula to set its allocation number. The allocation number increments from 0x0000 to 0xFFFF and wraps to 0x0000.

SPX manages this field; applications need not be concerned with it.

SPX Data Flow and Sequence

SPX allows both uni-directional and bi-directional communication.

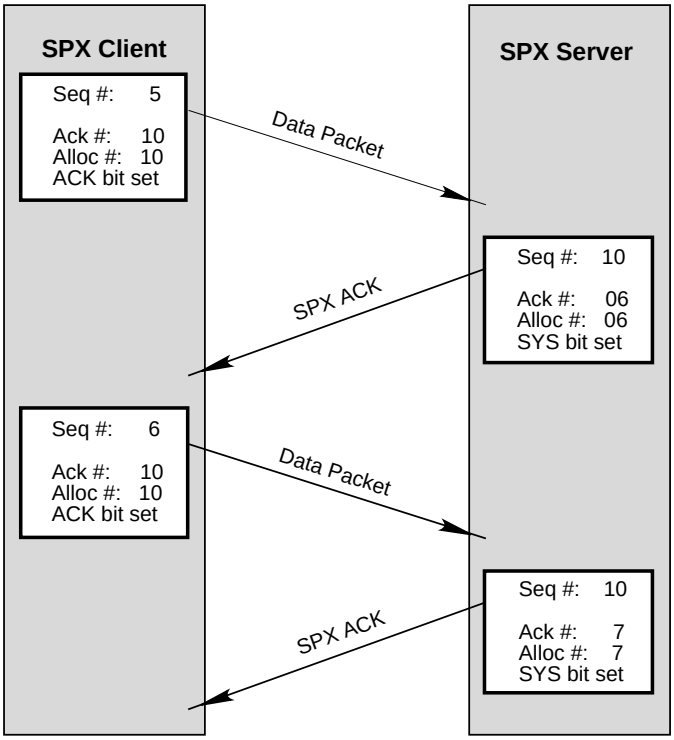
Uni-directional Communication

A source node sends SPX packets containing data. The SPX header has the acknowledge (ACK) bit set. Destination nodes respond with acknowledgment (ACK) packets indicating that the data packet was received and that the next packet in the sequence can be sent. The ACK packet header has the SYS bit set.

The SPX client transmits data packets to an SPX server and receives an acknowledge packet for each data packet sent. SYS packets do not consume sequence numbers.

Figure 3-3 illustrates uni-directional SPX communication.

Figure 3-3
SPX Uni-Directional Data Sequence

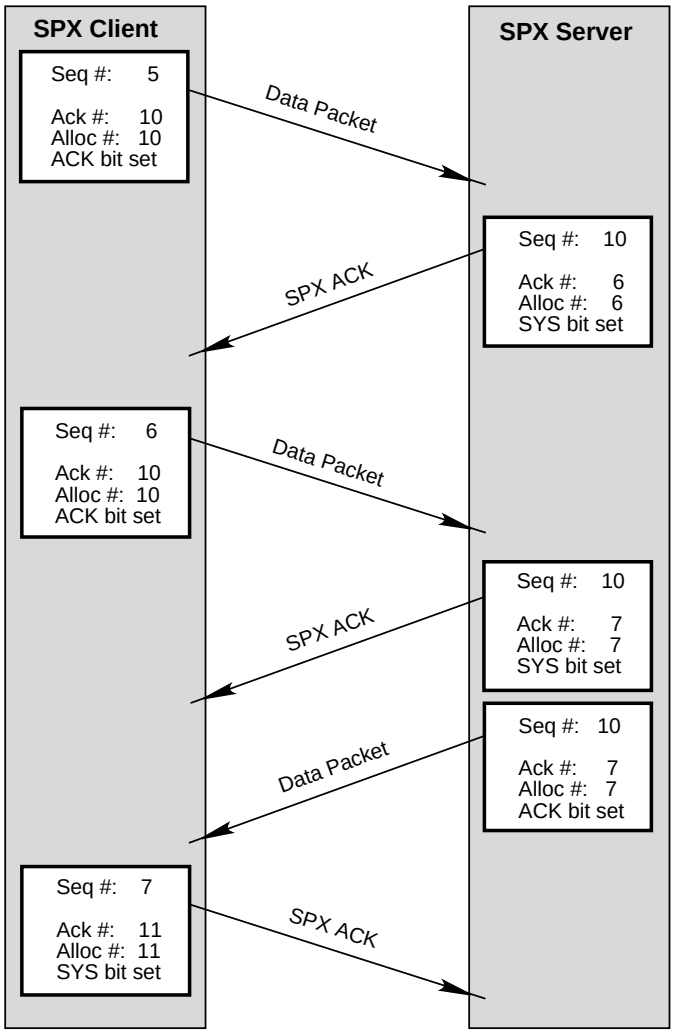


Bi-directional Communication

Data packets are sent by both nodes. ACK packets are sent by the receiving endpoint. An acknowledgment from the receiving end can be sent with a data packet as well as with a SYS packet. Each data packet must be acknowledged before the next packet of data is sent.

Figure 3-4 illustrates bi-directional SPX transmission between an SPX client and an SPX server.

Figure 3-4
SPX Bi-directional Data Sequence



SPX Flow Control

Flow control regulates the speed with which packets are sent and received by the sending and receiving processes.

SPX implements flow control between communicating applications by assigning an Allocation number (in the header) which is maintained by the receiving endpoint. The sending endpoint checks the Allocation number. SPX sends packets only until the local Sequence number equals the Allocation number of the remote partner.

Flow Control on Incoming Data

If an application cannot receive data as fast as the sender is sending it, SPX (on the receiving side) acknowledges the last received packet. SPX on the sending side queues up a read-side retry procedure that tries again.

If SPX cannot deliver packets on the second try, it increases the time interval for subsequent retries. After SPX retries the maximum number of times, it generates a disconnect.

Flow Control on Outgoing Data

If SPX is sending data faster than a remote transport endpoint can receive it, the receiving endpoint acknowledges the last data but does not increment the allocation number, which closes the SPX receive window.

The sender then monitors the connection for one of the following conditions:

- If the remote transport endpoint opens the window again, SPX begins transmitting data again.
- If the remote transport endpoint becomes inactive (the SPX watchdog detects this), SPX drops all pending data and generates a disconnect indication to the stream head.

- If the remote transport endpoint remains active but unable to receive data, the SPX watchdog continues to monitor the connection and either keeps it open indefinitely or until the endpoint becomes inactive.

SPX Connection Management

Since SPX is a connection-based service, SPX maintains a connection table that maps client SPX connections to the appropriate SPX application.

The SPX link to IPX is created and held together by the NetWare Protocol Stack daemon (NPSD). If NPSD dies, the architecture that links IPX and SPX is destroyed. When the link is destroyed, SPX drops all outbound data. Disconnect indications are then generated to all local endpoints.

To clear broken or inactive connections, SPX uses watchdog and timeout procedures.

SPX Watchdog

The SPX watchdog monitors for inactive connections. A connection is labelled inactive when a connection does not receive any packets for a set interval. The watchdog sends watchdog packets to the inactive connections and goes back to sleep.

When the watchdog awakens, it checks the inactive connections as follows:

- If the connection acknowledged the watchdog packet, the connection remains active.
- If the connection failed to acknowledge the watchdog packet while the watchdog process was asleep, the watchdog clears the connection.

SPX Timeout

SPX also clears active connections that fail to acknowledge a data packet. When a connection fails to acknowledge a packet within a specified amount of time, SPX resends the packet.

SPX resends the packet a specified number of times. After each retry, the wait interval is incremented until the interval reaches a maximum interval. If the connection fails to acknowledge the packet after the specified number of retries, SPX assumes the connection has failed and clears the connection.

chapter 4

Enhanced Sequenced Packet Exchange (SPXII) Protocol

What Is SPXII?

SPXII is the designator of Novell's enhanced version of the SPX protocol. (Because of naming constraints on some platforms, SPXII can also be designated as SPX2.)

SPX (discussed in Chapter 3) is a guaranteed delivery, connection-oriented transport protocol, and, like IPX, it is an implementation of the Xerox Network Standard (XNS) protocol specification. SPX has been supported by Novell since NetWare 2.0a.

In developing an enhanced successor to SPX (and at the same time removing its recognized limitations), the following requirements were specified:

- Backward compatibility with SPX
- Negotiation and use of larger packets
- True windowing
- Better support for the Transport Layer Interface (TLI / XTI).

Before discussing SPXII, the terminology that applies to network connections needs to be briefly clarified:

Server refers to the network node that is waiting or has received a connection request. **Host**, **passive endpoint**, and **listener** can be considered synonymous terms.

Client refers to the network node that is making or has made the connection request. It can also be referred to as the **active endpoint** or the **requester**.

Connection partner refers to the opposite endpoint and **connection endpoint** refers to either endpoint participating in the connection, without respect to its active or passive status.

Packet size includes both the header and the data.

The discussion in this chapter covers the following topics:

- SPXII features and enhancements
- SPXII packet structure
- SPXII packet header fields
- SPXII data flow
- SPXII connection management
- SPXII windowing
- SPXII programming interface

How SPXII Works

The SPXII features and enhancements are discussed following.

Backward Compatibility with SPX

Because SPXII has been designed to be compatible with SPX

- Applications written to SPXII can run in a mixed SPX/SPXII environment.
- Applications written to SPX can run with the SPXII driver.

Compatibility on the Wire

SPXII is compatible with SPX on the wire. SPXII achieves compatibility on the wire through bimodal operation. An SPXII server or client can communicate with an SPX server or client by detecting the type of connecting partner.

When an SPXII client requests a connection with a server, the client does not know whether the server is SPXII- or SPX-based. A bit in the Connection Control field of the SPXII header (called the SPX2 bit) on the connection request indicates whether the requester wants SPXII services. The client side uses this newly defined SPX2 bit then to indicate that it prefers an SPXII connection (no other modification is made to the connection request).

- An SPXII-based server recognizes the SPX2 bit and acknowledges the connection with the SPX2 bit set. The SPX2 bit remains set on all packets until the connection is terminated.

- A server that is not based on SPXII cannot recognize the SPX2 bit, and so it responds with a normal SPX acknowledgment. The client then responds with a normal SPX connection and does not set the SPX2 bit on subsequent packets.

Similarly, an SPXII server waiting for a connection request does not know whether the connecting clients will be SPXII-based or not. Therefore, the SPXII server examines the SPX2 bit for all connection requests and responds appropriately.

- For connection requests with the SPX2 bit set, the SPXII server sets the SPX2 bit on the connection acknowledgment.

- In all other cases, the server does not set the SPX2 bit and treats the connection as an SPX connection.

Programming Interface Compatibility

SPXII is compatible with SPX with regard to its TLI / XTI programming interface. SPXII supports all system calls that were supported by SPX, and applications written for SPX function properly with SPXII. The values returned in a few SPXII calls do vary, however, to reflect SPXII's larger message sizes and new functionality.

Large Packets

SPXII reduces the amount of traffic on the wire by allowing larger packets and by cutting down on the number of acknowledgments.

Packet size refers to the entire packet, including the IPX header as well as the SPXII data.

SPXII provides a message-level service rather than a packet-level service (as SPX provides). An application can hand SPXII large, non-packet sized messages which SPXII can then partition into appropriately sized packets, the size having been determined by negotiation.

The only size limitation is that which is imposed by the system (the Maximum Stream Message Size).

The original XNS specification (of which both IPX and SPX are implementations) defined the length of a network packet as 576 bytes—allowing 512 bytes of data and 64 bytes for the header. IPX faithfully incorporated this limitation, and in some early implementations, enforced it.

On the other hand, applications that used SPX were responsible for determining whether a larger packet could be used.

IPX no longer sets a maximum size on the packet. Today the LAN driver determines the maximum size of a packet and the SPXII driver handles packet size negotiation for the application. This means that applications using SPXII—which is now a message level service—no longer need to determine packet size.

Large Packet Negotiation

SPXII includes a mechanism for determining the largest packet size supported between endpoints. The maximum packet size is determined during connection establishment. Routers and bridges along a packet's route to its destination might not support the same packet size as the packet's LAN driver.

This size negotiation is accomplished by sending a packet of maximum driver size to the other endpoint. If the endpoint does not respond within an aggressive amount of time, SPXII sends a packet of smaller size. This continues until the receiving endpoint acknowledges the size negotiation packet. For example, on Ethernet the packet size sequence for negotiation is 1500, 1492, 1474, 1024, and 576 bytes.

SPXII also deals with the possibility of different send and receive packet sizes. When parallel routes exist, the path taken by a packet from point A to point B is not necessarily the route that will be taken from point B to point A. In such cases, packet size must be negotiated in both directions and then renegotiated whenever a route changes.

Windowing Protocol

Window size refers to the number of packets that can be sent before an acknowledgment is required. SPX has an effective send window size of one: after an endpoint (server) sends a packet, it waits for an acknowledgment.

SPXII, however, is a true windowing protocol. It uses the same header fields to fully support windowing and even to support different window management algorithms.

The SPXII windowing protocol allows the transmitter to send multiple packets before requesting an ACK. The receiver determines and maintains the window size for its half of the session. The receiving endpoint is “passive,” that is, it sends acknowledgments only when the transmitting endpoint sets the ACK bit in the header.

When the receiving endpoint receives packets, and if a packet or packets are missing, SPXII allows the receiving endpoint to request the sender to resend the missing packets.

Additionally, SPX has only the positive acknowledgment, whereas SPXII has both a negative acknowledgment (NAK) and a positive acknowledgment (ACK). The information derived from a negative acknowledgment allows a receiver to request that the transmitter resend a specific packet, or a range of packets, without retransmitting the full window.

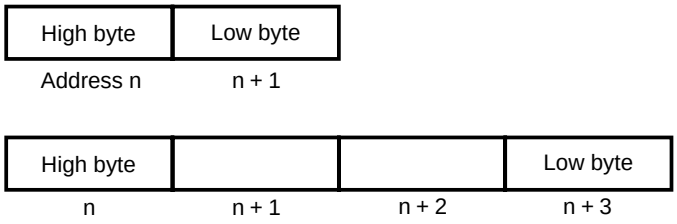
SPXII Packet Structure

The SPXII packet is a superset of the SPX packet. The header contains the same fields in the same positions, but the SPXII packet also includes a 2-byte Negotiation Size field. Additional flags are defined in the Connection Control field, especially the SPX2 bit that distinguishes SPXII from SPX packets.

SPXII uses an extended header that contains a 2-byte Negotiation Size field. The SPXII packet, therefore, consists of a 44-byte header followed by data. The minimum packet size is 44 bytes (the header only) and, if negotiation is not done during connection establishment, the maximum size is 576 bytes (header plus 532 bytes of data).

All the numeric fields in the header are byte-order sensitive, and the data must be sent in hi-lo or network order, as illustrated in Figure 4-1.

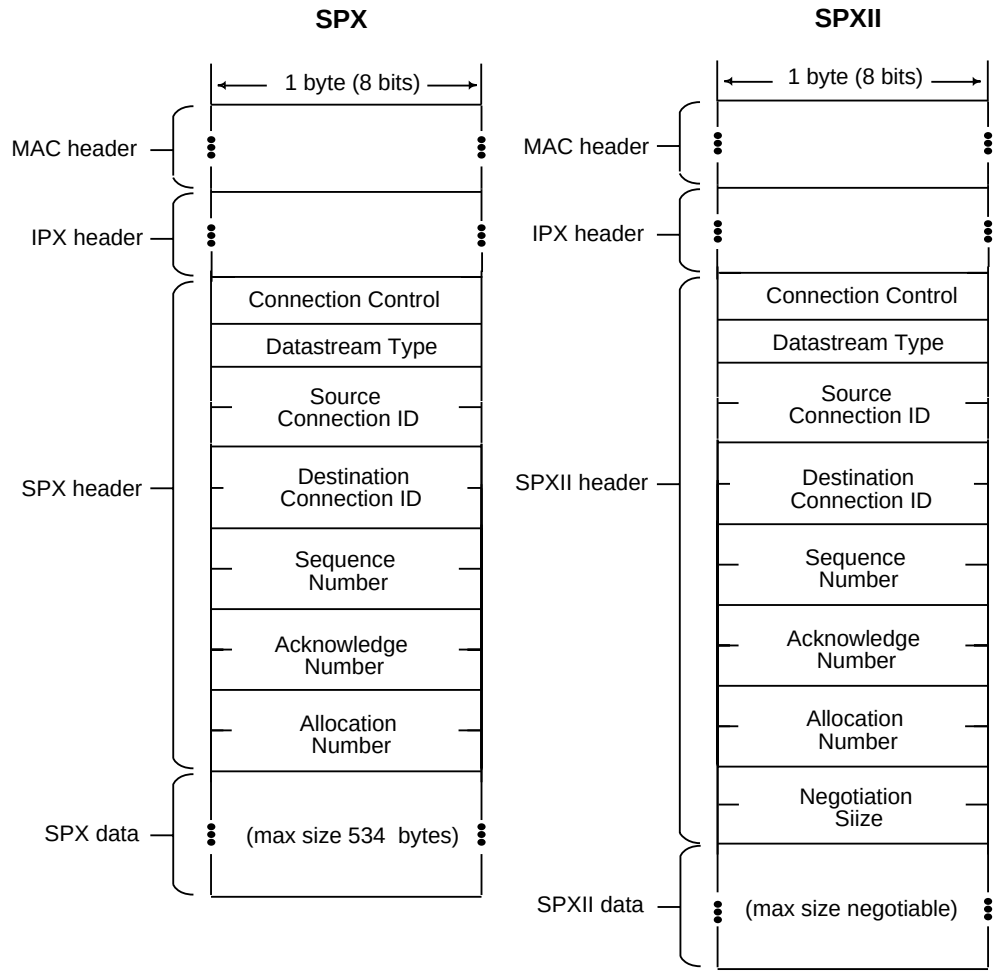
Figure 4-1
Byte Order



The content and structure of the packet's data portion are entirely the responsibility of the application using SPXII and can take any format.

Figure 4-2 illustrates the SPXII packet structure and shows how it differs from the SPX packet structure.

Figure 4-2
SPX and SPXII Packet Structure



The fields of the SPXII header are described in the following pages.

SPXII Header Fields

Table 4-1 shows the field definition for SPXII, including the IPX encapsulation (descriptions of each field follow the table).

Table 4-1
Packet Header Fields for IPX/SPXII

Field	Field Type	Byte Order	Type and Size	Value
Checksum	IPX	hi-lo	uint16 *	usually 0xFFFF
Length	IPX	hi-lo	uint16	42 - 65,535 bytes
Transport control	IPX		uint8	0 - 16 hops
Packet type	IPX		uint8	0 - 0xFF
Destination address.network	IPX	hi-lo	uint8[4]	0 - 0xFFFFFFFF
Destination address.node	IPX	hi-lo	uint8[6]	0 - 0xFFFFFFFFFFFF
Destination address.socket	IPX	hi-lo	uint8[2]	0 - 0xFFFF
Source address.network	IPX	hi-lo	uint8[4]	0 - 0xFFFFFFFF
Source address.node	IPX	hi-lo	uint8[6]	0 - 0xFFFFFFFFFFFF
Source address.socket	IPX	hi-lo	uint8[2]	0 - 0xFFFF
Connection control	SPX		uint8	(See description below)
Datastream type	SPX		uint8	(See description below)
Source connection ID	SPX	hi-lo	uint16	(See description below)
Destination connection ID	SPX	hi-lo	uint16	(See description below)
Sequence number	SPX	hi-lo	uint16	(See description below)
Acknowledge number	SPX	hi-lo	uint16	(See description below)
Allocation number	SPX	hi-lo	uint16	(See description below)
Negotiation size	SPXII	hi-lo	uint16	(See description below)

*Uint8, uint16 refer to unsigned integers. The length designator is given in the number of bits.

Checksum

Checksum is configurable. When turned on, it holds a checksum of the entire packet (including the IPX header, but not the Transport Control field). The checksum is on a per-connection basis because SPX/SPXII is a connection-oriented protocol. This can be set via the t_optmgmt structure in the **t_optmgmt** or **t_connect** call. The SPXII driver sets the IPX Checksum field to 0xFFFF.

Length

The SPXII driver sets the IPX Length field to the length of the entire packet (including the IPX header).

Transport Control

This field is used by the IPX router. The SPXII driver always sets this field to zero before sending the packet. Each router increments the field before sending the packet on. If the packet passes through 16 routers, the sixteenth router discards the packet.

Packet Type

The SPXII driver automatically fills in this field. For SPX/SPXII, it is always set to 5.

Destination Address

When establishing an SPXII connection, the application must pass the destination address to SPXII. When the connection has been established, the SPXII driver fills in the destination address with the 12-byte IPX address of the other endpoint (Network, Node, and Socket) for all subsequent outgoing packets on this connection.

Table 4-2 contains a list of the IPX address fields:

Table 4-2
IPX Address Fields

Field	Size	Byte Order
Network Address	uint8[4]	hi-lo

Table 4-2
IPX Address Fields

Field	Size	Byte Order
Node Address	uint8[6]	hi-lo
Socket Number	uint8[2]	hi-lo

Network Address is set to zero (0) when the destination node and the source node reside on the same network. When this is the case, the packet is not processed by an IPX router.

Node Address, for clients, contains the physical address of the destination node. If a physical network needs less than 6 bytes to specify a node address, the address should occupy the least significant portion of the field and the most significant bytes should be set to zero (0). Both SPX and SPXII explicitly disallow a broadcast node address (0xFF FF FF FF FF FF).

Socket Address is the intranode destination address. Sockets route packets to different processes within a single node.

For an explanation of IPX addressing, see “IPX Addressing” on page 2.

Source Address

The SPXII driver fills in the source address of the sender. The source address, like the destination address discussed above, is a 12-byte IPX address with Network, Node, and Socket fields.

Again, the Network Address field is set to zero (0) when the destination node and the source node reside on the same network.

In a client-server dialogue, the server node usually listens on a specific socket for connection requests. In such a case, the source socket is not necessarily the same or even significant. All that matters is that the server sends its replies to the source socket contained in the connection request packet. For example, all remote application servers have the same socket address, but requests to them can originate from any socket number.

Otherwise, the source address follows the same conventions as those for the destination address. For an explanation of IPX addressing, see “IPX Addressing” on page 2.

Connection Control

This field is in the SPX/SPXII packet header itself and contains single-bit flags used by SPXII to control the bidirectional flow of data across a connection. (SPX uses this field to indicate whether the packet is a system or application packet or the end of the message.)

Table 4-3 lists flag values.

Table 4-3
Flags for SPXII Connection Control

Value	Symbol	Description
0x01	XHD	Reserved for extender header
0x02	RES1	Reserved; do not use
0x04	NEG	Negotiate size request/response
0x08	SPX2	SPXII packet type
0x10	EOM	Set by SPX/SPXII to indicate end of message. This bit reflects the state of T_MORE from a TLI application.
0x20	ATN	Reserved for attention indication. (Not currently supported by SPXII.)
0x40	ACK	Set to request that the receiving partner acknowledge the receipt of this packet.
0x80	SYS	Set to identify a system packet. System packets are internal SPX packets. They are not delivered to an SPX application, and do not consume sequence numbers.

Datastream Type

SPXII uses this field to indicate the type of data found in the packet; it can use any of the values in Table 4-4. In this it differs from SPX, which ignores any values in this field other than 0xFE and 0xFF.

Note Although SPXII allows an application to define a value for the Datastream Type field, we strongly recommend that you do not use this field. We suggest instead that you add a control field to your application data packets.

Table 4-4 lists flag values.

Table 4-4
Flags for Datastream Type

Value	SPXII	SPX
0x00 - 0x7F	Defined by client	Defined by client
0x80 - 0xFB	Reserved	Defined by client
0xFD	Orderly release request	Defined by client
0xFE	Informed disconnect notification	End-of-Connection packet
0xFF	Informed disconnect acknowledgment	End-of-Connection acknowledgment

Source Connection ID

SPXII generates the identification number for the Source Connection ID field during connection establishment. The source endpoint of the packet assigns the value.

Destination Connection ID

SPXII generates the identification number for the Destination Connection ID field during connection establishment. The destination endpoint of the packet assigns the value.

SPXII uses Source and Destination Connection ID numbers to demultiplex packets from multiple connections that arrive on the same socket. All concurrently active connections on a given node are guaranteed to have unique connection ID numbers.

Sequence Number

Each side of the connection keeps its own sequence count.This field keeps a count of packets transmitted in one direction on a connection. The number wraps to zero after reaching 0xFFFF.

This field is set to 0x0 for SPXII packets that negotiate size, except during lost-packet error recovery.

SPXII manages this field; applications need not be concerned with it.

Acknowledge Number

SPXII uses this field to indicate the sequence number of the next packet expected to be sent from the other endpoint. Any packet with a sequence number less than the specified acknowledge number has been correctly received by SPXII and need not be retransmitted.

SPXII manages this field; applications need not be concerned with it.

Allocation Number

SPXII uses this field to

- Identify to the connection partner the largest sequence number that can be sent.
- Control the number of unacknowledged packets outstanding in one direction in the connection.

The receiving endpoint maintains this number, which is checked by the sending endpoint. SPXII sends packets only until the sequence number equals the allocation number.

SPXII manages this field; applications need not be concerned with it.

Negotiation Size

This field exists as part of the SPXII header on all packets, except on the connection request packet. It contains a length value, and the significance of the value depends on the type of packet. (If negotiation is not done during connection establishment, the value of this field should be 576, the default packet size.)

The Negotiation Size field is neither part of the SPX header nor part of the packet header for an SPXII connection request.

SPXII Data Flow

As a guaranteed delivery service, SPXII requires that the connection endpoints verify that each packet has been received. As a windowing protocol, SPXII also allows an endpoint to send multiple packets before requesting that the receiving endpoint acknowledge the received packets.

When the receiving endpoint receives packets, and a packet or packets are missing, SPXII allows the receiving endpoint to request that the sender retransmit the missing packets.

This section describes SPXII packet format and sequence:

- Data packet format
- SPXII ACK packet format
- SPXII NAK packet format
- Packet sequence without a NAK
- Packet sequence with a NAK

For information on connection management packets and sequences, see “SPXII Connection Management” on page 65.

Data Packet Format

Figure 4-3 on page 56 shows the fields and values of an SPXII data packet.

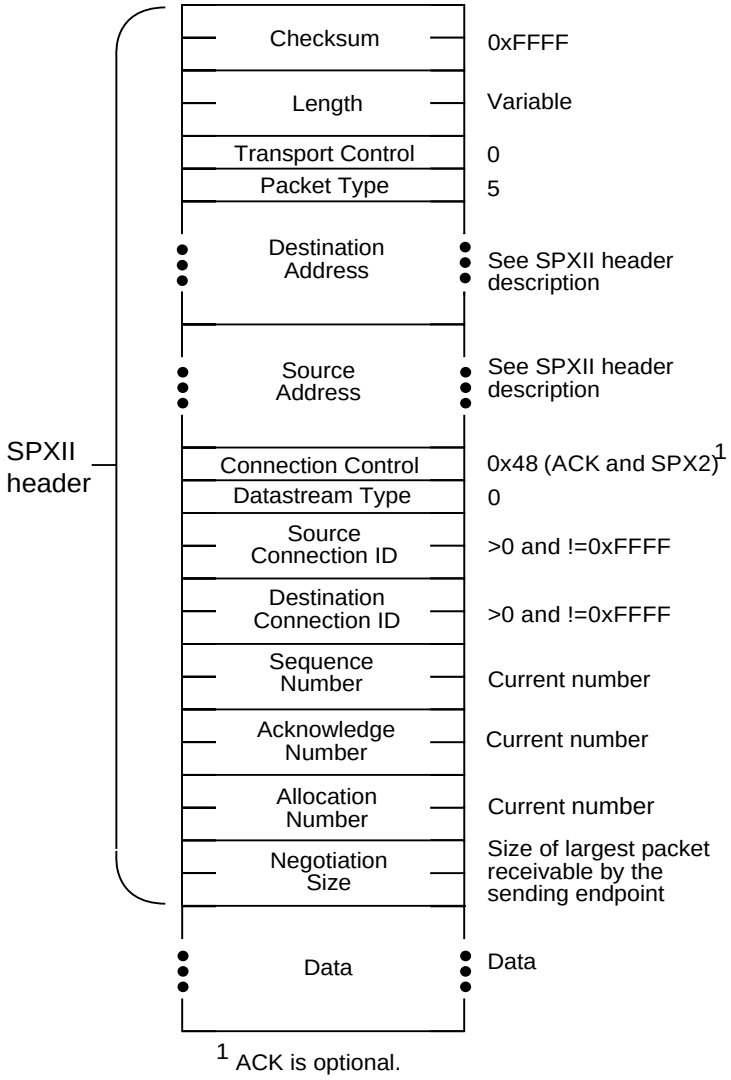
The value in the Length field reflects the actual size of the data packet. A data packet can vary from an SPXII header alone to an SPXII header plus the maximum negotiated data size (for a range of total packet size of 576 to 65,535 bytes).

The ACK bit in the Connection Control field is optional. For NetWare in a UNIX environment, the SPXII driver sets the ACK bit when any of the following conditions occur:

- The packet has the EOM bit set in Connection Control field.
- No more data is currently on the Streams write queue.
- This packet fills the receiver's last available buffer (according to the window size in the receiver's last ACK).

The Sequence Number field tracks the number of data packets the sender has transmitted. The first data packet has a sequence of zero. Once the driver has placed that packet on the sending queue, the driver increments the count so that the next data packet will have a sequence number of 1.

Figure 4-3
SPXII Data Packet



The Acknowledge Number field tracks the number of data packets the sender has received from the other endpoint. It is incremented on the reception of a packet and contains the value of the sequence number expected in the next data packet.

For example, the Acknowledge Number field would have the following values under the described conditions:

0	No data packets have been sent by the other endpoint.
4	Four data packets have been sent by the other endpoint (0,1,2,3) and the next data packet to arrive will have a Sequence Number of 4.

The Allocation Number indicates the current receive window size of the sender. Actual window size is calculated by subtracting the acknowledge number from the allocation number and adding one: (Alloc# - ACK#) + 1.

SPXII ACKs

Figure 4-4 on page 58 shows the fields and values of an SPXII acknowledgment (ACK) packet.

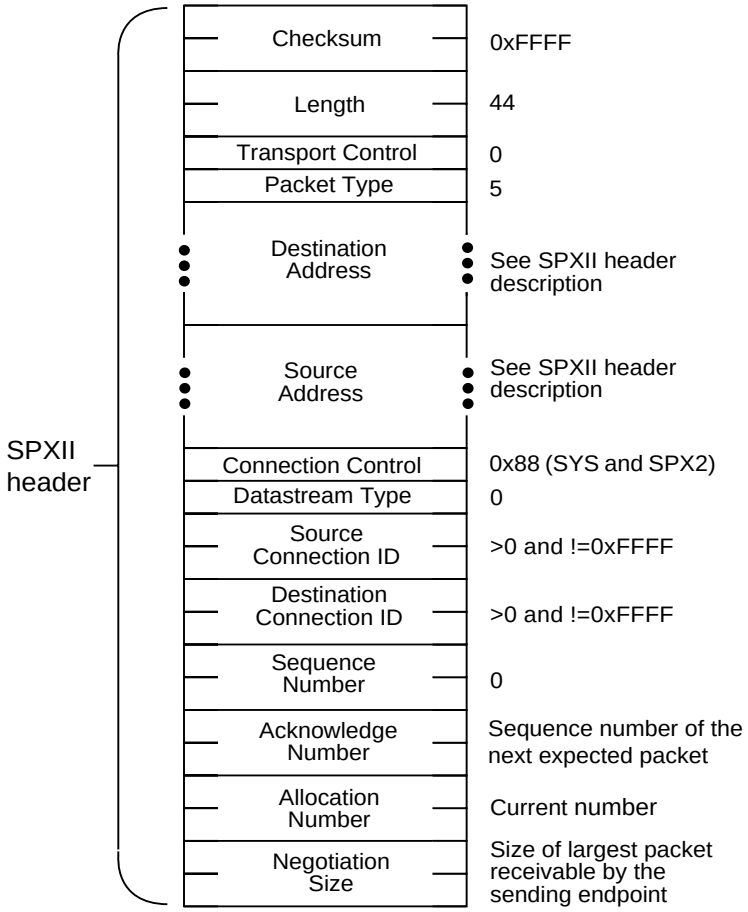
SPXII ACKs are used to acknowledge SPXII data packets.

Note

To determine whether any data packet received contains an implicit acknowledgment (“piggy-back” ACK), check whether the ACK or the Alloc numbers have been updated.

The SPXII ACK has the fields and values set as shown in Figure 4-4.

Figure 4-4
Fields and Values of an SPXII Data ACK



The Connection Control field has the SYS and SPX2 bit set.

The Sequence Number field is set to zero (0).

The Acknowledge Number field is set to acknowledge the data packet(s) received (this means that all data packets up to but not including the acknowledgment number have been received).

The Allocation Number field is set to the greatest Sequence Number that can be received.

SPXII NAKs

Figure 4-5 on page 60 shows the fields and values of an SPXII negative acknowledgment (NAK) packet.

For SPXII ACKs, the Sequence Number field is set to zero (0). For an SPXII NAK, however, the Sequence Number field is set to a non-zero value.

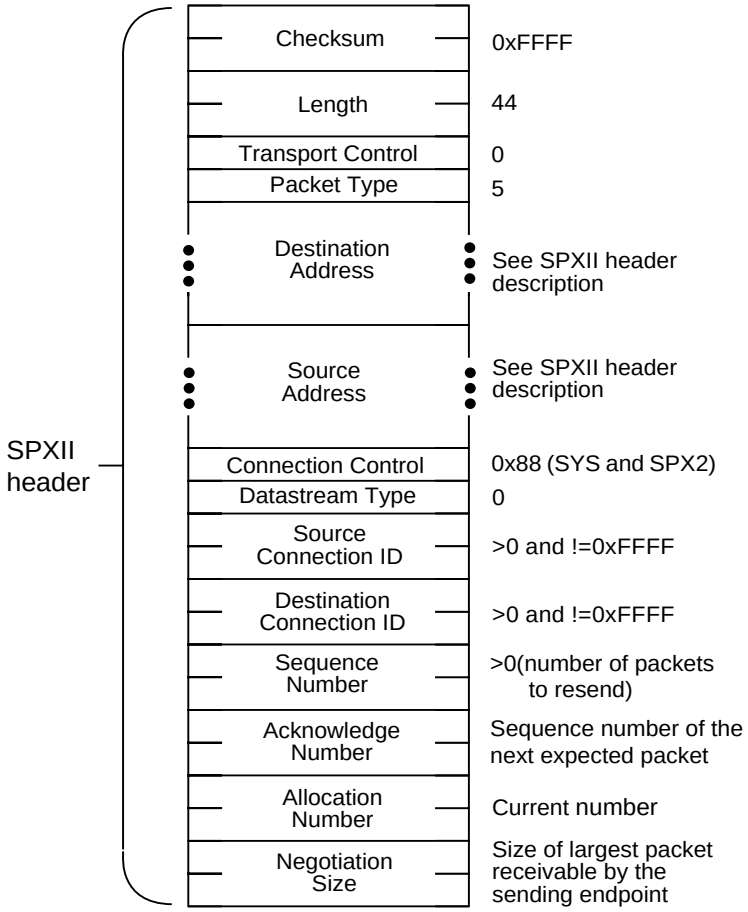
In the SPXII implementation for UNIX, the receiver sends a NAK if the sender requests an ACK but the packets in the sequence have not arrived. The receiver of the NAK (sender of data) uses the NAK to determine which packets have been lost.

The Sequence Number field contains the number of packets that need to be resent. For the example used in the previous paragraph, the SPXII driver would set the sequence number to 2 (packets 5 and 6 need to be retransmitted).

The Acknowledge Number field contains the sequence number of the first packet that failed to arrive. For example, if the receiver has received packets with sequence numbers of 1, 2, 3, 4, 7 and 8, the driver sets the acknowledgment number to 5.

The SPXII NAK packet consists only of an SPXII header (no data) with the fields and values set as shown in Figure 4-5.

Figure 4-5
Fields and Values of an SPXII NAK Packet



Sequence of Data Packets without a NAK

Figure 4-6 on page 62 illustrates the normal sequence of data packets between a client and a server (in other words, with an ACK and without a NAK).

In the case illustrated in Figure 4-6, the fields in the client's first data packet have the following values:

- The sequence number is set to 5. The client has already sent 5 data packets (0, 1, 2, 3, and 4) and they have been received up through number 4. The next packet expected is number 5.
- The allocation number is 12. Because window size is calculated with the formula (Alloc# - Ack#) +1, the client's window size is 3.

For the server, this means

- The server has sent 10 data packets because the client's first data packet shows that the acknowledge number is 10.
- The server has a window size of 5 because the server's first SPXII ACK shows that the allocation number is set to 10.

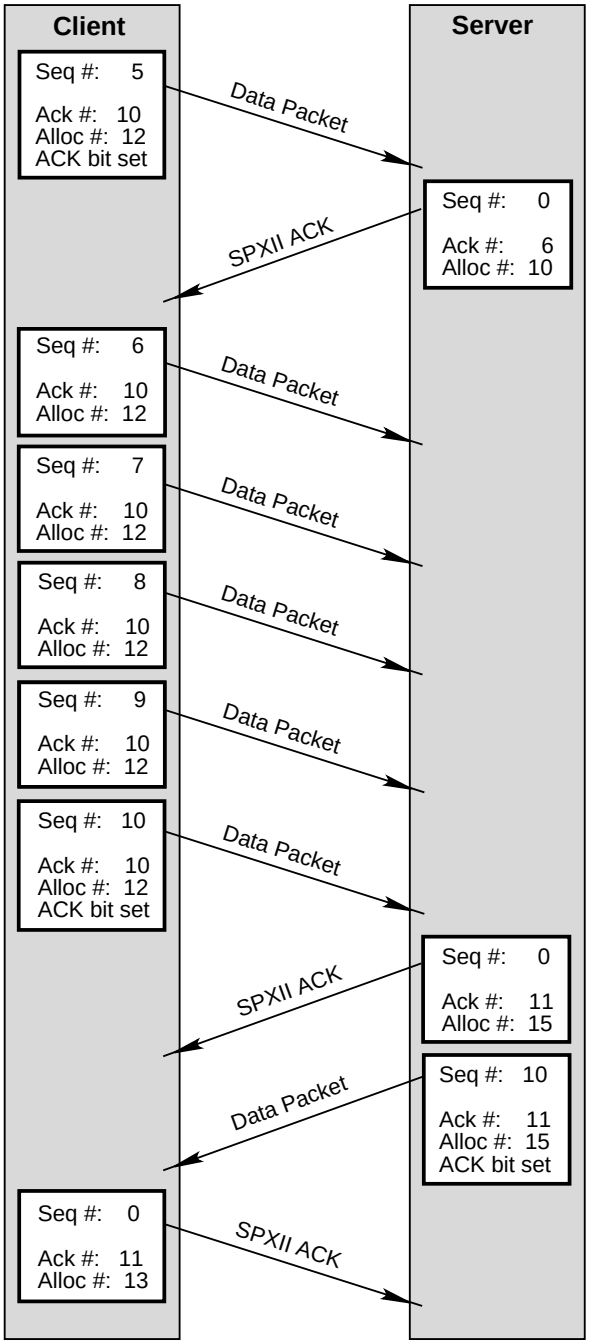
Notice that both the acknowledgment number and the allocation number increase with each received data packet. The server's second SPXII ACK packet shows this. Both numbers increase by 5.

The packet sequences in Figure 4-6 assume that the client and server are able to process each packet immediately and free the buffer. If the server is unable to process some of the packets before sending the SPXII ACK, the allocation number would be increased only by the number of free buffers.

For example, if only data packets 6, 7, and 8 were processed and their buffers freed, the allocation number would be 13 (rather than 15, which is shown in Figure 4-6).

A data packet can also acknowledge received packets. This type of packet is a "piggy-back" ACK. (We know it is not a NAK because it contains data.)

Figure 4-6
Normal Data Sequence



In the example in Figure 4-6, the server could have used the data packet to acknowledge the received packets. If the packets had been combined in a data packet with a “piggy-back” ACK, the fields would have the following values:

Seq:	10
Ack:	11
Alloc:	15
ACK bit:	set

Notice that these are the same values as the ACK packet. The client would use the value in the Acknowledge Number field as the ACK for packets 6 through 10.

Sequence of Data Packets with a NAK

Figure 4-7 on page 64 illustrates the packet sequence when a packet doesn’t arrive and an endpoint must answer with a negative acknowledgment.

It uses the same situation as Figure 4-6 on page 62. The client has a window size of 3, and the server has a window size of 5.

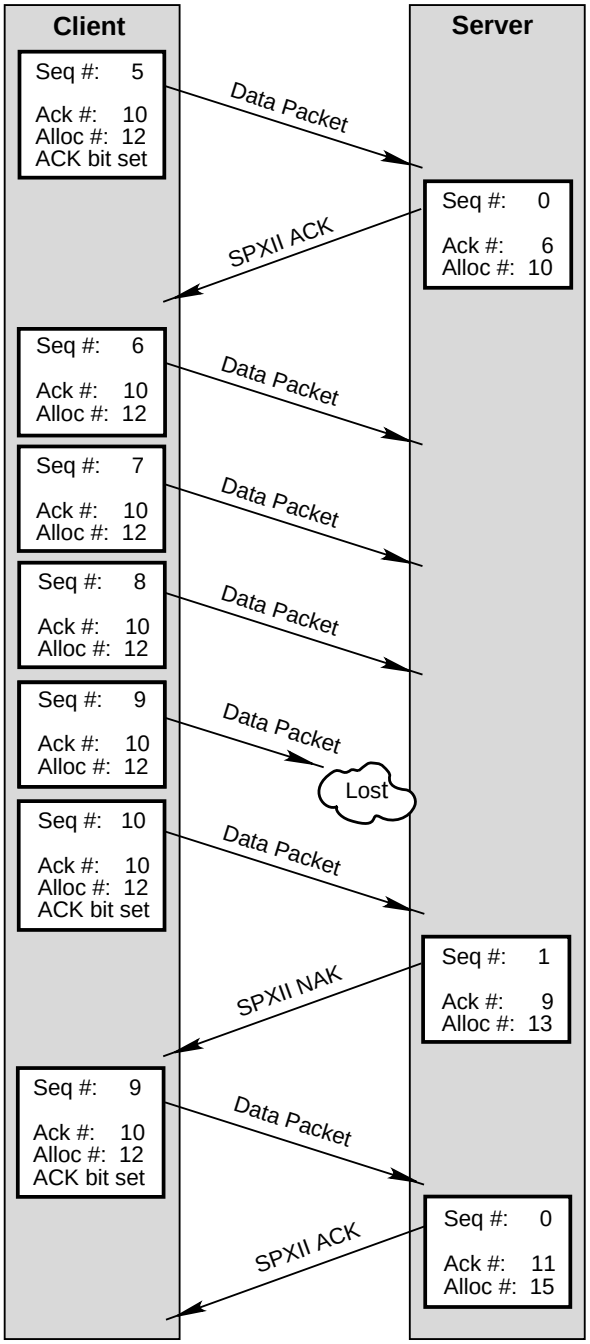
The scenario in Figure 4-7 begins when the server acknowledges packets numbered up through 4. The client has already acknowledged packets from the server up through number 9.

The server then ACKs data packet number 5, and the client sends five more data packets. However, in Figure 4-7, data packet number 9 never arrives and is lost.

When the client requests an ACK in data packet 10, the server sends a NAK, letting the client know that data packet 9 hasn’t arrived. The sequence number in the NAK indicates how many packets need to be resent. Since packet number 10 arrived, the server needs only one packet resent—packet number 9.

The ACK bit is automatically set on all retransmitted packets; therefore the ACK bit is set when the client resends packet number 9.

Figure 4-7
Data Sequence with a NAK



When the server ACKs the retransmitted data packet, notice that the ACK contains an ACK for both data packets 9 and 10. The server signals the client that packet 10 has also arrived by setting the acknowledge number to 11.

The allocation numbers in Figure 4-7 assume that the client and the server are able to process packets as soon as they arrive and that their buffers are free when sending an ACK or NAK.

The server's SPXII NAK indicates that data packets 6, 7, and 8 have been processed and their buffers freed. The buffer for data packet 10 is not free because an out-of-sequence packet can't be processed.

Note

There is no way to send a negative acknowledgment with a data packet (no "piggy-back" NAK).

SPXII Connection Management

In order to establish, maintain, and release a connection between two endpoints, SPXII requires the following types of connection management packets:

- Connection establishment packets (see next section)
- Session termination packets (see page 82)
- Watchdog packets (see page 92)
- Renegotiation packets (see page 96)

Once a connection has been established, the endpoints are in a data transfer state. For information on the format and sequencing of data packets, ACKs and NAKs, see "SPXII Data Flow" on page 54.

Connection establishment for SPXII includes negotiation of packet size. Because routers are non-deterministic with respect to the route used, it is possible to have parallel routes. Thus an SPXII connection can have different send and receive packet sizes. SPXII packet-size negotiation takes this into account and separately determines the client's and the server's send packet size.

Connection Establishment Packets

In order to establish a connection between two endpoints, SPXII requires the following types of connection establishment packets:

- Connection Request and ACK
- Session Negotiate and ACK
- Session Setup and ACK

This section not only describes the format of each packet, but the following packet sequences as well:

- SPXII to SPXII connection establishment sequences (with and without negotiation)
- Mixed SPX and SPXII connection establishment sequences

Connection Request Packet

A Connection Request packet is the first packet in the connection establishment sequence. The packet is sent by the endpoint requesting the connection (the client), and the client uses the packet to

- Request a connection
- Optionally request negotiation

Figure 4-8 on page 67 shows how the values in the IPX and SPXII headers are set for the Connection Request packet.

The SPXII Connection Request packet is the same size as an SPX connection request. Notice that the Length field in the IPX header is the same size as an SPX connection request and that the Negotiation Size field is not used.

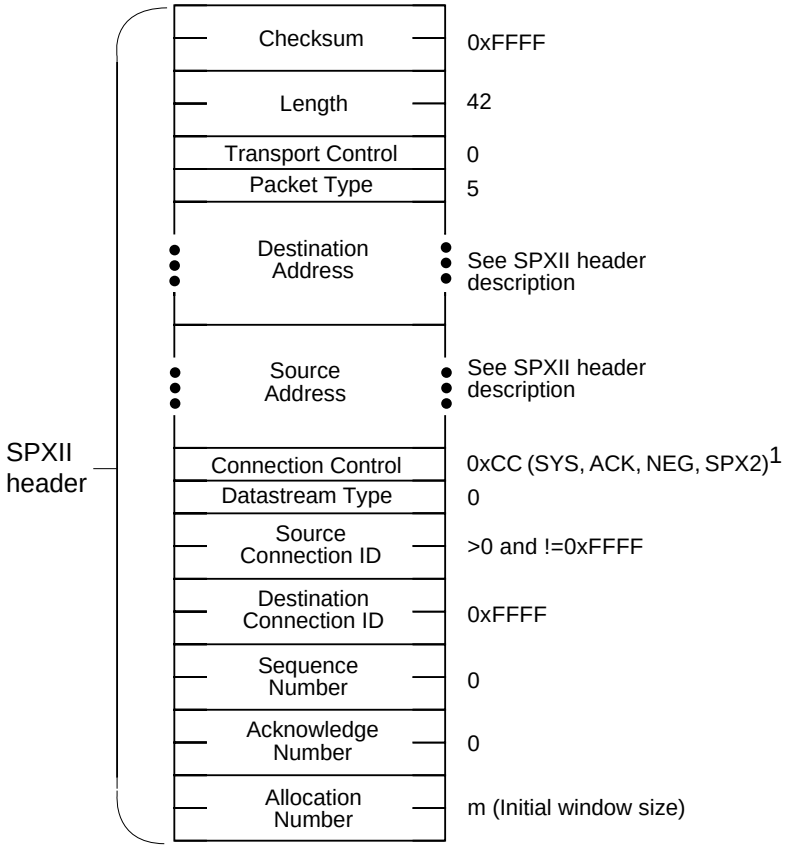
The SPXII Connection Request packet is the only SPXII packet that does not use the Negotiation Size field.

The Connection Control field can be set to request negotiation. If the client does not want to negotiate (because the connection is temporary or only small amounts of data will be transferred), the client sets this field to 0xC8 (SYS, ACK, and SPX2).

When appropriate, developers should have their client applications avoid the overhead of packet-size negotiation. If an application knows that all messages will be small (fewer than 534 bytes) or that the connection will be short-lived, the application can specify no negotiation for packet size. The TLI function **t_optmgmt**, negotiate protocol options, allows this. (This function is discussed on page 205.)

The TLI call can tell SPXII not to set the NEG bit of the Connection Control field in the Connection Request packet. With the NEG bit unset, the client and the server assume a packet size of 576 bytes and do not send any negotiation packets.

Figure 4-8
Fields and Values of the Connection Request Packet

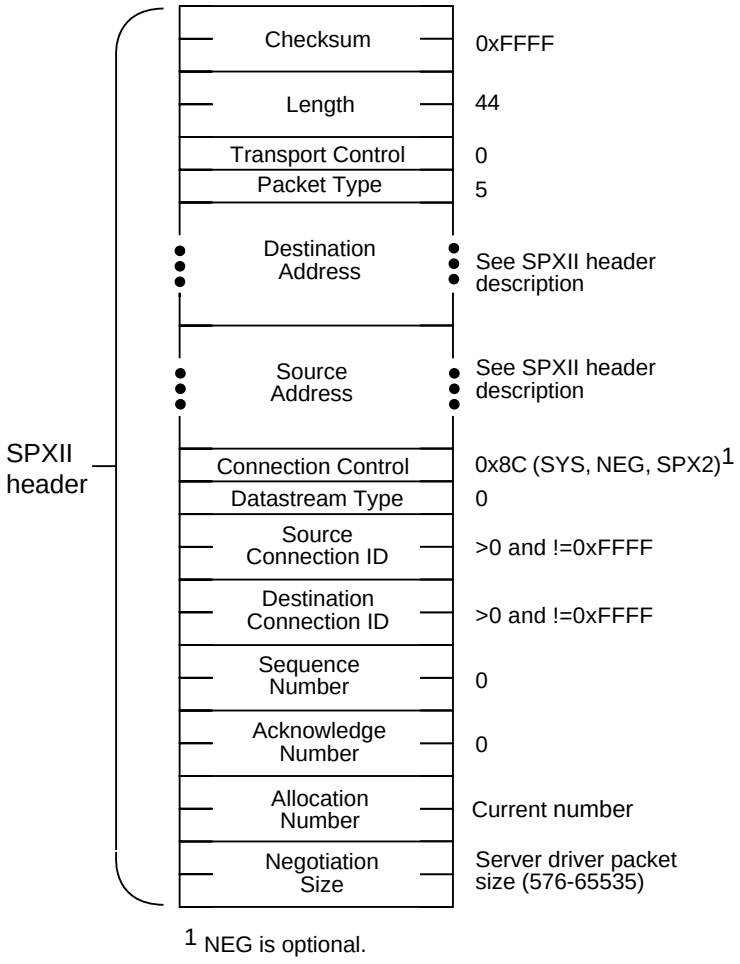


1 NEG is optional.

Connection ACK Packet

The server sends a Connection ACK packet as an immediate acknowledgment to a connection request. Figure 4-9 shows the fields and values of the packet's header.

Figure 4-9
Fields and Values of the Connection ACK Packet



Notice that the Length field has a value of 44. This packet uses the Negotiation Size field.

The server can either set the Connection Control field to 0x8C (SYS, NEG, and SPX2) to agree to negotiation or to 0x88 (SYS and SPX2) to skip negotiation.

- If the NEG flag is not set, the server is either agreeing with the client not to negotiate or informing the client that the server won't negotiate. When negotiation is declined, the next packet is a Session Setup packet sent by the server.

- If the NEG bit is set both in this packet and the Connection Request packet, both the client and server have agreed to negotiation, and the client can send a Session Negotiate packet.

The value of the Negotiation Size field depends upon whether negotiation is requested:

- If requested, the field is set to the maximum packet size of the server driver.

- If not requested, the field is set to 576.

Session Negotiate Packet

The Session Negotiate Packet allows the two endpoints to negotiate packet size as well as other information such as IPX checksum control and timeout values.

A client sends a Session Negotiate packet only if the following conditions are met:

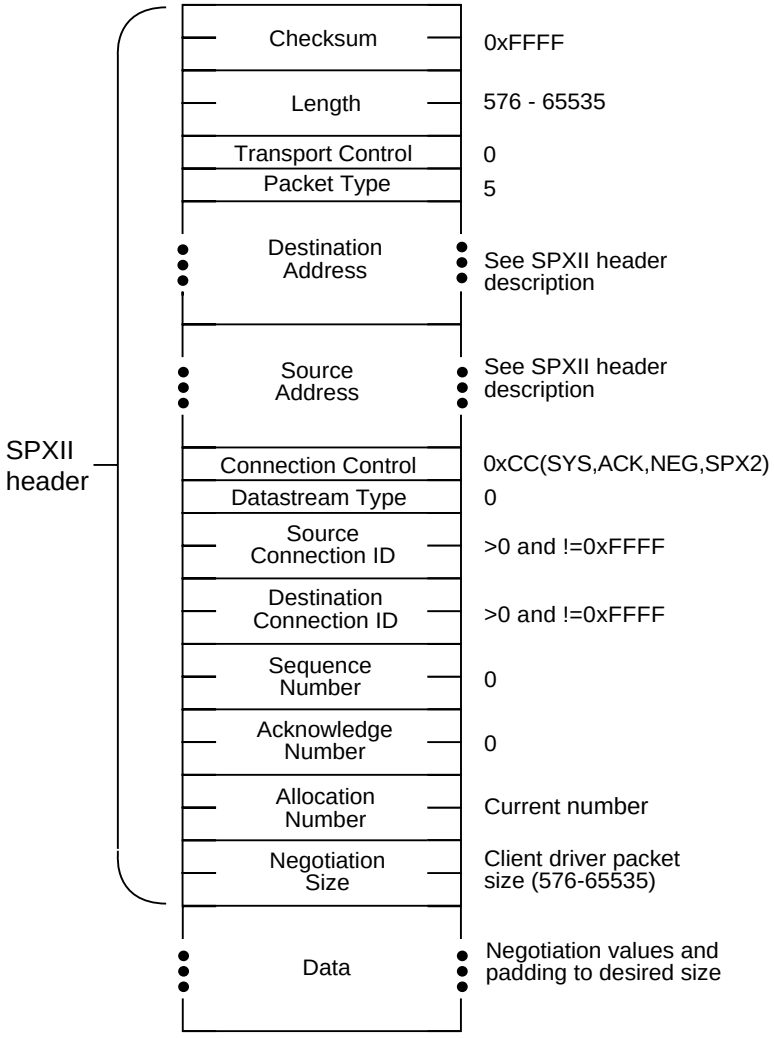
- The connection is still in the establishment phase.

- The NEG flag was set in the Connection Control field in both the Connection Request and Connection Acknowledgment packets.

If any of these conditions are not met, the client cannot send a Session Negotiate packet and must wait for the server to send a Session Setup packet.

Figure 4-10 shows the header fields and values for the Session Negotiate packet.

Figure 4-10
Fields and Values of the Session Negotiate Packet



The header is followed by *n* data bytes. The client SPXII driver pads the Data field so that the packet size equals the maximum packet size that both the server driver and client driver can support.

For example, if the client driver supports a packet size of 1024 and the server driver, 1474, the client SPXII driver pads the Data field so that the packet size is 1024 (the client can't send a larger packet).

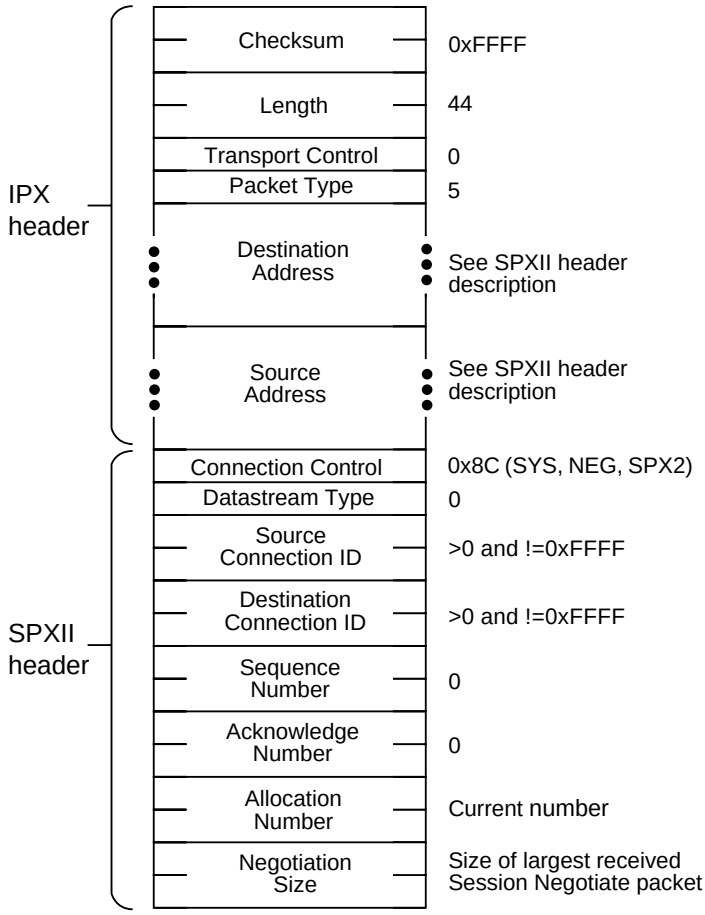
If the server does not acknowledge the Session Negotiate packet within an aggressive timeout period, the client reduces the size of the packet to the next logical driver size and then resends the packet.

Only the first 532 bytes of data can contain optional information to be negotiated between the endpoints. (See "Negotiating Other Values between Endpoints" on page 103.) The padding value is not significant and can be uninitialized data.

Session Negotiate ACK

When the server receives a Session Negotiate packet, it sends an ACK packet. Figure 4-11 shows the packet's fields and values.

Figure 4-11
Fields and Values of the Session Negotiate ACK Packet



The server uses the Length field in the IPX header to determine the size of the Session Negotiate packet. It then uses this value to set the value of the Negotiation Size field in the Session Negotiate ACK packet. If the server receives multiple Session Negotiate packets (because of delayed packets or lost ACKs), the server always sets the Negotiation Size field to the value of the largest received Session Negotiate packet.

Session Setup Packet

The Session Setup packet signals the successful establishment of the connection.

The server sends this packet when one of the following conditions is met:

- The server has sent a Session Negotiate ACK.
- The server has sent a Connection ACK and negotiation was not requested.

When the client receives this packet, the client informs the application that a connection has been established.

- For the client, the session is established and enters a data transfer state as soon as the client sends the Session Setup ACK.
- For the server, the session is established and enters a data transfer state as soon as the server receives Session Setup ACK.

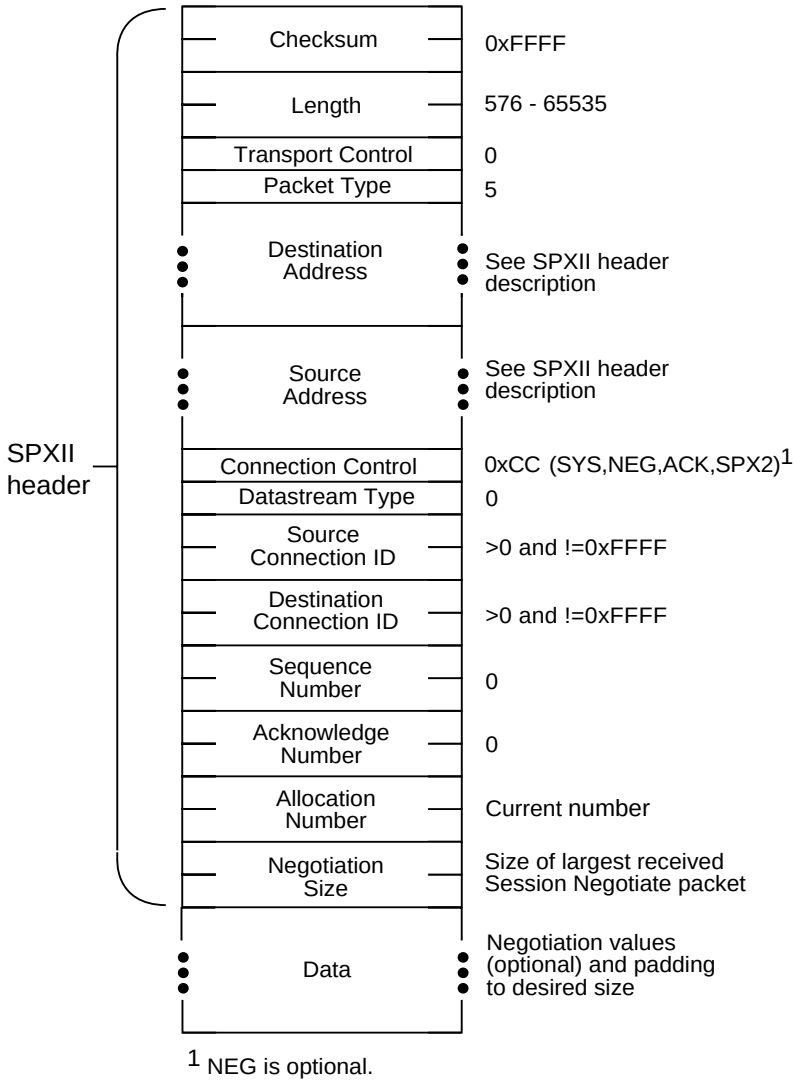
The header is followed by *n* data bytes. The server SPXII driver pads the Data field so that the packet size equals the maximum packet size that both the server driver and client driver can support.

For example, if the client driver supports a packet size of 1024 and the server driver, 1474, the server SPXII driver pads the Data field so that the packet size is 1024 (the client can't accept a larger packet).

If the client does not acknowledge the Session Setup packet within an aggressive timeout period, the server reduces the size of the packet to the next logical driver size and then resends the packet.

Figure 4-12 on page 74 shows the fields and values for the Session Setup packet.

Figure 4-12
Fields and Values of the Session Setup Packet



If the NEG flag is set in the Connection Control field, the first 532 bytes of data can contain optional information to be negotiated between the endpoints. (See “Negotiating Other Values between Endpoints” on page 103.) The padding value is not significant and can be uninitialized data.

The server can use the Session Setup packet to indicate that the server’s socket address has changed. (Applications often use one socket address for connection establishment and then assign a different socket number for each established session.)

The server’s socket address change is indicated in the Source Address field of the SPX header. The client must detect the socket change and send all future packets, including the Session Setup ACK, to this new socket address. When the socket address changes, the Connection IDs remain unchanged.

The server sets the Connection Control field to reflect the state of the connection and uses one of the following values:

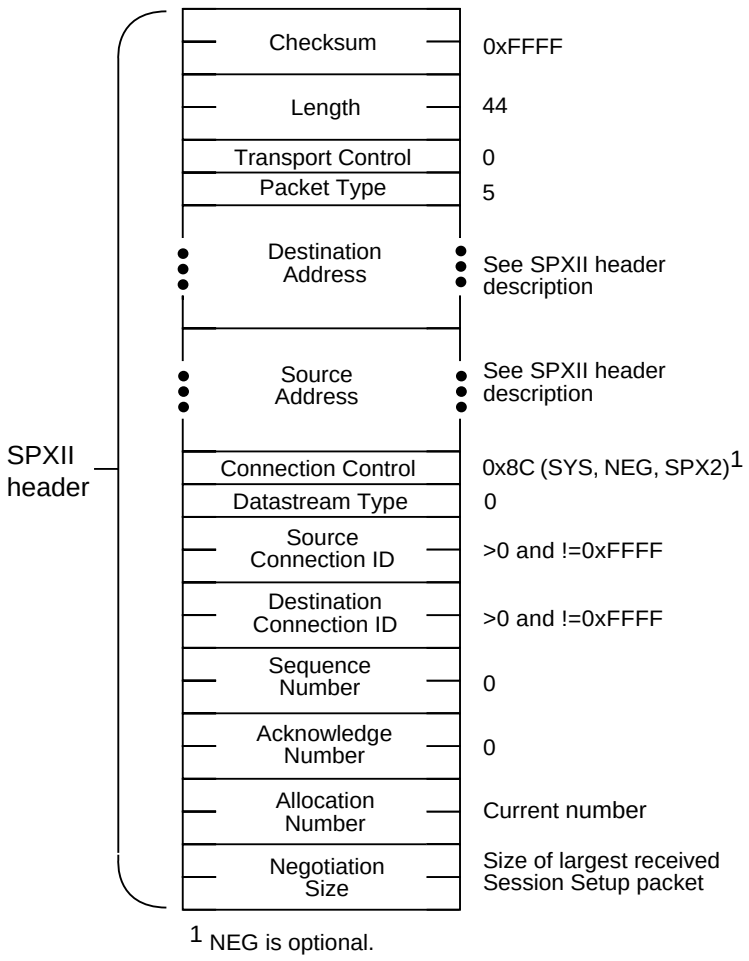
- 0xCC (SYS, ACK, NEG, and SPX2) when the connection is still in a negotiation state
- 0xC8 (SYS, ACK, and SPX2) when the connection is not in a negotiation state

The Sequence number must be set to zero. The client uses the Sequence number to differentiate between a Session Setup retry and a Renegotiate Request packet which happens to occur on the first data packet. (The first data packet has a sequence number of 0; if the first packet after session establishment is a Renegotiation Request packet, that packet would have a Sequence number of 1).

Session Setup ACK Packet

The client sends a Session Setup ACK as an acknowledgment to a Session Setup packet. Figure 4-13 illustrates the packet's header fields and values.

Figure 4-13
Fields and Values of the Session Setup ACK Packet



The client sets the Connection Control field to indicate the state of the connection.

- 0x8C (SYS, NEG, and SPX2) to reflect a negotiation state
- 0x88 (SYS and SPX2) to reflect a non-negotiation state

The client uses the Length field in the IPX header to determine the size of the Session Setup packet and to set the Negotiation Size field in the ACK. If the client receives multiple Session Setup packets (because of delayed packets or lost ACKs), the client always sets the Negotiation Size field to the value of the largest received Session Setup packet.

Packet Sequence for SPXII to SPXII Connection Establishment

This section describes the normal sequence of packets in establishing a connection between endpoints that are both using SPXII. The following types of packet sequencing are shown:

- A negotiation sequence
- A non-negotiation sequence

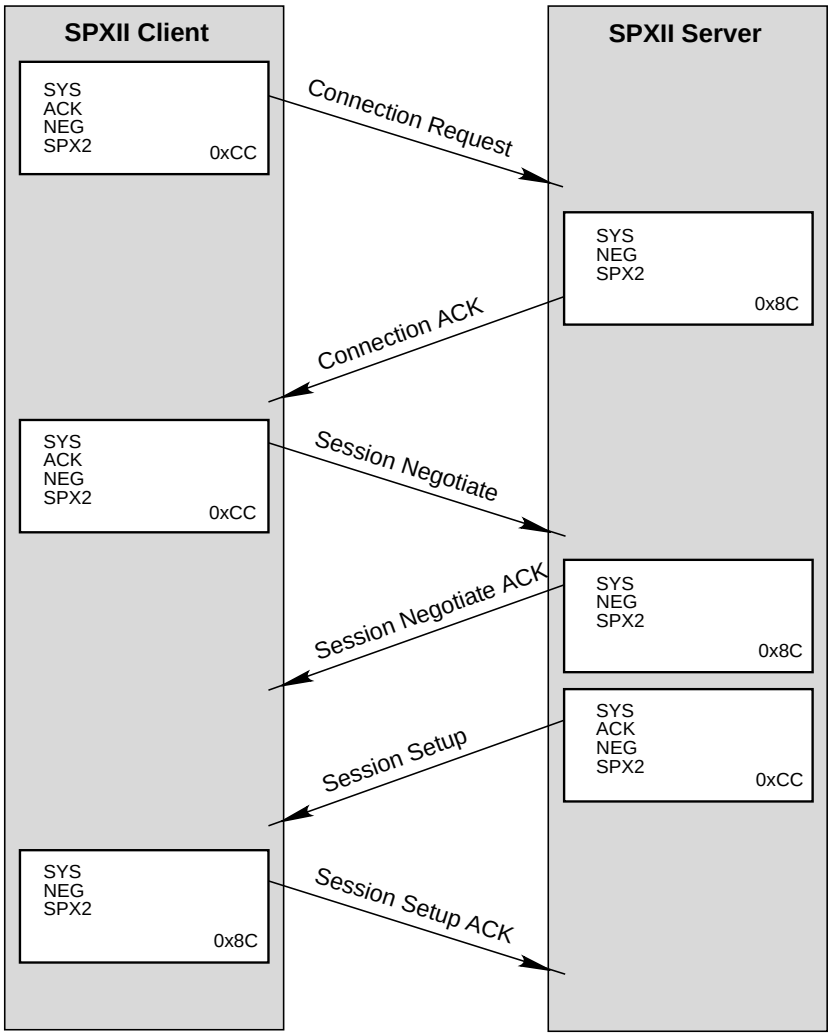
Negotiation Sequence. Figure 4-14 on page 78 illustrates the packet sequence between endpoints that want to negotiate packet size.

The client sends the first packet, a Connection Request packet with the SYS, ACK, SPX2, and NEG bits in the Connection Control field set. The server responds with a Connection ACK packet with the SYS, SPX2, and NEG bits set.

The client responds with a Session Negotiate packet with the SYS, ACK, NEG, and SPX2 bits set in the Connection Control field. The client sends the largest possible Session Negotiate packet.

Figure 4-14

SPXII Client to SPXII Server Connection Packets with Negotiation



The packet sequence in Figure 4-14 assumes that the server receives the first and largest Session Negotiate packet sent by the client.

If the client does not receive a Session Negotiate ACK from the server within an aggressive timeout period, the client sends a second Session Negotiate packet, reduced in size to the next logical driver size. The client continues to reduce the packet to the next logical driver size until an ACK is received or the minimum packet size is reached.

The client may optionally share information about other values (such as default timeout values and default retry counts) in the data field of the Session Negotiate packet. (See “Negotiating Other Values between Endpoints” on page 103.)

Once the server has sent the Session Negotiate ACK, the server sends the largest possible Session Setup packet.

The packet sequence in Figure 4-14 assumes that the client receives the first and largest Session Setup packet sent by the server.

If the server does not receive a Session Setup ACK from the client in an aggressive timeout period, the server sends a second Session Setup packet, reduced in size to the next logical driver size. The server continues to reduce the packet to the next logical driver size until an ACK is received or the minimum packet size is reached.

If the client does not receive a Session Setup packet within a specific time period, a session setup timeout occurs. This timeout will abort the connection establishment and notify the application of the failure. The time period begins with the reception of the Connection ACK. The driver uses the watchdog timer, which has a default of 60 seconds.

The server may optionally share information about other values (such as default timeout values and default retry counts) in the data field of the Session Setup packet. (See “Negotiating Other Values between Endpoints” on page 103.)

As soon as the server receives the Session Setup ACK, the session is established and is in data transfer state.

Without Negotiation. Either endpoint can select not to negotiate packet size in the following situations.

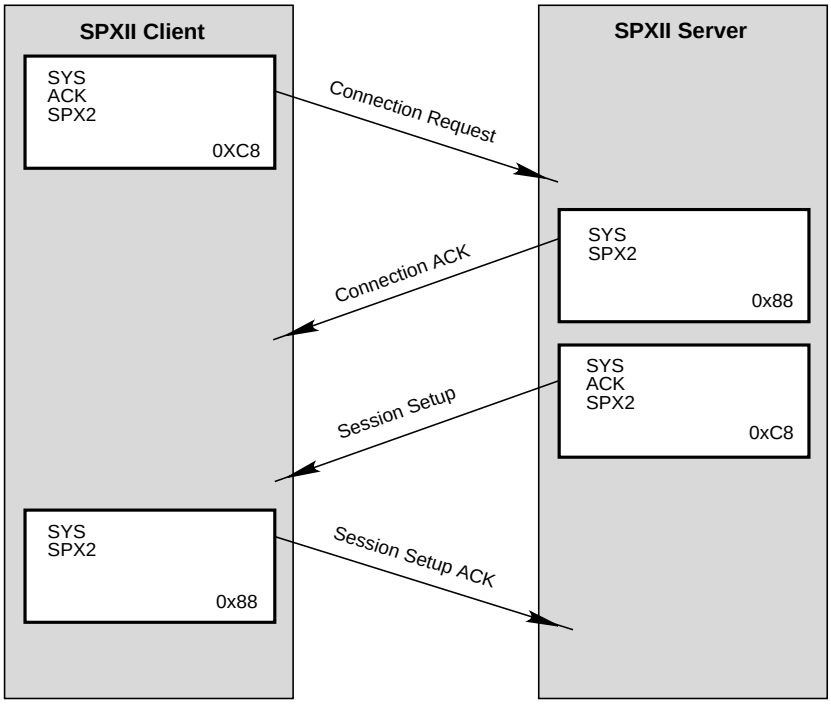
- If the client application wants to avoid the overhead of negotiation because the packets will be small (less than 576) or the connection will be short lived, the application requests that the NEG bit in the Connection Control field not be set in the Connection Request packet.

- If the server wants to avoid negotiation, the server does not set the NEG bit in the Connection ACK.

Figure 4-15 illustrates the packet sequence when the client does not request negotiation. The client sends the Connection Request packet without the NEG bit set in the Connection Control field. The server responds with the NEG bit not set in the Connection ACK.

Once the server has sent the Connection ACK, the server sends a Session Setup packet with the NEG bit not set. The client responds with a Session Setup ACK with the NEG bit not set.

Figure 4-15
SPXII Client to SPXII Server Connection Packets without Negotiation



If the client does not receive a Session Setup packet within a specific time period, a session setup timeout occurs. This timeout will abort the connection establishment and notify the application of the failure. The time period begins with the reception of the Connection ACK. The SPXII driver uses the watchdog timer, which has a default of 60 seconds.

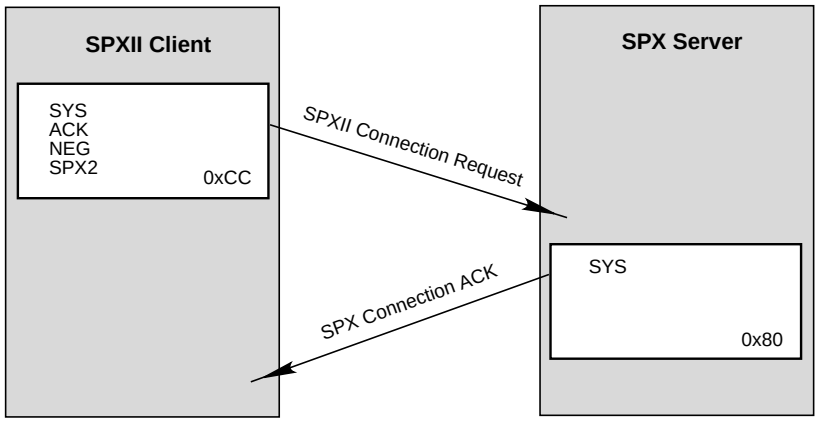
As soon as the server receives the Session Setup ACK, the session is established and is in data transfer state.

Packet Sequences for Mixed SPX and SPXII Connection Endpoints

SPXII is compatible with SPX because SPXII endpoints can connect to SPX endpoints and SPX endpoints can connect to SPXII endpoints.

SPXII Client to SPX Server. Figure 4-16 illustrates the packet sequence between an SPXII client and an SPX server.

Figure 4-16
Packet Sequence for an SPXII Client to an SPX Server



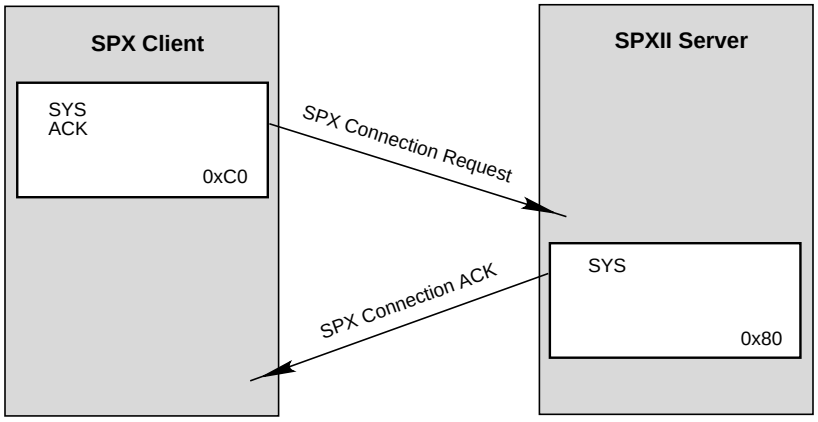
The client issues an SPXII Connection Request with the SPX2 bit set.

The SPX server ignores the SPX2 bit and returns a standard SPX Connection ACK.

When the client examines the ACK and notices that the SPX2 bit is not set, it reverts to standard SPX connection protocol.

SPX Client to an SPXII Server. Figure 4-17 illustrates the packet sequence between an SPX client and an SPXII server.

Figure 4-17
Packet Sequence for an SPX Client to an SPXII Server



The client issues an SPX Connection Request (which does not have the SPX2 bit set).

The server recognizes that the SPX2 bit is not set and returns a standard SPX Connection ACK and uses standard SPX connection protocol.

Session Termination Packets

SPXII supports the following types of session termination:

- Unilateral abort
- Informed disconnect
- Orderly release

Unilateral abort is used when one endpoint has detected a connection failure through retry failure or watchdog failure. No packets are exchanged; the detecting endpoint clears the connection and the resources it was using.

Informed disconnect differs from the unilateral abort by requiring the endpoints to exchange disconnect packets and is used when either of the following occurs:

- The application sends a send disconnect request (with a TLI **t_snddis** call).
- The SPXII driver detects a system failure on its side of the connection.

In SPX and in the AIX version of SPXII, an endpoint could potentially lose data under two conditions:

- When a disconnection indication arrives before all data is sent upstream to the application or before it is read by the application.
- When the application closes the local endpoint before all data is sent and acknowledged.

The new SPX Linger feature in the SPXII driver addresses both of these conditions. If a disconnect indication arrives from the remote endpoint, SPXII now attempts to send all data upstream to the application before taking action on the disconnection indication. If an application issues a close, SPXII delays the closing of the local endpoint until all data has been sent and acknowledged. In both cases, SPXII attempts to deliver all data, but when it is not possible to do so within the timeout period (default 120 seconds), SPXII completes the requested action.

Orderly release is used when the application calls the TLI **t_sndrel** function. It terminates the session only after both endpoints agree to the termination.

Informed Disconnect Packets

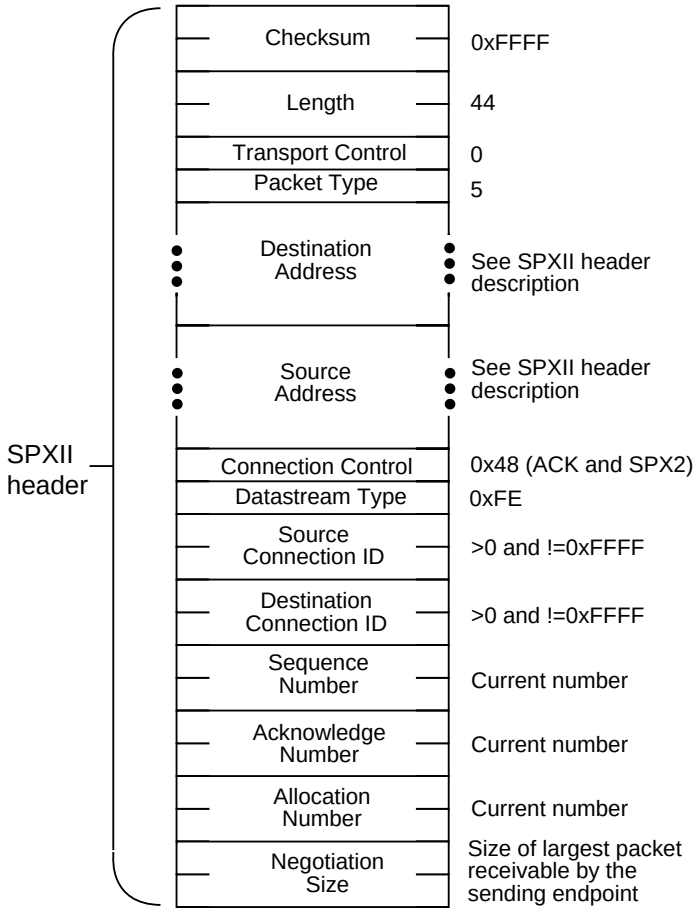
This section describes the following:

- Format of the Informed Disconnect packet
- Format of the Informed Disconnect ACK
- Packet sequence of an informed disconnect

Format of the Informed Disconnect Packet. The Informed Disconnect packet requires the exchange of disconnect packets between endpoints before the disconnect is performed. The informed disconnect can be generated by either endpoint.

Figure 4-18 shows the following fields and values of the packet:

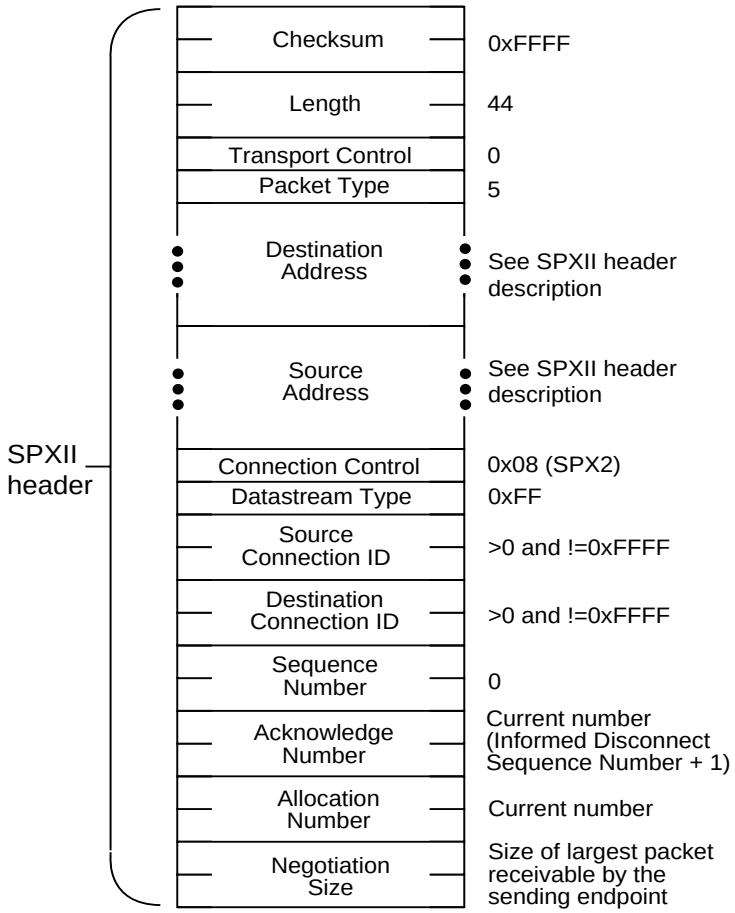
Figure 4-18
Fields and Values of the Informed Disconnect Packet



Format of the Informed Disconnect ACK. The Informed Disconnect ACK is sent in response to an Informed Disconnect Request. Once an Informed Disconnect Request is received, the endpoint can respond only with an Informed Disconnect ACK packet.

Figure 4-19 illustrates the following fields and values for this acknowledgment packet.

Figure 4-19
Fields and Values of the Informed Disconnect ACK

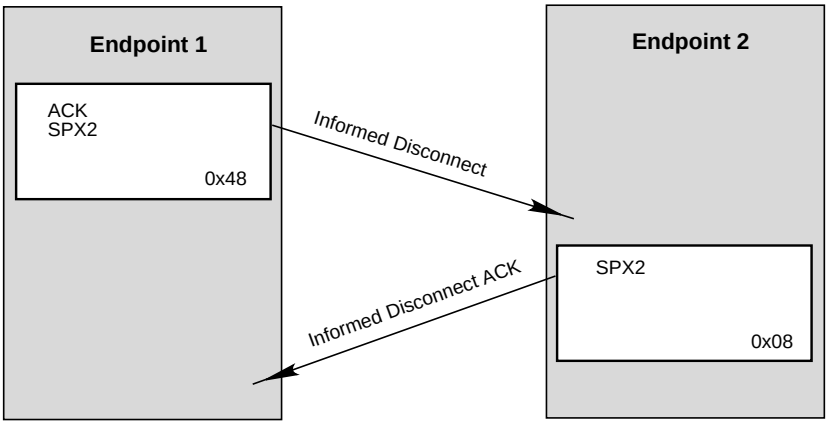


Packet Sequence for an Informed Disconnect. An informed disconnect terminates the session after the exchange of disconnect packets between the endpoints. The SPX Linger feature gives the connection partner an opportunity to complete any transmissions before the session is terminated.

Figure 4-20 below illustrates the packet sequence between the endpoints and assumes that Endpoint 2 receives the Informed Disconnect packet and responds with an ACK which Endpoint 1 receives.

If Endpoint 1 does not receive an ACK within the wait time for a data ACK, Endpoint 1 should retry sending the packet the same number of times it retries data packets. However, no route rediscovery or renegotiation are required. When all retries are exhausted, the endpoint performs a unilateral abort.

Figure 4-20
Informed Disconnect Sequence



Orderly Release Request Packets

The Orderly Release Request terminates the session after both endpoints agree to the termination. An orderly release permits the following:

- The endpoint receiving the release request has the opportunity to complete any data transmissions before the release is finalized.
- The endpoint sending the release request agrees to receive and process any data sent by the endpoint receiving the release request.

SPXII supports the orderly release facility through TLI. The t_info structure that is returned on the **t_open** and **t_getinfo** calls indicates that SPXII supports connection-mode service with the optional orderly release (T_COTS_ORD) in the type field.

TLI applications can take advantage of orderly release by using the **t_sndrel** (initiate orderly release) and **t_rcvrel** (acknowledge receipt of an orderly release indication) when terminating connections.

SPXII applications can connect to either SPX or SPXII endpoints. At connection time, an SPXII application can use the SPX2_OPTIONS structure to discover whether the connected endpoint is an SPX or SPXII endpoint:

- Servers should save this information from **t_listen**.
- Clients should save this information from **t_connect**.

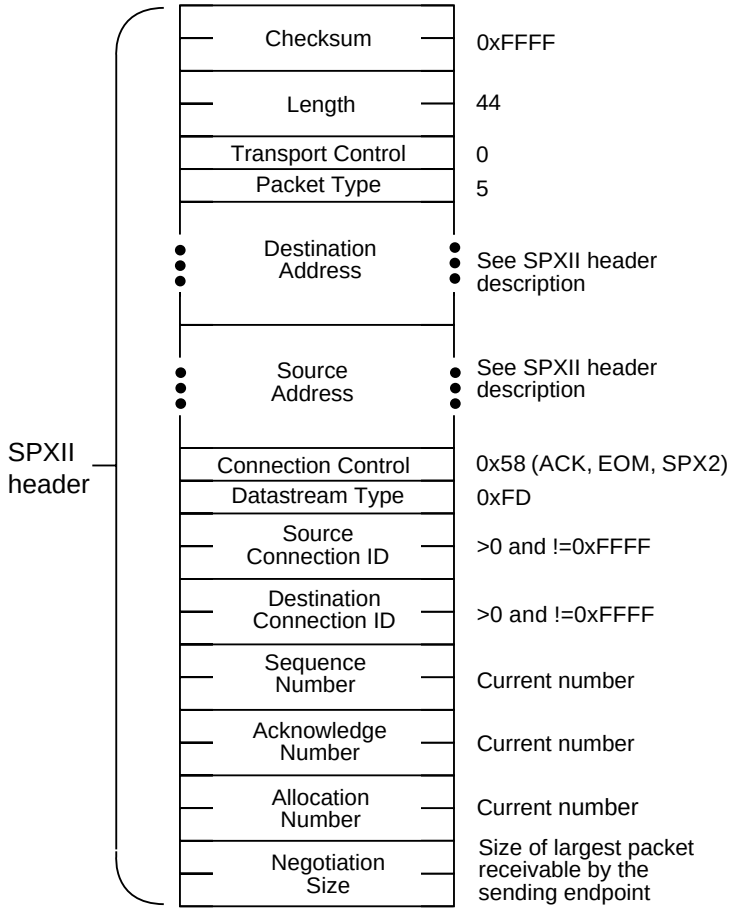
SPX does not support orderly release. If an SPXII application sends an Orderly Release Request to an SPX endpoint, the SPXII driver generates an “EPROTO” error, which causes all subsequent system calls to fail.

This section describes the following:

- Format of the Orderly Release Request packet
- Format of the Orderly Release ACK
- Packet sequence of an orderly release

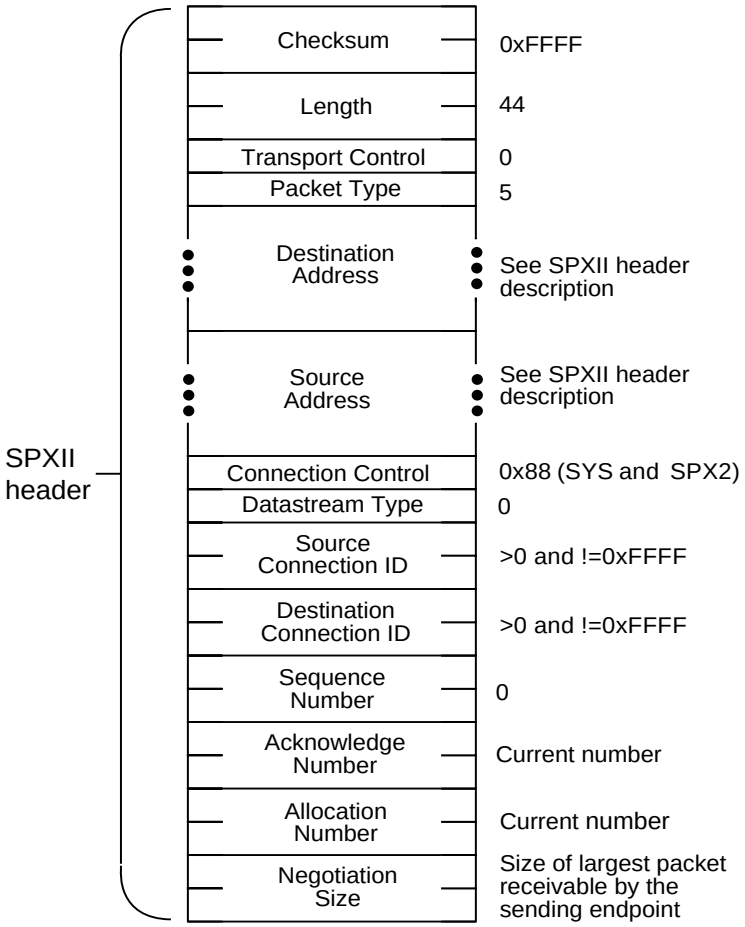
Format of the Orderly Release Request Packet. Figure 4-21 illustrates the packet's fields and values.

Figure 4-21
Fields and Values of the Orderly Release Request Packet



Orderly Release ACK. The Orderly Release ACK packet is the same as a normal data ACK packet. Figure 4-22 illustrates the packet's fields and values.

Figure 4-22
Fields and Values of the Orderly Release ACK



Packet Sequence for Orderly Release. An orderly release begins with one endpoint sending an Orderly Release Request packet.

Once the endpoint receiving the release request completes its data transmissions, it sends an Orderly Release Request back to the other endpoint.

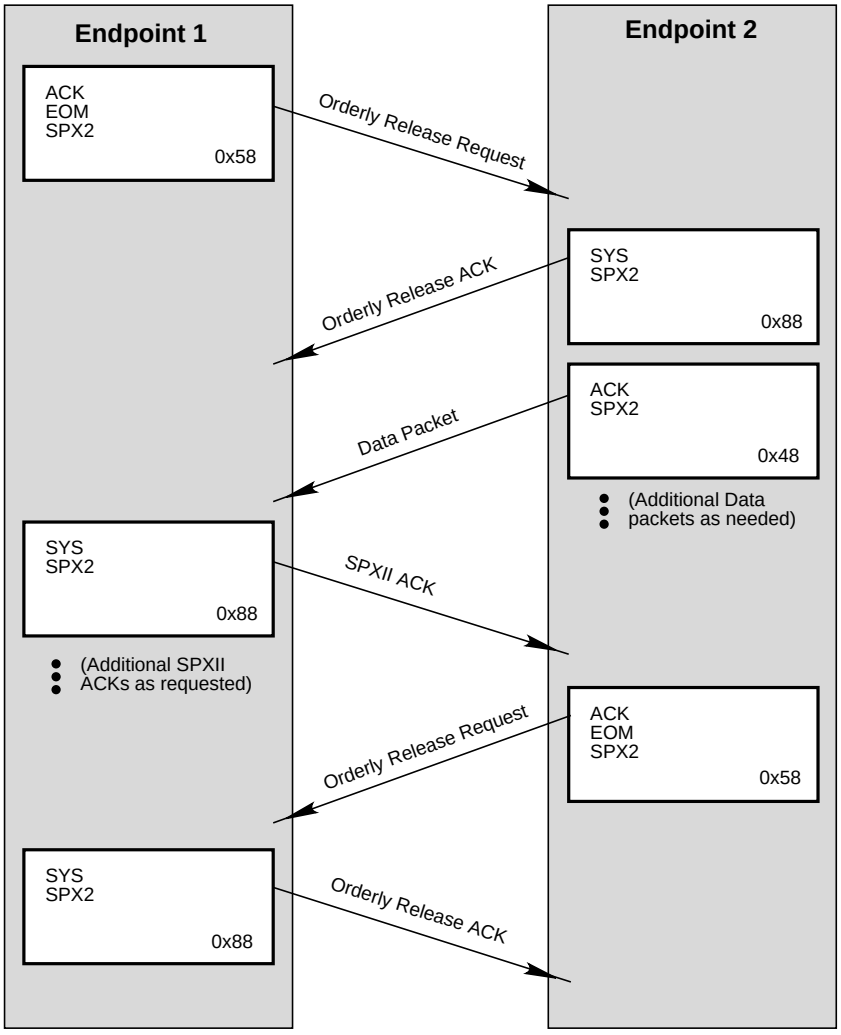
Following the second orderly release exchange, the session is terminated by both endpoints.

Figure 4-23 on page 91 illustrates the packet sequence between Endpoint 1 and Endpoint 2. This figure shows Endpoint 1 initiating the orderly release.

Endpoint 2 has more data to send before it wants to disconnect, so Endpoint 2 sends an Orderly Release ACK and a data packet, and then waits for the ACK.

Figure 4-23 shows only one data packet and ACK, but Endpoint 2 can send as many data packets as needed.

Figure 4-23
Orderly Release Sequence



As soon as Endpoint 2 receives the SPXII ACK for the data, it sends an Orderly Release Request.

When Endpoint 1 receives this packet, it knows that Endpoint 2 is ready to disconnect. Endpoint 1 sends an Orderly Release ACK and disconnects.

Endpoint 2 disconnects as soon as it receives the Orderly Release ACK from Endpoint 1.

If an Orderly Release ACK is not received, the Orderly Release Request should be retried with the same sequence as a normal data packet, with route rediscovery and renegotiation (if applicable). When all retries are exhausted, an endpoint can perform a unilateral abort.

An Orderly Release Request is not subject to window closure and must be sent even if the window is closed. An Orderly Release ACK is not subject to window closure.

Watchdog

SPX implemented the watchdog as an optional component on each connection. Under SPXII, connection watchdog management is required.

Also, SPXII has changed the watchdog algorithm to allow a more efficient exchange between endpoints. SPXII supports only the “Are you there?” request, rather than both the “Are you there?” and the elective “I am here” type packets that SPX sends.

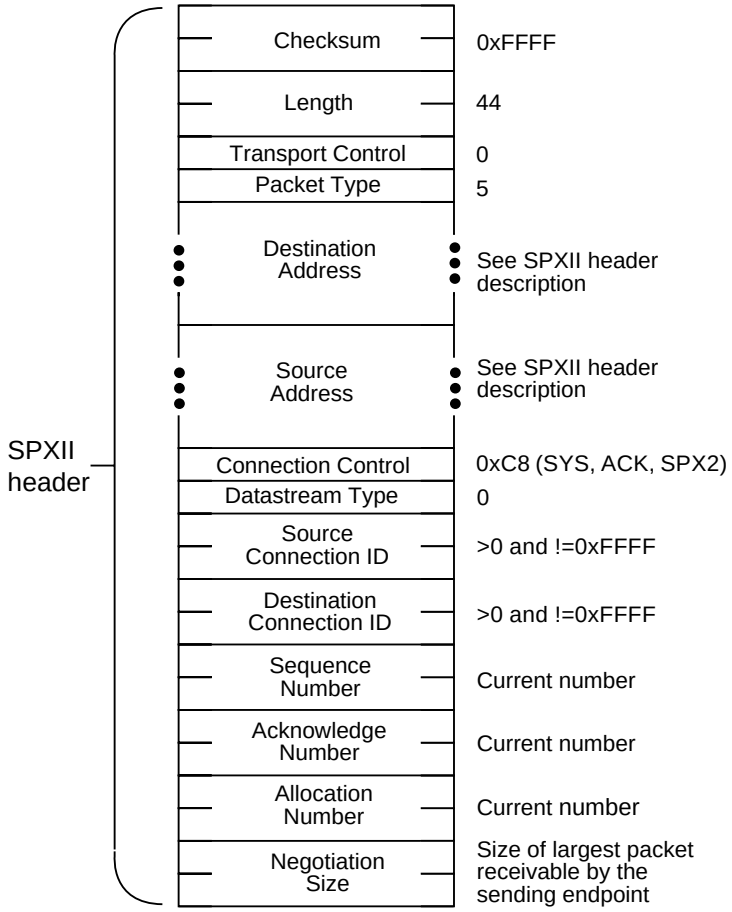
This section describes the following:

- Watchdog packet format
- Watchdog ACK
- Watchdog algorithm
- Watchdog during connection establishment

Watchdog Packet Format

A Watchdog packet consists of an SPXII header with the SPX2, SYS, and ACK bits set in the Connection Control field. Figure 4-24 illustrates the packet.

Figure 4-24
Fields and Values of the Watchdog Packet

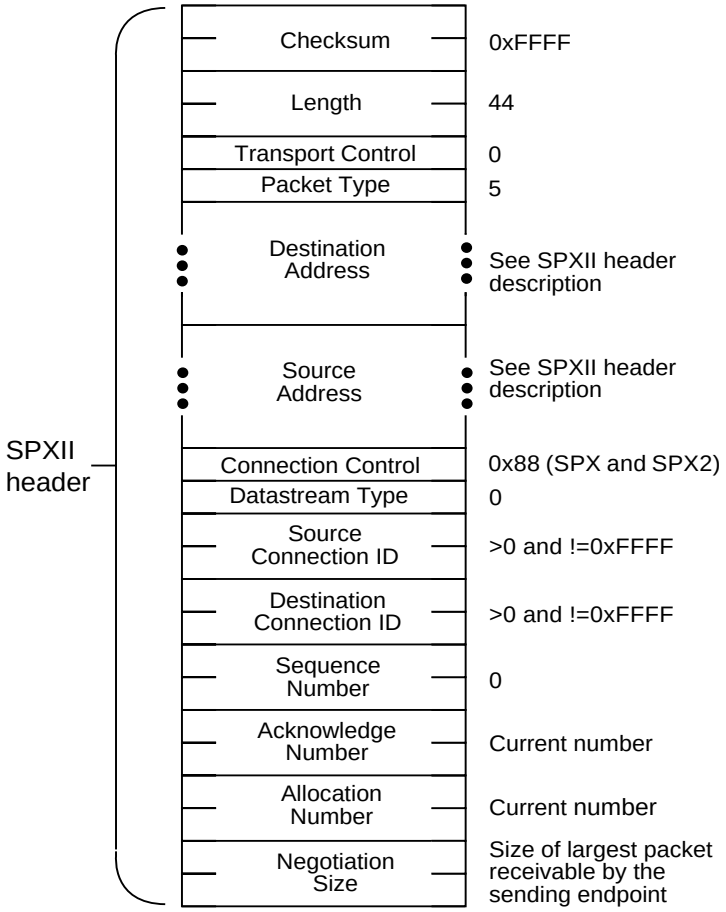


A data ACK can be "piggy-backed" on a watchdog packet by increasing the Acknowledge number to reflect all packets received (all packets up to but not including the acknowledge number have been received).

Watchdog ACK

The receiver of a Watchdog packet responds with a Watchdog ACK which consists of an SPXII header with the SPX2 and SYS bits set in the Connection Control field and the Sequence Number set to 0. Figure 4-25 illustrates the fields in this packet.

Figure 4-25
Fields and Values of the Watchdog ACK



SPXII Watchdog Algorithm

The watchdog routine is a passive element in SPXII, just as it is in SPX. Watchdog packets are sent only if there is an extended period with no traffic on the session.

The default watchdog timeout (the minimum time between the last packet traffic on the session and the first watchdog packet) is 60 seconds. Any packet that arrives for a session resets the watchdog timer for that session. This includes system packets as well as user data packets. (Thus an acknowledgment of transmitted data would reset the timer.)

Before the watchdog algorithm terminates a connection, it must perform the standard number of retries with the appropriate timeouts between attempts, just as SPXII would do for a data packet. Then, if the connection partner has not responded, the watchdog algorithm must attempt to locate a new route, just as it would for a normal data packet.

If the watchdog algorithm has completed all these steps and still has not received a response from the connection partner, the algorithm assumes that the partner is unreachable and unilaterally terminates the connection.

Session Watchdog during Connection Establishment

SPXII considers a connection to exist once an endpoint has both session IDs. This condition happens after either the receipt of a connect request at the server or the receipt of a connection ACK at the client.

If a connection exists, it should be monitored by the watchdog system. This is necessary because a connection can fail after the initial packet exchange, but before any other packet exchange (such as size negotiation or session acknowledgment) has taken place.

The watchdog system is also aware that the socket address of the destination endpoint may change after the Session Setup packet exchange. For more information, see “Session Setup Packet” on page 73.

Renegotiation

Renegotiate Request packets are used after connection establishment to renegotiate the packet size if the route has changed. Optional information can only be negotiated during connection establishment. For information on negotiating optional information, see “Negotiating Other Values between Endpoints” on page 103.

Renegotiate Request Packet

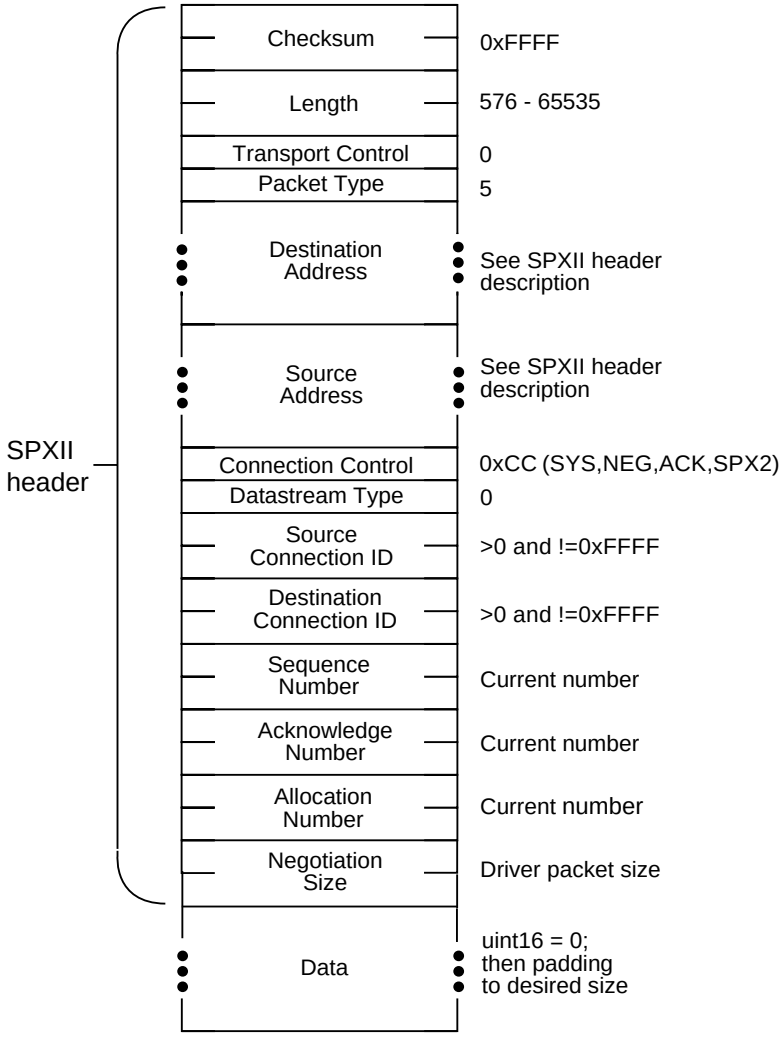
The Renegotiate Request packet can be sent only when both of the following conditions are met:

- The connection is established and is in a data transfer state.
- The NEG bit was set in the Connection Control field of the Connection Request and Connection ACK packets.

Either endpoint can send the packet any time the endpoint detects a route change. This usually occurs after retry failure and when the endpoint successfully locates a new route.

Figure 4-26 illustrates the fields and values of the Renegotiate Request packet.

Figure 4-26
Fields and Values of the Renegotiate Request Packet



The header is followed by *n* bytes of data so that the packet size equals the driver packet size or smaller as required by negotiation retries. The first `uint16` of the data must be zero (0) to indicate no optional information.

The padding value is not significant and can be uninitialized data. The Sequence Number field contains the next sequence number not already used by a data packet.

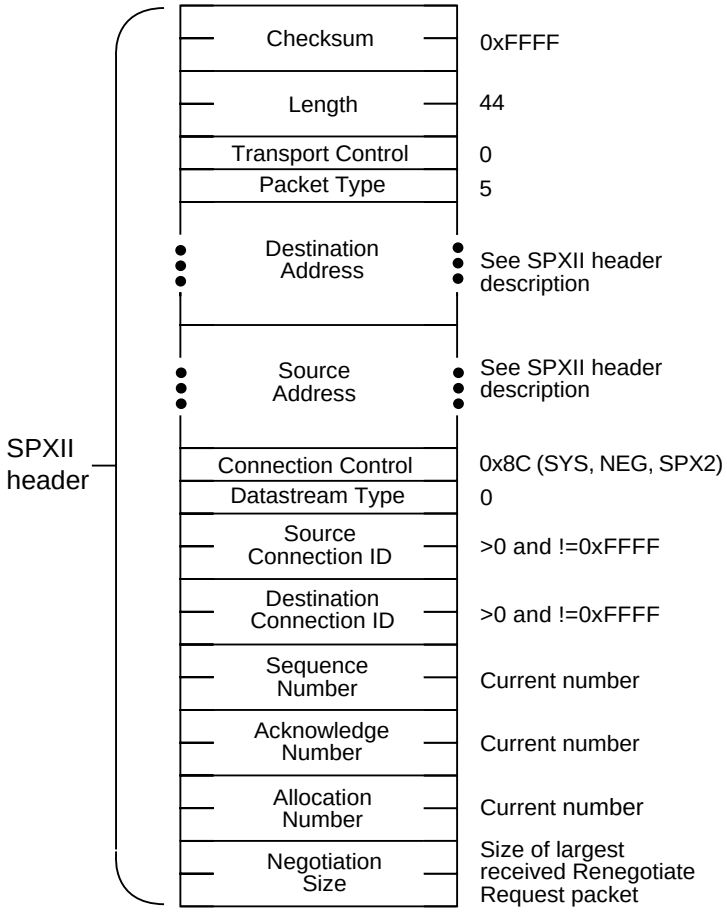
Renegotiate ACK

When an endpoint receives a Renegotiate Request packet, the endpoint determines the size of the packet by accessing the Length field in the IPX header. It then uses this to set the value in the Negotiation Size field of the ACK.

If an endpoint receives multiple Renegotiate Request packets, the endpoint always sets the Negotiation Size field to the value of the largest received Renegotiate Request packet.

Figure 4-27 illustrates the fields and values of the Renegotiate ACK.

Figure 4-27
Fields and Values of the Renegotiate ACK



The Acknowledge Number contains the appropriate number for the last data packet received. All data packets buffered by the receiver, because of out-of-order packet processing, must be discarded.

The Allocation Number contains the appropriate number relative to the Sequence Number of the Renegotiate Request.

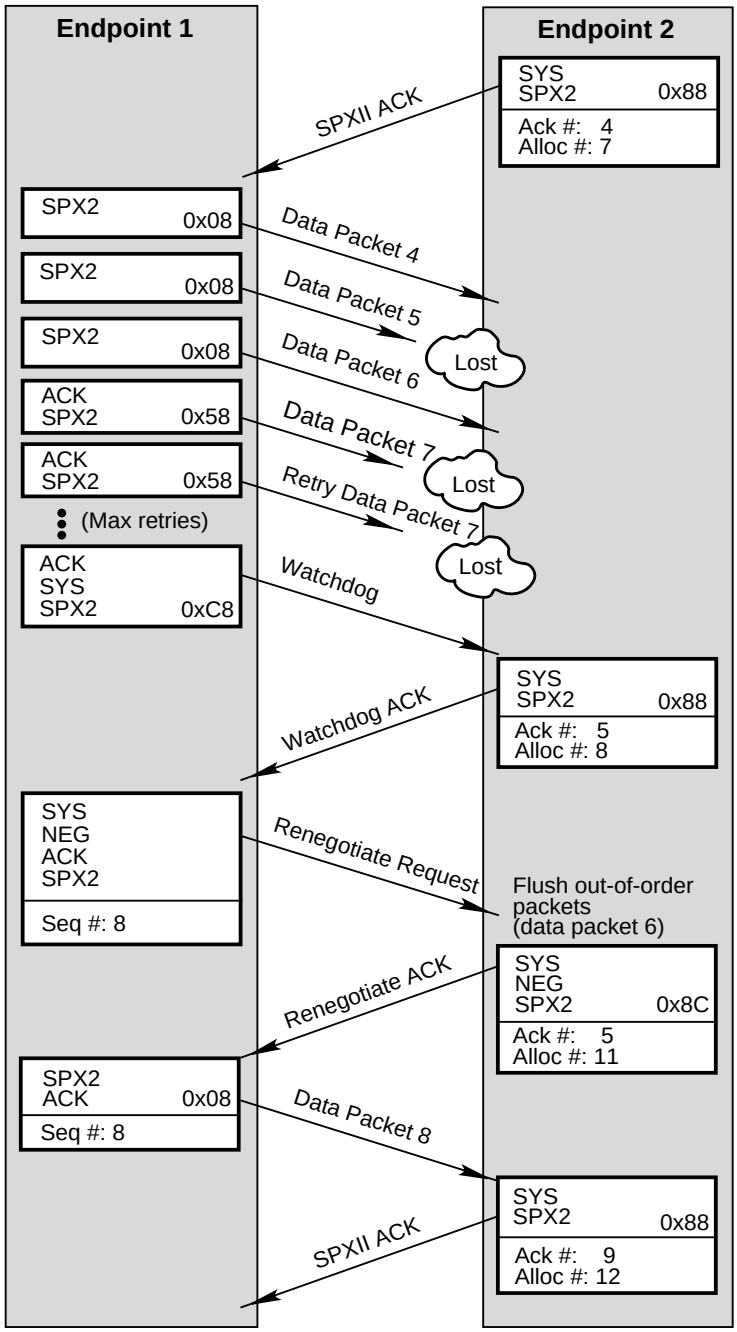
The Sequence Number of the first valid data packet after successful renegotiation is the same sequence number contained in that endpoint's Renegotiate Request packet.

Packet Sequence for Renegotiation

Either endpoint can initiate a renegotiation whenever the endpoint detects a route change. The endpoint usually detects the route change after a retry failure and initiates the renegotiation when route relocation has been successful.

Figure 4-28 on page 101 illustrates the flow of packets between Endpoint 1 (the initiater of the renegotiation) and Endpoint 2.

Figure 4-28
Renegotiation Sequence



The scenario in Figure 4-28 starts with Endpoint 2 acknowledging Data Packet 3 (with a window size of four).

Endpoint 1 then starts sending the next group of data packets. Two packets, data packets 5 and 7, never arrive at Endpoint 2. Packets are lost or destroyed when transmission errors corrupt the data, networks become overloaded, or changing routes cause the packets to be undeliverable because they are too large.

Endpoint 1 tries to resend Data Packet 7 the maximum number of retries. When Endpoint 2 does not respond, Endpoint 1 queries IPX for a valid route. When IPX returns with a valid route (either the old route or a new route), Endpoint 1 sends a Watchdog packet to Endpoint 2.

Endpoint 2 immediately responds with a Watchdog ACK. This Watchdog ACK also acknowledges that Endpoint 2 received Data Packet 4 (Acknowledge number minus 1 equals the last received data packet).

Endpoint 1 now assumes that a route change has occurred (because the Watchdog packet was delivered) and sends a Renegotiate Request packet.

When Endpoint 2 receives this packet, it flushes the out-of-sequence data packet (Data Packet 6) and sends a Renegotiate ACK with the acknowledge number set to the last in sequence data packet (Data Packet 4) plus 1. Endpoint 2 also adjusts the allocation number in the ACK so that the Allocation number minus the Sequence number of the Renegotiate Request packet equals a valid window size (in this example, four).

The normal window size formula of (Alloc#-ACK#) +1 is not valid on Renegotiate ACK packets.

When Endpoint 1 receives the Renegotiate ACK, it starts resending the data. It starts with the data from Data Packet 5, but the sequence number is now 8 because 8 was the sequence number of the Renegotiate Request packet. Because of potential packet size differences before and after the renegotiation, the packet boundaries of the retransmitted data may not be the same as when the data was sent the first time.

Endpoint 2 updates its Acknowledge number when it receives the first data packet from Endpoint 1. This update number is shown in the SPXII ACK for Data Packet 8.

If the Renegotiate Request packet is not acknowledged within an aggressive timeout period (which should be a function of the driver speed and the number of routers between the endpoints), the endpoint reduces the size of the packet to the next smaller logical driver size and resends the packet.

If multiple Renegotiate Request packets are received by an endpoint because of delayed packets or lost ACKs, the Negotiation Size field in the Renegotiate ACK contains the size of the largest Renegotiate Request packet received.

Only one renegotiation is permitted for a given sequence number. Otherwise it is impossible to differentiate between a delayed Renegotiate Request and a subsequent Renegotiate Request.

Negotiating Other Values between Endpoints

Since all sessions that negotiate packet size during connection establishment send negotiation packets from the client to the server and from the server to the client, the SPXII drivers can share other information in these packets. (For information on how an application communicates information to the SPXII driver, see “TLI Differences between SPX and SPXII” on page 160.)

Values (other than size) can be negotiated *only* at connection setup time (that is, at the start of the session). All subsequent negotiation packets must contain a zero (0) in the first `uint16` of the data portion of the packet.

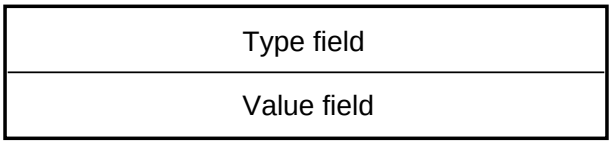
SPXII drivers can share the following types of information:

- Time to net
- IPX checksum control
- Default retry counts
- Route information

The first `uint16` (hi-lo format) in the data portion of the Negotiate Size packet or the Session Setup packet is a count of the number of Negotiate Value fields contained in the packet. Each Negotiate Value field is composed of two fields: a Type field and a Value field.

Figure 4-29 illustrates the simplest format—one byte for the Type field and one byte for the Value field.

Figure 4-29
Negotiate Value Format



The fields are described in detail below, with the Value field discussed first.

Value

The Value field contains the value of the option that is being negotiated. The Value is of variable length, ranging from one byte to 255 bytes.

The Size subfield of the Type field specifies its length.

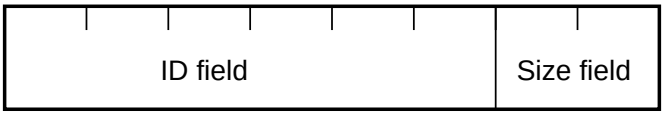
Type

The Type field specifies the option that is being negotiated. It consists of a single byte with two subfields: ID and Size.

- The ID subfield consists of the upper six bits of the Type field and specifies the identification number (ID number).
- The Size subfield consists of the lower two bits of the Type field and specifies the length of the Value field.

Figure 4-30 illustrates the format of the Type field.

Figure 4-30
Type Field Format



ID subfield. This subfield alone has no meaning. The ID number has meaning only in combination with the Size subfield. For example, there could be Type field with an ID of 1 and the size of `uint8` for a value of 0x04. This would not conflict with a Type field with an ID of 1 and size of `uint16`. Since this one would have a value of 0x05, these would not conflict.

Table 4-5 illustrates how ID numbers can be used with different size values to produce unique types.

Table 4-5
ID Numbers and Size Values

Range of ID Values (Decimal)	Size Subfield Values (Binary)
0-62	00
0-62	01
0-62	10
0-62	11

Both the ID field and the Size field are extensible. However, since most options will be communicated in only 2 bytes, combining the ID and size into one byte reduces the amount of space needed for the majority of the options.

Size subfield. This subfield can contain one of four possible values. Table 4-6 lists these values and their corresponding meanings.

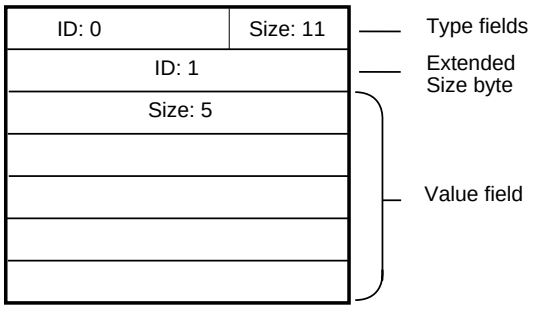
Table 4-6
Size Subfield Values

Binary	Description
00	uint8
01	hi-lo uint16
10	hi-lo uint32
11	extended size (length to follow)

When the Size subfield contains a binary 11 (meaning extended size), the `uint8` immediately preceding the Value field will contain the size of Value field in bytes. For example, if the Size field is set to 11 binary and the byte immediately preceding the Value field is 5, then the Value field is 5 bytes long.

Figure 4-31 illustrates this format.

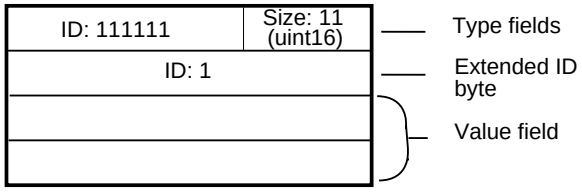
Figure 4-31
Extended Size Format



This ID subfield can contain one of 64 possible values. A value of 63 (111111 binary) indicates an Extended ID. When used, the extended ID immediately follows the Type field.

Figure 4-32 illustrates this format.

Figure 4-32
Extended ID Format



This extended ID field contains a number between zero and 255 and is unique for each size. For example, there could be a Value of size `uint8` with an extended ID of 1 and another Value of size `uint16` with an extended ID of 1, and these two would not conflict.

By using the upper six bits of the Type field, SPXII drivers can check for extended types without having to mask and shift. They can compare the type to binary 11 11 11 00; if it is equal to this value, the type is an extended ID.

Extended Value Combinations

There will be times when negotiation packets will contain an extended size and an extended ID. The maximum size of a Negotiate Value field would be the Type field followed by an Extended ID field followed by an Extended Size field followed by a Value field.

Figure 4-33 illustrates this format.

Figure 4-33
Extended Value Format

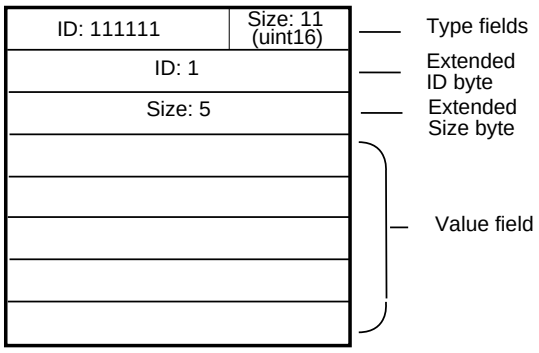


Table 4-7 lists the possible fields in a Negotiate Value and their maximum length in bytes.

Table 4-7
Negotiate Value Format

Field	Length (in bytes)	Description
Type	1	ID number/size combination
Extended ID	1	Optional
Extended size	1	Optional
Value	1 - 255	Number of bytes is determined by the Type and Extended size fields

Currently Defined Types

Table 4-8 lists the currently defined values for the Type field expressed in hexadecimal. (The ID and Size entries are given for clarity. These are derived from the eight bits that make up the Type field.)

Table 4-8
Type Numbers

Type	ID	Size	Description
0x02	0000 00	10 (uint32)	Time to Net. The time value returned by the Get Local Target or Get Route call expressed in microseconds. This value is communicated from the client to the server to eliminate the need for the server to call Get Local Target simply to get the time-to-net for initial timeouts. It is not necessary for the server to return this information to the client as it was provided unilaterally for the server's use.
0x04	0000 01	00 (uint8)	IPX Checksum. This is a boolean field. TRUE (non-zero) indicates that the endpoint wants IPX checksum verification.

Developers should reserve with Novell any Type Numbers that they want to add to ensure that their Type has a unique ID and size combination.

Disparate Versions of SPXII

Disparate versions of SPXII can communicate variables that the other is unaware of by using the following algorithm:

- If an endpoint receives a Type that it does not recognize, it can skip the Value field by looking at the Size subfield to find the number of bytes to skip. (An endpoint indicates that it does not recognize a particular Type by not returning data for that Type in the reply packet.)
- If an endpoint does not receive a Type / Value pair for a newly defined variable, the endpoint could do one of the following:
  - Use a default value.
  - Reject the connection if the information was critical. Although this destroys backward compatibility, it could be done in extreme circumstances.

SPXII Windowing Algorithm

One of the means by which SPXII increases transmission speed is its windowing protocol. SPXII implements the windowing protocol with the following features:

- Positive and negative acknowledgments
- Variable window size
- Improved error recovery

Positive and Negative Acknowledgments

Previous versions of SPX have had a positive acknowledgment only; negative acknowledgments were handled via a timeout on the sending side. SPXII introduces a negative acknowledgment packet to allow the receiver to inform the sender that one or more packets have been missed.

In SPXII the receiver is "passive" with respect to acknowledgments (ACK and NAK) and generates acknowledgments only when the ACK bit is set.

For more information on packet formats and packet sequences, see "SPXII Data Flow" on page 54.

Variable Window Size

SPX is usually used with a window size of one.

In SPXII, however, the window size is variable, and the receiver is responsible for calculating and maintaining the window size for its half of the session. (That is, the receiver at each endpoint maintains the window size for packets received by that endpoint.) The receiver then communicates this number to the other endpoint transmitter via the allocation number field in the SPXII header by adding the calculated window size to the current sequence number.

The transmitter is allowed to send packets while the sequence number is less than or equal to the allocation number.

The receiver is free to change the window size during the session with the stipulation that the receiver can never reduce an already granted window. The only way for the receiver to reduce the window size is to allow the acknowledge number to grow without increasing the allocation number.

For example, the receiver could not send a packet indicating an allocation number of 10 and then send a subsequent packet indicating an allocation number of 8. However, the receiver could send a subsequent packet indicating an allocation number of 10 even though the acknowledge number may have increased by 2. Such a packet would reduce the window size by 2.

Default Window Size

The SPXII driver in a UNIX environment has a default window size of 8. Applications can change the default during connection establishment by passing a **t_optmgmt** function.

Closing and Reopening a Window

Because of resource limitations or other reasons, the receiving side may need to close the data receive window completely. Only the data receive window may be closed. The driver always retains enough system resources to receive system and informed disconnect packets.

The receiver closes the data window by sending a packet indicating that the next sequence number expected is greater than the next allocation number. When this happens, the receiver is responsible for the following:

- Reopening the window when the flow condition has cleared
- Handling the possibility that the reopen packet may be lost

Once a receiver has closed the window, the receiver reopens a window by sending either a normal data packet or a system packet. The allocation number in the packets must be a number greater than or equal to the sequence number.

A packet which reopens a window may be lost. The SPXII driver handles such lost packets in one of the following ways:

- If the reopen packet is a data packet, the normal data packet retry mechanism will ensure delivery.
- If the reopen packet is a system packet, the packet is not retransmitted by the data packet retry mechanism.

  The watchdog will eventually send a series of system packets. These packets are sent at an increased frequency (about 3 seconds between queries). The watchdog packet will have the allocation number set to reopen the window. The watchdog system remains in this increased frequency state until a data packet arrives.

Error Recovery

Previous versions of SPX relied solely on a timeout mechanism for error recovery. Every SPX data packet had the ACK bit set. If the data packet was not acknowledged within a generous time period, SPX would retransmit the packet.

If, after several retries, the packet still had not been acknowledged, SPX would terminate the session. SPX would also terminate the session whenever it detected an error it didn't know how to handle.

SPXII, however, specifies how various error conditions will be handled.

Data Packet Timeout

With the SPXII windowing protocol, the only packets which have a timeout period associated with them are those packets that request an acknowledgment.

If a connection partner fails to acknowledge a packet, the sender retries the packet RETRY_COUNT/2 (default is 10/2 or 5) times, each time increasing the round trip time value by 50 percent, up to the MAX_RETRY_DELAY (default is 5 seconds).

If the packet still has not been acknowledged after the retries, the sender attempts to locate a new route to the connection partner.

If no route is available, the connection is terminated. If a new route is located or the old route exists, the sender immediately launches a watchdog packet requesting an acknowledgment.

- If the watchdog packet is acknowledged, the sender begins packet size negotiation, followed by retransmission of the unacknowledged data.
- If the watchdog packet or subsequent retries are not acknowledged, the connection is aborted with a Unilateral Abort.

A data packet timeout has the following sequence:

1. Retry the data packet RETRY_COUNT / 2 times, increasing the timeout value before each retry.
2. Attempt to locate a new route. (If no route is available, terminate the session.)
3. If a route is available, send a watchdog packet (and retries as needed).

SPXII Programming Interface

The SPXII driver can open either "/dev/nspx2" or "/dev/nspx". It is readable and writable by everyone.

This device node is accessed with either the UNIX TLI/XTI specification or ioctls.

For SPXII programming information, see the following:

- Chapter 7, "TLI/XTI for SPX/SPXII," on page 159.
- Chapter 10, "SPX/SPXII ioctls," on page 285.

chapter 5

Service Advertising Protocol (SAP)

What Is SAP?

SAP provides a way for service nodes, such as file servers, print servers, and gateway servers, to register their services and addresses in a Server Information Table and have these services advertised across an internetwork.

SAP is a distance vector protocol where the destinations are text strings. End nodes can listen for SAP or make queries to get a specific service or all services.

SAP provides dynamic registration of services. Periodic broadcast packets provide the service information and addresses for the Server Information Table. When a server goes down or fails to send a periodic broadcast packet, that service is removed from the table.

SAP services are accessed via a SAP agent (all NetWare servers and routers have a SAP agent). In a UNIX environment, the SAP daemon (SAPD) is the SAP agent. A SAP agent echoes the server information to all networks on the internetwork. Other SAP agents use the information to update and maintain their own Server Information Tables.

The discussion in this section covers the following topics:

- How SAP works
- SAP packet structure
- SAP information aging
- SAP information flow
- SAP packet types

- Periodic information broadcasts
- SAP programming information

How SAP Works

SAP is responsible for maintaining a list of servers and services on an internetwork which it then advertises. The easiest way to understand Service Advertising is to liken the Server Information Table to a telephone directory. NetWare clients must first have the number (address) of a service before they can initiate a session with that service.

When a NetWare server first comes up, it sends out a broadcast packet with its server information stored in the Server Information Structure. This packet informs the SAP agent that the server is now available. The SAP agent uses the information in the packet to add the server to its Server Information Table and then echoes the information to other SAP agents on the internetwork.

SAP also provides a way to terminate Service Advertising. When a server goes down, it sends a packet indicating the service is no longer available.

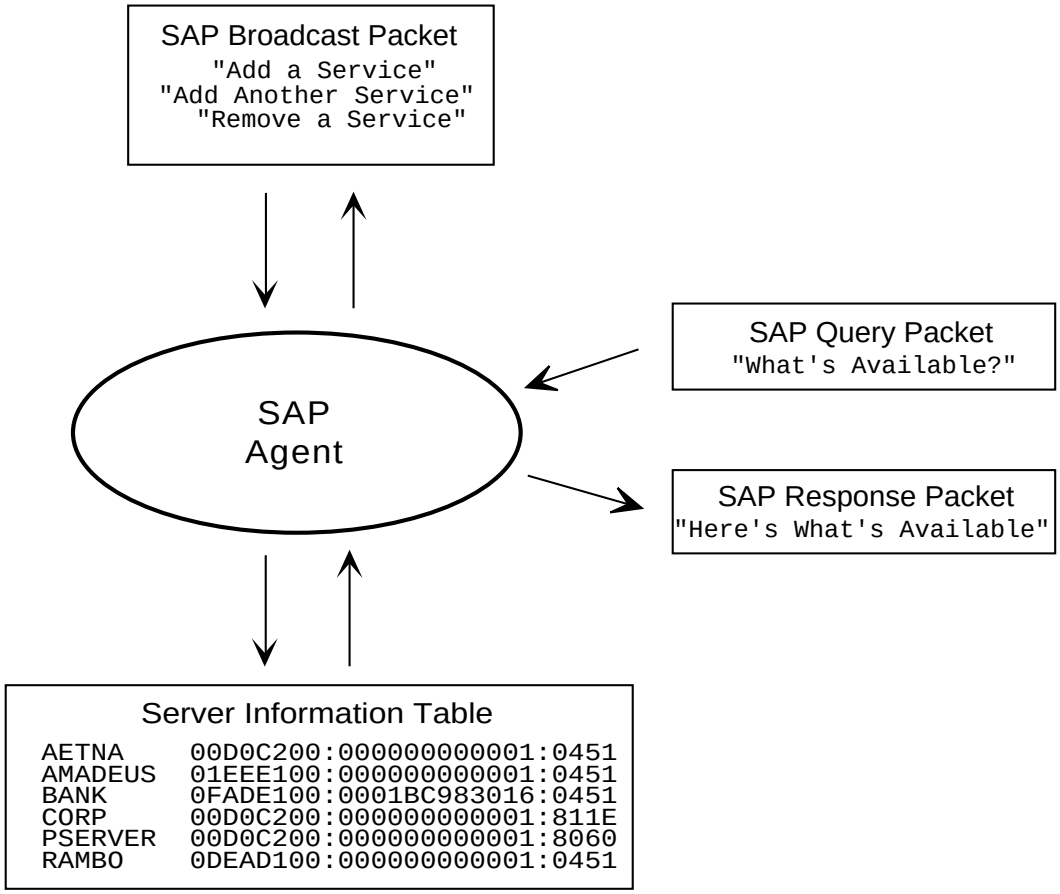
After the server's initial broadcast packet, the server is then responsible for sending out a periodic broadcast (every 60 seconds) to let the SAP agents know that its services are still available.

Any server or service that does not send a periodic broadcast packet within three minutes is presumed "downed." That server's information is then removed from the SAP agent's table, and the SAP agent sends out a broadcast indicating that the server is down.

In this way, SAP agents use the arrival or non-arrival of these broadcast packets to keep their Server Information Tables up-to-date.

Figure 5-1 on page 115 illustrates how a SAP agent uses broadcast packets to keep the Server Information Table up-to-date and how it provides query and response service.

Figure 5-1
Dynamic Registering of Services



When a SAP agent is initialized, it binds to IPX socket 0x0452 and sends a request (in the form of a SAP packet) for all SAP agents to send their server information. The SAP agent then builds a Server Information Table from the reply packets.

Every 60 seconds, SAP agents send local broadcasts of the server information for which they are the best or only source for a network. If a new server sends out a SAP packet saying that it is available, the SAP

agent verifies that this server is not in its Server Information Table. The SAP agent then adds this server to its Server Information Table and echoes this new server information to all connected networks.

Servers loaded on a UNIX machine can make the SAP daemon responsible for their advertising duties. The **SAPAdvertiseMyServer** function (documented on page 260) adds the information for such servers to the SAP daemon's tables. Their broadcasts are sent out with the SAP daemon's broadcasts until one of the following occurs:

- The server process notifies the SAP daemon that it is going down.
- The SAP daemon determines that the server process is no longer running because the process that requested advertising is no longer alive.
- The UNIX machine is rebooted.

For example, a NetWare server, an NVT2 (Novell Virtual Terminal) server, and an NDS (NetWare Directory Services) server all notify the SAP daemon when they come up. The SAP daemon then performs the advertising. The Print Server (shown in the Server Information Table in Figure 5-1) also uses the SAP daemon for advertising.

A server can also notify SAP that it is a Permanent Service server, in which case it is automatically re-advertised after the system boots.

Obtaining Service Names and Addresses

Querying a SAP agent is not the only way to obtain service addresses. It is up to you as an application developer to decide which is better for the application under development: to obtain network addresses from SAP or from the NetWare server's object database (explained below).

Querying a SAP Agent

Using a SAP agent is conceptually similar to using the yellow pages of the telephone directory. All services are grouped according to types of services. The SAP agent provides faster access to information about *types* of servers than does the object database.

Depending upon the type of query, the SAP agent returns one of the following:

- The names and addresses of all available services
- The names and addresses of servers of a specific type
- The name and address of the nearest server of a particular type

Querying the Bindery or Directory Services

Clients can also query a NetWare server's object database which contains definitions for entities such as file servers, print servers, or any other entity that has been given a name.

For NetWare 4.x servers, that database is the Directory. For NetWare 3.x servers, it is the Bindery. Directory Services maintains backward compatibility via bindery emulation mode.

Using the NetWare Bindery or Directory can be likened to using the white pages in the telephone directory. This provides faster access to information about a *particular* server than the SAP agent.

Depending upon the type of query, one of the following is returned:

- The name and address of a particular server
- The names and addresses of all servers of a particular type

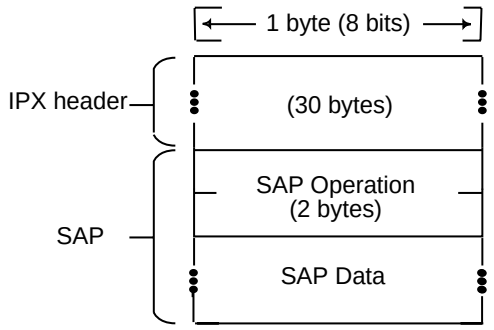
Actually, it is the interaction between SAP agents and the NetWare server's object database that keeps the latter up to date. When a NetWare server is initialized, it queries the memory copy of the Server Information Table to keep the object database updated.

SAP Packet Structure

SAP is implemented using the IPX datagram protocol. The SAP information for querying or advertising servers becomes the data portion of the IPX packet. The SAP header is defined in "include/sys/sap_app.h".

Figure 5-2 illustrates the layout of the SAP packet.

Figure 5-2
SAP Packet Structure



IPX Header

The IPX header is 30 bytes and contains fields for checksum, length, transport control, packet type, destination address (network, node, socket), and source address (network, node, socket). For more information about the IPX header, see “IPX Header Fields” on page 7.

SAP Operation

The SAP Operation field is two bytes long and determines the format of the rest of the packet. The SAP Operation field identifies four types of packets.

Operation Type	Constant	Hex Value
General Service Query	SAP_GSQ	0x0001
General Service Response	SAP_GSR	0x0002
Nearest Service Query	SAP_NSQ	0x0003
Nearest Server Response	SAP_NSR	0x0004

Note

Previous versions of SAP defined the periodic broadcast as a fifth packet type. Because the periodic broadcast packet uses the same format as a General Service Response packet, it is no longer described as a separate packet type. The difference was only that of context. However, the #define has been left in “include/sys/sap_app.h” to provide backward compatibility.

Server Information Structure

Each structure contains information for a particular advertising server.This SAPS (Server Information) structure is defined in the "sap_app.h" file in the "include/sys" directory. It has the following format:

```
#define SAP_MAX_SERVER_NAME_LENGTH 48

typedef struct saps_s {
   uint16    serverType;
   uint8     serverName[SAP_MAX_SERVER_NAME_LENGTH];
   ipxAddr_t serverAddress;
   uint16    serverHops;
} SAPS, *SAPSP;
```

The number of Server Information structures that are passed can be determined by subtracting the length of the IPX header and Operation field from the length of the packet and then dividing the remainder by the size of the Server Information structure. The number of structures depends on the maximum packet size used on a network. Seven structures fit in a 576-byte packet.

The fields of the Server Information structure (described below) store the information obtained from SAP response packets.

Server Type

This field is in hi-lo order and contains the object type of the advertising server.

NetWare 3.x servers maintain an object database called the Bindery. Although NetWare 4.x servers maintain a distributed object database called the Directory, they operate in bindery emulation mode by default to maintain backward compatibility.

Novell administers object types. If you are developing an advertising application server, contact Novell to be assigned a unique object type for your server.

Table 5-1 lists some of Novell's common object types.

Table 5-1
Common Server Object Types

Bindery Object	Type (hex)
File Server	0x0004
Job Server	0x0005
Gateway	0x0006
Print Server	0x0007
Archive Server	0x0009
Remote Bridge Server	0x0024
Target Service Agent	0x002E
Advertising Print Server	0x0047
NetWare Access Server	0x0098
Communications Server	0x0130
Named Pipes/SQL Server	0x0200
NVT2 Server	0x0247
Time Synchronization Server	0x026B
Directory Services Server	0x0278
Wildcard (only for General Service Queries)	0xFFFF (All types respond)

Server Name

This field contains the unique name (per server type within the internetwork) of a server that provides a service (file, printing, database management, archiving, and so on). The name of the advertising server must be at least 2 characters long and (because the name is a NULL-terminated string) cannot be more than 47 characters.

Server Address

This field contains the server's complete IPX address: Network, Node, and Socket. For more information, see "IPX Addressing" on page 2.

Hops to Server

This field contains the distance vector. In other words, the number of hops is the distance to the server, defined in terms of the number of intermediate networks (the number of routers that exist between the client and the server). For NWS, since the SAP daemon is the SAP agent, this is in the number of routers between the server and SAPD.

Initially the field is set to 1; each time the packet passes through an intermediate network, the field is incremented by one.

SAP Information Aging

For each entry in their Server Information Table, SAP agents maintain a timer field (Time Since Change) in an internal structure. This field implements aging for the Server Information Table. It tracks the number of periodic intervals that have elapsed since information was received regarding a particular entry in the table.

When information is received concerning a server, the field is reset. When the time interval in which no data is received exceeds three minutes, it times out. The SAP agent assumes that the server is down, removes the server from the table, and then broadcasts a SAP packet to indicate that the server is down.

In cases where a server goes down unexpectedly (for example, hardware failure, power glitch, power outage) without sending a "going down" broadcast, the server is removed from the Server Information Table via this process.

The layout of the network cabling affects how servers are aged and removed from the table:

- If the SAP agent is connected to a LAN that is cabled so that it loops back on itself *and* where there is more than one source or route to the server, the SAP agent maintains information about all routes.

Each route is aged separately. For example when the fastest or primary route goes down, that route is removed from the table and the next fastest route (one with fewest hops and ticks) is marked as the primary route.

A server is kept in the table as long as an active route exists.

- If the SAP agent is connected to a Wide Area Network (WAN), an asynchronous or X.25 link, the SAP agent can be configured to expect to receive only changed information from that link.

  Server routes that have been added to its tables from the WAN's SAP agent are *not* aged and remain in the table until that agent sends changed information indicating that either the route or server is down.

SAP Information Flow

On an internetwork, SAP agents perform the following two major functions:

- Keep other SAP agents up-to-date on the available servers
- Answer queries about available servers

This section describes the flow of this information. It first describes the flow of broadcast information as SAP agents keep each other informed. It then describes the flow of information in a query / response sequence.

SAP Broadcasts

SAPD broadcasts a request onto each of its directly connected LANs for information about other servers that exist on the internetwork.

Figure 5-3 illustrates the initial flow of information to SAPD as the NetWare protocol stack is initialized and the SAP daemon comes up.

Figure 5-3
Initial Flow of Information

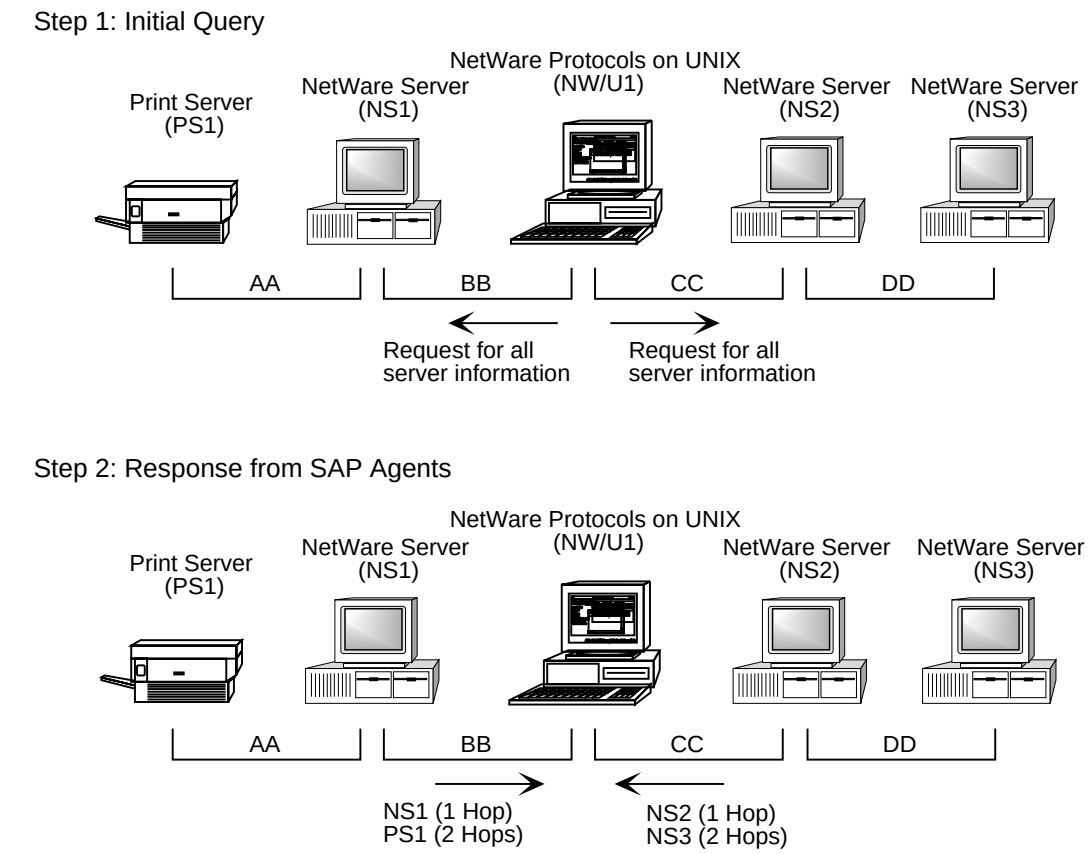
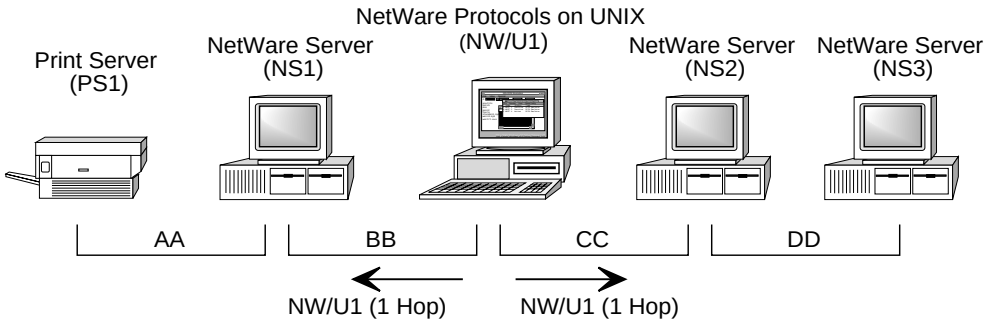


Figure 5-3 also illustrates the SAP daemon requesting the SAP agents on LANs BB and CC to send their server information. This is in the form of a General Service query packet for servers of all types. NS1 responds by sending information about the servers on its half of the network and NS2 sends information about its half of the network. NSI and NS2 send as many General Service response packets as are needed to send information about all known servers.

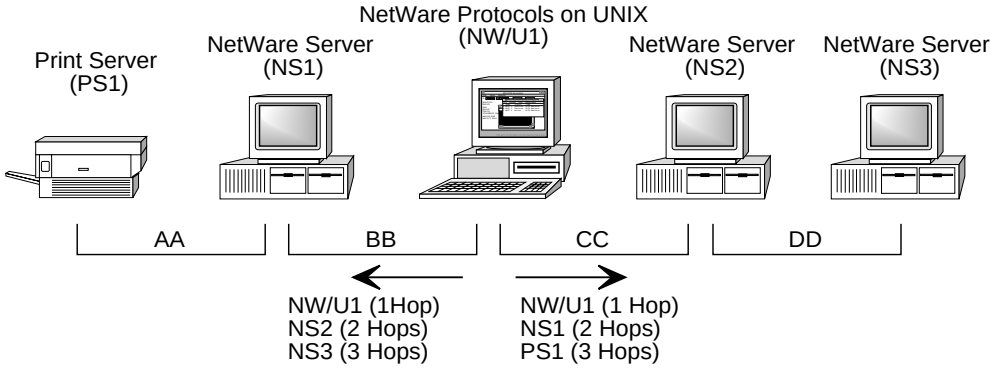
Figure 5-4 illustrates what happens when the server running the NetWare protocol stack is initialized and how information about local services flows from SAPD.

Figure 5-4
Flow of SAP Information from SAPD

Step 3: Server Initialization with NetWare Protocols on UNIX



Step 4: Periodic Broadcasts from SAPD



SAP agents use a split horizon algorithm to determine which SAP information to send to connected LANs. This algorithm ensures that the SAP agent does not broadcast information back to the same SAP agent from which the information was obtained.

For example, in Figure 5-4 on page 124, NW / U1 would never broadcast SAP information about NS2 or NS3 to NS2 since NW / U1 first received that information from NS2. NW / U1 will send both NS1 and NS2 information about itself because it didn't obtain that information from either.

Nearest Server Query

Clients using IPX must have the address of a server before they can establish a connection. One of the first packets a client sends out requests the Nearest Server.

To get the nearest NetWare server, the client would send out a SAP packet with the fields set as follows:

- Operation field set to 0x0003 (SAP_NSQ)
- Server type field set to 0x0004 (FILE_SERVER_TYPE)

When SAPD receives one of these packets, it searches its entire Server Information table for a server of the specified type with the fewest ticks (obtained from RIP) and the fewest hops. The only exception to a total search is when SAPD finds a local server of the specified type. In such a case, SAPD knows immediately that this is the nearest server and sends a response.

If SAPD does not have a local server of the specified type, it sends a response only if it determines that it is the best source for the nearest server selected.

For example in Figure 5-4 on page 124, NW/U1 would not respond to a Get Nearest Print Server query from a client on network BB. SAPD knows that it got the information from NS1 and that NS1 is the best source of the information. However, NW/U1 would answer such a query from a client on network CC.

The response packet would have its fields set as follows:

- Operation is set to 0x0004 (SAP_GSR)
- One Server Information Structure with settings that were obtained from the Server Information Table.

SAP Packet Types

As previously mentioned, the value in the SAP Operation field indicates whether the rest of the packet will have a query or a response format and then what the scope of the query or response is.

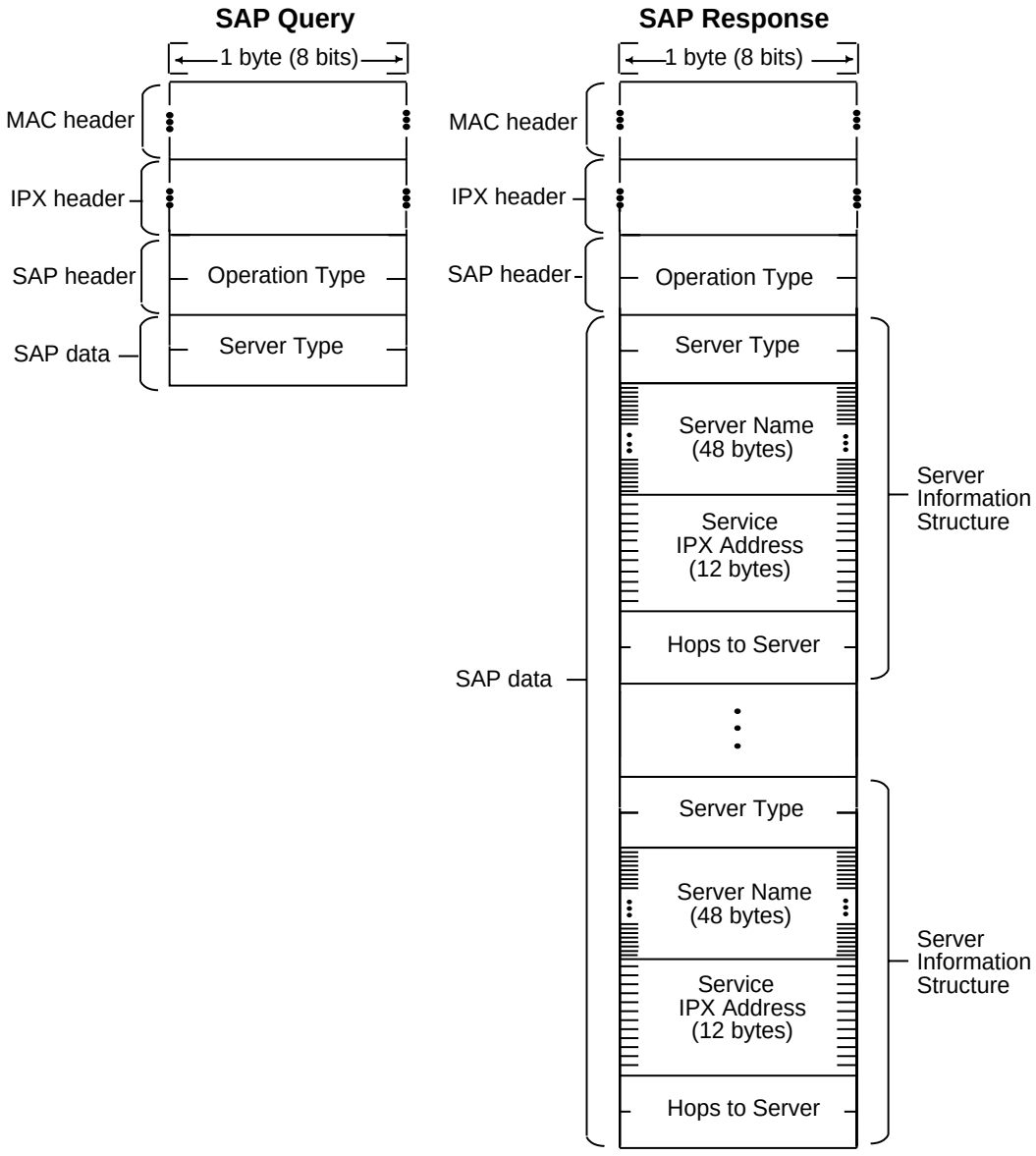
There are four types of SAP packets:

Operation Type	Constant	Hex Value
General Service Query	SAP_GSQ	0x0001
General Service Response	SAP_GSR	0x0002
Nearest Service Query	SAP_NSQ	0x0003
Nearest Server Response	SAP_NSR	0x0004

The length and content of the SAP Data depends on whether the packet is a SAP query or a SAP response. Query packets have 2 bytes of SAP data, whereas a response packet can have up to 448 bytes of SAP data, typically up to seven Server Information Structures for a 576-byte packet (the default IPX packet size). If a larger packet size has been negotiated, SAP can fill the packet with as many entries as it can hold.

Figure 5-5 illustrates both SAP packet structures, query and response.

Figure 5-5
Structure of SAP Packet Types



The SAP header fields and packet types are described in the sections following.

SAP Header

This Operation Type field indicates the SAP Type (query or response) and the scope (nearest or all). This field is also byte-order sensitive, and the data must be sent in hi-lo order, as shown in Figure 1-1 on page 4.

SAP Data

Both query and response packets have a Server Type field.

For the query packet, the server type is the object type of the server. See Table 5-1 on page 120 for possible values for this field.

However, although the Server Type field in the response packet is also the object type of the server, the field itself is the first field in a Server Information structure.(server type, server name, service IPX address, and the number of hops to the server). The Server Information structure is described on page 119.

A response packet can contain as many Server Information Structures as will fit in the packet.

SAP Query Packets

SAP query packets are used to discover the identities of servers on the internetwork.

Client applications use SAP queries to obtain the addresses of available servers via SAP queries. Using the IPX address from the response, the client application can then establish a session with a server.

Although application servers usually use SAP queries only on startup, if an application server needs to establish a session with another server, the application server would use a SAP query to obtain the address of the other server.

The fields for a SAP Query packet are listed in Table 5-2 and described below:

Table 5-2
SAP Query Packet

Field	Size	Byte order
Operation Type (SAP Query)	uint16	hi-lo
Server Type	uint16	hi-lo

SAP Query Operation

This field also indicates the scope of the query, whether it is general (for all servers) or specific (for the nearest server of a particular type). It can be set to one of the following values:

- 0x0001 for General Service Query packet (SAP_GSQ) is used to query local servers and routers. The query can be for all servers or all servers of a particular type or it can be directed to a specific node.

Note

In a response to broadcast requests (identified by the Destination Node Address in the IPX header as 0xFFFFFFFFFFFF), the split horizon algorithm described in the *IPX Router Specification* is followed.

However, in responding to any direct requests (the Destination Node Address in the IPX header is not broadcast nor multicast), the split horizon algorithm is not followed.

- 0x0003 for a Nearest Server Query packet (SAP_NSQ) is used to query for the nearest server of a particular type. The response to this query is always information about a single server.

Server Type

This field follows the SAP Query Type field and is used to differentiate the types of servers and limit the scope of the query. Server types are object types and are assigned by Novell. Some of the more common object types are listed in Table 5-1 on page 120.

Note

Server types are object types and are administered by Novell. Application developers who create servers that advertise services must apply to Novell to be assigned a unique server type.

SAP Response Packets

Because SAP response packets are sent in response to SAP queries, the types of responses match the types of queries. Each response packet has a field that indicates the type of the SAP response and one or more Server Information Structures.

Table 5-3 lists the elements of the SAP response format.

Table 5-3
SAP Response Format

Field	Size	Byte Order
Operation Type (SAP Response)	uint16	hi-lo
Server Information Structures	SAPS[n]	hi-lo

SAP Response Operation

This field indicates whether the response is to a general query (for any server regardless of type) or to a specific query (for a server of a particular type). It can be set to one of the following values:

- 0x0002 for a General Service Response (SAP_GSR) answers a General Service Query packet. SAPD responds to all General Service Query packets in behalf of all NetWare servers and all application servers that use SAP.

Previous systems defined a fifth packet type, a Periodic Broadcast packet. Because its format is the same as a General Service Response, it has been eliminated as a separate type. However, the #define has been left in the include file to provide backward compatibility.

- 0x0004 for a Nearest Server Response (SAP_NSR) is used to send an answer in response to a Nearest Server Query packet. This packet usually contains just one Server Information Structure. SAPD will find the nearest server of the requested type. If SAPD determines it is the best source for information about the server, it responds to the query.

Server Information Structure

This field contains an array of one or more (up to a maximum of seven) Server Information Structures. Each structure contains information for a particular advertising server.

The fields of the Server Information Structure are described beginning on page 119.

Periodic Broadcasts

Periodic Broadcast packets are used to advertise that a service is available and to remove services that are no longer available. When a server first comes up, it sends one of these packets out to let the SAP agents know that its services are now available. The SAP agents use this first broadcast to add the server to their Server Information Tables and to propagate the new server information to all their networks.

The server then sends out a periodic broadcast at regular intervals to let the SAP agents know that its services are still available. SAP agents use these broadcasts to keep their Server Information Tables up-to-date.

Any server or service that does not send a broadcast for three minutes is presumed "downed." That server's information is then removed from the SAP agent's table, and the SAP agent sends out a broadcast indicating that the server is down.

When a server is going down, the server sends a periodic broadcast with the Server Hops field set to 16. The "16" signals SAP agents to remove that server from their tables.

Periodic Broadcast packets have the same format as a General Service Response packet, but the Operation Type (SAP Response) field is set to 0x0002 (SAP_PIB). Otherwise, the difference between the two packets is functional: a General Service Response packet is a solicited response, whereas a Periodic Broadcast packet is unsolicited.

SAP Programming Interface

For SAP programming information, see Chapter 8, "SAP Library"on page 233.

chapter 6

TLI/XTI for IPX

Overview

This chapter describes the functions and structures used by NetWare's Internetwork Packet eXchange (IPX) under the UNIX Transport Layer Interface (TLI). IPX is NetWare's connectionless datagram protocol.

The following discussion assumes a working knowledge of both NetWare and TLI / XTI. It includes the following sections:

- IPX-specific information for TLI functions
- TLI structures
- Sequence of TLI functions
- IPX considerations
- TLI reference for IPX

For information on the IPX protocol, packet structure and fields, see Chapter 1, "Internetwork Packet Exchange (IPX) Protocol."

IPX-Specific Information for TLI Functions

In the UNIX environment, IPX is accessed via TLI. This chapter documents only the TLI functions that are transport-provider layer specific for IPX.

Developers should also have the "ipx_app.h" file.

Other TLI calls are handled transparently by the TLI library. For information about TLI calls not discussed in this chapter, see *AIX Version 4 Technical Reference Vol. 4: Communications (SC23-2617-01).*

TLI Data Structures

The following structures from the UNIX Transport Interface Library are transport-provider specific for IPX:

t_bind	t_optmgmt
t_info	t_unitdata

For a detailed description of these structures, see *Network Programming Interfaces.*

Each structure is included in the "Remarks" section of the documentation for the function that uses the structure.

Sequence of TLI Functions

To use IPX, a UNIX process calls the following functions in the listed order:

1. Open the IPX driver "/dev/ipx" using the **t_open** call.

   The **t_open** call returns a file descriptor (*fd*).

2. Bind *fd t*o a static or dynamic socket using the **t_bind** call.

   t_bind (*fd, &bind, &bind*)

3. If necessary, set options using **t_optmgmt** call.

   t_optmgmt (*fd, &req, &ret*)

   The **t_optmgmt** function is optional, but if it is used, it must be used after the **t_bind** function and before the **t_unbind** function.

4. To send or receive data, use **t_sndudata** (*fd, &ud*) or **t_rcvudata** (*fd, &ud, &flags*).

5. On exiting, unbind *fd* using a **t_unbind** call.

   The **t_unbind** call is optional.

6. On exiting, close *fd* using a **t_close** call.

In addition, it is often necessary to allocate memory for the various data structures used by these functions. The **t_alloc** function is used to allocate memory and the **t_free** function should be called to free the memory. The suggested use for **t_alloc** is included in code examples presented with the documentation for each function.

The IPX driver generates unitdata error messages. The **t_rcvuderr** call functions as specified by TLI. If a signal is sent to the IPX application during a TLI system call, the TLI system call fails with errno set to EINTR.

"TLI Reference for IPX" on page 135 describes the use of each of these functions with IPX. *Network Programming Interfaces* describes the standard use of these functions; refer to it for information not found in this document.

IPX Considerations

IPX follows the state diagram in *Network Programming Interfaces* for a connectionless service.

The device name for UNIX IPX is "/dev/ipx".

An internal network number is required where there are either multiple LANs or multiple frame types. In such a case, before starting the NetWare protocol stack, use the **nwcm** utility to configure the NetWare protocol stack by assigning an IPX internal network number.

```
nwcm -s ipx_internal_network = "n"
```

TLI Reference for IPX

This section describes how to use the following TLI functions with IPX.

t_bind	page 137	Binds a socket to a given transport endpoint.
t_open	page 143	Establishes a transport endpoint connected to a transport provider.
t_optmgmt	page 146	Manages transport protocol specific options.

t_rcvudata	page 149	Receives message sent using **t_sndudata**.
t_sndudata	page 154	Sends message to specified transport user.

For a description of the standard use of these functions or for functions not described in this document, see *Network Programming Interfaces*.

t_bind

Binds a socket to a transport endpoint.

Syntax

```
#include “ipx_app.h”

int t_bind(
   int ipxFd,
   struct t_bind *req,
   struct t_bind *ret )
```

Parameters

(IN) *ipxFd*

Passes the file descriptor that was returned by **t_open**.

(IN) *req*

Passes a pointer to (or the address of) a t_bind structure that in turn points to a structure that contains the requested address. The socket value in the structure is initialized to either a static socket number (one you have been assigned) or to zero (to obtain a dynamic socket number).

(IN) *ret*

Passes a pointer to (or the address of) a t_bind structure that in turn points to a structure that contains the IPX address. The pointer can point to and be the same structure that contains the requested address, *req*.

(OUT) *ret*

Receives the full IPX address: network address, node address, and socket number of the bound transport endpoint.

Return Values

0	Successful
-1	Unsuccessful

If **t_bind** returns an error, both *errno* and *t_errno* may be set. *t_errno* may be set to one of the following:

TBADADDR	The requested socket number is in use.
TNOADDR	There are no unused dynamic socket numbers. The IPX user should try again.
TOUTSTATE	This connection is in a state that invalidates a **t_bind** request.
TSYSERR	A system error occurred during the **t_bind** call. See the values for *errno*.

If *t_errno* is set to TSYSERR, *errno* may be set to the following:

ENOSR	No message buffers were available to acknowledge the bind request.

See *Network Programming Interfaces* for other possible errors.

Remarks

The **t_bind** function binds an endpoint to an IPX socket. This means that it associates a protocol address with a given transport endpoint.

The **t_bind** function is supported as documented in *Network Programming Interfaces.* with the following additions.

The t_bind structure has the following format:

```
struct t_bind {
   struct netbuf  addr;
   unsigned int   qlen;
};
```

IPX does not use the *qlen* field. It should be set to zero.

The netbuf structure has the following format:

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

The **t_bind** call requires that a pointer to an ipxAddr_t type structure be passed in the *req* t_bind structure (*req.addr.buf* field).

The ipxAddr_t structure has the following format:

```
typedef struct ipxAddress {
   unsigned char net[ 4 ];
   unsigned char node[ 6 ];
   unsigned char sock[ 2 ];
}ipxAddr_t;
```

The **t_bind** call allows an application to bind to a socket number, which can be either dynamic or static. IPX keeps track of which socket number is bound to which transport endpoint.

- A dynamic socket number is an unused socket number returned by the IPX driver and is guaranteed to be a unique unused number among the IPX endpoints. The range is a value from 0x4000 to 0x7FFF.

- A static socket number can be requested in the *req* t_bind structure. If the socket number is unused, it is granted and returned in the *ret* t_bind structure. The range is 0x8000 to 0xFFFF and numbers are assigned by Novell.

Static Socket Numbers

To obtain a static socket number, complete the following steps:

1. Allocate an ipxAddr_t structure.

2. Set the socket value in the ipxAddr_t structure.

   The example on the following page uses two #defines to specify the socket number and to ensure that the socket number is passed in hi-lo format.

3. Allocate a t_bind structure.

4. Initialize the t_bind structure's fields. The *req.addr.buf* field must point to the ipxAddr_t structure allocated in Step 1.

5. Make the **t_bind** call by passing the *ipxFd* value returned in your **t_open** call and by passing the address of the t_bind structure allocated in Step 3 as both the *req* and the *ret* values.

The IPX driver looks at the socket field in the ipxAddr_t structure for the IPX user's desired socket number. The socket number must be passed in hi- lo byte order.

If the socket number desired is not currently being used by another IPX user, the IPX driver returns the local net, local node, and the allocated or requested socket number in the corresponding fields of the ipxAddr_t structure of the *ret.addr.buf* field.

Only one IPX endpoint can bind to a given socket number at a time. If the user tries to bind to a socket that has already been bound to, an error results and the bind fails.

Services written to run over IPX generally have well-known or static socket numbers associated with them. (Contact Novell to obtain a static socket number.) By having static socket numbers, IPX users ensure that their server and client application types match.

Another method to coordinate servers and clients is to use the Service Advertising Protocol (SAP). For programming information, see Chapter 8, "SAP Library,"on page 233.

Dynamic Socket Number

A dynamic socket number is an unused socket number and is guaranteed to be a unique unused number among the IPX endpoints. A dynamic socket is a value from 0x4000 to 0x7FFF.

There are two methods for obtaining a dynamic socket number.

- If a dynamic socket is wanted, set the *sock* field in the ipxAddr_t structure to zero (0).

  The IPX driver attempts to allocate a dynamic socket number.

- If a dynamic socket is wanted and you do not need to know the value of the socket number, pass NULL as *req*.

  The IPX driver assumes then that the IPX user has requested a dynamic socket number, and it tries to allocate and to assign a dynamic socket number.

Regardless of which method you choose, if you do not need the address that has been bound, you can pass NULL as *ret*.

Example

```
#define SOCKET_TO_BIND_HIGH 0x45 /*high order byte */
#define SOCKET_TO_BIND_LOW 0x00 /*low order byte */

struct t_bind bind;
ipxAddr_t localAddress;

localAddress.sock[0] = SOCKET_TO_BIND_HIGH;
localAddress.sock[1] = SOCKET_TO_BIND_LOW;

bind.addr.len = sizeof(ipxAddr_t);
bind.addr.maxlen = sizeof(ipxAddr_t);
bind.addr.buf = (char *)&localAddress;
bind.qlen = 0;

if (t_bind(ipxFd, &bind, &bind)<0) {
t_error( “t_bind failed”);
..
..
}
```

State

The state follows the state diagram in *Network Programming Interfaces.*

See Also

t_open
t_unbind

t_open

Establishes a transport endpoint connected to a transport provider.

Syntax

```
#include “ipx_app.h”

int t_open(
   char          *ipxPath,
   int           oflag,
   struct t_info *ipxInfo )
```

Parameters

(IN) *ipxPath*

Passes a pointer to the path of the IPX driver. The path on most systems is “/dev/ipx”.

(IN) *oflag*

Passes the option flags for the opened stream.

(IN) *ipxInfo*

Passes the address of a t_info structure. See below for the format of the structure.

(OUT) *ipxInfo*

Receives the initialized t_info structure. See below for the format of the structure.

Return Values

>0	Successful
-1	Unsuccessful

If the **t_open** call is successful, it returns a file descriptor that identifies the local transport endpoint. Refer to *Network Programming Interfaces* for any errors that occur.

Remarks

The **t_open** function creates a local transport endpoint and returns protocol-specific information associated with that endpoint. It also returns a file descriptor that serves as the local identifier of the endpoint.

This function is supported as documented in *Network Programming Interfaces.*

The **t_open** function returns a TLI information structure upon the successful return of an open. The t_info structure contains the following information about IPX:

Table 6-1
IPX Information in the t_info Structure

Field	Value	Description
addr	12 (bytes)	This is the number of bytes required for an IPX address, which consists of three components: network address 4 bytes node address 6 bytes socket number 2 bytes
options	3 (bytes)	This is the maximum number of bytes in an options buffer.
tsdu	n	Size in bytes of a LAN's maximum transport service data unit. If connected to more than one LAN, the smallest value is returned.
etsdu	-2	Not supported.
connect	-2	Not supported.
discon	-2	Not supported.
servtype	T_CLTS	The service type for IPX is always T_CLTS (connectionless mode service).

The **t_optmgmt** function cannot be used to negotiate any of the values listed above.

The *tsdu* value can be obtained for a specific LAN by using the **t_getinfo** call after the endpoint has been bound using **t_bind**.

Example

```
int           ipxFd;
char  *ipxPath = "/dev/ipx";
struct t_info ipxInfo;

if ((ipxFd=t_open(ipxPath,O_RDWR,&ipxInfo))<0) {
t_error( "t_open failed");
..
..
}
```

State

The state follows the state diagram in *Network Programming Interfaces*.

See Also

t_bind
t_close

t_optmgmt

Manages protocol-specific options.

Syntax

```
#include “ipx_app.h”

int t_optmgmt(
   int     ipxFd,
   struct t_optmgmt *req,
   struct t_optmgmt *ret )
```

Parameters

(IN) *ipxFd*

Passes the file descriptor that was returned by **t_open**.

(IN) *req*

Passes a pointer to (or the address of) the structure that contains a pointer to an ipxAddr_t structure.

(IN) *ret*

Passes a pointer to (or the address of) the structure that will be initialized to the local IPX address.

(OUT) *ret*

Receives the local IPX address in the ipxAddr_t structure.

Return Values

0	Successful
-1	Unsuccessful

Refer to *Network Programming Interfaces* for errors that may occur with this call.

Remarks

The **t_optmgmt** function enables the user to obtain the local IPX address. (IPX does not support any negotiable options.)

This call is supported as documented in *Network Programming Interfaces* except that it returns the local address instead of negotiating options.

The t_optmgmt structure has the following format:

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

IPX does not use the *flags* field. It should be set to zero.

The netbuf structure has the following format:

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

The *len* and *maxlen* fields must be initialized to the size of an ipxAddr_t structure. The *buf* field returns all the local information about this transport endpoint: source network address, source node address, and source socket number.

The *req.buf* and *ret.buf* fields must point to a structure large enough to hold an ipxAddr_t (12 bytes). Upon successful completion, *ret.buf* contains the source information.

The first four bytes are the local network address; the next six bytes, the node address; and the last two bytes, the local socket number. All these address numbers are in hi-lo byte order.

The local socket number is valid only if this local endpoint has already been bound.

Example

```
struct t_optmgmt optionsRequest;
ipxAddr_t        localIpxAddress;

optionsRequest.opt.maxlen = sizeof(ipxAddr_t);
optionsRequest.opt.len = sizeof(ipxAddr_t);
optionsRequest.opt.buf = (char *)&localIpxAddress;

/* flags are not used with the IPX options request */

optionsRequest.flags = 0;

/* ipxFd is the file descriptor of an opened IPX device */

if( t_optmgmt( ipxFd, &optionsRequest, &optionsRequest )<0) {
t_error( “t_optmgmt failed”);
..
..
}
```

State

The state follows the state diagram in *Network Programming Interfaces.*

See Also

t_open

t_rcvudata

Receives a message sent using **t_sndudata**.

Syntax

```
#include “ipx_app.h”

int t_rcvudata(
   int     ipxFd,
   struct t_unitdata *ud,
   int     *flags )
```

Parameters

(IN) *ipxFd*

Passes the file descriptor that was returned by **t_open**.

(IN) *ud*

Passes a pointer to a t_unitdata structure. See “Remarks” for the format of this structure.

(IN) *flags*

Passes a pointer to an integer. The flag should be set to zero (0).

(OUT) *ud*

Receives the information in the t_unitdata structure. The *ud.udata.buf* field points to the data that was sent with the IPX packet.

(OUT) *flags*

Indicates if a complete data unit has been received.

Return Values

0	Successful
-1	Unsuccessful

If **t_rcvudata** fails, refer to **t_rcvuderr** for more information, which is documented in *Network Programming Interfaces*. See this guide for additional errors that may occur with the **t_rcvudata** call.

Remarks

The **t_ rcvudata** (receive unit data) function enables transport users to receive data units from other users.

This call is supported as documented in *Network Programming Interfaces*.

The t_unitdata structure has the following format:

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

The netbuf structure has the following format:

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

The address (*addr.buf* field) must point to an ipxAddr_t structure, which has the following format:

```
typedef struct ipxAddress{
   unsigned char net[ 4 ];
   unsigned char node[ 6 ];
   unsigned char sock[ 2 ];
} ipxAddr_t;
```

This call is the reverse of **t_sndudata**. The address of the sender is returned to the *ud.addr* field. The packet type is in the *ud.opt* field, and the packet data is in the *ud.udata* field.

The *len* field of *opt*, *udata*, and *addr* is set according to the incoming packet. The amount of data received in the IPX packet is in *ud.udata.len*.

If a complete data unit is received by the **t_rcvudata** call, *flags* will be set to 0. If more data remains to be received *flags* will be set to T_MORE.

Because IPX is a datagram service, there is no flow control on incoming data. If the IPX application cannot service the incoming data as fast as it is generated, the IPX driver drops the excess incoming packets.

Both static and dynamic allocation are illustrated in the examples below.

Example 1

```
/* This example uses statically allocated data buffers. */

struct t_unitdata ud;
unsigned char  ipxPacketType;
unsigned char  ipxDataBuf[IPX_MAX_DATA];
ipxAddr_t    sourceAddress;
int          flags = 0;

/* When the t_rcvudata unblocks, ipxPacketType will have the packet type
** from the IPX packet. */

ud.opt.len = sizeof(ipxPacketType);
ud.opt.maxlen = sizeof(ipxPacketType);
ud.opt.buf = (char *)&ipxPacketType;
```

```
/* When the t_rcvudata unblocks, sourceAddress will have the IPX address
of ** the datagram sender */

ud.addr.len = sizeof(ipxAddr_t);
ud.addr.maxlen = sizeof(ipxAddr_t);
ud.addr.buf = (char *)&sourceAddress;

/* When the t_rcvudata unblocks, ipxDataBuf will contain the data in the
** IPX packet */

ud.udata.len = IPX_MAX_DATA;
ud.udata.maxlen = IPX_MAX_DATA;
ud.udata.buf = (char *)&ipxDataBuf[0];
if (t_rcvudata (ipxFd, &ud, &flags)<0) {
t_error( “t_rcvudata failed” );
..
..
}
```

Example 2

```
/* This example uses dynamic allocation and checks for receipt of a
** complete data unit. */

struct t_unitdata  *ud;
int flags = 0;

if ((ud = (struct t_unitdata *) t_alloc (ipxFd,
   T_UNITDATA, T_ALL)) == NULL)
{
t_error ( “t_alloc failed”);
.
.
.
}
do
{
if (t_rcvudata ( ipxFd, ud, &flags) <0 ) {
   t_error ( “t_rcvudata failed”);
   ..
   ..
}
..
..
} while ( flags == T_MORE );
```

State

The state follows the state diagram in *Network Programming Interfaces.*

See Also

t_alloc
t_rcvuderr
t_sndudata

t_sndudata

Sends a message to specified transport user.

Syntax

```
#include “ipx_app.h”

int t_sndudata(
   int               ipxFd,
   struct t_unitdata *ud )
```

Parameters

(IN) *ipxFd*

Passes the file descriptor that was returned by **t_open**.

(IN) *ud*

Passes a pointer to (or address of) a t_unitdata structure. See “Remarks” for the structure’s format.

Return Values

0	Successful
-1	Unsuccessful

If the IPX driver cannot allocate memory, it sets *t_errno* to TSYSERR and *errno* to ENOSR. Errors can also cause *t_errno* to be set to one of the following values:

TOUTSTATE	The application hasn’t done a **t_bind**. A **t_bind** must be done before issuing a **t_sndudata**.
TBADADDR	The destination address is smaller than the size of an IPX address (ipxAddr_t structure).
TACCES	The destination network was not found in the routing tables.

TNOADDR	The allocation of streams memory failed. Check whether more kernel memory buffers are needed.

Errors can also cause the error field of the t_uderr structure to be set. Refer to **t_rcvuderr** for more information on how to receive these error messages. The **t_rcvuderr** function is documented in *Network Programming Interfaces.*

Remarks

The **t_sndudata** (send unit data) function enables transport users to send a self-contained data unit to the user at the specified protocol address. This call is supported as documented in *Network Programming Interfaces.*

The t_unitdata structure has the following format:

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

The netbuf structure has the following format:

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

The address (*addr.buf* field) must point to an ipxAddr_t structure, which has the following format:

```
typedef struct ipxAddress {
   unsigned char net[ 4 ];
   unsigned char node[ 6 ];
   unsigned char sock[ 2 ];
} ipxAddr_t;
```

The destination network address, node address, and socket number must be filled in this IPX address structure by the user program. The numbers must be in hi-lo order.

The IPX user sets the packet type of the outgoing IPX packet by passing 1 byte specifying the packet type in the options (*opt)* field. The IPX driver fills in the other fields in the outgoing IPX packet: checksum (see note below), length, transport control, source network address, source node address, and source socket number.

To enable checksums on a packet, the IPX user must send 3 bytes of options in the outgoing IPX packet. This includes 1 byte specifying the packet type and 2 bytes initialized to IPX_CHECKSUM_TRIGGER. The IPX driver will see that the checksum is requested, calculate it, and insert the number into the header.

IPX does not support EXPEDITED data. All data is sent on a first-come, first-served basis.

A successful return by the **t_sndudata** call doesn't guarantee that the data has been sent. A successful return guarantees only that the data has been queued up to be sent. The **t_sndudata** call returns before the data has actually been transmitted, so the application must take care not to close the connection before the data has been transmitted.

If the socket is closed before the data has been transmitted, the data is dropped. Before closing the connection, you should verify that the endpoint has received all the data and is ready to close the connection.

For UNIX applications that communicate with an unknown machine type, the byte order of data sent requires attention.

For example, if a UNIX application running on an 80386 CPU does a **t_sndudata**, the data portion of the IPX packet is sent across the wire in 80386 (lo-hi) order. This presents a problem if the receiving CPU (for example, a 68030) doesn't account for the lo-hi order.

If *ud.udata.len* equals zero (0), no packet is sent.

Example

```
#define MAX_DATA_SIZE 546

ipxAddr_t  remoteEndpointAddress;
unsigned char   ipxPacketType;
unsigned char   ipxData[IPX_MAX_DATA_SIZE];
struct t_unitdata ud;

/* There are three approaches to obtaining the address of the endpoint you
** wish to send to.

** 1. You can query a NetWare bindery for the server type you want.( This
** method assumes that you have established a connection to a NetWare
** server. Use the NWScanProperty function with NET_ADDRESS as the
** searchPropertyName.)

** 2. You can also create a file that maps a server name to an address.

** 3. You can use any appropriate method for discovering the endpoint's
** address.

** You must allocate an ipxAddr_t structure and initialize the fields to
** the endpoint's address before making the t_sndudata call. The following
** example code assumes that the endpoint has the following address: network
** address = 0x89415810, node address = 0x1, socket number = 0x500.*/

remoteEndpointAddress.net[0] = 0x89;
remoteEndpointAddress.net[1] = 0x41;
remoteEndpointAddress.net[2] = 0x58;
remoteEndpointAddress.net[3] = 0x10;
remoteEndpointAddress.node[0] = 0x00;
remoteEndpointAddress.node[1] = 0x00;
remoteEndpointAddress.node[2] = 0x00;
remoteEndpointAddress.node[3] = 0x00;
remoteEndpointAddress.node[4] = 0x00;
remoteEndpointAddress.node[5] = 0x01;
remoteEndpointAddress.sock[0] = 0x05;
remoteEndpointAddress.sock[1] = 0x00;

ipxPacketType = 0;

ud.opt.len = sizeof(ipxPacketType);
ud.opt.maxlen = sizeof(ipxPacketType);
ud.opt.buf = (char *)&ipxPacketType;

ud.addr.len = sizeof(ipxAddr_t);
ud.addr.maxlen = sizeof(ipxAddr_t);
```

```
ud.addr.buf = (char *)&remoteEndpointAddress;

ud.udata.maxlen = IPX_MAX_DATA_SIZE;

/* actual number of data bytes sent */

ud.udata.len = IPX_MAX_DATA_SIZE;
ud.udata.buf = (char *) ipxData;

if ( t_sndudata( ipxFd, &ud)<0) {
t_error( “t_sndudata failed \n”);
..
..
}
```

State

The state follows the state diagram in *Network Programming Interfaces.*

See Also

t_bind
t_rcvudata

chapter 7

TLI/XTI for SPX/SPXII

Overview

This chapter describes the functions and structures used by NetWare's Sequenced Packet Exchange (SPX / SPXII) protocol under the UNIX Transport Layer Interface (TLI). Application developers should have a working knowledge of both NetWare and TLI / XTI. The chapter includes the following sections:

- TLI differences between SPX and SPXII
- SPX / SPXII-specific TLI functions
- TLI data structures
- Sequence of TLI functions
- SPX / SPXII considerations
- TLI reference for SPX / SPXII

For information on the SPXII protocol, packet structure and fields, and data flow, windowing and packet size negotiation, see Chapter 4, "Enhanced Sequenced Packet Exchange (SPXII) Protocol" on page 41.

To compare the SPXII protocol with the earlier SPX protocol, see Chapter 3, "Sequenced Packet Exchange (SPX) Protocol" on page 27.

Application developers should use the enhanced SPXII protocol, which is backward compatible with SPX. The SPXII driver can open either /dev/nspx2 or /dev/nspx. (Code samples, with two exceptions, use /dev/nspx2.)

References to SPX/SPXII should be understood generically. The exception is when a distinction between SPX and SPXII is the subject of the discussion.

TLI Differences between SPX and SPXII

While TLI applications written for SPX will function on SPXII, their developers should be aware of the following:

- Orderly release changes
- Option management structure changes
- Compatibility procedures

Orderly Release Differences

SPXII supports the TLI orderly release functions; SPX does not. If SPXII applications use the orderly release functions, precautions must be taken because they can connect to both SPXII and SPX endpoints. Before issuing an orderly release packet, an SPXII application should verify that the connected endpoint is an SPXII endpoint. If the endpoint is an SPX endpoint, the SPXII driver generates an EPROTO error, which causes all subsequent system calls to fail.

At connection time, an SPXII application can use the SPX2_OPTIONS structure to discover whether the connected endpoint is an SPXII or an SPX endpoint. The application needs to save this information so that it can issue the proper sequence of function calls for a release.

In SPX and in the AIX version of SPXII, an endpoint could potentially lose data under two conditions:

- When a disconnection indication arrives before all data is sent upstream to the application or before it is read by the application.
- When the application closes the local endpoint before all data is sent and acknowledged.

The new SPX Linger feature in the SPXII driver addresses both of these conditions.

If a disconnect indication arrives from the remote endpoint, SPXII now attempts to send all data upstream to the application before taking action on the disconnection indication.

If an application issues a close, SPXII delays the closing of the local endpoint until all data has been sent and acknowledged.

In both cases, SPXII attempts to deliver all data, but when it is not possible to do so within the timeout period (default 120 seconds), SPXII completes the requested action.

Differences in the t_optmgmt Structure

TLI provides the following two mechanisms for exchanging option information between the protocol stack and the application:

- The **t_optmgmt** function
- The t_call structure

SPXII has increased the size of the t_optmgmt structure and added options for such features as windowing and packet size negotiation. On the other hand, SPX uses these TLI methods to negotiate only three options in the t_optmgmt structure.

Because the SPXII driver has been implemented so that it is backward compatible with SPX, it is capable of using either the SPX / TLI structure or the SPXII / TLI structure. For more information, see “Device Selection Procedures” on page 167.

The SPX and SPXII t_optmgmt structures are described below.

SPX t_optmgmt Structure

The following t_optmgmt structure applies to SPX:

```
typedef struct spx_optmgmt {
     uint8   spxo_retry_count;
     uint8   spxo_watchdog_flag;
     uint16  spxo_min_retry_delay;
} SPX_OPTMGMT;
```

Table 7-1
SPX Fields in the t_optmgmt Structure

Field	Description
spxo_retry_count	The value in this field specifies how many times SPX will resend an unacknowledged packet before concluding that the destination node is not functioning properly.
spxo_watchdog_flag	The value in this field determines whether the SPX watchdog is activated. Setting this field to zero disables the watchdog.
spxo_min_retry_delay	The value in this field specifies the initial timeout value before SPX resends a data packet. The SPXII driver does not support this option.

SPXII t_optmgmt Structure

While SPX has three options, SPXII uses an SPX2_OPTIONS structure which has 13 fields that an application may need to set or examine.

```
typedef struct spx2_options {
   uint32  versionNumber;
   uint32  spxIIOptionNegotiate
   uint32  spxIIRetryCount;
   uint32  spxIIMinimumRetryDelay;
   uint32  spxIIMaximumRetryDelta;
   uint32  spxIIWatchdogTimeout;
   uint32  spxIIConnectionTimeout;
   uint32  spxIILocalWindowSize;
   uint32  spxIIRemoteWindowSize;
   uint32  spxIIConnectionID;
   uint32  spxIIInboundPacketSize;
   uint32  spxIIOutboundPacketSize;
   uint32  spxIISessionFlags;
} SPX2_OPTIONS;
```

The fields in this structure are defined in Table 7-2 on the following page. The comments in the structure definition use the following symbols and indicate whether

- The attribute is read (r), write (w), or read / write (r / w)
- The field is applicable to the **t_optmgmt** function (o), the t_call structure (c), or both (o / c)

Whether an application uses the **t_optmgmt** function or the t_call structure to set values depends on the application's needs and the state of the connection.

- An application can use the **t_optmgmt** function only during the T_IDLE state (clients, between **t_bind** and **t_connect**; servers, between **t_bind** and **t_listen**).

  During this time, the application is able to negotiate the **t_optmgmt** values, and the SPXII driver returns information about the adjustments.

- When an application uses the t_call structure during **t_connect** or **t_listen**, the protocol stack uses any valid **t_optmgmt** values, but is unable to inform the application if the requested values were ignored because they were invalid.

  Applications use t_call when they know their requested values are valid and they want to avoid the overhead of negotiation.

Each field is described below.

Table 7-2
SPXII Fields in the t_optmgmt Structure

Field	Comment	Description
versionNumber	(r/w, o/c) Must be set to OPTIONS_VERSION	This field is a monotonically increasing number which can be used by the protocol to determine the fields supported by the application. Each time the structure is enhanced this version number will be increased. It is recommended that for transparency, an SPXII TLI-based application use **t_alloc** and **t_getinfo** to allocate and determine the size of the options structure rather than the C operator sizeof(SPX2_OPTIONS).

Table 7-2 *continued*
SPXII Fields in the t_optmgmt Structure

Field	Comment	Description
spxIIOptionNegotiate	(r/w, o/c) Exchange options and negotiate packet size	This field specifies whether the application wants to exchange option information with the remote endpoint and negotiate size. This field has the following values: SPX_NEGOTIATE_OPTIONS (default) SPX_NO_NEGOTIATE_OPTIONS
spxIIRetryCount	(r/w, o/c) Number of retries on data packets	When a transmission failure for a data packet is detected, SPXII will resend the data. This field specifies the number of attempts SPXII will make before unilaterally aborting the connection. A value of zero indicates to SPXII to use the current default value. (The default is 10.)
spxIIMinimumRetryDelay	(r/w, o/c) Minimum retry timeout, in milliseconds	Setting this field to nonzero indicates that the application wants to override the internal round trip time calculation algorithm and wants to specify a minimum timeout value before SPXII resends a data packet. A value of zero indicates to SPXII to use the current round trip time as the retry delay.
spxIIMaximumRetryDelta	(r/w, o/c) Maximum retry delta, in milliseconds	The value of this field is added to *spxIIMinimumRetryDelay* or to the current round trip time (if *spxIIMinimumRetryDelay* is zero) to determine the maximum retry delay. A value of zero indicates to SPXII to use the current default. value. (The default is 5 seconds.)
spxIIWatchdogTimeout	(r/w, o/c) Number of milliseconds to wait before 1st watchdog	This value determines the amount of time the watchdog algorithm will allow to pass on a client connection before sending a watchdog query packet to determine if the other side is still available. This option is not implemented in the SPXII driver; the driver uses a default value of 60 seconds.
spxIIConnectionTimeout	(r/w, o/c) Number of milliseconds to wait for full connection setup	This value determines the amount of time after a successful connect request before the session setup packet must arrive. This option is not implemented in the SPXII driver; the driver uses a default value of 60 seconds.

Table 7-2 *continued*
SPXII Fields in the t_optmgmt Structure

Field	Comment	Description
spxIILocalWindowSize	(r/w, o/c) Number of packets in data window	This specifies the size, in packets, of the local endpoint receive window. A value of zero indicates that SPXII will determine the receive window size. (The default for the SPXII driver is 8.)
spxIIRemoteWindowSize	(r, c) Number of packets in data window	This is an information-only field and is valid only after a connection has been established. It is the size, in packets, of the remote endpoint's receive window.
spxIIConnectionID	(r, c) Valid only after connection is established	This is an information-only field and is valid only after a connection has been established. It is the local endpoint connection ID.
spxIIInboundPacketSize	(r, c) Size of packets coming from other endpoint	This is an information-only field and is valid only after a connection has been established. It is the size in bytes for incoming packets. This value may change if SPXII has to renegotiate after a route change. There is no way to inform an application when packet size changes because of renegotiation.
spxIIOutboundPacketSize	(r, c) Size of packets being sent to other endpoint	This is an information-only field and is valid only after a connection has been established. It is the size in bytes for outgoing packets. This value may change if SPXII has to renegotiate after a route change. There is no way to inform an application when packet size changes because of renegotiation.
spxIISessionFlags	Session characteristic options (see description for r/w)	This is a bit field and contains a set of flags used to control characteristics of the SPXII packets on the wire. These characteristics might include packet checksums, data encryption, or data signing. The following flags have been defined: SPX_SF_NONE 0x00 SPX_SF_IPX_CHECKSUM 0x01 //(r/w, o/c) SPX_SF_SPX2_SESSION 0x02 //(r, c) After a connection is established, this field indicates whether the connection is an SPXII or an SPX connection.

Compatibility Procedures

Although the SPXII driver is backward compatible, SPX applications can't use all the features in SPXII without some modification. Also, some coding practices that worked with SPX may allow the application to access only the SPX features in SPXII.

Developers should be aware of the following:

- Allocation procedures for TLI structures
- Datastream type differences
- Device selection procedures

Allocation Procedures for TLI Structures

An SPXII / TLI-based application should use **t_alloc** and **t_getinfo** to allocate and determine the size of the options structure rather than using stack variables and the C operator sizeof(SPX2_OPTIONS).

Since the size of the structure changed between SPX and SPXII, using the TLI functions allows the application to remain compatible with both SPX and SPXII, as well as future releases of SPXII which may modify the SPX2_OPTIONS structure.

Datastream Type Differences

SPXII has a defined value for 0xFD (Orderly Release request) and reserves the values from 0x80 through 0xFB in the Datastream field of the SPX / SPXII header.

SPX allowed applications to use values from 0x00 through 0xFD. SPX applications which use the reserved or defined values need to be modified.

Device Selection Procedures

The SPXII driver has been implemented so that an application can use either SPXII or SPX t_optmgmt structure.

To request the SPXII t_optmgmt structure, an application uses one of the following methods:

- Opens "/dev/nspx2"
- Issues the SPX_SPX2_OPTIONS ioctl (for information on SPX ioctls, see Chapter 10, "SPX/SPXII ioctls")

To request the SPX t_optmgmt structure from the SPXII driver, an application uses the following method:

- Opens "/dev/nspx"

SPX/SPXII Specific Information for TLI Functions

In the UNIX environment, SPX/SPXII is accessed via TLI. The TLI functions that are transport-provider layer specific for SPX/SPXII are documented in this chapter.

The order and use of these calls is described in *Network Programming Interfaces*, which you should also refer to for TLI/XTI calls and other information not found in this document.

Developers should also have the "spx_app.h" file.

Because SPXII is a session-based service and SPX is a connection-based service, both deliver data reliably. In addition, the transport user is notified if any errors occur during data transmission. This service is an attractive feature for applications that require relatively long-lived, datastream-oriented interaction. Upon encountering a data transmission error, SPX/SPXII retries a given number of times before closing the connection and notifying the connection user. SPX/SPXII also notifies the user if a disconnection indication is received from the remote connection endpoint.

Because the actual values for retries and the intervals between retries are tunable by the system administrators, the values will vary from host system to host system.

SPX / SPXII can function in both synchronous and asynchronous modes. The application developer determines the mode.

TLI Data Structures

The Transport Layer Interface to SPX / SPXII uses the following TLI structures which require transport provider-specific data:

t_bind	t_info
t_call	t_optmgmt
t_discon	

Each structure is included in the "Remarks" section of the documentation for the function that uses the structure. For more information about these structures, see *Network Programming Interfaces*.

Sequence of TLI Functions

The state of the endpoint follows the connection mode table in *Network Programming Interfaces*.

The type of application determines the sequence and the types of TLI calls.

- **Server applications** need to open a transport endpoint, listen for client connections, and then service client requests.
- **Client applications** need to open a transport endpoint, establish a connection with a server, and then request and receive information from the server.

The steps for these two types of procedures using a synchronous mode are outlined in the following sections.

Server Applications

To create a synchronized UNIX server application, follow the sequence outlined below:

1. Open the SPXII driver "/dev/nspx2" using the **t_open** call.

   The **t_open** call returns a file descriptor (*fd*).

2. To bind *fd* to a well-known socket number (so that subsequent client connection requests can arrive on it), use the **t_bind** call.

3. To obtain a second file descriptor (*fd2*), use the **t_open** call with the device set to the SPXII driver"/dev/nspx2".

4. To bind *fd2* to a dynamic socket number, use the **t_bind** call.

5. If necessary, set options using **t_optmgnt** call.

   The **t_optmgmt** function is optional, but if it is used, it must be used after the **t_bind** function and before the **t_accept** function.

6. To listen for incoming connection requests on *fd*, use the **t_listen** call.

   If clients are supposed to discover the server address, the server application must use SAP to advertise its services. For programming information, see Chapter 8, "SAP Library."

7. Upon receiving a connection request, fork a child.

8. The parent should close *fd2* and return to Step 3.

   The parent continues looping through Steps 3 to 7 until the parent exits (skips to Step 13).

   The forked child completes Steps 8 through 12.

9. To accept the connection request, the forked child issues a **t_accept.**

   t_accept (fd, fd2, call).

10. The forked child should close *fd* with **t_close.**

11. The forked child uses **t_snd** or **t_rcv** to send or receive data on *fd2*.

12. The forked child listens for or sends a disconnection indication using **t_rcvdis** or **t_snddis.**

13. On exiting, the forked child unbinds *fd2* using a **t_unbind** call and closes *fd2* using a **t_close** call.

14. On exiting, the parent unbinds *fd* using a **t_unbind** call and closes *fd* using a **t_close** call.

Client Applications

To create a synchronized UNIX client application, follow the sequence outlined below:

1. Open the SPXII driver "/dev/nspx2" using the **t_open** call.

   The **t_open** call returns a file descriptor (*fd*).

2. Obtain the address of the server you want to connect to.

   The client application can use any appropriate method for obtaining the endpoint's address:

   - Create a file that maps a server name to an address.
   - Use SAP API functions (for programming information, see Chapter 8, "SAP Library")
   - Scan a NetWare server's bindery for the destination address if the application has already established a connection with that particular server. (See the **NWScanProperty** function in *NetWare Library Reference for C* and then use NET_ADDRESS as *searchPropertyName*.)

     Although NetWare 4.x servers have Directory Services, they run in bindery emulation mode as a default.

3. Bind to a static or dynamic socket using a **t_bind** call.

4. If necessary, set options using **t_optmgnt** call.

   The **t_optmgmt** function is optional, but if it is used, it must be used after the **t_bind** function and before the **t_connect** function.

5. Send a connection request to the server using a **t_connect** call.

6. Use **t_snd** to send or **t_rcv** to receive data on *fd*.

7. Listen for or send a disconnection request. Use **t_rcvdis** to receive a disconnect request or **t_snddis** to send a disconnect request.

8. On exiting, use the **t_unbind** call to unbind the file descriptor obtained in Step 1 and then close it using the **t_close** call.

SPX Considerations

The SPXII driver is accessible via "/dev/nspx2" or "/dev/nspx". These are the nodes that applications open to access SPX/SPXII.

- Opening "/dev/nspx2" allows an expanded set of options (as described in **t_optmgmt** on page 205).

- Opening "/dev/nspx" provides a set of options compatible with SPX (see **t_optmgmt**).

After a connection is established with TLI, an application can pop "timod" and push "tirdwr" to get a read/write interface for SPXII. For more information, refer to *Network Programming Interfaces*.

TLI Reference for SPX

This section describes how to use the following TLI functions (listed in alphabetical order) with the SPXII driver.

t_accept	page 173	Accepts a connection request.
t_bind	page 177	Binds a socket to a given transport endpoint.
t_connect	page 186	Establishes a connection with an SPX/SPXII server application at a specified destination.
t_listen	page 195	Enables an SPX/SPXII application server to receive connection requests from SPX/SPXII clients.

t_open	page 201	Establishes a transport endpoint connected to a transport user.
t_optmgmt	page 205	Manages protocol-specific options.
t_rcv	page 211	Receives data over an established transport connection.
t_rcvdis	page 214	Returns a disconnect indication from the remote transport endpoint.
t_rcvrel	page 219	Acknowledges receipt of an orderly release indication.
t_snd	page 222	Send data over a transport connection.
t_snddis	page 226	Aborts a connection or rejects a connection request.
t_sndrel	page 230	Requests orderly release of a connection (SPXII only).

For a description of the standard use of these functions or for functions not described in this document, see *Network Programming Interfaces*.

t_accept

Accepts a connection request.

Synopsis

```
#include “spx_app.h”

int t_accept (
   int              spxFd,
   int              spxFd2,
   struct t_call    *call )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the endpoint that is receiving connection requests.

(IN) *spxFd2*

Passes the file descriptor of the endpoint on which the connection is to be established.

(IN) *call*

Passes a pointer to (or the address of) a t_call structure that the SPX server application used in its **t_listen** call. This structure contains the address of the remote transport endpoint, the remote endpoint’s connection ID, its allocation number, and the sequence number of the connection request.

Return Values

0	Successful
-1	Unsuccessful

If **t_accept** returns an error, *t_errno* may be set to one of the following.

TBADF	The specified file descriptor does not refer to a transport endpoint, or the application is illegally accepting a connection on the same endpoint on which the connection indication arrived.
TOUTSTATE	The local transport endpoint or the accepting stream (specified by the accepting *fd*) is not in the appropriate state for issuing **t_accept.**
TBADSEQ	The connection request specified by the sequence number in the call structure is invalid.
TLOOK	An asychronous event has occurred on the transport endpoint referenced by *spxFd* and requires immediate attention.
TSYSERR	A system error has occurred during execution of this function; check *errno* for possible further information.

Remarks

An SPX / SPXII server application uses the **t_accept** call to accept a client connection request. The SPX / SPXII server application uses **t_snddis** to reject the connection request.

This function works as specified in *Network Programming Interfaces*.

For server applications (applications that wait for incoming connection requests), we recommend that the server application use two file descriptors. The server uses one file descriptor and socket to listen for incoming connection requests and uses another separate file descriptor and socket to accept a connection request each time a connection request arrives.

Usually the endpoint that listens for connections opens the SPXII driver and binds to a well-known socket (*spxFd*). Upon receiving a connection request, the SPX / SPXII server opens another file descriptor (*spxFd2*), binds to a dynamic socket, and issues a **t_accept**. When both the **t_accept** and the **t_connect** calls are successful, the client's **t_connect** call returns the dynamic socket number of the server. This procedure causes all client / server traffic to be on the server's dynamic socket.

A connection request can be accepted on the same *fd* as the listen *fd* (*spxFd=spxFd2*), but this is not recommended. Only a single connection request can be accepted if file descriptors are equal. The SPXII driver drops any further connection requests to the local transport endpoint (*spxFd*) because this endpoint is now in the data transfer state.

If a client's connection request timeout is short, the client may time out before the SPXII server application can issue a **t_accept** call. If the client connection request times out before the server application issues a **t_accept** call, you need either to extend the client's connection request timeout or to reduce the server's delay between the **t_listen** and the **t_accept**.

The SPXII driver detects a transmission failure of a connection request acknowledge and resends it.

If the **t_accept** call fails with *t_errno* equal to TOUTSTATE or TSYSERR, the SPXII driver marks the outstanding connection request (that the **t_accept** was replying to) invalid. If a TOUTSTATE or TSYSERR occurs, the SPXII server application should not retry the **t_accept**.

If any other errors occur, the SPXII server application can retry the **t_accept**.

Example

```
{
   char          *spxDevice = "/dev/nspx2";
   int           spxFd;
   int           spxFd2;
   struct t_info spxInfo;
   SPX2_OPTIONS  *reqOpts;

   if ((spxFd2 = t_open(spxDevice, O_RDWR, &spxInfo)) < 0) {
      t_error("t_open failed");
      exit(-1);
   }
   ..
   ..

   /* Bind to dynamic socket. I don't want to know what address I am
   ** bound to. */

   ..
   ..
```

```
    if ((t_bind(spxFd2, NULL, NULL)) < 0) {
       t_error("t_bind failed");
       exit(-1);
    }
    ..
    ..

     /*
     ** spxFd is the file descriptor representing the stream that the
     ** connection request arrived on. The call structure is also the
     ** same call structure that was returned from the t_listen when
     ** the connection request arrived. Some options can be set/changed
     ** on the t_accept call, but no confirmation will be returned.
     */

    reqOpts = (SPX2_OPTIONS *)rcvcall->opt.buf;
    reqOpts->spxIILocalWindowSize = 12;
    reqOpts->spxIIRetryCount = 8;
    reqOpts->spxIIMinimumRetryDelay = 250;  /* 1/4 second */
    reqOpts->spxIIMaximumRetryDelta = 2000; /* 2 seconds */

    if ((t_accept(spxFd, spxFd2, rcvcall)) < 0) {
       t_error("t_accept failed");
       if (t_errno == TLOOK) {
          lookVal = t_look(spxFd);
          printLookVal(lookVal);
       }
       exit(-1);
     }
}
```

State

The state after a successful connection establishment is T_DATAXFER for both the client and server. An unsuccessful **t_accept** call leaves the state T_IDLE.

See Also

t_connect
t_getstate
t_optmgmt
t_listen
t_open

t_bind

Binds a socket to a given transport endpoint.

Syntax

```
#include “spx_app.h”

int t_bind (
   int             spxFd,
   struct t_bind   *req,
   struct t_bind   *ret )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *req*

Passes a pointer to a t_bind structure. The t_bind structure contains the socket number to bind to. The socket value in the structure is initialized either to a static socket number (an assigned number) or to zero (to obtain a dynamic socket number).

The application can set *req* to NULL to obtain a dynamic socket (*qlen* is assumed to be zero).

(IN) *ret*

Passes a pointer to a t_bind structure. The local endpoint address is returned in the structure to which *ret* is pointing.

If the application does not care to which address it was bound, it can set *ret* to NULL.

(OUT) ret

Returns the local endpoint’s address: network address, node address, and socket number.

Return Values

0	Successful
-1	Unsuccessful

If **t_bind** returns an error, *t_errno* may be set to one of the following.

TBADF	The specified file descriptor does not refer to a transport endpoint.
TOUTSTATE	This transport endpoint is in a state that invalidates a **t_bind** request.
TBADADDR	Either the address passed down was not the same size of an ipxAddr_t, or the size of the address was not zero (NULL bind pointer).
TNOADDR	There are no unused dynamic socket numbers. The SPX user should try again later.
TACCES	The socket number requested was in use.
TBUFOVFLW	The number of bytes allowed for the return argument is not sufficient to store the value of that argument.
TSYSERR	A system error has occurred during the execution of this function. Check *errno* for possible further information.

Remarks

The **t_bind** call associates a protocol address with a given transport endpoint. This call binds the endpoint to an SPX/SPXII socket. This call also directs the transport provider to begin accepting incoming connection requests.

This function works as specified in *Network Programming Interfaces* with the additions explained below.

The t_bind structure has the following format:

```
struct t_bind {
  netbuf     addr;
  unsigned   qlen;
};
```

The *qlen* field is used to indicate the total number of outstanding connection requests allowed on this endpoint.

- Applications that do *not* service connection requests should set this field to zero (0).
- Applications that service connection requests should set this field to one (1).

For additional information, see "Outstanding Connection Requests" on page 181.

The netbuf structure has the following format:

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

For a **t_bind** call, a pointer to an ipxAddr_t structure must be passed in the *req.addr.buf* field to bind to a static socket. If binding to a dynamic socket, a NULL pointer can be passed.

The *i*pxAddr_t structure has the following format:

```
typedef struct ipxAddress{
   uint8    net[4];
   uint8    node[6];
   uint8    sock[2];
} ipxAddr_t;
```

The **t_bind** call allows an endpoint to bind to a socket number, which can be either dynamic or static. SPX keeps track of which socket number is bound to which transport endpoint.

- A dynamic socket number is an unused socket number returned by the SPXII driver and is guaranteed to be a unique unused number among the IPX/SPX endpoints. A dynamic socket is a value from 0x4000 to 0x7FFF.

  If a dynamic socket is wanted, set the *sock* field in the ipxAddr_t structure to zero.

- A static socket number can be requested. If it is unused, it is granted and returned in the *ret* structure. Static socket numbers are in the range of 0x8000 to 0xFFFF and are assigned by Novell. If your application requires a static socket number, contact Novell for an assignment.

  If a static socket number is desired, set the *sock* field in the ipxAddr_t structure to the assigned socket number.

The *net* and *node* fields do not need to be filled in the ipxAddr_t structure, but the *sock* field must be initialized to either a static or dynamic socket value.

Static Socket Numbers

Services written to run over SPX/SPXII generally have well-known or static socket numbers associated with them. (Again, contact Novell to obtain an assignment for a static socket number for your application.) By having static socket numbers, SPX/SPXII users can be sure that their server and client application types match.

To bind to a static socket number, complete the following steps.

1. Allocate a t_bind structure for *req* and *ret*. They can be the same structure.

2. Set the socket value in the ipxAddr_t structure before making the **t_bind** call. The example code uses two #defines to specify the socket number and to ensure that the socket number is passed in hi-lo format.

3. Initialize the structure's fields. The *req.addr.buf* field must point to the ipxAddr_t structure allocated in Step 1.

4. Make the **t_bind** call by passing the *spxFd* value returned in the **t_open** call and by passing the address of the t_bind structure allocated in Step 1.

The SPXII driver looks at the socket field in the ipxAddr_t structure for the SPX user's desired socket number. The socket number must be passed in hi-lo byte order.

If the socket number desired is not currently being used by another IPX/SPX user, the SPXII driver returns the local network address, local node address, and the allocated or requested socket number in the corresponding fields of the ipxAddr_t structure of the *ret.addr.buf* field.

Only one IPX/SPX endpoint can bind to a given socket number at a time. If the user tries to bind to a socket that has already been bound to, an error results and the bind fails.

Another method to coordinate servers and clients is to use the Service Advertising Protocol (SAP). For programming information, see Chapter 8, "SAP Library," on page 233.

Dynamic Socket Number

Two methods can be used to have the SPXII driver allocate and assign a dynamic socket number.

- Pass zero (0) in the *sock* field of the ipxAddr_t structure
- Set *req* to NULL—when you do not need to know the value of the socket number

Outstanding Connection Requests

An outstanding connection request is a connection request that has arrived and has been delivered to the UNIX application, but to which the application has not responded with a connection request acknowledge (**t_accept**) or connection request reject (**t_snddis**).

The *qlen* field in the t_bind structure indicates to SPXII the total number of outstanding connection requests allowed on this transport endpoint.

SPXII allows up to a specified number of outstanding connection requests per transport endpoint. This is currently set to 5. Even if the UNIX application requests more than 5, only 5 are given.

Although you are allowed to have more than one outstanding connection request, we recommend that you have only one.

If a value greater than 1 is specified in the *qlen* field during a **t_bind**, a connection request can arrive from a remote transport endpoint, making the **t_listen** unblock.

If another connection request arrives between the time the **t_listen** unblocks and the **t_accept** is issued, the **t_accept** fails, saying that an event has occurred. You will not be able to **t_accept** the connection requests until all pending connection requests have been retrieved from the stream head using **t_listen.** A **t_bind** with *qlen* equal to 1 should be issued to avoid this outcome.

Example 1 shows how to bind to a dynamic socket, while Example 2 shows how to bind to a specific or static socket number.

Example 1

```
 /* Bind to dynamic socket. I don't want know what address I am
 ** bound to.
 */
{
   ..
   ..
   if ((t_bind(spxFd, NULL, NULL)) < 0) {
      t_error("t_bind failed");
      exit(-1);
   }
   ..
   ..
}
```

Example 2

```
/* This example shows how to bind to the specific socket 0x4500.
** The SPXII driver fills in the net, node fields of the IPX address,
** and returns the full address.
*/

/*
** Bind to static socket 0x4500; then print full address after t_bind.
*/

#define SOCKET_TO_BIND_HIGH 0x45
#define SOCKET_TO_BIND_LOW 0x00
{
   int            spxFd;
   struct t_bind  *bind_req;
   struct t_bind  *bind_ret;
   ipxAddr_t      *ipxAddr;
   ..
   ..

   /*
   ** Allocate structures for t_bind request
   */

if ((bind_req = (struct t_bind *)t_alloc(spxFd, T_BIND, T_ALL)) == NULL ) {
   t_error("t_alloc of T_BIND request structure failed");
   exit(-1);
}

   /*
   ** Allocate structures for t_bind return values
   */

if ((bind_ret = (struct t_bind *)t_alloc(spxFd, T_BIND, T_ALL)) == NULL ) {
   t_error( "t_alloc of T_BIND return structure failed");
   exit(-1);
}

   /*
   ** qlen 0 for clients, 1-5 for servers. qlen is the # of
   ** outstanding connect indications allowed.
   */
```

```
bind_req->qlen = 0;
bind_req->addr.len = sizeof(ipxAddr_t);
ipxAddr = (ipxAddr_t *)bind_req->addr.buf;
ipxAddr->sock[0] = SOCKET_TO_BIND_HIGH;
ipxAddr->sock[1] = SOCKET_TO_BIND_LOW;

if (t_bind(spxFd, bind_req, bind_ret) < 0) {
   t_error( “t_bind failed”);
   exit(-1);
}

   /*
   ** Print t_bind returned values
   */

   fprintf(stderr,“\nt_bind returned:\n”);

   /* qlen */
   fprintf(stderr,“\t%4d for qlen from %s\n”, bind_ret->qlen, spxDev);

   /* number of address bytes returned */
   fprintf(stderr,“\t%4d bytes of address from %s\n”,
      bind_ret->addr.len, spxDev);
   ipxAddr = (ipxAddr_t *)bind_ret->addr.buf;

   /* network */
   fprintf(stderr,“\tBound to address:\n\t net 0x%02X%02X%02X%02X\n”,
       ipxAddr->net[0],ipxAddr->net[1],ipxAddr->net[2],ipxAddr->net[3]);

   /* node */
   fprintf(stderr,“\t node 0x%02X%02X%02X%02X%02X%02X\n”,
      ipxAddr->node[0],ipxAddr->node[1],ipxAddr->node[2],
      ipxAddr->node[3],ipxAddr->node[4],ipxAddr->node[5]);

   /* socket */
   fprintf(stderr,“\t socket 0x%02X%02X\n”, ipxAddr->sock[0],
      ipxAddr->sock[1]);

   /*
   ** free structures used for t_bind
   */

   t_free((char *)bind_req, T_BIND);
   t_free((char *)bind_ret, T_BIND);
   ..
   ..
}
```

State

After a successful bind, the state is T_IDLE.

After an unsuccessful bind, the state is T_UNBND unless *t_error* was TOUTSTATE.

See Also

t_open
t_optmgmt
t_unbind

t_connect

Establishes a connection with an SPX/SPXII server application at a specified destination.

Syntax

```
#include “spx_app.h”

int t_connect (
   int            spxFd,
   struct t_call  *sndcall,
   struct t_call  *rcvcall )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *sndcall*

Passes a pointer to a t_call structure that contains the IPX address of the server to which the SPX / SPXII client wants to connect.

If *spxFd* is an SPXII “ / dev / nspx2” file descriptor, then the sndcall structure can contain an option structure (SPX2_OPTIONS) to change the current options.

(IN) *rcvcall*

Passes a pointer to a t_call structure that will contain the server’s connection information upon successful completion of the call.

(OUT) *rcvcall*

Receives the server’s connection information: network address, node address, socket number, connection ID, and allocation number.

If *spxFd* is an SPXII “ / dev / nspx2” file descriptor, then the rcvcall structure will contain an option structure (SPX2_OPTIONS) with the current options.

Return Values

0	Successful
-1	Unsuccessful

The SPXII driver tries to connect with the remote transport endpoint. After trying a given number of times without receiving an acknowledgment, the SPXII driver generates a disconnection indication of TLI_SPX_CONNECTION_FAILED (refer to **t_rcvdis**). If this error occurs, the state of the stream is set to T_IDLE.

See *Network Programming Interfaces* for other possible errors.

Remarks

A client SPX/SPXII application uses the **t_connect** call to request a connection to an SPX/SPXII server application at a specified destination. This call may be executed in either synchronous or asynchronous mode.

SPX/SPXII supports both synchronous and asynchronous modes according to the specifications in *Network Programming Interfaces.* This function works as specified in that manual.

In the synchronous mode, the call waits for the server's response before returning control to the client. In asynchronous mode, the call initiates connection establishment but returns control to the client before a response arrives. The **t_rcvconnect** function must be used to complete an asynchronous connection.

The t_call structure has the following format:

```
struct t_call {
  struct netbuf  addr;
  struct netbuf  opt;
  struct netbuf  udata;
  int            sequence;
};
```

The SPXII driver does not use the udata structure. Its fields should be initialized. You must set the *udata.len* and *udata.maxlen* fields to zero and set the *udata.buf* field to NULL.

The SPXII driver does not use the *sequence* field.

The netbuf structure has the following format:

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

For standard information about the t_call and netbuf structures, see *Network Programming Interfaces.*

The **t_connect** call uses the following t_call structures:

- sndcall
- rcvcall

The *sndcall.addr.len* and *sndcall.addr.maxlen* fields must be initialized to the size of an ipxAddr_t or equivalent structure. The *sndcall.addr.buf* field must point to an ipxAddr_t structure. The ipxAddr_t structure must be initialized to the server's IPX address.

The ipxAddr_t structure has the following format:

```
typedef struct ipxAddress{
   uint8    net[4];
   uint8    node[6];
   uint8    sock[2];
} ipxAddr_t;
```

All information passed in the ipxAddr_t structure must be in hi-lo byte order. See Figure 4-1 on page 46 for an illustration of byte order.

The *sndcall.opt.len* and *sndcall.opt.maxlen* fields must be initialized either to the size of a valid option structure or to zero. SPXII supports two different option structures, one for SPX (SPX_OPTS) and the other for SPXII (SPX2_OPTIONS).

See page 206 for the t_optmgmt option structure formats.

The SPX_OPTS structure has the following format:

```
typedef struct spxopt_s {
   unsigned char  spx_connectionID[2];
   unsigned char  spx_allocationNumber[2];
} SPX_OPTS;
```

Both endpoints must support orderly release before an application can use the orderly release calls. Although older versions of SPX did not support orderly release, the *spxIISessionFlags* can be used to determine whether both endpoints support orderly release. The *spxIISessionFlags* in the opt (SPX2_OPTIONS) structure should be saved if the application wants to use orderly release. See **t_sndrel** on page 230 for further information.

The **t_connect** call sends an SPX / SPXII connection request to the IPX address specified in *sndcall.addr*.

If the rcvcall structure is passed in **t_connect**, the server's address and connection information are returned.

If an rcvcall structure is passed, the *maxlen* and *buf* variables must be set appropriately to receive the ipxAddr_t and option structures.

The server's IPX address is returned in the *rcvcall.addr.buf* field, while the server's connection ID and allocation number are passed back in the *rcvcall.opt.buf* field.

Example 1 below is for an SPXII connection request, while Example 2 is for SPX.

Example 1

```
{
   char *spx2Device = “/dev/nspx2”;
   int            spxFd;
   uint32         spxIISessionFlags;
   ipxAddr_t      *serversAddress;
   struct t_call  *sndcall;
   struct t_call  *rcvcall;
   SPX2_OPTIONS   *reqOpts;
   SPX2_OPTIONS   *retOpts;

   ..
   ..
   ..

   /* Open an spxII device */
   if ((spxFd = t_open(spx2Device, O_RDWR, &spxInfo)) <0) {
      t_error(“t_open failed”);
      exit(-1);
   }

   /* Bind to dynamic socket */
   if ((t_bind(spxFd, NULL, NULL)) < 0) {
      t_error( “t_bind failed”);
      exit(-1);
   }

   /* Allocate call structures */
   if ((sndcall = (struct t_call *)t_alloc(spxFd, T_CALL, T_ALL))==NULL) {
      t_error( “t_alloc of T_CALL failed”);
      exit(-1);
   }

   if ((rcvcall = (struct t_call *)t_alloc(spxFd, T_CALL, T_ALL))==NULL) {
      t_error( “t_alloc of T_CALL failed”);
      exit(-1);
   }

   /* The first step in making a connection is to obtain the address of
   ** the SPXII user you want to establish the connection with. There are
   ** three approaches to obtaining an address.
   ** 1. You can query a NetWare bindery for the server type you want.
   ** ( This method assumes that you have established a connection with
   ** a NetWare file server. Use the NWScanProperty function with
   ** NET_ADDRESS as the searchPropertyName. )
   ** 2. You can create a file that maps a server name to an address.
```

```
** 3. You can use any appropriate method for discovering the endpoint's
** address, for example, the SAP APIs.
** You must allocate an ipxAddr_t structure and initialize the
** fields to the endpoint's address before making the t_connect
** call. The following example code assumes that the server has
** the following address: network = 0x0101038B, node = 0x01,
** socket = 0xDEAD.*/

serversAddress = (ipxAddr_t*)sndcall->addr,buf;
serversAddress.net[0] = 0x01;
serversAddress.net[1] = 0x01;
serversAddress.net[2] = 0x03;
serversAddress.net[3] = 0x8B;
serversAddress.node[0] = 0x00;
serversAddress.node[1] = 0x00;
serversAddress.node[2] = 0x00;
serversAddress.node[3] = 0x00;
serversAddress.node[4] = 0x00;
serversAddress.node[5] = 0x01;
serversAddress.sock[0] = 0xDE;
serversAddress.sock[1] = 0xAD;
sndcall->addr.len = sndcall->addr.maxlen;

/*
** Change SPXII options on t_connect
*/
reqOpts = (SPX2_OPTIONS *)sndcall->opt.buf;
reqOpts->versionNumber = OPTIONS_VERSION;
reqOpts->spxIIOptionNegotiate = SPX2_NEGOTIATE_OPTIONS;
reqOpts->spxIIRetryCount = 7;
reqOpts->spxIIMinimumRetryDelay = 500;  /* 1/2 second */
reqOpts->spxIIMaximumRetryDelta = 2000; /* 2 seconds */
reqOpts->spxIILocalWindowSize = 10;
sndcall->opt.len = sndcall->opt.maxlen;

if ((t_connect(spxFd, sndcall, rcvcall)) < 0) {
   t_error( "t_connect failed");
   if (t_errno == TLOOK) {
      lookVal = t_look(spxFd);
      printLookVal(lookVal);
   }
   exit(-1);
}
/*
** Upon successful completion, rcvcall->opt.buf will have the
** connection identification number of the server and the current
** options.
*/
```

```
   retOpts = (SPX2_OPTIONS *)rcvcall->opt.buf;
   retOpts = (SPX2_OPTIONS *)rcvcall->opt.buf;

   /* Save spxII session flags, needed for orderly release */
   spxIISessionFlags = retOpts->spxIISessionFlags;

   fprintf(stderr,“Servers Window size:----------- %06d\n”,
      retOpts->spxIIRemoteWindowSize);
   fprintf(stderr,“Servers connection ID Number:-- %06d\n”,
      GETINT16(retOpts->spxIIConnectionID));
   fprintf(stderr,“Inbound Packet size:----------- %06d\n”,
      retOpts->spxIIInboundPacketSize);
   fprintf(stderr,“Outbound Packet size:---------- %06d\n”,
      retOpts->spxIIOutboundPacketSize);
   fprintf(stderr,“SPXII Session Flags: ---------- 0x%04X\n”,
      retOpts->spxIISessionFlags);

   t_free((char *)rcvcall, T_CALL);
   t_free((char *)sndcall, T_CALL);
}
```

Example 2

```
{
   char          *spxDevice = “/dev/nspx”;
   int           spxFd;
   ipxAddr_t     serversAddress;
   struct t_call *call;
   SPX_OPTS      *ret_Opts;
   uint16        connectionId;
   ..
   ..
   ..

    /* Open an spx device */
   if ((spxFd = t_open(spxDevice, O_RDWR, &spxInfo)) <0) {
      t_error( “t_open failed”);
      exit(-1);
   }

    /* Bind to dynamic socket */
   if ((t_bind(spxFd, NULL, NULL)) < 0) {
      t_error( “t_bind failed”);
      exit(-1);
   }
```

```
if ((spxFd=t_open( spxDevice, O_RDWR, &spxInfo))<0) {
   t_error( “t_open failed” );
   ..
   ..
}

/* Allocate call structure */
if ((call = (struct t_call *)t_alloc(spxFd, T_CALL, T_ALL))==NULL) {
   t_error( “t_alloc of T_CALL failed”);
   exit(-1);
}

serversAddress = (ipxAddr_t*)call->addr,buf;
serversAddress.net[0] = 0x01;
serversAddress.net[1] = 0x01;
serversAddress.net[2] = 0x03;
serversAddress.net[3] = 0x8B;
serversAddress.node[0] = 0x00;
serversAddress.node[1] = 0x00;
serversAddress.node[2] = 0x00;
serversAddress.node[3] = 0x00;
serversAddress.node[4] = 0x00;
serversAddress.node[5] = 0x01;
serversAddress.sock[0] = 0xDE;
serversAddress.sock[1] = 0xAD;

call->addr.len = call->addr.maxlen
call->opt.len = 0;
call->opt.buf = NULL;

/*
** We pass the address of the call structure for both the request
** and return fields so that the call->opt.buf field will be filled
** in with the server’s information.
*/
if ((t_connect(spxFd, call, call)) < 0) {
   t_error( “t_connect failed”);
   if (t_errno == TLOOK) {
      lookVal = t_look(spxFd);
      printLookVal(lookVal);
   }
   exit(-1);
}

/*
** Upon successful completion, call->opt.buf will have the
** connection identification number of the server.
*/
```

```
    ret_Opts = (SPX_OPTS *)call->opt.buf;
    memcpy(&connectionId, ret_Opts->spx_connectionID,
       sizeof(connectionId));
    fprintf(stderr,“Servers connection ID number: %06d\n”,
       GETINT16(connectionId));
    t_free((char *)call, T_CALL);
}
```

State

The state after a successful connection establishment is T_DATAXFER for both the client and server. The state after an unsuccessful connection establishment is T_IDLE.

See Also

t_accept
t_listen
t_open
t_optmgmt
t_sndrel

t_listen

Enables an SPX/SPXII application server to receive connection requests from SPX/SPXII clients.

Syntax

```
#include"spx_app.h"

int t_listen (
   int            spxFd,
   struct t_call  *rcvcall )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *rcvcall*

Passes a pointer to a t_call structure. The fields in the structure must be initialized to the proper size of the incoming data. See "Remarks" for information about the fields and the appropriate values for the fields.

(OUT) *rcvcall*

Receives the address of the remote endpoint, its connection ID, its allocation number, and—if *spxFd* is an SPXII file descriptor—an SPX2_OPTIONS structure. See "Remarks" for more information.

Return Values

0	Successful
-1	Unsuccessful

See *Network Programming Interfaces* for other possible errors.

Remarks

The **t_listen** call enables an SPX/SPXII application server to receive connection requests from SPX/SPXII clients. A successful **t_listen** call returns the client's address, connection ID, and allocation number. The application server sends a response back to the client: either a **t_accept** to accept the connection or a **t_snddis** to reject the connection.

This function works as specified in *Network Programming Interfaces* with the following additions.

The t_call structure has the following format:

```
struct t_call {
  struct netbuf  addr;
  struct netbuf  opt;
  struct netbuf  udata;
  int            sequence;
};
```

The **t_alloc** call will allocate memory for all needed structures. It will initialize the *buf* pointers and *maxlen* fields for all netbuf structures. The *len* field of the netbuf structure must be initialized by the application for buffers sent to SPX/SPXII.

The netbuf structure has the following format:

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

Use of **t_alloc** to allocate structures will help ensure the compatibility of user programs with future releases of SPXII.

On return, the *addr.buf* field will point to an ipxAddr_t structure with the following format:

```
typedef struct ipxAddress{
   uint8    net[4];
   uint8    node[6];
   uint8    sock[2];
} ipxAddr_t;
```

A value is placed in the *rcvcall.sequence* field by SPX/SPXII. If the application wants to accept the connection request, the sequence field must be the same for the **t_accept** call. If the application wants to reject the connection request it can use the **t_snddis** call with the same sequence field from the **t_listen** call.

If **t_listen** returns successfully, *rcvcall.addr* points to an ipxAddr_t structure that contains the *net*, *node*, and *sock* of the remote transport endpoint requesting the connection. The *net*, *node*, and *sock* are in hi-lo byte order. The *rcvcall->opt.buf* points to the option structure (SPX2_OPTIONS or SPX_OPTS) returned by SPX. It will contain information about the connection.

The SPX_OPTS structure has the following format:

```
typedef struct spxopt_s {
   unsigned char  spx_connectionID[2];
   unsigned char  spx_allocationNumber[2];
} SPX_OPTS;
```

See page 208 for the SPX2_OPTIONS structure format.

Both endpoints must support orderly release before an application can use the orderly release calls. Although older versions of SPX did not support orderly release, the *spxIISessionFlags* can be used to determine whether both endpoints support orderly release. The *spxIISessionFlags* in the *opt* (SPX2_OPTIONS) structure should be saved if the application wants to use orderly release.

For further information, see **t_sndrel** on page 230 and **t_optmgmt** on page 205.

The **t_listen** call retrieves any connection requests residing on the stream head. The **t_listen** call can function synchronously or asynchronously.

- When the call functions synchronously, it blocks until a connection request comes in.
- When the call functions asynchronously, it checks for a connection request and returns failure if no connection requests exist.

SPX/SPXII ensures that each connection indication is unique by dropping any duplicate connection requests. A duplicate request is a request that comes from the same network, node, socket, and source connection ID as a previous request.

When a **t_connect** call has been received from a client, the SPX/SPXII server can either accept or reject the connection request.

- To accept the connection request, the SPX/SPXII server uses the **t_accept** call. The sequence number from the **t_listen** call structure should be the same for the **t_accept** call.
- To reject the connection, the SPX/SPXII server uses the **t_snddis** call. The sequence number in the call structure used for the **t_snddis** and sequence number in the **t_listen** call structure must be the same.

SPX/SPXII sends a terminate connection indication if the application issues a **t_snddis** after a **t_listen** return.

Example

```
{
   int            spxFd;
   int            len;
   uint32         spxIISessionFlags;
   struct t_call  *rcvcall;
   SPX2_OPTIONS   *retOpts;
   ipxAddr_t      *ipxAddr;
   ..
   ..
```

```
if((rcvcall = (struct t_call *)t_alloc(spxFd, T_CALL, T_ALL))==NULL) {
  t_error( “t_alloc of T_CALL failed”);
  exit(-1);
}

rcvcall->addr.len = rcvcall->addr.maxlen;
rcvcall->opt.len = rcvcall->opt.maxlen;
rcvcall->udata.len = 0;
len = rcvcall->opt.maxlen;
/*
** Since this is a synchronous call, the call will block until a
** connection request comes in. When the call returns, the
** rcvcall->addr will contain the remote address.
** The rcvcall->opt.buf will be a pointer to the option structure
** (SPX_OPTS or SPX2_OPTIONS). See t_optmgmt for the structure
** formats. If this call were in asynchronous mode, the t_listen
** call will return fail if no connection requests have arrived,
** or success if one has arrived.
*/

/*
** Listen for a connect request
*/
if ((t_listen(spxFd, rcvcall)) < 0) {
   t_error( “t_listen failed”);
   if (t_errno == TLOOK) {
      lookVal = t_look(spxFd);
      printLookVal(lookVal);
   }
   exit(-1);
}

ipxAddr = (ipxAddr_t *)rcvcall->addr.buf;
fprintf(stderr,“\tConnect Request from:\n\t net 0x%02X%02X%02X%02X\n”,
   ipxAddr->net[0],ipxAddr->net[1],
   ipxAddr->net[2],ipxAddr->net[3]);
fprintf(stderr,“\t node 0x%02X%02X%02X%02X%02X%02X\n”,
   ipxAddr->node[0],ipxAddr->node[1],ipxAddr->node[2],
   ipxAddr->node[3],ipxAddr->node[4],ipxAddr->node[5]);
fprintf(stderr,“\t socket 0x%02X%02X\n”, ipxAddr->sock[0],
   ipxAddr->sock[1]);
```

```
    retOpts = (SPX2_OPTIONS *)rcvcall->opt.buf;
    /* Save spxII session flags, which are needed for orderly release */
    spxIISessionFlags = retOpts->spxIISessionFlags;
    fprintf(stderr,"Clients Window size:---------- %06d\n",
       retOpts->spxIIRemoteWindowSize);
    fprintf(stderr,"Clients connection ID Number:-- %06d\n",
       GETINT16(retOpts->spxIIConnectionID));
    fprintf(stderr,"SPXII Session Flags: ---------- 0x%04X\n",
       spxIISessionFlags);

    ..
    ..
     /* Accept this connect request on the same fd, only one connection */
     rcvcall->udata.len = 0;
     rcvcall->opt.len = len;

     if ((t_accept(spxFd, spxFd, rcvcall)) < 0) {
        t_error("t_accept failed");
        if (t_errno == TLOOK) {
           lookVal = t_look(spxFd);
           printLookVal(lookVal);
        }
        exit(-1);
     }
     t_free((char *)rcvcall, T_CALL);
}
```

State

When **t_listen** returns with a connect indication, the state will be T_INCON.

See Also

t_accept
t_bind
t_connect
t_optmgmt
t_snddis
t_sndrel

t_open

Establishes a transport endpoint for a specified transport provider.

Syntax

```
#include “spx_app.h”

int  t_open (
   char             spxDevice,
   int              oflag,
   struct t_info    *spxInfo )
```

Parameters

(IN) *spxDevice*

Passes a pointer to the path of the SPXII driver.

(IN) *oflag*

Passes the option flags for the opened stream.

(IN) *spxInfo*

Passes a pointer to the t_info structure. See “Remarks” for the format of the t_info structure.

(OUT) *spxInfo*

Receives the SPXII protocol information as a t_info structure. See “Remarks” for the format of the t_info structure.

Return Values

>0	Successful
-1	Unsuccessful

See *Network Programming Interfaces* for other possible errors.

If the **t_open** is successful, the value returned is a file descriptor that identifies the local transport endpoint. This document uses the variable *spxFd* to refer to this value.

If **t_open** returns an error, *t_errno* may be set to one of the following.

TSYSERR	A system error has occurred during execution of this function. Check *errno* for possible further information.
TBADFLAG	An invalid flag was specified.

Remarks

The **t_open** function creates a local transport endpoint and returns protocol-specific information associated with that endpoint as well as a file descriptor that serves as the local identifier of the endpoint. Both server and client applications can use this call to open a transport endpoint.

This function works as specified in *AIX Version 4 Technical Reference Vol. 4: Communications (SC23-2617-01)*. SPX / SPXII can be used either synchronously or asynchronously.

The path and name of the clonable SPXII device is "/dev/nspx2" or "/dev/nspx". The difference between opening "nspx2" and "nspx" depends on the options that are allowed in other TLI calls:

- Opening "nspx2" allows an expanded set of options associated with SPXII (see the **t_optmgmt** documentation).
- Opening "nspx" allows a set of options compatible with SPX (see the **t_optmgmt** documentation).

A successful **t_open** call returns a TLI information structure. The t_info structure contains the following information about SPX / SPXII.

Table 7-3
SPX/SPXII Information in the t_info Structure

Field	Value	Description
addr	12 (bytes)	This is the number of bytes required for an IPX address. The address consists of three components: network address 4 bytes node address 6 bytes socket number 2 bytes
tsdu	-1	An unlimited amount of data can be sent during a connection.
etsdu	-2	Not supported.
connect	-2	Not supported.
discon	-2	Not supported.
servtype	T_COTS_ORD	The service type for SPXII is always T_COTS_ORD. SPXII is a connection-oriented service with orderly release.
(nspx) options	4	SPX supports 4 bytes of option data. For a description, see **t_optmgmt**.
(nspx2) options	52	SPXII supports 52 bytes of option data. For a description, see **t_optmgmt**.

Example

```
{
  char   *spxDevice = “/dev/nspx2”;
  int    spxFd;
  struct t_info   spxInfo;
   ..
   ..
```

```
    /* Open spxII with expanded options */
    if ((spxFd = t_open(spxDevice, O_RDWR, &spxInfo)) < 0) {
        fprintf(stderr,“open of %s failed \n”,spxDevice);
        t_error( “t_open failed”);
        exit(-1);
    }
    ..
    ..
}
```

State

A **t_open** call changes the state of the service connection to T_UNBND (unbound).

See Also

t_bind
t_close
t_optmgmt

t_optmgmt

Manages protocol-specific options.

Syntax

```
#include “spx_app.h”

int t_optmgmt (
   int                spxFd,
   struct t_optmgmt   *req,
   struct t_optmgmt   *ret )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *req*

Passes the address of the t_optmgmt structure that contains the requested values for each option.

(IN) *ret*

Passes the address of a t_optmgmt structure that will contain the granted values for each option.

(OUT) *ret*

Receives the granted values in the t_optmgmt structure.

Return Values

0	Successful
-1	Unsuccessful

If **t_optmgmt** returns an error, *t_errno* may be set to one of the following:

TBADOPT	The size of the structure was less than the size of the SPX or SPXII option structure, or there has been an internal SPX/TLI error.
TBADFLAG	The flag specified is invalid.
TOUTSTATE	This request was issued in some state other than T_IDLE.

Remarks

The **t_optmgmt** function enables the user to get, verify, or negotiate protocol options with the transport provider.

SPXII supports two different option structures: one for SPXII and the other for SPX. The available options depend on whether you use "nspx2" or "nspx". The "nspx2" set of options is an expanded set, while the "nspx" set of options is compatible with older versions of SPX.

This function works as specified in *Network Programming Interfaces*. This call has one negotiable option that enables the SPX / SPXII user to set the maximum number of retries when the SPXII driver tries to deliver data reliably to the opposite transport endpoint.

The t_optmgmt structure has the following format:

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

The netbuf structure has the following format:

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

The **t_optmgmt** call uses two t_optmgmt structures: a request (*req*) structure and a return (*ret*) structure. The same structure can be passed as both the request structure and the return structure.

Although the SPX structure (SPX_OPTMGMT) is the same as in previous releases of SPX, it should not be used for newly written applications. SPX_OPTMGMT is available only for compatibility reasons and has the following format:

```
typedef struct spx_optmgmt {
     uint8   spxo_retry_count;
     uint8   spxo_watchdog_flag;
     uint16  spxo_min_retry_delay;
} SPX_OPTMGMT;
```

The SPXII structure (SPX2_OPTIONS) is shown on the following page. It is used for calls **t_listen**, **t_accept**, **t_connect** and **t_optmgmt**. Because this structure is expandable in future versions of SPXII, a variable should never be declared directly (such as "struct SPX2_OPTIONS spxoptions;"). Likewise, the size of the structure should never be taken (for example, "sizeof(SPX2_OPTIONS);").

The **t_alloc** function should always be used to allocate the t_optmgmt structure. The size of the *opt.buf* in the t_optmgmt structure can be determined by either checking the value of the *options* field in the t_info structure used during **t_open**, or by checking *opt.maxlen* after the **t_alloc** call.

If the TLI calls are not used to allocate and determine the size, a TBUFOVFLW error could be generated when an application is run with a newer version of SPXII.

Only some of the SPXII structure elements are valid with the **t_optmgmt** call. The others are used for **t_listen**, **t_accept**, and **t_connect**.

The SPXII structure (SPX2_OPTIONS) has the following format.

Table 7-4
The SPX2_OPTIONS Structure

Type	Field	Description
uint32	versionNumber *	Must be set to OPTION_VERSION
uint32	spxIIOptionNegotiate *	Exchange options and negotiate packet size with other endpoint
uint32	spxIIRetryCount *	Number of transmit retries on data packets
uint32	spxIIMinimumRetryDelay *	Minimum retry timeout, in milliseconds
uint32	spxIIMaximumRetryDelta *	Maximum retry delta, in milliseconds
uint32	spxIIWatchdogTimeout	This is a SYSTEM parameter for UNIX SPXII
uint32	spxIIConnectionTimeout *	Number of milliseconds to wait for full connection setup
uint32	spxIILocalWindowSize *	Number of data packets in receive window
uint32	spxIIRemoteWindowSize	Remote endpoints initial receive window size
uint32	spxIIConnectionID	Valid only after connection is established
uint32	spxIIInboundPacketSize	Maximum receive packet size
uint32	spxIIOutboundPacketSize	Maximum transmit packet size
uint32	spxIISessionFlags	Session characteristic options

Note * Structure elements valid for **t_optmgmt**.

The *flags* field in the t_optmgmt structure must be initialized to the appropriate value. The **t_optmgmt** call supports the following flags: T_NEGOTIATE, T_CHECK, and T_DEFAULT.

Example

```
     /*
     ** Change SPXII local window size and retry count.
     ** If SPX device, change retry count only.
     */

{
     struct t_optmgmt  *req_opts;
     struct t_optmgmt  *ret_opts;
     SPX2_OPTIONS      *retIIOpts;
     SPX2_OPTIONS      *reqIIOpts;
     SPX_OPTIONS       *retOpts;
     SPX_OPTIONS       *reqOpts;
     int               len;
     ..
     ..

     /* Get proper size structure for request values t_optmgmt */
     if ((req_opts = (struct t_optmgmt *)
               t_alloc (spxFd, T_OPTMGMT, T_ALL)) == NULL ) {
        t_error( “t_alloc failed”);
        exit(-1);
     }
     /* Get proper size structure for return values from t_optmgmt */
     if ((ret_opts = (struct t_optmgmt *)
               t_alloc (spxFd, T_OPTMGMT, T_ALL)) == NULL ) {
        t_error( “t_alloc failed”);
        exit(-1);
     }
     len = req_opts->opt.maxlen;
     /*
       ** Get the DEFAULT options. Have defaults returned in request structure
     **/
     req_opts->flags = T_DEFAULT;
     req_opts->opt.len = len;
     if ((t_optmgmt(spxFd, req_opts, req_opts))<0) {
         fprintf (stderr,
             “t_optmgmt failed to %s failed t_errno= %d errno= %d\n”,
             spxDev, t_errno, errno);
         t_error (“t_optmgmt failed”);
         exit(-1);
     }
```

```
    if (len == sizeof(SPX_OPTMGMT)) {
       /*
       **   SPX: Change retry count to 5.
       */
       reqOpts = (SPX_OPTMGMT *)req_opts->opt.buf;
       reqOpts->spxo_retry–count = 5;
    } else {
       /*
       **   SPXII: Change retry count to 5 and window size to 12.
       */
       reqIIOpts = (SPX2_OPTIONS *)req_opts->opt.buf;
       reqIIOpts->spxIIRetryCount = 5;
       reqIIOpts->spxIILocalWindowSize = 12;
    }
    req_opts->flags = T_NEGOTIATE;
    req_opts->opt.len = len;
    if ((t_optmgmt(spxFd, req_opts, ret_opts))<0) {
       fprintf (stderr,
          “t_optmgmt failed to %s failed t_errno= %d errno= %d\n”,
             spxDev, t_errno, errno);
        t_error (“t_optmgmt failed”);
        exit(-1);
    }
    t_free((char *)req_opts, T_OPTMGMT);
    t_free((char *)ret_opts, T_OPTMGMT);
    ..
    ..
}
```

State

The **t_optmgmt** call must be issued with the endpoint in the T_IDLE state. The state does not change on the successful completion of the call.

See Also

t_accept
t_connect
t_listen
t_open

t_rcv

Receives data over an established transport connection.

Synopsis

```
#include “spx_app.h”

int t_rcv (
   int          spxFd,
   char         *buf,
   unsigned int nbytes,
   int          *flags )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *buf*

Passes the address of a buffer that will receive the data.

(IN) *nbytes*

Passes the maximum number of data bytes expected to be received.

(IN) *flags*

Passes the address of an integer that will indicate whether there is more data to receive. SPX/SPXII does not support the T_EXPEDITED flag.

(OUT) *flags*

Receives the flag (T_MORE) that indicates whether there is more data to receive.

Return Values

>=0	Successful. Returns the number of bytes accepted by SPX/SPXII
-1	Unsuccessful

If the **t_rcv** is successful, the value returned is the actual number of data bytes received. Refer to *Network Programming Interfaces* for additional errors that may occur with this call.

If the watchdog determines that the remote transport endpoint is no longer participating in the connection, the watchdog generates a disconnect indication which causes **t_rcv** to return with a T_LOOK error. (See **t_rcvdis** for more information.)

Remarks

The **t_rcv** function works as specified in *Network Programming Interfaces*, except that the T_EXPEDITED flag is never set.

SPX/SPXII does perform flow control. If SPX/SPXII determines that the remote endpoint is sending data faster than the application can receive the data, a packet is sent notifying the remote endpoint to stop sending data. When the application receives all of the data queued for it, SPX/SPXII notifies the remote endpoint to begin sending data again.

The watchdog in the SPXII driver prevents blocking forever for a packet. For example, suppose a local transport endpoint blocks waiting for an incoming packet. While the local transport blocks, the remote transport endpoint goes away before sending the packet. In this event, the local transport endpoint could block forever. However, the watchdog solves this problem by periodically checking all active connections.

State

The **t_rcv** call is allowed only in the T_DATAXFER state. The state does not change on successful completion of the call.

See Also

t_look
t_open
t_rcvdis
t_snd

t_rcvdis

Returns a disconnect indication from the remote transport endpoint.

Synopsis

```
#include “spx_app.h”

int t_rcvdis (
   int              spxFd,
   struct t_discon  *discon )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *discon*

Passes the address of a t_discon structure.

(OUT) *discon*

Receives the information about the disconnect in the t_discon structure.

Return Values

0	Successful
-1	Unsuccessful

If any of the following conditions occur, a disconnect indication is generated and passed to the stream head.

In the following situations, the reason integer of the t_discon structure is set accordingly:

TLI_SPX_CONNECTION_FAILED	The remote transport endpoint fails to acknowledge any transmission. This is generated by the SPX/SPXII watchdog after failing to connect to the remote transport endpoint.
or	The SPXII driver could not reliably deliver the data or connection request. The remote transport endpoint doesn't acknowledge transmissions.
or	SPXII has tried a number of times to allocate memory and has failed.
TLI_SPX_CONNECTION_TERMINATED	No error occurred. An SPX/SPXII terminate connection packet was received from the remote transport endpoint.

Remarks

The **t_rcvdis** function works as specified in *Network Programming Interfaces* with the following additions:

The t_discon structure has the following format:

```
struct t_discon {
  struct netbuf  udata;
  int            reason;
  int            sequence;
};
```

The SPXII driver does not use the *udata* field in the t_discon structure. SPX/SPXII doesn't support the transmission of any user data with a disconnect request.

Upon receiving a disconnect request, SPX/SPXII sets the stream to a T_IDLE state.

Because a transmission error or a disconnect indication can arrive at any moment from the remote transport endpoint, the application needs to be aware of the asynchronicity of the disconnect indication arrival.

Example 1

```
/* The following code has two examples. The first example shows
** how to accept a disconnect indication that you know has arrived.
** The second example shows how to loop while you wait for a
** disconnect indication to arrive.
*/

{
   struct t_discon *discon;
   ..
   ..

    if((discon = (struct t_discon *)t_alloc(spxFd, T_DIS, T_ALL))==NULL) {
        t_error( “t_alloc of T_DIS failed”);
        exit(-1);

   discon->udata.len = 0;
   if(t_rcvdis(spxFd, discon) < 0) {
      t_free((char *)discon, T_DIS);
      t_error( “t_rcvdis failed”);
      exit(-1);
   }

   switch( discon->reason) {
   case TLI_SPX_CONNECTION_TERMINATED:
      fprintf(stderr,”Connection terminated by remote endpoint.\n”);
      ..
      ..
      break;
   case TLI_SPX_CONNECTION_FAILED:
      fprintf(stderr,”Connection failed.\n”);
      ..
      ..
      break;
   }
   t_free((char *)discon, T_DIS);
   t_close(spxFd);
}
```

Example 2

```
/*
** This example shows how to loop while you wait for a disconnect
** indication to arrive.
*/

{
   int             lookVal;
   struct t_discon *discon;
   ..
   ..

   while(lookVal = t_look(spxFd)) {
      if (lookVal < 0) {
         t_error( “t_look failed”);
         exit(-1);
      }
      if (lookVal == 0) {            /* Nothing there. Wait */
         sleep(1);
         continue;
      }
      if (lookVal == T_DISCONNECT) {
         if((rcvdis=(struct t_discon *)t_alloc(spxFd,T_DIS,T_ALL)) == 0) {
            t_error( “t_alloc of T_DIS failed”);
            exit(-1);
         }
         rcvdis->udata.len = 0;
         if(t_rcvdis(spxFd, rcvdis) < 0) {
            t_free((char *)rcvdis, T_DIS);
            t_error( “t_rcvdis failed”);
            exit(-1);
         }
         switch( discon->reason) {

         case TLI_SPX_CONNECTION_TERMINATED:
            fprintf(stderr,“Connection terminated by other endpoint\n”);
            ..
            ..
            break;

         case TLI_SPX_CONNECTION_FAILED:
            fprintf(stderr,“Connection failed.\n”);
            ..
            ..
            break;
         }
```

```
            t_free((char *)rcvdis, T_DIS);
            break;
         }
         if (lookVal == T_ORDREL) {
            fprintf(stderr, “got T_ORDREL\n”);
            if(t_rcvrel(spxFd) < 0) {
               t_error( “t_rcvrel failed”);
               exit(-1);
            }
            break;
         } else {
            fprintf(stderr, “Not T_DISCONNECT or T_ORDREL %d\n”,lookVal);
            ..
            ..                /* Take care of event */
            ..
            continue;
         }
      }
      t_close(spxFd);
   }
```

State

The state after a **t_rcvdis** call is T_IDLE.

See Also

t_connect
t_listen
t_snddis

t_rcvrel

Acknowledges receipt of an orderly release indication.

Synopsis

```
#include “spx_app.h”

int t_rcvrel (
   int spxFd )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

Return Values

0	Successful
-1	Unsuccessful

If **t_rcvrel** returns a error, *t_errno* may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TNOREL	No orderly release indication currently exists.
TLOOK	An asychronous event has occurred on the transport endpoint referenced by *spxFd* and requires immediate attention.
TSYSERR	A system error has occurred during execution of this function; check *errno* for possible further information.
TNOTSUPPORTED	This function is not supported by the underlying transport provider.

Remarks

The **t_rcvrel** function is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user should not attempt to receive more data because such an attempt will block forever.

This call works as specified in *Network Programming Interfaces.*

Because the other endpoint might send a T_DISCONNECT, a user should not loop waiting only for an orderly release indication.

If a loop is desirable, **t_look** should be used to detect T_INREL or T_DISCONNECT.

If a **t_sndrel** call has not been issued by the user, the user can continue to send data over the connection.

If the connection is not an SPXII connection, TNOREL is returned.

The correct procedure to orderly terminate an SPXII connection is for both endpoints to send an orderly release request when there is no more data to send. Then both continue to read data until an orderly release indication is received. At this point both endpoints are finished sending and receiving data, and the connection state is set to T_IDLE.

Example

```
{
   int  spxFd;
   ..
   ..
   /*
   ** Receive data loop
   */
   while ((rcvbytes = t_rcv(spxFd,dataBuf,sizeof(dataBuf),&flags))!= -1 )
{
     fprintf(stderr,“received %d bytes Do something with them\n”,rcvbytes);
   }

   /* t_rcv failed. Find out why */
   if (t_errno == T_LOOK) {
      lookVal = t_look(spxFd);
      switch (lookVal) {
         case T_ORDREL:
```

```
            /*
            ** If Orderly Release, receive it and continue
            */
               if(t_rcvrel(spxFd) < 0) {
                  t_error( “t_rcvrel failed”);
                  exit(-1);
               }
               break;
            case T_DISCONNECT:
            /*
            ** If disconnect, exit now
            */
               if(t_rcvdis(spxFd) < 0) {
                  t_error( “t_rcvrel failed”);
                  exit(-1);
               }
               exit();
            default:
                fprintf(stderr,“t_look return %d in receive loop\n”,lookVal);
               exit();
         }
      } else {
          t_error(“t_rcv failed “);
          exit(-1);
      }
      /*
      ** Send more data; then send Orderly Release request
      */
      ..
      ..
      t_close(spxFd);
  }
```

State

If an orderly release request has been sent, the state will change from T_OUTREL to T_IDLE.

If an orderly release request has *not* been sent, the state will change from T_DATAXFER to T_INREL.

See Also

t_sndrel

t_snd

Sends data over a transport connection.

Syntax

```
#include “spx_app.h”

int t_snd (
   int           spxFd,
   char          *buf,
   unsigned int  nbytes,
   int           flags )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

(IN) *buf*

Passes the address of the buffer that contains the data to be sent.

(IN) *nbytes*

Passes the actual number of bytes to be sent.

(IN) *flags*

Passes a flag (T_MORE) that indicates whether there is more data to send. (SPX/SPXII does not support the T_EXPEDITED flag.)

Return Values

>= 0	Successful. Returns the number of bytes accepted by SPX/SPXII
-1	Unsuccessful

If the remote transport endpoint has gone away or fails to acknowledge the transmitted data, the SPXII driver generates a disconnect indication and changes the state of the local transport endpoint to T_IDLE.

Following the TLI specification, any **t_snd** issued in the T_IDLE state is dropped by the SPXII driver. From the SPX/SPXII user's point of view, this indication is noticed only if that user issues some call *other* than **t_snd**.

The SPX/SPXII user should check the return code in the disconnect indication to make sure it is TLI_SPX_CONNECTION_TERMINATED.

The following errors can occur during the send request. These errors lock the stream and disable the local transport endpoint. Only *errno* is set to the following value (*t_errno* is not affected):

EPROTO	The data request was issued from a state other than T_DATAXFER. This transport endpoint is no longer valid and must be closed. *or* The size of the data request header or data portion of the message received by SPXII was invalid (too small or too large). This transport endpoint is no longer valid and must be closed.

Remarks

The **t_snd** function works as specified in *Network Programming Interfaces.* All data is sent on a first-come, first-served basis.

During the period of the connection, SPX/SPXII uses the most recent round-trip delay multiplied by 1.5 as the timeout for the next retransmission.

The T_MORE flag is supported by SPX/SPXII.

- If the UNIX application sets the T_MORE flag when doing a **t_snd**, the EOF bit is *not* set in the SPX/SPXII packet header.
- If the receiving client is a UNIX machine, the T_MORE flag is set in the **t_rcv** call.

- If the receiving client is a DOS/OS2 application, the EOF bit is *not* set in the SPX/SPXII packet header.

SPXII breaks any large send requests into a series of maximum packets for the given connection.

Example

```
/*
** SpxII will partition large messages into appropriately
** sized packets.
*/

#define TRANS_BUFFER_SIZE 4096
int            flags;
int            bytesRead;
unsigned char  readBuffer[TRANS_BUFFER_SIZE];
char           *someFileString = “someFileName”;
FILE           *fp;

/* Open the file to send */
if ((fp=fopen(someFileString, “r+b”)) == NULL) {
   perror(“open failed “);
   ..
}

/* While there is data in the file, tell the remote endpoint that
** there is still data for this transmission */
flags = T_MORE;
while (!feof(fp)) {
   bytesRead=fread(readBuffer, 1, TRANS_BUFFER_SIZE, fp);
   /*
   ** spxFd2 is the local endpoint that has been bound and
   ** connected to a server */
   if (t_snd(spxFd2, readBuffer, bytesRead, flags)<0) {
      t_error( “t_snd failed “);
      ..
      ..
   }
   ..
   ..
}

   /* Check that we haven’t been cut off by remote end. NOTICE that the
   ** return code is greater than zero if we received a disconnect
   ** indication.
   */
```

```
if (t_rcvdis(spxFd2, &disconnectInfo)>=0) {
   printf( “remote endpoint aborted connection \n”);
   ..
   ..
}
/* Send one byte with the T_MORE flag turned off to indicate EOF */
flags = 0;
if (t_snd(spxFd2, readBuffer, 1, flags)<0) {
   t_error( “t_snd failed “);
   ..
   ..
}
```

State

The **t_snd** call is allowed only in the T_DATAXFER state. If any errors occur, the state moves to T_IDLE.

See Also

t_open

t_rcv

t_snddis

Aborts a connection or rejects a connection request.

Syntax

```
#include “spx_app.h”

int t_snddis (
   int            spxFd,
   struct t_call  *call)
```

Parameters

(IN) *spxFd*

Passes the file descriptor that was returned by **t_open**.

(IN) *call*

Passes the address of a t_call structure.

Return Values

0	Successful
-1	Unsuccessful

Refer to *Network Programming Interfaces* for errors that may occur with this call.

Remarks

The **t_snddis** function sends a connection termination request, which initiates the abortive release of a transport connection. It is used when the application wishes to abort or break a connection. This call can be issued by either transport user, and it may also be used to reject a connect request during the connection establishment phase.

This function works as specified in *Network Programming Interfaces.*

A **t_snddis** call generates an SPX/SPXII terminate connection request and releases all outstanding data messages on both the local and remote transport endpoint.

After an application issues a **t_snddis**, the application can do a **t_unbind** and **t_close** immediately without waiting.

SPX/SPXII doesn't support the function of sending address options or user data along with a disconnect request.

The correct procedure for terminating an SPX/SPXII connection is for both transport endpoints to correlate the moment that the connection is no longer needed, and then use **t_snddis** and **t_rcvdis** to terminate the connection or use orderly release now supported in SPXII.

The code samples illustrate the following:

- Example 1 shows how to terminate a connection that is in a T_DATAXFER state.
- Example 2 shows how to deny a connection request.

Example 1

```
{
   ..
   ..
   /*
   ** If you want to terminate the current connection and the
   ** state is T_DATAXFER, you do not need to send a pointer to
   ** the t_call structure.
   */

   if(t_snddis(spxFd, (struct t_call *)NULL) < 0) {
      t_error( "t_snddis failed ");
      exit(-1);
   }
   ..
   ..
   t_close(spxFd);
}
```

Example 2

```
/*
** If you wish to deny a connection request, you must supply the
** sequence number. The sequence number is in the sequence field of
** the t_call structure. The code below listens for connect request
** the uses the option structure returned by the listen to test if
** the other endpoint is using spxII. The t_listen sets the sequence
** field of the t_call structure, we use the same structure that
** t_listen returned to deny the connect request if the other
** endpoint is not using spxII.
*/

{
   struct t_call  *rcvcall;
   SPX_OPTMGMT    *retOpts;
   ..
   ..
   ..
   ..
   if((rcvcall = (struct t_call *)t_alloc(spxFd, T_CALL,T_ALL)) == NULL) {
      t_error( “t_alloc of T_CALL failed”);
      exit(-1);
   }

listenAgain:

   rcvcall->addr.len = rcvcall->addr.maxlen;
   rcvcall->opt.len =  rcvcall->opt.maxlen;
   rcvcall->udata.len = 0;

/*
** Listen for a connect request
*/
   if ((t_listen(spxFd, rcvcall)) < 0) {
      t_error( “t_listen failed”);
      exit(-1);
   }

/*
** Deny connection request if other endpoint is not spxII
*/
   retOpts = (SPX2_OPTIONS *)rcvcall->opt.buf;
   if (!(retOpts->spxIISessionFlags & SPX_SF_SPX2_SESSION)) {
      rcvcall->addr.len =  0;
      rcvcall->udata.len =  0;
      rcvcall->opt.len = 0;
      if(t_snddis(spxFd, rcvcall) < 0) {
```

```
            t_error( “t_snddis failed “);
            t_free((char *)rcvcall, T_CALL);
            exit(-1);
         }
         ..
         ..
         ..
         ..
         ..
         goto listenAgain;
      }
      t_free((char*)rcvcall, T_CALL);
      ..
      ..
      ..
      ..
      ..
}
```

State

Regardless of the success of the call, the state is T_IDLE.

See Also

t_connect
t_listen
t_rcvdis

t_sndrel

Requests an orderly release of an SPXII connection.

Syntax

```
#include “spx_app.h”

int t_sndrel (
   int spxFd )
```

Parameters

(IN) *spxFd*

Passes the file descriptor of the local transport endpoint.

Return Values

0	Successful
-1	Unsuccessful

If **t_sndrel** returns an error, *t_errno* may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	O_NDELAY or O_NONBLOCK was set, but flow control prevented SPXII from accepting the function at this time.
TSYSERR	A system error has occurred during execution of this function; check *errno* for possible further information.
TNOTSUPPORTED	This function is not supported by the underlying transport provider.

Note If **t_sndrel** is issued from an invalid state or if the connection is not an SPXII connection, SPXII will generate a fatal EPROTO error. However, this error may not occur until a subsequent reference to SPXII.

Remarks

The **t_sndrel** function is used to initiate an orderly release of a transport connection and indicates to the transport provider (SPXII) that the transport user (application) has no more data to send.

Note This function is supported *only* when both endpoints are using SPXII.

This function works as specified in *Network Programming Interfaces.* A **t_sndrel** call sends a orderly release request, indicating to both endpoints that the user sending the orderly release will not send any more data. However, a user may continue to receive data if an orderly release indication has not been received.

Even if the service type T_COTS_ORD is returned on **t_open** or **t_getinfo**, **t_sndrel** may return EPROTO if the connection is not an SPXII connection. The *spxIISessionFlags* returned in the opt structure from the **t_listen** or the **t_connect** function calls can be examined to determine whether the connection is SPXII.

The correct procedure for an orderly termination of an SPXII connection is for both endpoints to send an orderly release request when there is no more data to send. Then both continue to read data until an orderly release indication is received. At this point, both endpoints are finished sending and receiving data, and the connection state is set to T_IDLE.

Example

```
{
   int spxFd;
   ..
   ..

    /* All done sending data. Send orderly release. */
       if(t_sndrel(spxFd) < 0) {
          t_error( “t_sndrel failed” );
          exit(-1);
       }
    /*
```

```
    ** Receive more data and wait for orderly release indication.
    */
    ..
    ..
    t_close(spxFd);
}
```

State

If an orderly release indication has been received, the state will change from T_INREL to T_IDLE after **t_sndrel** is called.

If an orderly release indication has not been received, the state will change from T_DATAXFER to T_OUTREL after **t_sndrel** is called.

See Also

t_rcvrel

chapter 8

SAP Library

Overview

One of the design goals for NetWare transports for the UNIX environment has been to provide an easy to use application programming interface for NetWare service advertising and service queries. The SAP library has been written to accomplish this goal.

The SAP library provides functionality in two modes. The mode used by the library is dependent upon the status of the NetWare SAP daemon (SAPD).

- If the SAP daemon is *not* active, the SAP library is unable to advertise any services to the network.

  In this mode, all service queries are handled by passing the request to the network and processing the responses.

- If the SAP daemon is active, all of the SAP library functions are supported.

  In this mode, the SAP library is an Inter Process Communication (IPC) library designed for processes running on the same platform as the NetWare protocol stack. It provides fast and convenient access to the information that the SAP daemon maintains.

As these functions become part of the Application Programming Interface (API) and Transport products, application developers can use them for the following tasks:

- Obtaining SAP information
- Authorizing the SAP daemon to advertise an application server
- Informing the SAP daemon that the application server is going down

For example, SAP implementations for UNIX processes, such as the Server Advertiser daemon and the Print Server daemon, use these functions.

With the SAP daemon running, none of the SAP library query functions send SAP packets out on the wire; they all obtain their information directly from tables maintained by the SAP daemon in a mapped memory region. Those functions that use mapped memory automatically attach to the mapped memory region at the beginning of each call and detach at the end of the call.

For those processes that desire multiple accesses to the SAP information, **SAPMapMemory** allows the process to attach to the mapped memory region and remain attached to the mapped memory region until a detach function is used.

All functions in the SAP library use this mapped memory region except for the following:

SAPAdvertiseMyServer
SAPNotifyOfChange
Advertise Service
ShutdownSAP

The functions that use the mapped memory region can only read information; they cannot update or write to it. Hence, the functions that require the updating of information (the four functions listed above) do not use the mapped memory region.

Reference for SAP Functions

The following functions are defined in the “sap_app.h” file or in the “sap_dos.h” file:

Table 8-1
Descriptions of SAP Functions

Name	Description
SAPMapMemory	Attaches to SAP daemon mapped memory
SAPUnmapMemory	Detaches from SAP daemon mapped memory

Table 8-1 *continued*
Descriptions of SAP Functions

Name	Description
SAPStatistics	Gets SAP daemon statistics
SAPGetAllServers	Gets all server information
SAPGetNearestServer	Gets information for the nearest server of a specific type
SAPGetChangedServers	Gets changed server information
SAPNotifyOfChange	Registers a callback function to be activated if server information has changed
SAPGetServerByAddr	Gets server information by address
SAPGetServerByName	Gets server information by name
SAPAdvertiseMyServer	Starts (or stops) the advertising of a service of a specific type
SAPListPermanentServers	Gets information about servers that have been advertised with an Advertise Forever flag
SAPGetLanData	Gets LAN statistics for NetWare management
SAPPerror	Prints error message
AdvertiseService	Native NetWare-compatible function; advertises a service of a specific type
ShutdownSAP	Native NetWare-compatible function; discontinues advertising of all services advertised by the calling process
QueryServices	Native NetWare-compatible function; allows General and Nearest Server Queries of specific server types

The functions are documented in the sections below.

SAPMapMemory

Attaches to SAP daemon's mapped memory.

Syntax

```
#include "sap_app.h"

int SAPMapMemory()
```

Parameters

None

Return Values

0	Successful
-7	Unable to find "nwconfig" file; or *nwconfig* parameter not present
-22	Unable to generate mapped memory ID key
-23	Unable to get mapped memory ID
-24	Unable to attach to mapped memory

Remarks

SAPMapMemory causes the process to map to the SAP daemon's mapped memory region in a read-only mode

This function always returns successfully when the SAP daemon is not running.

Example

```
int ret;
ret = SAPMapMemory();
```

See Also

SAPUnmapMemory

SAPUnmapMemory

Detaches from SAP daemon mapped memory.

Syntax

```
#include “sap_app.h”

void SAPUnmapMemory()
```

Parameters

None

Return Values

None

Remarks

SAPUnmapMemory causes the process to unmap from the SAP daemon mapped memory region.

Example

```
SAPUnmapMemory ();
```

See Also

SAPMapMemory

SAPStatistics

Gets SAP daemon statistics.

Syntax

```
#include “sap_app.h”

int SAPStatistics(
    SAPD    *sapstats)
```

Parameters

(OUT) *sapstats*

Pointer to the address of the SAPD structure.

Return Values

0	Successful
-10	Not supported (SAP daemon not running)

Remarks

SAPStatistics returns a structure filled with statistics about the SAP process and the available services. The *sapstats* argument specifies the address of the SAPD structure which receives the statistics.

This function is supported only when the SAP daemon is running.

The SAPD structure contains the following fields:

```
typedef struct sap_data {
   time_t   StartTime;
   pid_t    SapPid;
   uint16   Lans;
   uint8    MyNetworkAddress[IPX_ADDR_SIZE];
   int32    ConfigServers;
```

```
    clock_t  RevisionStamp;
    int32    ServerPoolIdx;
    uint32   ProcessesToNotify;
    uint32   NotificationsSent;
    uint32   TotalInSaps;
    uint32   GSQReceived;
    uint32   GSRReceived;
    uint32   NSQReceived;
    uint32   SASReceived;
    uint32   SNCReceived;
    uint32   GSIReceived;
    uint32   NotNeighbor;
    uint32   EchoMyOutput;
    uint32   BadSizeInSaps;
    uint32   BadSapSource;
    uint32   TotalInRipSaps;
    uint32   BadRipSaps;
    uint32   RipServerDown;
    uint32   TotalOutSaps;
    uint32   NSRSent;
    uint32   GSRSent;
    uint32   GSQSent;
    uint32   SASAckSent;
    uint32   SASNackSent;
    uint32   SNCAckSent;
    uint32   SNCNackSent;
    uint32   GSIAckSent;
    uint32   BadDestOutSaps;
    uint32   SrvAllocFailed;
    uint32   MallocFailed;
} SAPD, *SAPDP;
```

To display the information maintained by the SAPD structure, see the **nwsapinfo** utility in the *Utilities* manual.

Table 8-2 describes the control information fields of the SAPD structure (this is internal information concerning configuration and local requests).

Table 8-2
SAPD Control and Miscellaneous Information Fields

Field	Description
SapPid	Process ID of SAP daemon process
Lans	Number of connected LANs, including internal LAN
MyNetworkAddress	Workstation network address
ConfigServers	Total configured server entries
RevisionStamp	Revision of last update
ServerPoolIdx	Index to next unused server entry
ProcessesToNotify	Number of processes to notify of changes
NotificationsSent	Number of notifications sent to processes
GSIReceived	Number of local SAP Get Mapped Memory ID requests
GSIAckSent	Number of Get Mapped Memory ID ACKs
TotalInRipSaps	Total “RIP network down” packets received
BadRipSaps	Bad “RIP network down” packets received
RipServerDown	Server set to “down” due to RIP interaction

The statistical fields in the SAPD structure contain counts only for over-the-wire requests and responses. Table 8-3 describes each of these statistical fields.

Table 8-3
SAPD Statistical Fields

Statistical Type	Field	Description
	StartTime	Time in seconds since SAPD was started
Packets Received	*TotalInSaps*	Total SAP packets received
	GSQReceived	General Server Query packets received
	GSRReceived	General Server Reply packets received
	NSQReceived	Nearest Server Query packets received
	SASReceived	Number of local servers that have requested SAP to advertise their service (SAS requests)
	SNCReceived	Number of local processes that have requested notification of changes (SNC requests)
	NotNeighbor	Packets received from sources not on LAN. If all SAP agents on the LAN are functioning correctly, this should be zero.
	EchoMyOutput	Broadcast packets sent by SAPD which were echoed back to SAPD by the LAN driver. This should be zero.
	BadSizeInSaps	Packets received which have an incorrect packet size. This should be zero.
	BadSapSource.	Packets received which have a bad source address
Packets Sent	*TotalOutSaps*	Total SAP packets sent
	NSRSent	Nearest Server Response packets sent
	GSRSent	General Service Reply packets sent
	GSQSent	General Service Query packets sent
	SASAckSent	ACKs sent in response to SAS requests (see the SASReceived field)

Table 8-3 *continued*
SAPD Statistical Fields

Statistical Type	Field	Description
	SASNackSent	NAKs (negative acknowledgments) sent in response to SAS requests (see the SASReceived field)
	SNCAckSent	ACKs sent in response to SNC requests (see the SNCRequest field)
	SNCNackSent	NAKs sent in response to SNC requests (see the SNCRequest field)
	BadDestOutSaps	SAP packets sent which had an invalid destination network address
Memory Error	*SrvAllocFailed*	Number of server allocation request failures. If greater than zero, indicates memory problems. The NetWare protocol stack needs to be downed, SAPD reconfigured for a larger shared memory region, and the NetWare protocol stack started again.
	MallocFailed	Number of Malloc request failures. If greater than zero, indicates memory problems. The NetWare protocol stack needs to be downed, SAPD reconfigured for a larger shared memory region, and the NetWare protocol stack started again.

Example

```
SAPD sapstats;
ret = SAPStatistics(&sapstats);
```

SAPGetAllServers

Gets all server information.

Syntax

```
#include “sap_app.h”

int SAPGetAllServers(
    uint16    ServerType,
    int       *ServerEntry,
    SAPI      *ServerBuf,
    int       MaxEntries)
```

Parameters

(IN) *ServerType*

Specifies either a type of server or ALL_SERVER_TYPE to obtain information on all servers. Servers types are defined in the include file and are listed in Table 8-5 on page 262.

(IN / OUT) *ServerEntry*

Pointer to an index value that indicates the position in SAP responses from which the next *MaxEntries* will be returned. Modified on return. Should initially be set to 0.

(OUT) *ServerBuf*

Specifies the address of a buffer of size (sizeof(SAPI) * *MaxEntries*) which will be filled with SAPI entries.

(IN) *MaxEntries*

Specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

Return Values

>=0	Successful
< 0	Unsuccessful

Remarks

SAPGetAllServers fills the provided buffer with one or more SAPI structures. The SAPI structure contains information about the server type requested. All integer values are returned in machine order, including those values in the netInfo_t structure. However, the *serverAddress* field is returned in network order.

The SAPI structure has the following format.

```
typedef struct sap_info {
   uint16    serverType;
   uint8     serverName[SAP_MAX_SERVER_NAME_LENGTH];
   ipxAddr_t serverAddress;
   uint16    serverHops;
   netInfo_t netInfo;
} SAPI, *SAPIP;
```

The SAPI structure is similar to the information obtained from SAP information packets except that it includes an additional structure, netInfo_t, which describes the local network used to access the server. All values in the netInfo_t structure are returned in machine order.

The netInfo_t structure has the following format:

```
typedef struct netInfo {
   uint32   netIDNumber;
   uint16   timeToNet;
   uint8    hopsToNet;
   uint8    netStatus;
   uint32   lanIndex;
} netInfo_t;
```

Table 8-4 describes the fields of the netInfo_t structure.

Table 8-4
netInfo_t Fields

Field	Description
netIDNumber	Network address
timeToNet	Number of ticks to the network (Tick = 1/18 second)
hopsToNet	Intermediate networks
netStatus	Network status (defined in “ripx_app.h”)
lanIndex	Index to the network’s LANs

If your host configuration is set so that the SAP daemon is running, the netInfo_t structure will be filled. If SAPD is not active, **SAPGetAllServers** retrieves its information from the network and the netInfo_t structure is then set to NULL.

The *ServerEntry* argument must be set to zero on the first call and is updated by the **SAPGetAllServers** function. The updated value should be passed on subsequent calls. You should not modify the contents of *ServerEntry* except to set the initial value of zero.

If successful, the function returns the number of SAPI entries placed in *ServerBuf*. The *ServerEntry* argument is set to the index of the next server entry to be read when the next function call is made. All server entries have been returned when the function return value is less than the value of *MaxEntries* or zero.

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
ServerType = FILE_SERVER_TYPE;
ServerEntry = 0;
MaxEntries = 1;

ret = SAPGetAllServers (ServerType, &ServerEntry, &ServerBuf, MaxEntries);
```

SAPGetNearestServer

Gets information for the nearest server of a specific type.

Syntax

```
#include “sap_app.h”

int SAPGetNearestServer(
    uint16  ServerType,
    SAPI    *ServerBuf)
```

Parameters

(IN) *ServerType*

Specifies the type of server requested.

(OUT) *ServerBuf*

Pointer to the address of a SAPI structure to receive the server entry information.

Return Values

1	Successful
0	No servers of the specified type exist.
<0	Unsuccessful

Remarks

SAPGetNearestServer returns the nearest server of the type specified. The *ServerType* argument specifies the type of server requested. The ALL_SERVER_TYPE constant is not a legal value for *ServerType* on this call. The *ServerBuf* argument specifies the address of a SAPI structure to receive the server entry information. Server types are defined in the include file and listed in Table 8-5 on page 262.

If your host configuration is set so that the SAP daemon is running, the netInfo_t structure will be filled.

If SAPD is not active, **SAPGetAllServers** retrieves its information from the network and the netInfo_t structure is then set to NULL.

If successful, the function returns 1 (the number of server entries placed in *ServerBuf*). If no servers of the specified type exist, the function returns 0. If the function fails, it returns a negative number which is the negative of the error code.

Example

```
ServerType = PRINT_SERVER_TYPE;
ret = SAPGetNearestServer (ServerType, &ServerBuf);
```

SAPGetChangedServers

Gets changed server information.

Syntax

```
#include “sap_app.h”

int SAPGetChangedServers(
    uint16   ServerType,
    int      *ServerEntry,
    SAPI     *ServerBuf,
    int      MaxEntries,
    uint32   RevisionStamp,
    uint32   *NewRevisionStamp)
```

Parameters

(IN) *ServerType*

Specifies either the type of Server desired or ALL_SERVER_TYPE to obtain information on all servers. Server types are defined in the include file and listed in Table 8-5 on page 262.

(IN / OUT) *ServerEntry*

Pointer to an index value that indicates position in SAP responses from which the next *MaxEntries* will be returned. Modified on return. Should initially be set to zero (0).

(OUT) *ServerBuf*

Pointer to the address of a buffer of size (sizeof(SAPI) * *MaxEntries*) which will be filled with SAPI entries.

(IN) *MaxEntries*

Specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

(IN) *RevisionStamp*

Address of the *NewRevisionStamp* value returned by the previous series of calls to **SAPGetChangedServers**. See “Remarks” below.

(IN / OUT)*NewRevisionStamp*

Pointer to the value returned to be used in the next series of calls. See “Remarks” below.

Return Values

>=0	Successful
-10	Not supported (SAP daemon not running)

Remarks

SAPGetChangedServers returns information about servers that have changed since the last time the function was called.

This function is supported only when the SAP daemon is running.

Each server entry in the mapped memory region is stamped with a revision or change stamp. Server entries are returned that have a value greater than the value of *RevisionStamp* passed on the function call. Information about servers that are no longer alive (HOPS >= 16) will also be returned if the server *RevisionStamp* value is greater than the function *RevisionStamp* value. Dead servers are never purged from the mapped memory region. When that server is reactivated, the same entry is updated to show the new status.

All server entries are maintained in the mapped memory region by server name and server type.

The *ServerEntry* argument must be set to zero on the first call and is updated by **SAPGetChangedServers** function. The updated value should be passed on subsequent calls. You should not modify the contents of *ServerEntry* except to set the initial value to zero.

The *MaxEntries* argument specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

The *RevisionStamp* argument is the *NewRevisionStamp* value returned by the previous series of calls to **SAPGetChangedServers**. Server entries with a revision stamp greater than the function’s *RevisionStamp* are copied to *ServerBuf*. This includes services that are no longer active (hops = SAP_SHUTDOWN). If the function’s *RevisionStamp* is set to

zero, all server information is returned. This field is not updated until you update its value. This allows you to make a series of calls to retrieve all changed servers.

The *NewRevisionStamp* argument returns a value to be used in the next series of calls. The value of *NewRevisionStamp* should be set to zero (0) before the first function call in a series is made. Its value will be updated by calling **SAPGetChangedServers**. Subsequent function calls will not alter the value. This allows multiple function calls to be used to retrieve all the changed servers. After all changed servers are retrieved, the process can be notified when additional changes are available. At this time, the value of *NewRevisionStamp* returned on the last set of calls is used for *RevisionStamp* and *NewRevisionStamp* is set to zero.

If successful, the function returns the number of SAPI entries returned in *ServerBuf*. The *ServerEntry* argument is set to the index of the next server entry to be read when the next function call is made. All server entries have been returned when the function return value is less than the *MaxEntries* or zero. The *NewTimeStamp* returns the current time stamp value if the field is set to zero; otherwise, its value is not changed.

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
ServerType = FILE_SERVER_TYPE;
ServerEntry = 0;
MaxEntries = 1;
RevisionStamp = 0;
NewRevisionStamp = 0;
ret = SAPGetChangedServers (ServerType, &ServerEntry, &ServerBuf,
    MaxEntries, RevisionStamp, &NewRevisionStamp);
```

See Also

SAPNotifyOfChange

SAPNotifyOfChange

Registers a callback function to be activated if server information changes.

Syntax

```
#include “sap_app.h”

int SAPNotifyOfChange(
    int       Signal,
    void      (*Function)(int),
    uint16    ServerType)
```

Parameters

(IN) *Signal*

Specifies the signal that will be used to notify the process when a change has occurred.

(IN) *(*Function)(int)*

Specifies the callback function that will be invoked when a change has occurred. See “Remarks” below.

(IN) *ServerType*

Set to either a type of server or ALL_SERVER_TYPE. Server types are defined in the include file and listed in Table 8-5 on page 262.

Return Values

0	Successful
- 5	Duplicate registration of callback function
-10	Not supported (SAP daemon not running)

If unsuccessful, the function returns a negative number which is the negative of the error code.

Remarks

SAPNotifyOfChange gives control to the specified function when one or more server entries have changed. This function allows the process to maintain very accurate SAP information and is used by the Server Advertiser.

This function is supported only when the SAP daemon is running.

The *Signal* argument specifies the signal that will be used to notify the process when a change has occurred. **SAPNotifyOfChange** registers the process for the indicated signal.

If *Signal* value is SAP_STOP_NOTIFICATION, notification of changes will be discontinued, and the callback function will be unregistered.

The *Function* argument specifies the callback function that will be invoked when a change has occurred. The specified function will typically use **SAPGetChangedServers** so that the process can obtain information about the changed servers.

If the *Signal* argument is set to SAP_STOP_NOTIFICATION, the *Function* argument is ignored. The callback function needs no knowledge of the actual signal mechanism involved. This mechanism is set up when **SAPNotifyOfChange** is invoked.

The *ServerType* argument is set to either a type of server or to ALL_SERVER_TYPE to obtain information on all servers.

Example

```
void Function (int Sig)
{
    return;
}
:
:
    ServerType = FILE_SERVER_TYPE;
    ret = SAPNotifyOfChange (SIGUSR1, Function, ServerType);
```

See Also

SAPGetChangedServers

SAPGetServerByAddr

Gets server information by address.

Syntax

```
#include “sap_app.h”

int SAPGetServerByAddr(
    ipxAddr_t   *ServerAddr,
    uint16      ServerType,
    int         *ServerEntry,
    SAPI        *ServerBuf,
    int         MaxEntries)
```

Parameters

(IN) *ServerAddr*

Pointer to the address structure of the requested server.

(IN) *ServerType*

Specifies either the type of server desired or ALL_SERVER_TYPE to obtain information on all servers with that name. Server types are defined in the include file and listed in Table 8-5 on page 262.

(IN / OUT) *ServerEntry*

Pointer to an index value that indicates position in SAP responses from which the next *MaxEntries* will be returned. Modified on return. Should initially be set to zero (0).

(OUT) *ServerBuf*

Pointer to the address of a buffer of size (sizeof(SAPI) * *MaxEntries*) which will be filled with SAPI entries.

(IN) *MaxEntries*

Specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

Return Values

>=0	Successful
< 0	Unsuccessful

Remarks

SAPGetServerByAddr requests server entries by address and type. If the specified type is ALL_SERVER_TYPE, then all servers at the specified address are returned. For example, a NetWare file server and its print server can be on the same machine.

If your host configuration is set so that the SAP daemon is running, the netInfo_t structure will be filled. If SAPD is not active, **SAPGetServerByAddr** retrieves its information from the network and the netInfo_t structure is then set to NULL.

The *ServerAddr* argument specifies the net and node address of the requested service. The socket number is not checked. Therefore, all services on a host machine will have the same address.

The *ServerType* argument specifies either the type of server desired or ALL_SERVER_TYPE to obtain information on all servers at that address.

The *ServerEntry* argument must be set to zero (0) on the first call and is updated by **SAPGetServerByAddr**. The updated value should be returned on subsequent calls. You should not modify the contents of *ServerEntry* except to set the initial value to zero.

The *ServerBuf* argument specifies the address of a buffer of size (sizeof(SAPI) * *MaxEntries)* which will be filled with SAPI entries.

The *MaxEntries* argument specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

If successful, the function returns the number of SAPI entries placed in *ServerBuf*. The *ServerEntry* argument is set to the index of the next server entry to be read when the next call is made. All server entries have been returned when the function return value is zero or less than the *MaxEntries*.

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
memset ((char*) &addr, 0, sizeof(ipxAddr_t));
addr.net[0] = 0x01;
addr.net[1] = 0x23;
addr.net[2] = 0x45;
addr.net[3] = 0x67;
addr.node[5] = 0x01;
ServerType = FILE_SERVER_TYPE;
ServerEntry = 0;
MaxEntries = 1;
ret = SAPGetServerByAddr (ServerAddr, ServerType, &ServerEntry,
&ServerBuf,
    MaxEntries);
```

See Also

SAPGetServerByName

SAPGetServerByName

Gets server information by name.

Syntax

```
#include “sap_app.h”

int SAPGetServerByName(
    char     *ServerName,
    uint16   ServerType,
    int      *ServerEntry,
    SAPI     *ServerBuf,
    int      MaxEntries)
```

Parameters

(IN) *ServerNam*e

Pointer to the NULL-terminated name of the requested server.

(IN) *ServerType*

Specifies either the type of server desired or ALL_SERVER_TYPE to obtain information on all servers with that name.

(IN / OUT) *ServerEntry*

Pointer to an index value that indicates position in SAP responses from which the next *MaxEntries* will be returned. Modified on return. Should initially be set to zero (0).

(OUT) *ServerBuf*

Pointer to the address of a buffer of size (sizeof(SAPI) * *MaxEntries*) which will be filled with SAPI entries.

(IN) *MaxEntries*

Specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

Return Values

>=0	Successful
-1	Server Type invalid
-2	Server Name too long / too short
< 0	Unsuccessful

Remarks

SAPGetServerByName requests server entries by name and type. If the specified type is ALL_SERVER_TYPE, then all servers by that name are returned. SAP allows servers to have the same name as long as they are of different types. For example, a NetWare file server and its print server could have the same name.

If your host configuration is set so that the SAP daemon is running, the netInfo_t structure will be filled. If SAPD is not active, **SAPGetAllServers** retrieves its information from the network and the netInfo_t structure is then set to NULL.

Some limited wildcard capabilities are allowed in this function. If the final character of *ServerName* is an "*" character, the string is matched up only to the "*" character.

The *ServerName* argument specifies the NULL-terminated name of the requested service. If the name contained in *ServerName* is not less than SAP_MAX_SERVER_NAME_LENGTH, the service requested will be rejected.

The *ServerType* argument specifies either the type of server desired or ALL_SERVER_TYPE to obtain information on all servers with that name.

The *ServerEntry* argument must be set to zero (0) on the first call and is updated by **SAPGetServerByName**. The updated value should be returned on subsequent calls. You should not modify the contents of *ServerEntry* except to set the initial value to zero.

The *ServerBuf* argument specifies the address of a buffer of size (sizeof(SAPI) * *MaxEntries)* which will be filled with SAPI entries.

The *MaxEntries* argument specifies the maximum number of SAPI entries which can be put in *ServerBuf*.

If successful, the function returns the number of SAPI entries placed in *ServerBuf*. The *ServerEntry* argument is set to index of the next server entry to be read when the next call is made. All server entries have been returned when the function return value is zero or less than the *MaxEntries*.

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
strcpy (ServerName, “TEST_SERVER”);
ServerType = FILE_SERVER_TYPE;
ServerEntry = 0;
MaxEntries = 1;
ret = SAPGetServerByName (ServerName, ServerType, &ServerEntry,
&ServerBuf,
    MaxEntries);
```

See Also

SAPGetServerByAddr

SAPAdvertiseMyServer

Advertises (or stops the advertising of) a service of a specific type on the internetwork.

Syntax

```
#include “sap_app.h”

int SAPAdvertiseMyServer(
    uint16    ServerType,
    char      *ServerName,
    uint16    Socket
    int       Action)
```

Parameters

(IN) *ServerType*

Specifies the type of server assigned by Novell for the server’s service class.

(IN) *ServerName*

Pointer to the NULL-terminated name of the server to be advertised (maximum of 48 characters including NULL).

(IN) *Socket*

Specifies the socket number at which the advertised service may be accessed.

(IN) *Action*

Specifies the type of action the SAP daemon is to perform.

Return Values

0	Successful
- 3	Invalid *Action* flag
- 7	Unable to obtain NetWare configuration file path
-10	Not supported (SAP daemon not running)
-11	Service in use; try again
-40	Unable to allocate local memory
-42	Server to unadvertise not found
-43	No permission to advertise / unadvertise server

Remarks

SAPAdvertiseMyServer causes the named server to be advertised by the SAP daemon. To start the advertising process, this call needs to be made once only.

This function is supported only when the SAP daemon is running.

This function does not use mapped memory, but sends a message via the protocol stack to the SAP daemon. The message is acknowledged by the SAP daemon to ensure that it was received.

Only the root user has permission to advertise using the SAP_ADVERTISE_FOREVER flag, and only root can unadvertise a server advertised with this flag.

When a server is advertised with the SAP_ADVERTISE_FOREVER flag, the server description is written to a file called “sapouts” which resides in the NetWare configuration directory. When NetWare services are started, this file is read and the services are automatically re-advertised. Servers can be removed from the “sapouts” file *only* via a **SAPAdvertiseMyServer** call with the *Action* flag set to SAP_STOP_ADVERTISING.

To obtain a list of servers that are permanently advertised, use the **SAPListPermanentServers** function.

If SAP_ADVERTISE is set in *Action*, the SAP daemon places the PID of the advertising process in the advertise table entry.

If SAP_ADVERTISE_FOREVER is set in *Action*, the SAP daemon places its own PID in the table.

The **kill** (*pid*, 0) system call is used to determine if the process that made the **SAPAdvertiseMyServer** call is still active. As long as the process is active, the services for that process will be advertised. When the process terminates, the services for that process will be marked as HOPS = 16, or down.

The *ServerType* argument specifies the type of server to be advertised. See Table 8-5 following for some common values (contact Novell to be assigned a number for new services).

Table 8-5
Common Server Types

Defined Constants for SAP daemon	Value	Server Type
FILE_SERVER_TYPE	0x0004	NetWare Server
PRINT_SERVER_TYPE	0x0047	Print Server
BTRIEVE_SERVER_TYPE	0x004B	Btrieve Server
ACCESS_SERVER_TYPE	0x0098	NetWare Access Server
OLD_NVT_SERVER_TYPE	0x009E	NVT over NVT protocol
I386_SERVER_TYPE	0x0107	386 NetWare (3.x)
SPX_NVT_SERVER_TYPE	0x0247	NVT2 over SPX/SPXII protocol
TIME_SYNC_SERVER_TYPE	0x026B	Time Synchronization
DIRECTORY_SERVER_TYPE	0x0278	Directory Server

The *ServerName* argument specifies the NULL-terminated name of the server to be advertised. If the name contained in *ServerName* is not less than SAP_MAX_SERVER_NAME_LENGTH, the request to advertise the server will be rejected.

The *Socket* argument is the socket number to which clients may make service requests. If *Socket* is zero (0), however, the effect of the **SAPAdvertiseMyServer** call is a notification to the network without providing real service.

The *Action* argument specifies the type of action the SAP daemon is to perform. Three values are valid:

Table 8-6
Flags for SAP daemon Action

Flag	Description
SAP_ADVERTISE	Advertise my server while my process lives or until it is discontinued
SAP_ADVERTISE_FOREVER	Advertise my server until it is discontinued
SAP_STOP_ADVERTISING	Discontinue advertising my server

If successful, the function returns a zero (0); otherwise, it returns a negative number which is the negative of the error code.

Example

```
strcpy (ServerName, “MY_SERVER”);
Socket = 0;
ServerType = AIX_TYPE;
Action = SAP_ADVERTISE_FOREVER;
ret = SAPAdvertiseMyServer (ServerType, ServerName, Socket, Action);
```

See Also

SAPListPermanentServers

SAPListPermanentServers

Gets a list of servers that are permanently advertised.

Syntax

```
#include “sap_app.h”

int SAPListPermanentServers(
    char      *ServerEntry,
    PersistList_t    *ServerBuf,
    int       MaxEntries)
```

Parameters

(IN / OUT) *ServerEntry*

Pointer to an index value that indicates the position in SAP responses from which the next *MaxEntries* will be returned. Modified on return. Should initially be set to 0.

(OUT) *ServerBuf*

Specifies the address of a buffer of size (sizeof(PersistList_t) **MaxEntries*) which will be filled with PersistList_t entries.

(IN) *MaxEntries*

Specifies the maximum number of PersistList_t entries which can be put in *ServerBuf*.

Return Values

>=0	Successful
- 7	Unable to find / read NetWare configuration file path
-28	Error opening “sapouts” file
-29	Unable to read “sapouts” file

Remarks

SAPListPermanentServers fills the provided buffer with one or more PersistList_t structures. The PersistList_t structure contains information about servers that have been advertised with the SAP_ADVERTISE_FOREVER flag. These servers are listed in a file called “sapouts” in the NetWare configuration directory. This function reads the records stored in that file. All numerical values are returned in machine order.

This function is meaningful only when the SAP daemon is running.

The PersistList_t structure has the following format:

```
typedef struct PersistList{
   uint8   ServerName[SAP_MAX_SERVER_NAME_LENGTH];
   uint16  ServerType;
   uint16  ServerSocket;
} PersistList_t;
```

If successful, **SAPListPermanentServers** returns the number of PersistList_t entries placed in *ServerBuf*. The *ServerEntry* argument is set to the index of the server entry to be read when the next function call is made. All server entries have been returned when the function return value is less than the value of *MaxEntries* or zero (0).

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
ServerEntry = 0;
MaxEntries = 1;

ret = SAPListPermanentServers(&ServerEntry, &ServerBuf, MaxEntries);
```

See Also

SAPAdvertiseMyServer

SAPGetLanData

Gets LAN statistics for NetWare management.

Syntax

```
#include “sap_app.h”

int SAPGetLanData(
    int     lanNumber,
    SAPL    *LanDataBuffer)
```

Parameters

(IN) *lanNumber*

Specifies the network to return information about. See “Remarks” below.

(OUT) *LanDataBuffer*

Pointer to the address of the SAPL structure to fill with data about the LAN.

Return Values

1	Successful.
0	No data available for specified LAN
-10	Not supported (SAP daemon not running)

Remarks

SAPGetLanData returns LAN statistics information.

This function is supported only when the SAP daemon is running.

The *lanNumber* argument specifies the network to return information about. LAN 0 is always the internal network. The other LAN numbers are assigned sequentially starting with 1. You can get the number of LANs from a **SAPStatistics** call.

If no internal LAN exists, the *LanDataBuffer* is set to zeros and a successful (1) code is returned. The *LanDataBuffer* argument is the address of the SAPL structure to fill with data about the LAN.

The SAPL structure has the following format:

```
typedef struct SapLanData
{
   uint16   LanNumber;
   uint16   UpdateInterval;
   uint16   AgeFactor;
   uint16   PacketGap;
   int32    Network;
   int32    LineSpeed;
   uint32   PacketSize;
   uint32   PacketsSent;
   uint32   PacketsReceived;
   uint32   BadPktsReceived;
} SAPL, *SAPLP;
```

Table 8-7 describes the SAPL fields.

Table 8-7
Descriptions of SAPL Fields

Field	Description
LanNumber	LAN number
UpdateInterval	Periodic update interval in seconds
AgeFactor	Number of periodic update intervals to miss before marking the server “down”
PacketGap	Time in milliseconds between packets. Time is zero for a WAN, nonzero for a LAN.
Network	Network number for this LAN
LineSpeed	Linespeed in MBS. If the sign bit is set, KBS. Currently always zero.
PacketSize	Packet size that will be sent on this LAN
PacketsSent	Number of packets sent

Table 8-7 *continued*
Descriptions of SAPL Fields

Field	Description
PacketsReceived	Number of packets received
BadPktsReceived	Number of bad packets received

If successful, the function returns a 1. If the LAN number specified does not exist, the function returns a zero (0).

If an error occurs, it returns a negative number which is the negative of the error code.

Note

To display the information maintained in the SAPL structure, see the **nwsapinfo** utility in the *Utilities* manual.

Example

```
ret = SAPGetLanData (1, &LanDataBuffer);
```

SAPPerror

Prints error message.

Syntax

```
#include “sap_app.h”

int SAPPerror(
    int    saperr,
    char   *text)
```

Parameters

(IN) *saperr*

SAP error number.

(IN) *text*

Pointer to the address of the text to be prepended to the error message.

Return Values

>0	Successful.
-1	An error has occurred.

Remarks

SAPPerror prints an error message for errors returned from the SAP library. The *saperr* argument returns the error code as it is returned from one of the SAP functions. The *text* argument specifies the address of the text to be prepended to the error message returned by **SAPPerror**.

If an error occurs, the function returns a -1.

AdvertiseService

Advertises a service of a specific type on the internetwork.

Syntax

```
#include “sap_dos.h”

int AdvertiseService(
    uint16    ServerType,
    char      *ServerName,
    uint8     *Socket)
```

Parameters

(IN) *ServerType*

Specifies the type of server assigned by Novell for the server’s service class.

(IN) *ServerName*

Pointer to the NULL-terminated name of the server to be advertised (maximum of 48 characters, including NULL).

(IN) *Socket*

Pointer to a 2-byte array that specifies the socket number at which the advertised service may be accessed.

Return Values

0	Successful
- 3	Invalid Action flag
- 7	Unable to obtain NetWare configuration file path
- 8	Invalid socket
-10	Not supported (SAP daemon not running)
-11	Service in use; try again

-40	Unable to allocate local memory
-42	Server to unadvertise not found
-43	No permission to advertise / unadvertise server

Remarks

AdvertiseService is compatible with native NetWare and it causes the named server to be advertised by the SAP daemon. This call needs to be made only once to start the advertising process.

This function is supported only when the SAP daemon is running.

This function does not use mapped memory, but sends a message via the protocol stack to the SAP daemon. The message is acknowledged by the SAP daemon to ensure that it was received.

Only the root user has permission to advertise using the SAP_ADVERTISE_FOREVER flag, and only root can unadvertise a server advertised with this flag.

When a server is advertised with the SAP_ADVERTISE_FOREVER flag, the server description is written to a file called “sapouts” which resides in the NetWare configuration directory. When NetWare services are started, this file is read and the services are automatically re-advertised. Servers can be removed from the “sapouts” file *only* via a **SAPAdvertiseMyServer** call with the *Action* flag set to SAP_STOP_ADVERTISING.

To obtain a list of servers that are permanently advertised, use the **SAPListPermanentServers** function.

If SAP_ADVERTISE is set in *Action*, the SAP daemon places the PID of the advertising process in the advertise table entry.

If SAP_ADVERTISE_FOREVER is set in *Action*, the SAP daemon places its own PID in the table.

The SAP daemon places the PID of the advertising process in the advertise table entry. The **kill** (*pid*, 0) system call is used to determine if the process that made the **AdvertiseService** call is still active.

As long as the process is active, the services for that process will be advertised. When the process terminates, the services for that process will be marked as HOPS = 16, or down.

The *ServerType* argument specifies the type of server to be advertised. For some common values, see Table 8-5 below. Contact Novell to be assigned a number for new services.

Table 8-8
Common Server Types

Defined Constants for SAP daemon	Value	Server Type
FILE_SERVER_TYPE	0x0004	NetWare Server
PRINT_SERVER_TYPE	0x0047	Print Server
BTRIEVE_SERVER_TYPE	0x004B	Btrieve Server
ACCESS_SERVER_TYPE	0x0098	NetWare Access Server
OLD_NVT_SERVER_TYPE	0x009E	NVT over NVT protocol
I386_SERVER_TYPE	0x0107	386 NetWare (3.x)
SPX_NVT_SERVER_TYPE	0x0247	NVT over SPX/SPXII protocol
TIME_SYNC_SERVER_TYPE	0x026B	Time Synchronization
DIRECTORY_SERVER_TYPE	0x0278	Directory Server

The *ServerName* argument specifies the NULL-terminated name of the server to be advertised. If the name contained in *ServerName* is not less than SAP_MAX_SERVER_NAME_LENGTH, the request to advertise the server will be rejected.

The *Socket* argument is the socket number to which clients may make service requests. If *Socket* is zero (0), an invalid socket error will be returned.

The service will be advertised until it is discontinued or until the advertising process terminates.

If successful, the function returns a zero (0); otherwise, it returns a negative number which is the negative of the error code.

Example

```
strcpy (ServerName, “MY_SERVER”);
Socket[0] = 0x40;
Socket[1] = 0x47;
ServerType = PRINT_SERVER_TYPE;
ret = AdvertiseService (ServerType, ServerName, Socket);
```

See Also

ShutdownSAP

SAPAdvertiseMyServer

ShutdownSAP

Discontinues advertising of all services advertised by the calling process.

Syntax

```
#include “sap_dos.h”

int ShutdownSAP(void)
```

Parameters

None

Return Values

0	Successful

Remarks

ShutdownSAP is compatible with native NetWare and it instructs the SAP daemon to discontinue advertising of all services advertised by the calling process, if any. This call always returns a zero (0).

Example

```
ret = ShutdownSAP ();
```

See Also

AdvertiseService

SAPAdvertiseMyServer

QueryServices

Gets all server information.

Syntax

```
#include “sap_dos.h”

int QueryServices(
    uint16        QueryType,
    uint16        ServerType,
    int           ReturnSize,
    SAP_ID_PACKET *ServiceBuffer)
```

Parameters

(IN) *QueryType*

Specifies type of SAP request to broadcast.

(IN) *ServerType*

Specifies either a type of server or ALL_SERVER_TYPE to obtain information on all servers.

(IN) *ReturnSize*

Specifies the size in bytes of the buffer pointed to by *ServiceBuffer.*

(OUT) *ServiceBuffer*

Pointer to the address of a buffer which will be filled with SAP_ID_PACKET entries.

Return Values

-9	Unable to allocate local memory
-3	returnSize < sizeof(SAP_ID_PACKET)
-2	Invalid QueryType

-1	No servers found
>0	Successful
<0	Unsuccessful

Remarks

QueryServices is compatible with native NetWare and it fills the provided buffer with one or more SAP_ID_PACKET structures.

The number of structures returned is calculated by *ReturnSize* / sizeof(SAP_ID_PACKET). If less than one, a -1 value is returned.

The SAP_ID_PACKET structure contains information about the server type requested. All integer values are returned in machine order.

The SAP_ID_PACKET structure has the following format:

```
typedef struct {
   uint16  serverType;
   char    serverName[SAP_MAX_SERVER_NAME_LENGTH];
   uint8   network[IPX_NET_SIZE];
   uint8   node[IPX_NODE_SIZE];
   uint16  socket;
   uint16  hops;
} SAP_ID_PACKET;
```

When the SAP daemon is not running, **QueryServices** retrieves its information from the network.

QueryServices does not provide an index value for retrieving SAP information in small batches. If more replies are received than can be placed in *ServiceBuffer*, the replies are discarded.

If successful, the function returns the number of SAP_ID_PACKET entries placed in *ServiceBuffer*.

If an error occurs, the function returns a negative number which is the negative of the error code.

Example

```
QueryType = SAP_NSQ;
ServerType = FILE_SERVER_TYPE;
ReturnSize = sizeof(SAP_ID_PACKET);

ret = QueryServices (QueryType, ServerType, ReturnSize, &ServiceBuffer);
```

See Also

SAPGetAllServers

SAPGetNearestServer

chapter 9

IPX Direct Interface

Overview

This chapter describes the ioctl commands used by NetWare's IPX driver for the UNIX environment. The ioctls described in this chapter are a subset of TLI/XTI and apply either to the IPX Socket Multiplexer or to the IPX LAN Router.

Sockets direct packets to different processes within a single node. Applications bind sockets and then send and receive data via one of the following programming interfaces:

- **TLI/XTI for IPX**

  The Transport Layer Interface (TLI) is more easily ported to other transports. Programming information specific to IPX is provided in Chapter 6, "TLI/XTI for IPX,"on page 133.

- **IPX Direct Interface**

  The direct interface to IPX bypasses some of the overhead involved in portability issues. Only the latter of the two mechanisms is documented in this chapter.

  - **getmsg/putmsg** allows an application to transfer data to a process on a per packet basis and leaves no residue. Refer to the AIX *Operating System API Reference* for programming information.

  - **ioctl** commands allow an application to send and receive parameters that specify directly to IPX how data is to be transferred to processes. All ioctl commands described in this chapter are issued using the **`STREAMS I_STR`** ioctl.

For information on the IPX protocol, packet structure, and fields, see Chapter 1, "Internetwork Packet Exchange (IPX) Protocol."

IPX Driver in the UNIX Environment

The IPX driver has two parts: an IPX Socket Multiplexer and an IPX LAN Router. A set of ioctl commands applies to each.

IPX Socket Multiplexer

The Socket Multiplexer is the entity in the IPX driver that delivers an IPX packet to the appropriate application. (If a version of IPX is running that has no socket multiplexer, these ioctls will fail.)

A socket is assigned to a single UNIX process. (A process can use multiple sockets, but a socket cannot be shared among multiple processes.)

Table 9-1 lists the ioctls that apply to the Socket Multiplexer.

Table 9-1
ioctls for IPX Socket Multiplexer

ioctl Command	Description
IPX_SET_SOCKET	Binds an open file handle to an IPX socket (must release before next bind)
IPX_BIND_SOCKET	Binds an open file handle to an IPX socket (multiple binds allowed)
IPX_UNBIND_SOCKET	Releases a bound socket from an open file handle

IPX LAN Router

The LAN Router is the entity in the IPX driver that delivers an IPX packet on or between LANs configured in IPX. Table 9-1 lists the ioctl commands that apply to the LAN Router.

Table 9-2
ioctls for IPX LAN Router

ioctl Command	Description
IPX_GET_NET	Returns 4 byte IPX network address for the internal LAN
IPX_GET_NODE_ADDR	Returns 6 byte IPX node address for the internal LAN

Table 9-2
ioctls for IPX LAN Router

ioctl Command	Description
IPX_GET_LAN_INFO	Gets LAN state, mux, dlInfo, and RIP/SAP information
IPX_GET_CONFIGURED_LANS	Gets configured LANs
IPX_STATS	Gets IPX LAN and socket statistics

Reference to IPX ioctls

Developers should have access to the “ipx_app.h” and the “lipmx.app.h” files, which contain the structures and descriptions.

IPX_SET_SOCKET

This command binds an open file handle to an IPX socket. This ioctl allows exactly one socket to be bound for the process. To bind another socket, you must first release the socket by using the **IPX_UNBIND** command. When you use this command, IPX fills in the socket number on all packets you send.

IPX_BIND_SOCKET

This commands binds an open file handle to an IPX socket and allows multiple sockets to be bound to the process. The command is used once for each socket to be bound.

IPX_UNBIND_SOCKET

This command releases a bound socket from an open file handle. This ioctl can be used multiple times to bind / release different sockets to the same stream.

When you use this command, you must fill in the socket number on all packets you send. If the non-zero socket number you fill in is invalid, the packet is discarded.

Note If you send zero as a socket number, an M_ERROR is sent to the stream head with a status of EINVAL.

This command uses IpxSetSocket_t structure to specify the number of the socket to unbind.

```
typedef struct {
  uint16 socketNum;
} IpxSetSocket_t;
```

IPX_GET_NET

This command returns the 4-byte IPX network address for the internal LAN. The IPX Network Address structure returns with the *IPX Network* field filled in. The address is in network order (hi-lo).

```
typedef struct {
  IpxNet_t myNetAddress;
} IpxNetAddr_t;
```

IPX_GET_NODE_ADDR

This command returns the 6- byte IPX node address for the internal LAN. The IPX Node Address structure returns with the *IPX Node* field filled in. The address is in network order (hi-lo).

```
typedef struct {
  IpxNode_t myNodeAddress;
} IpxNodeAddr_t;
```

IPX_GET_LAN_INFO

This command returns the lanInfo_t structure which includes information about the specific LAN, such as RIP and SAP information, datalink information, and state. This structure and all elements of the structure are contained in the “lipmx_app.h” file.

```
typedef struct lanInfo {
   uint32   lan;   /* Fill in for GET */
   uint32   state; /* This value returned by GET */
   uint32   streamError;/* This value returned by
     /* GET */
```

```
   uint32    network; / * Fill in for SET, returned by
     /* GET mach order */
   uint32    muxId;  / * Fill in for SET, returned by
     /* GET */
   uint8     nodeAddress[6];  /* Ignored by SET,
     /* dlInfo value used, returned by GET */
   dlInfo_t  dlInfo;   /* DataLink Layer info,
     /* supplied by the user, returned by GET */
   ripSapInfo_t ripSapInfo; /* RIP and SAP lan info,
     /* Fill in for SET, returned by GET */
} lanInfo_t;
```

IPX_GET_CONFIGURED_LANS

This command returns the IpxConfiguredLans_t structure with fields filled.

The IpxConfiguredLans_t structure is defined in “lipmx.app.h” and has the following format:

```
typedef struct {
   uint32 lans; /*Current number of configured LANs*/
   uint16 maxHops;
} IpxConfiguredLans_t;
```

IPX_STATS

This command gets IPX statistics from two structures: IpxLanStats _t and IpxSocketStats_t. Developers must provide a data area large enough to hold both structures.

The **ipxinfo** utility calls **IPX_STATS** to retrieve these structures and print them out in human readable form. If the IPX Socket Multiplexer is not present, all values in IpxSocketStats_t will be zero (0).

The IpxLanStats _t and IpxSocketStats_t structures contain information about both LANs and sockets. The structures describing socket statistics are in the “ipx_app.h” file. The structures describing LAN statistics are in the “lipmx_app.h” file.

On return, the IpxLanStats_t structure precedes the IPXSocketStats_t structure.

chapter **10** ***SPX/SPXII ioctls***

Overview

This chapter describes the ioctl commands used by NetWare's SPXII driver in a UNIX environment. These ioctls are a subset of TLI / XTI and allow an application to retrieve information or statistics about SPX / SPXII or a specific SPX / SPXII connection.

Although ioctls are not required when using TLI / XTI, they can be used to set up transfer data or tear down a connection with another endpoint. Some ioctls described in this chapter are provided for compatibility with older versions of SPX; others for setting or retrieving addition information or statistics. All ioctl commands described in this chapter are issued using the **`STREAMS I_STR`** ioctl.

SPX / SPXII is NetWare's connection-oriented, reliable transport protocol. For information on the enhanced SPXII protocol, packet structure and fields, and data flow, windowing and packet size negotiation, see Chapter 4, "Enhanced Sequenced Packet Exchange (SPXII) Protocol,"on page 41.

Information on the SPX protocol, packet structure and fields, and data flow is provided for purposes of comparison in Chapter 3, "Sequenced Packet Exchange (SPX) Protocol,"on page 27.

For information not contained in this chapter, refer to *Operating System API Reference* or to *STREAMS Modules and Drivers*.

Table 10-1 lists the ioctls that apply to SPX/SPXII.

Table 10-1
SPX/SPXII ioctls

ioctl Commands	Description
SPX_GS_MAX_PACKET_SIZE	Returns the maximum data packet size that can be sent to the wire
SPX_GS_DATASTREAM_TYPE	Sends a value in the Datastream byte in the header so that sender's packets are passed to the application with SPX/SPXII header preceding data
SPX_T_SYNCDATA_IOCTL	Allows the sender to get an ACK/NAK for data sent as part of the ioctl
SPX_CHECK_QUEUE	Allows an application to find out if all data has been sent and acknowledged
SPX_GET_STATS	Allows an application to get the current SPXII driver statistics
SPX_SPX2_OPTIONS	Notifies the SPXII driver to use SPX2 rather than SPX options
SPX_GET_CON_STATS	Returns the current statistics for an SPX/SPXII connection

Reference to SPX ioctls

Developers should have access to the "spx_app.h" file, which contains the structures and descriptions.

SPX_GS_MAX_PACKET_SIZE

This command allows the sender to determine the maximum DATA packet size (MAX PACKET - SPXII header) that can be sent over the connected interface.

Pass buffer to hold the return value (`uint32`).

This command returns the maximum data size that can be sent to the wire.

SPX_GS_DATASTREAM_TYPE

Sending Side. This command allows the sender to send a value in the Datastream Type byte in the SPX/SPXII header. After this ioctl, SPX/SPXII will take the first byte of data from the message (**t_snd**) and put it in the Datastream Type field of the header.

If SPXII needs to break up the message into smaller packets, the same Datastream byte is on fragments of the message.

Receiving Side. After this ioctl, any packets received from the sending endpoint will be passed up to the application with the SPX or SPXII header in front of the data.

The receiving application is responsible for knowing the size of the header: 42 bytes of SPX header or 44 bytes for SPXII header. The header size can be determined at connection time. The **t_connect** and **t_listen** calls can inform the application through the SPX2_OPTIONS structure if the connection is SPX (42-byte header) or if the connection is SPXII (44-byte header).

This command requires no data.

SPX_T_SYNCDATA_IOCTL

This command allows the sender to get an ACK/NAK for data sent as part of the ioctl.

Pass data to send on the wire.

This command returns ioctl ACK after data is sent and acknowledged. If data cannot be delivered, it returns ioctl NAK.

SPX_CHECK_QUEUE

This command allows an application to find out if all data has been sent and acknowledged. It can be used to determine when all data has been sent and acknowledged rather than using the Orderly Release mechanism.

Pass buffer to hold the return value (`uint32`).

This command returns one of the following:

- Size of last unacknowledged packet sent
- One (1) if there are more packets to send
- Zero (0) if there are no more packets to send (this means that all data has been sent and acknowledged)

SPX_GET_STATS

This command allows an application to get the current SPXII driver statistics.

Pass spxStats_t structure uninitialized.

This command returns spxStats_t with current SPXII driver statistics.

SPX_SPX2_OPTIONS

This command is used to notify the SPXII driver to use SPX2_OPTIONS instead of SPX_OPTS. It is needed only if "/dev/nspx" was used on **t_open**.

If "/dev/nspx2" was used on **t_open**, the SPXII driver uses SPX2 options by default.

This command requires no data.

SPX_GET_CON_STATS

This command allows an application to get the current statistics of an SPX/SPXII connection.

Pass spxConStats_t structure initialized with the requested connection number (`uint16`) in the first field.

This command returns spxConStats_t structure with current values for the requested connection.

chapter 11

NCP Extensions

What Are NCP Extensions?

A set of functions called the NetWare Core Protocol provides the procedures that a server's NetWare operating system follows to accept and respond to workstation requests. Collectively, these routines provide the fundamental NetWare services. Each routine is numbered and referred to as an NCP.

"Extended" NCPs (NCP Extensions) provide a mechanism by which an application, rather than the NetWare server, can respond to NCP Extensions coming in to the server.

NCP Extensions are now available for Novell Network Services 4.1 for AIX (NNS). When the NWS server is running on AIX, the NetWare clients that are already authenticated to a Directory tree are provided with a transport mechanism to access applications running on the AIX application server.

The NCPX Handler library for NNS allows AIX programmers to write an AIX application and register the services of this application as NCP Extensions. This extends the services provided by the AIX OS and at the same time maintains the advantages associated with NCPs.

Because Extended NCPs provide a simple authenticated method of communication between NetWare client and the NNS server that is similar to remote procedure calls, they can be used in place of other commonly-used NetWare transports such as IPX/SPX or TIRPC (Transport-Independent Remote Procedure Calls).

The API functions in the NCPX Handler library are identical to those provided for the native NetWare SDK, but the execution environment for server-side programs is different.

There are two sides to NCP Extensions:

- Server side

  The service-providing side of the distributed application can run as

  - An AIX application with an NWS server
  - An NLM application on a native NetWare server

  The NCPX application registers its services as an NCP Extension. The server-side NCPX program (also referred to as an NCPX Handler) provides the hooks into the native NetWare or AIX services.

  The application must be loaded on each server that provides the NCP Extension. An NCPX program that is loaded on one server cannot register NCP Extensions on a remote server.

- Client side

  The client side uses the NCPX Handler's callback service by calling the registered NCP Extension. The client can be one of the following:

  - An NLM running on a native NetWare server and which is acting as a client
  - An application running on a DOS or Windows workstation
  - Another AIX program acting as a client

  The client can also obtain information about an NCP Extension by invoking NCP 36 to scan and retrieve meta-information via the query data buffer.

Novell has published APIs for the NetWare server, clients, and AIX. For more information on DOS, Windows, OS/2, and UNIX as NetWare clients, see *NetWare Library Reference for C: Client Functions* in the NetWare 4 Client SDK, which is provided with the online AIX SDK documentation. For information on NLMS, see *Using Novell Network Services 4.1 for AIX for NLM Applications* in the NetWare 4 Server SDK. The AIX SDK and the NetWare 4 SDKs can be located at Novell's web site: http://www.novell.com.

This chapter covers the following topics:

- How NCP Extensions work
- Components of an NCPX program
- Registering and calling an NCP Extension
- NCPX on a AIX execution environment
- Writing an NCPX Handler program
- Programming issues
- NCPX Handler library reference

Potential Uses

Two key features of Extended NCPs suggest potential uses:

- NCP Extensions allow NetWare clients to access services that an AIX application provides.
- NCP Extensions use the existing connection of the client.

Client-Server Applications

NCP Extensions work well for client-server applications, since they allow the service-providing program, 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 NCPX server program to search the database for a certain record. The function registered as the NCP callback would interpret the request, process the search, and return the related information to the client.

In contrast, a client application downloads the database file and the client workstation performs the search.

IPX/SPX Alternative

NCP Extensions can simplify communication. By using the client's existing authenticated connection to the NWS server, developers are freed from the necessity of setting up communication sockets. In some cases, developers can use NCP Extensions in situations where they are currently using IPX or SPX.

The disadvantage is that NCP Extensions take up a connection. If your application doesn't establish a NetWare NCP connection, and you don't want to establish one, use IPX and SPX instead.

Another disadvantage to NCP Extensions is that communication must always be initiated by the client. (With IPX and SPX, either the client or the server can initiate communication.)

Advantages

To summarize, the advantages of NCP Extensions are as follows:

- They allow NetWare clients to access virtually any AIX service.
- They use an existing connection with a NetWare server.

  This eliminates the need to establish a separate communications session with the server. With this existing connection comes authentication to the Directory tree, packet signing, checksumming, and all other features that pertain to the NCP connection to the server.
- They allow use of arbitrary message sizes.

How NCP Extensions Work

When an application uses NCP Extensions, the following events occur in a client-server paradigm:

- The NCPX client sends an NCP request to the NWS server.
- The NWS server delivers the request to the NCPX Handler program for processing.

- The NCPX Handler program delivers the reply message to the NWS server.
- The NWS server delivers the reply message to the client.

Components of an NCPX Program

An NCPX program runs on an NWS server (as an AIX process) and is designed to handle processing of application-specific NCPs. This program links with the NCPX Handler library to import functions used to register the callback routines.

An NCPX program consists of the following components or functions:

- Query data buffer
- NCP callback
- Reply buffer manager callback (optional)
- Connection event callback (optional)

The callbacks are functions in the developer-written program, designed to process events related to NCP processing. The NCP callback is required for calling an NCP Extension.

Query Data Buffer

The query data buffer is a 32-byte buffer that is allocated by the NCPX Handler library when an Extension is registered. A pointer to the buffer is returned to the NCPX application, and the NCPX application can store information in this buffer.

The query data buffer can be also be used as a “passive, one way” information channel from the server-side NCPX program to the NCPX client program. When clients query the NWS server for information on registered NCP Extensions (invoke NCP 36), the NWS server requires only the query data buffer to return the information.

The query data buffer becomes the sole communication mechanism when an NCP callback is not registered.

NCP Callback

The NCP callback is a routine that runs on the NWS server.When this callback is registered with the NCP Extension, the NCPX Handler library calls this routine whenever the client calls a request function (invoking NCP 37). The NCP callback interprets the message sent by the client, processes the request, and returns information which the NCPX library then returns to the client.

Reply Buffer Manager Callback

The reply buffer manager callback is a routine that determines what to do with the reply buffer after the information in the buffer has been sent to the client. If this callback is registered with the NCP Extension, the NCPX Handler library calls this routine after it has copied the information in the buffer to NWS shared memory.

The reply buffer manager can free the reply buffer, or it can return it to a free list of buffers; the implementation is determined by the NCP callback and the reply buffer manager.

Connection Event Callback

The connection event callback is currently called only when a connection is freed or logged out. If this callback is registered with the NCP Extension, the NCPX Handler library calls this routine to determine when a connection has been freed or logged out.

The connection event callback can use this information to determine if the connection belongs to a client that is being serviced by the NCP callback, and if so, what action to take to clean up that connection's state.

The parameters to this routine are

- Connection (on which the event is happening)
- Event type

Callback Combinations

Not all combinations of NCPX callbacks are useful. The following table lists all combinations of callback components and the level of NCPX service provided in each case.

Case	NCP Callback	Reply Buffer Manager Callback	Connection Event Callback	Description
1	No	No	No	Clients can retrieve query data buffer with NCP 36. The NCPX program places data in the query data buffer.
2	Yes	No	No	Clients can send NCP 37 requests which are then dispatched to NCP callback.
3	Yes	Yes	No	NCP callback allocates buffers, and the reply buffer manager reclaims them.
4	No	Yes	No	Nonsense. (Because there is no NCP callback, the reply buffer manager will never be called.)
5	No	No	Yes	Not very useful. (Allows the NCPX program to monitor logout and connection “frees.”)
6	Yes	No	Yes	Clients can send NCP 37 requests which are then dispatched to NCP callback. Can use the connection event callback.
7	Yes	Yes	Yes	NCP callback allocates buffers, and the reply buffer manager reclaims them.Can use the connection event callback.
8	No	Yes	Yes	Nonsense. (Because there is no NCP callback, the reply buffer manager will never be called.)

“Programming Issues” on page 314 provides further information on the concerns you need to address in determining which Handler components to use in your NCPX application.

Identifying NCP Extensions

NCPX applications must register their services with the NWS server. They identify themselves to the server with names and IDs.

NCP Extension Names

Every NCP Extension must have an identifying name. The following rules apply for naming the NCP Extensions:

- The name is case-sensitive.
- The name can be any text character string up to 32 bytes long, not counting the NULL terminator.
- The name must be unique.

To guarantee uniqueness, you should register your NCP Extension's name through Novell's Developer Support.

Problems can occur if two service-providing NCPX applications use the same name for their NCP Extensions. The clients accessing the Extensions would face ambiguity; they would not know whether the Extension they see registered is the one they want.

Note also that the NCPX Handler library refuses to register an Extension with a name which matches that of an already-registered Extension.

NCP Extension IDs

IDs are also required to identify NCP Extensions. The following rules apply to NCP Extension IDs:

- They are unique.
- They can be dynamically assigned by the server when a Handler registers NCP Extensions (using the name of the NCP Extension).

  These dynamic IDs are determined by the NWS server on a first-come, first-served basis. If an NCP Extension that is using dynamic IDs is deregistered and then registered again, it has a different ID.

NCP Extension IDs increase monotonically. 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.

Novell's Developer Support does *not* assign IDs which are dynamically assigned by the server when the NCP is registered. These IDs are not attached to a specific NCP Extension.

- They can be assigned IDs that NCPX applications use to identify NCP Extensions when they register the Extensions by ID.

  Because these well-known IDs are the same each time an NCP Extension is registered, they can be used to identify a specific NCP Extension.

IDs are assigned by Novell's Developer Support to guarantee that the ID is unique and that the ID is within the valid range.

Problems can occur if two service-providing NCPX applications use the same ID for their NCP Extensions. The clients accessing the Extensions would face ambiguity; they would not know whether the Extension they see registered is the one they want.

Note also that the NCPX Handler library refuses to register an Extension with an ID which matches that of an already registered Extension.

Registering an NCP Extension

Before clients can use the services of a server-side NCPX program, the program must first register an NCP Extension with the NWS server via the NCPX Handler library.

The following occurs:

1. The NCPX application calls a registration function (provided by the NCPX Handler library). Three of the parameters may be functions that can be called as part of the service.

   The parameters to the call include

   - NCP Extension name

- NCP Extension ID (optional)
- Pointer to the NCP callback (or NULL)
- Pointer to the reply buffer manager callback (or NULL)
- Pointer to the connection event callback (or NULL)
- Pointer to the query data pointer

2. The NCPX Handler library validates the parameters. If all parameters are valid, the NCPX Handler library creates a new NCP Extension and allocates the query data buffer.

3. A pointer to the query data buffer is returned as the NCP Extension "handle" (to be used to deregister the NCP Extension).

Multiple NCP Extensions can be registered. After an NCP Extension is registered with the NWS server, 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.

Calling the NCP Extension

For a description of the client functions referenced in the following scenario, see *NetWare Library Reference for C: Client Functions* in the AIX SDK, included with the AIX online documentation. They are also described in *Using Novell Network Services 4.1 for AIX for NLM Applications* in the NetWare 4 Server SDK, located at Novell's web site: http://www.novell.com.

When the client uses the service of an NCP Extension, the following occurs:

1. The client sends an NCP Extension request (invoking NCP 37).
   - If the client sends an NCP Extension request with **NWSendNCPExtensionRequest**(), the client's 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 (up to four.)

Either way, the data is sent across the wire as a stream of bytes.

2. The NWS server via the NCPX Handler library creates the needed request and reply buffers.

   If the request buffer is large, it is sent in fragments to the NWS server. The NWS server reassembles the fragments, making the fragmentation transparent to NCPX application.

3. The NCPX Handler library calls the NCP callback function.

   The NCPX Handler library passes pointers to the request and reply buffers. The NCP callback reads the request buffer, performs processing, and fills in the reply buffer. (If a reply buffer manager is being used, the NCP callback routine allocates space for the reply message and passes a pointer to the allocated buffer back to the NCPX Handler library.)

   The reply buffer manager callback can be thought of as a second part of the NCPX Handler. The reply buffer manager can free buffers and reset counters and semaphores that the NCP callback has set. (For example, if the NCP callback has set a semaphore for a buffer, the reply buffer manager can signal or free the semaphore.)

4. The NCPX Handler library sends the reply information to the client.

   If the reply data is large, the NCPX Handler library sends it across the wire in fragments. The client's **NWSendNCPExtensionRequest**() or **NWSendNCPExtensionFraggedReques**t() reassembles the packet, making the fragmentation transparent to the client program.

The data in the reply buffer is sent to the client only if the NCP callback returns SUCCESSFUL.

5. The NCPX Handler library frees the buffers it has created.

   When the NCP Extension request is completed, the NCPX Handler library frees the buffers it has allocated for the request and reply data. When new requests come in, the NCPX Handler library allocates new buffers.

Client's View of an NCP Extension

The view from the client is different from that of the server-side NCPX application. The client does not need to know the details of how the NCP Extension works. The client needs to know only the protocol for sending requests and receiving replies.

For a description of the functions referenced in the following scenario, see *NetWare Library Reference for C: Client Functions* in the online AIX SDK (also included in the NetWare 4 Client SDK), or see *Using Novell Network Services 4.1 for AIX for NLM Applications* in the NetWare 4 Server SDK. The NetWare 4 SDK manuals are also located at Novell's web site: http://www.novell.com.

The client accesses the services of an NCP Extension in the following ways:

1. The client checks to see if the NCP Extension has been registered (invoking NCP 36).

   A client cannot use an NCP Extension until it has been registered. The client can use **NWGetNCPExtensionInfo**() or **NWScanNCPExtensions**() to see if the NCP Extension has been registered. This also returns the NCP Extension's ID.

2. The client sends a request to the NCP Extension with either **NWSendNCPExtensionRequest**() or **NWSendNCPExtensionFraggedRequest**() and uses the information that was returned (invoking NCP 37).

3. The client 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.

Service Provider's View of an NCP Extension

The server-side NCPX application does not need to know what the client process looks like; it needs to know only the format of the request coming from the client and how to format the reply:

1. The NCPX application registers the NCP Extension with the NCP callback and optionally with the reply buffer manager callback and the connection event callback. The query data buffer is returned from the NCPX Handler library. (Multiple NCP Extensions can be registered.)

2. When the NCP callback function is called, it finds the request in a buffer that the NWS server has allocated via the NCPX Handler library. The NCP callback processes the request and places the reply in another buffer(s) that the NWS server returns to the clients.

3. If a reply buffer manager is used, the reply buffer manager callback routine is called after the data in the reply buffer(s) has been copied to NWS shared memory. When the NWS server calls the reply buffer manager, the reply buffer manager determines what to do with the buffer(s) where the reply is stored.

4. If the connection event callback routine is called, it determines whether the connection event affects the NCPX server program, and takes appropriate action.

5. The NCPX application updates the information in the query data buffer.

6. The NCPX application deregisters the NCP Extension when it no longer wants to provide the service or when the service-providing NCP callback process is unloaded.

NCPX in a AIX Execution Environment

The NCPX Handler library for NNS provides the same APIs as those it provides for NLMs for the native NetWare SDK. The differences for the server-side programs in the AIX execution environment are described in this section.

Functions in the server-side NCPX Handler library are documented in “NCPX Handler Library Reference” beginning on page 319.

Process Model

NCPX applications are tightly coupled with the NNS running on AIX. These Handler programs first invoke registration functions to “attach” themselves as service routines for Extended NCPs coming in to the NWS server. Then they call a dispatch loop (EventLoop) to begin processing incoming Extended NCPs. Multiple NCP Extensions can be registered can be registered before calling the EventLoop (as long as each registered NCP Extension is deregistered before program

termination).When Extended NCPs arrive at the NWS server, the dispatch loop awakens and invokes the registered callback routines in the NCPX Handler code to service the request. The EventLoop code will properly dispatch the incoming Extended NCPs to the appropriate callback functions.

Like the NCPX NLMs running on native NetWare, NCPX applications on AIX are tightly coupled to the server; certain kinds of faults in the application programs could cause the rest of the NWS server to fail.

Unlike NCPX NLMs running on Native NetWare, NCPX applications on AIX run as a single thread. More specifically, the functions in the NCPX Handler library provided for the NWS server on AIX are not thread-safe. Although, technically speaking, you can have multiple threads running in your NCPX application, only one thread at a time can be allowed to call the NCPX functions.

This limitation also affects the EventLoop function: If a thread is “in” the EventLoop, only that thread is allowed to call NCPX functions (during callback processing). If other threads call NCPX functions, a deadlock will be the likely result.

Although you can have other threads running “background” tasks in the NCPX application, remember that only one thread at a time can call into the NCPX Handler library routines.

Handler Parents and Children

The NCPX application is composed of two processes:

- Parent
- Child

The only duty of the parent process is to wait for the child to die and then clean up data structures which were *not* cleaned up by the child program (for example, in the case of a core dump).

When the child exits, the parent receives notification of the child’s termination (returns from a **wait**() system call). The parent can then perform housecleaning steps required to free any leftover resources which were in use by the child when it died.

This mechanism protects the overall NWS server from errant NCPX application programs. If a child program malfunctions and exits prematurely (for example, dumps core), the parent is there to make sure the child's "mess" in the NWS server's internal state is cleaned up.

The child process is forked by the parent. The child runs the developer-written **HandlerMain**(), which contains the application-specific NCPX code.

All management of handler parents and children is performed by the NCPX Handler library routines; application developers do not need to write any code to manage parents and children.

System administrators need to be aware of this "dual process" nature of NCPX programs so that they don't inadvertently try to kill those "extra" (parent) NCPX processes and then wonder why the "real" (child) NCPX Handler is also exiting.

EventLoop

NCPX applications are event-driven. Once initialization is complete, the program calls **NCPX_EventLoop**() to service all incoming requests. **NCPX_EventLoop**() is the "workhorse" of an NCPX application. The EventLoop receives "work" information from the NWS server via an event queue stored in the server's shared-memory segment. It is the EventLoop's job to dispatch the "work" to the appropriate Handler callback function(s) and return the results to the NWS server (which then returns the results to the client which originated the request).

The NCPX application should not delay between the NCP Extension registration step and the invocation of the EventLoop. This is because Extended NCP requests begin queuing for services as soon as the Extension is registered. The EventLoop services these queued requests. If an NCPX application program registers an Extension and does *not* call the EventLoop, requests will queue up and the clients will "hang" waiting for service from the server.

The EventLoop returns to the server-side application in the following circumstances:

- An internal error occurs in the EventLoop code.
- All NCP Extensions are deregistered (during callback processing).
- The NWS server goes down.

- The system administrator sends the NCP application a signal to shut down (the **`kill`** command without parameters).

Signals

Although NCPX applications receive several signals, they don't need to respond to them because they are all intercepted by the NCPX Handler library routines. The NCPX Handler library contains code to respond properly to the various signals required by the server architecture.

Server-side application code should *not* manipulate these signals! Tampering with the NCPX Handler library's signal configuration will likely cause a malfunction of the server.

The following table contains the signals that are processed by the NCPX Handler library:

SIGHUP	Causes the Handler program to exit immediately. This signal is delivered to the NCPX application by the NetWare server when the server requires the handler to exit without cleaning up (such as when the server faults internally and wants to dump a core image without having the NCPX applications change the shared-memory contents).
SIGQUIT	Causes the Handler program to dump a core image. This signal is delivered to the NCPX application when the NWS server thinks the handler has faulted and should generate a core dump.
SIGTERM	Causes the EventLoop to return. The NCPX application should deregister extensions and exit. The NWS server delivers this signal to the NCPX application when the NWS server is being shut down.

SIGINT	Causes the EventLoop to return. The NCPX application should deregister Extensions and exit. System administrators send this signal when they want the NWS server to remain “up” while the Extended NCP service provided by the NCPX application is discontinued. You should tell system administrators that they can terminate NCPX Handlers from the AIX command line with the AIX **`kill`** command (default, without parameters).

Never send a SIGKILL signal to an NCPX Handler process (**`kill -9`**). Doing so causes the handler to die an untimely death with no cleanup, which will leave server internal data structures in an inconsistent state. If a Handler process is prematurely terminated (unclean), the undesired side effect is likely to be the malfunctioning of the server.

NCPX Handler processes are a part of the running NetWare server, and they should be treated with the same care and respect afforded all other server daemons and processes.

Privileges

The NWS server requires a certain set of AIX process privileges to run. Since NCPX Handler processes are a part of the NWS server, they also must run with privileges. The NCPX program won’t run if the required privileges are missing.

NCPX programs require the following privileges:

- P_OWNER
- P_DACREAD
- P_DACWRITE
- P_FILESYS
- P_MOUNT
- P_MULTIDIR
- P_SETUID
- P_SYSOPS
- P_DRIVER
- P_RTIME
- P_FSYSRANGE
- P_TSHAR
- P_CORE

The NCPX Handler library initialization code checks to ensure that the process is running with the correct privileges. If any of the requisite privileges are missing, the initialization code will print an error message and exit the program.

System administrators who are running NCPX Handlers with their NWS servers must grant privileges to the NCPX application. This can be done in either one of the two "AIX-defined" ways of granting privileges:

- Executable fixed privileges assigned with the **filepriv** command.
- Privileges derived from entries in the Trusted Facilities Manager (TFM) database.

Either method works. The most direct way to grant the required privileges is to use the **filepriv** command. However, this method has the drawbacks associated with this "setuid style" of assigning privileges. That is, any user with execute permissions can execute the program, which will then run with the granted privileges.

In addition, using the **filepriv** command is less flexible than the TFM method, which is more secure and more flexible. The TFM method allows the administrator to restrict privileges to certain users and/or roles defined in the TFM database. Using this method means that privileges can be restricted on a per-user or a per-role basis.

Shared Memory

The NWS server maintains a shared-memory segment which is used to keep most server state information. For NCPX programs to become a "part of the server," they must attach to this shared memory segment to gain access to server internal data structures.

The attachment procedure is handled by the NCPX Handler library initialization routines, so NCPX programs do not need to be concerned with this task.

However, application developers should be aware that the shared-memory segment is present in the address space of each and every NCPX Handler process. If the application inadvertently writes to any area of this shared memory segment, the NWS server will probably crash or malfunction. (This is not unlike the kind of behavior native

NetWare encounters when rogue NLMs stomp on areas of memory not assigned to them.)

NCPX programs are allowed to write to the shared memory segment of the NWS server, and access is limited to NCPX Handler library routines. The reason is that NCPX Handler library routines make extensive use of this shared memory segment for the registration and dispatch of Extended NCPs. The overhead associated with the un-mapping of the shared memory segment on a per-NCP basis would incur a substantial performance penalty on the servicing of Extended NCPs. Thus, NCPX applications, like native NetWare programs, are "trusted" extensions to the NWS server and presumed to be "trustworthy."

With NCP Extensions on AIX, the probability of an NCPX program bringing the entire AIX server down is extremely low. The reason for this low level of probability—compared with native NetWare—is that the NCPX programs run in separately scheduled and managed processes. The NCPX programs are not running on core OS threads as they are in native NetWare.

Nevertheless, the NWS server's shared memory segment represents a "window of liability" for NCPX applications. If the NCPX program stomps on the critical server state information stored in this segment, the NWS server will probably malfunction. Thus, developers need to ensure that their programs do *not* misbehave and do *not* violate this "trust."

NEMUX File Descriptor

A kernel module called NEMUX monitors the NWS server process execution and performs low-level dispatch of packets to NCP service engines. Since NCPX programs are a part of the server, they need to open a communication channel with NEMUX. The file descriptor is the NCPX program's handle to the NEMUX channel. This channel is opened by the NCPX Handler library initialization routines, so server-side programs do not need to perform this task.

NWS server control signals are delivered to the NCPX Handler process via the channel with NEMUX. The NCPX Handler library also uses this channel to send reply packets to clients.

Miscellaneous Requirements

The NCPX Handler library routines require a global character-array variable called *ExecName*[]. The string defined by this variable is included in various NCPX Handler library error messages as a means to identify the program which caused the error.

To meet this requirement, a programmer should include a definition like this in the application code:

```
char ExecName[] = “NCPX Program”;
```

Limitations

The architecture for the NWS server imposes some limitations on NCPX application. These are described below.

Single Threading

As discussed in “Process Model” on page 301, the NCPX Handler library functions are *not* thread-safe. Due to internal implementation constraints in NNS, NCPX applications cannot access the NCPX Handler library functions with more than a single thread.

This limitation extends to **NCPX_EventLoop**(): Only the thread that is “in” the EventLoop is allowed to call NCPX functions (during callback processing). Other threads can run background tasks in the NCPX Handler, but if other threads call NCPX functions, a deadlock will likely result.

Cannot Loop Back

Constraints imposed by NWS internal implementation also prevent the dispatch of multiple simultaneous NCP callbacks at once.

The EventLoop is a single thread of control, and hence cannot process “loopback” requests. What this means is that you cannot make NCP Extension request calls from within NCP callback routines.

Although it’s safe to call the registration and deregistration functions from an NCPX callback routine, it’s *not* safe to call NCPX client-side

functions (functions which invoke NCP Extensions on the NWS server) when running as apart of the server. Such “loopback” processing will surely hang the Handler and all clients using the NCPX application.

Size of NCPX Pool

In the NWS server’s shared-memory segment, a pool of memory is allocated specifically for NCPX data structures, message buffers, and the LightWeight message queues. This pool has a small “default” size, which imposes no significant additional shared memory requirements on NWS servers that do not need to use NCPX.

The pool is allowed to grow to sixteen megabytes (16 MB), subject to the availability of space in the NWS server’s shared memory segment. If the space is available in the shared memory heap, the space will be available for use by the server’s NCPX Handler library routines.

The Server’s overall shared memory size is controlled by the *shm_size* **nwcm** parameter. The default value for this parameter is small, usually 4 MB. This default size leaves little room for NCPX use. If your application makes heavy use of NCP Extensions on your server, you should tell system administrators to increase *shm_size* to make additional shared memory available for NCPX use. (System administrators also need to adjust the AIX system tunable *SHMMAX* to make the shared memory available to the NWS server.)

The maximum size of the NCPX pool is 16 MB. This means that the NCPX pool might grab as much as 16MB of server shared memory and link it onto the pool’s free list, making the memory unavailable to the rest of the NWS server. This condition will occur only on servers that use NCPX heavily.

To guarantee correct operation with heavy NCPX activity, system tell administrators to set their *SHMMAX* and *shm_size* parameters to 16 MB *plus* the normal *shm_size* requirement of the server. This allows the NCPX pool to grab 16MB from the shared memory pool, while leaving the remaining memory available for other NWS server operations.

Writing an NCPX Program

A server-side NCPX program is designed to handle processing of application-specific NCPs. This Handler program links with the

libncpx_han.so and libncpx_cmn.so libraries to import APIs used to register the NCPX Handler callback routines. (See “NCPX Handler Library Reference” on page 319 for a description of the routines.)

The application portion of the Handler contains **HandlerMain**(), which is called by the libncpx_han library to begin the application. (The **main**() function is provided by the libncpx_han library and provides process-startup functionality.)

The two processes associated with a Handler program, the parent and child, are discussed in “Handler Parents and Children” on page 302

The minimum NCPX program must perform the following steps:

1. Call **NWRegisterNCPExtension**() to register the NCP Extension(s) and their associated callback routines.

2. Call **NCPX_EventLoop**() to dispatch the NCPs.

3. On return from EventLoop, call **NWDeRegisterNCPExtension**() to deregister the NCP Extension(s).

Note

Multiple NCP Extensions can be registered before calling the EventLoop. The EventLoop code will properly dispatch the incoming Extended NCPs to the appropriate callback functions. Remember that each registered NCP Extension must be deregistered before program termination.

These steps are performed via **HandlerMain**(), which is called by the NCPX Handler library code in the Handler child process *after* the library has initialized resources required to support NCPX processing.

A callback routine is one of three developer-written functions in the NCPX Handler. These are functions designed to process events related to NCP processing. In the Handler process, the EventLoop code blocks on a LightWeight message queue until a message is available. When a message is received, the EventLoop “calls back” into the developer-written function to process the event.

Three types of callbacks can be associated with registered NCP Extensions:

- NCP— called when an NCP 37 message is received.
- Connection event—called when a connection is cleared or logged-out by the server.
- Reply buffer manager—called after the NCP response is sent.

  The reply buffer manager's "post processing" of the NCP allows the application code to release buffer resources which were allocated for the reply packet during processing of the NCP.

Callback functions are optional. An NCP Extension can be registered which has *no* callbacks. In this case, the only service provided by the Extension is the maintenance of the 32-byte *queryData* buffer which clients can retrieve using NCP 36 calls. However, Extensions are typically registered with at least the NCP Extension and connection event callbacks.

There is one caveat: In the NWS implementation, using a reply buffer manager callback actually decreases performance, due to architectural constraints on the allocation of reply buffers. (In native NetWare, use of a reply buffer manager can increase performance.) For further discussion, see "Reply Buffer Manager" on page 315.

The next section contains sample code for an NCPX Handler program. Subsequent sections discuss compiling and running NCPX Handler programs.

Code Example

The following minimal NCPX Handler program compiles, but does no useful work. You can use this "shell" as a template for your own NCPX applications.

```
#include <sys/ncpx_app.h>

/* Name of this program which can be referenced globally */
char            ExecName[] = "NCPX test program";

/
**********************************************************************/
```

```
BYTE
NCP_callback(NCPExtensionClient *client,
             void *requestData,
             LONG requestDataLength,
             void *replyData,
             LONG *replyDataLength)
{
    printf("NCP_Callback called for client %u, task %u.\n",
           client->connection,
           client->task);

    return 0;   /* SUCCESS! */
}

/
*****************************************************************************/

void
ConnectionEvent_callback(LONG connection,
                         LONG eventType)
{
    printf("ConnectionEvent_callback called for connection %u, event
0x%08lx.\n",
           connection,
           eventType);
}

/
*****************************************************************************/
/
*****************************************************************************/

int
HandlerMain( int argc, char *argv[] )
{
    int ccode;
    void *queryData;

    NCPX_EventLoopState el_state;

    /*****************************************/
    /* Register the extension. */

    ccode = NWRegisterNCPExtension("TEST EXTENSION",
                                   NCP_callback,
                                   ConnectionEvent_callback,
                           NULL,   /* No reply-buffer-manager callback. */
```

```
                                1,   /* major version */
                                2,   /* minor version */
                                3,   /* revision */
                                &queryData);
    if ( ccode != 0) {
        printf("%s had failure (ccode %d) registering NCP Extension.\n",
               ExecName, ccode);
        exit(1);
    }

    /*****************************************/
    /* Commence processing of NCPs: callbacks will come when we have work
to do. */

    NCPX_EventLoop( &el_state);

    /*****************************************/
    /* DeRegister the extension. */

    ccode = NWDeRegisterNCPExtension( queryData);
    if ( ccode != 0) {
        printf("%s had failure (ccode %d) DEregistering NCP Extension.\n",
               ExecName, ccode);
        exit(1);
    }

    return 0;
}
```

Compiling an NCPX Handler

A few special command-line options must be used when compiling NCPX Handler programs. These options ensure that the proper libraries are linked with the Handler.

The command-line options are listed in the table following:

-Kthread	This option ensures that proper C runtime startup code is included in the code (this is necessary to properly enable the mutual-exclusion locks used in the NWS server library code).
-lncpx_han	This option causes the NCPX Handler library to be linked with the NCPX Handler program.

-lnwu	This option causes additional NWS server libraries to be linked with the NCPX Handler program.
-lm	This option causes the math library to be linked with the NCPX Handler program (because some NWS server internal library routines use the math library, you must link it with NCPX Handler programs).

As an example, the following command can be used to compile the program shown in the code example beginning on page 311:

```
cc -Kthread -o  ncpx_prog  \
   ncpx_prog.c -l ncpx_han -lnwu -lm
```

Running an NCPX Handler

NCPX applications are invoked just like any other AIX program: that is, by typing the name of the program at the command line. To run the Handler program compiled above, use a command line such as the following:

```
ncpx_prog
```

The NCPX Handler library initialization code will check the process privileges, daemonize the process (fork and run in the background, returning the shell prompt to you), and perform the steps necessary to associate the Handler process with the running server. (If the NWS server isn't running when you run the NCPX application, the program exits with an error message indicating that the server isn't up.)

After the process initialization is complete, **HandlerMain**() runs, which runs your application-specific code to register NCP Extensions. When the initialization and registration of Extensions is complete, EventLoop runs, which causes the NCPX Handler process to sleep until an incoming Extended NCP is received by the server. At that point, the NCPX Handler library dispatches the work to the registered callback functions.

Programming Issues

The issues you must address in writing an NCPX application are discussed below.

Reply Buffer Manager

If you are going to use a reply buffer manager, specify it when you register the NCP Extension with the NWS server. The reply buffer manager is a routine that the NCPX Handler library calls after it has copied the NCPX Handler's reply to server shared memory.

The reply buffer manager does not allocate reply buffers. However, it can free the buffers that the Handler allocates.

Reply buffers are allocated in the following ways:

- The NCPX Handler library creates a single reply buffer and passes its address to the NCPX Handler.
- The NCPX Handler allocates a single reply buffer and returns a pointer to this buffer.
- The NCPX Handler allocates multiple reply buffers and returns a structure pointing to all of them.

If your NCP Extension does not use a reply buffer manager, the NCPX Handler library allocates a reply buffer that is the size specified by the client. The NCPX Handler library then passes a pointer to the allocated buffer as a parameter into your NCPX Handler. The Handler places its reply into the buffer, and the NCPX Handler library sends the data in the buffer to the client.

If your NCP Extension is going to return fragmented data, it must use a reply buffer manager. In this case, the NCPX Handler sets its *replyData* parameter to point to a structure containing pointers to multiple fragments. The Handler also sets its *replyDataLen* parameter to REPLY_BUFFER_IS_FRAGGED. The NCPX Handler library 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, documented in the client reference for **NWSendNCPExtensionFraggedRequest**(). The difference is that the structure the Handler returns can have more than four elements in its *fragList*. (The client is limited to four fragments, but the Handler has no limits to the number of fragments it can return.)

In native NetWare, using a reply buffer manager can significantly increase performance and decrease the amount of machine CPU

resource required to service an NCP Extension request. This performance gain is realized because the native NetWare NCPX libraries are able to copy data directly from the application-allocated buffer to the LAN driver, with no intervening copies. However, in the NNS implementation, using a reply buffer manager for NCPX on NWS can significantly *decrease* performance, due to architectural constraints on the allocation of reply buffers.

In native NetWare, reply data is copied into shared-memory before being returned to the client.

In NCPX on NWS, all reply data must be placed in the NWS server's shared-memory segment before being sent back to the client. To make the reply buffer available to the NCP service engines (which actually send the data back to the client), the data must be placed in the shared-memory segment.

- If you are *not* using a reply buffer manager, the reply buffer is allocated in shared-memory by the NCPX Handler library routines. The NCP callback fills in the shared-memory buffer directly.

- If you *are* using a reply buffer manager, the NCPX Handler library must copy the process-local memory allocated by the NCPX callback into the shared-memory reply buffer. This represents an additional copy of the buffer (process-local to shared memory).

Therefore, in NCPX on NWS, you will see better NCPX service performance if your NCPX application does *not* use a reply buffer manager.

Even given the performance considerations outlined above, there are good reasons for using a reply buffer manager. Some of these reasons are outlined below.

1. A reply buffer manager is required for 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 NCPX Handler simply returns a structure with 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.

2. A reply buffer manager serves as the second part of the NCPX Handler.

   In the case described in the previous paragraph, the 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.

Connection Event Callback

The connection event callback keeps track of when connections are freed or logged out. Depending on your application's service, this information could be important. For example, a service that limits 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.

On the other hand, a service that allows unlimited access may not be concerned with who is using it. If keeping track of connection status is not important to you, do not register a connection event callback when you register the NCP.

Deregistering Before Unloading

Before an NCPX server program unloads, it should deregister all NCP Extensions that it has registered.

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.

Registering Multiple NCP Extensions

Some service-providing NCPX programs offer more than one service. For example, if the service is a database, the following services could be made available:

- 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 with a switch statement?

If you choose to register five NCP Extensions, you must create five names for them. If you choose to use one NCP Extension, you need to create only one name.

If you choose to register one name and use a switch statement, your code might look like the following:

```
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);
}
}
```

NCPX Handler Library Reference

The APIs available on the native NetWare server have not been ported; there is no native-style NetWare CLIB.NLM. NCPX applications must use the equivalent AIX services. (Existing native NetWare NCPX NLMs will probably have to be rewritten to use the AIX services rather than native NetWare's CLIB interface.)

The NCPX Handler library for NNS (libncpx) provides the NCPX "interface" to the NWS server. It is loaded on AIX with the NNS package and consists of the following AIX shared-object libraries.

- libncpx_han.so

  This library contains APIs available to NCPX programs. It includes APIs for the registration and deregistration of NCP Extensions. It also contains code for the **main**() function and for the NCPX event-dispatch loop.

  NCPX programs actually link a number of other server libraries to provide packet-handling and connection-management functions. All of these extra libraries are "hidden" behind the interface provided by the NCPX Handler library, but the other libraries are required by the libncpx libraries in order to perform their tasks.

- libncpx_svr.so

  This library contains functions available to support NCPX dispatch in the NCP engines and the NWS NetWare daemon. This library is required by the NCP engine and "nwserver" executables, but not by applications.

- libncpx_cmn.so

  This library contains common support routines for both libncpx_han.so and libncpx_svr.so. Executables linking either of those libraries automatically link this one too.

Overview of Library routines

The NCPX Handler library provides several types of services. (The prototypes for the functions described below can be found in /usr/include/sys/ncpx_app.h.)

Initialization

The NCPX Handler library contains the **main**() function of the program. This means the library initialization routines run *before* the developer-written application code.

The initialization code performs several tasks which are required to make the NCPX program act as part of the server, including privilege checking, shared-memory attachments, NEMUX channel initialization, signal-handler initialization, etc.

The developer-written application code must begin with the function **HandlerMain**(*argc*, *argc*, *envp*). When the NCPX Handler library has finished initializing the process as part of the NWS server, it calls the developer-written **HandlerMain**() with the usual arguments (*argc*, *argc*, *envp*).

Registration

Several native NetWare-compatible functions are provided for the registration and deregistration of NCP Extensions:

NWRegisterNCPExtension()

NWRegisterNCPExtensionByID()

NWDeRegisterNCPExtension()

EventLoop

NCPX_EventLoop() controls the dispatch of incoming Extended NCPs to the developer-written callback functions.

After **HandlerMain**() has finished application-specific initialization and NCP Extension registration, it must immediately call **NCPX_EventLoop**() to begin the processing of Extended NCPs. As soon as the NCP Extension is registered, requests begin queuing for services, which are performed by the EventLoop. If an NCPX Handler registers an Extension and does not call the EventLoop, requests will queue up and the clients will “hang” waiting for service from the server.

The EventLoop returns to the developer-written application when there is an internal error occurs in the EventLoop code, or when all Extensions are deregistered, the Handler receives a signal to shut down, or when the server goes down.

Client Identification

One of the parameters to the NCP callback is the connection number. **NCPX_GetObjectName**() is provided to translate a connection number into the distinguished name of the object attached to the connection. This provides NCPX Handlers with a means of identifying the client for which a request is being processed.

Connection Status

Two functions are provided which return information about the "login state" of a client:

- **ConnectionIsLoggedIn**() determines whether the client originating a request is "logged-in"—that is, both authenticated to the Directory tree and licensed to the NNS server.

- **ConnectionIsAuthenticatedTemporary**() determines whether a client originating a request is in the "temporary authenticated" state.

Each function returns TRUE (nonzero) if the connection meets the specified condition, otherwise FALSE (zero).

When a client "attaches" (establishes a service connection) to a NNS server, the client first authenticates to the Directory tree. This is a "temporary authenticated" state. When the client then logs in to the server, the client is both authenticated and licensed.

Clients are *not* required to log in to the NWS server before sending NCPX packets to the server. They can be merely authenticated to the Directory tree.

Child Detachment

NCPX Handlers are allowed to **fork**() to create independent child processes. These children processes are *not* allowed to maintain any association with the server.

When a Handler **fork**()s, the child inherits all of the parent's attachments and associations with the server (shared memory, open files, signal disposition, etc). If these associations are not removed by

detaching the child process from the NWS server, the child prevents the server from going down.

To detach the forked child from the server, the forked child *must* call **NCPX_DetachForkedChildFromServer**()—This is not an optional step!

This library routine detaches the forked child process from the NWS server and allows the child process to continue running after the server is shut down. This functionality should be used whenever an NCPX Handler needs to fork off independent processes.

Index to NCPX Functions

NCPX_EventLoop

Controls dispatch of incoming Extended NCPs.

Syntax

```
#include “ncpx_app.h”

int  NCPX_EventLoop(
     NCPX_EventLoopState *exitReason);
```

Parameters

exitReason

(OUT) Passes a pointer to a variable, the value of which identifies the reason why the EventLoop has exited.

Return Values

0	Successful
non-zero	Unsuccessful

Remarks

NCPX_EventLoop() controls the dispatch of incoming Extended NCPs to the developer-written callback functions. It also returns the results to the NWS server (which then returns the results to the client which originated the request).

NCPX_EventLoop() should be called after **HandlerMain**() has finished application-specific initialization and NCP Extension registration. Calling **NCPX_EventLoop**() is not optional and should not be delayed because Extended NCPs begin queuing for services as soon as the Extension is registered.

NCPX_EventLoop() has one parameter: This is a pointer to a variable of type *NCPX_EventLoopState*. When the EventLoop returns, it will set the *NCPX_EventLoopState* variable to a value identifying the reason why the EventLoop has exited.

```
typedef enum {
   EL_RUNNING = 0, /* Never returned. */
   EL_EXIT_ERROR,
   EL_EXIT_ALL_HANDLERS_DEREGISTERED,
   EL_EXIT_SERVER_GOING_DOWN,
   EL_EXIT_SIGNAL_SHUTDOWN,
} NCPX_EventLoopState;
```

Example

```
int
HandlerMain( int argc, char *argv[] )
{
    int ccode;
    void *queryData;

    NCPX_EventLoopState el_state;

    /*****************************************/
    /* Register the extension. */

    ccode = NWRegisterNCPExtension("TEST EXTENSION",
                              NCP_callback,
                              ConnectionEvent_callback,
                         NULL,    /* No reply-buffer-manager callback. */
                              1,   /* major version */
                              2,   /* minor version */
                              3,   /* revision */
                              &queryData);
    if ( ccode != 0) {
        printf("%s had failure (ccode %d) registering NCP Extension.\n",
               ExecName, ccode);
        exit(1);
    }

    /*****************************************/
    /* Commence processing of NCPs: callbacks will come when we have work
to do. */

    NCPX_EventLoop( &el_state);

    /*****************************************/
    /* DeRegister the extension. */
```

```
    ccode = NWDeRegisterNCPExtension( queryData);
    if ( ccode != 0) {
        printf("%s had failure (ccode %d) DEregistering NCP Extension.\n",
                ExecName, ccode);
        exit(1);
    }

    return 0;
}
```

NWRegisterNCPExtension

Using a specific name, register an NCP Extension with Novell Network Services 4.1 for AIX.

Syntax

```
#include “ncpx_app.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) Specifies the name of the NCP Extension.

NCPExtensionHandler

(IN) Specifies the function that is to be executed when the NCP Extension is called with **NWSendNCPExtensionRequest**(). (See “Remarks” for the use of NULL in this field.)

ConnectionEventHandler

(IN) Specifies the function that to be called when a connection is logged out or terminated on the server. (See “Remarks” for the use of NULL in this field.)

ReplyBufferManager

(IN) Specifies a reply buffer manager that can be used to manage buffers used to reply to NCP Extension requests. (See “Remarks” for the use of NULL in this field.)

majorVersion

(IN) Specifies the major version number of the service provider.

minorVersion

(IN) Specifies the minor 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) Receives a pointer to a 32-byte area that the NCPX Handler library allocated. This buffer is used by the service provider to return up to 32 bytes of information to the client.

Return Values

0	(0x00)	SUCCESSFUL
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 was longer than the 32-byte limit

Remarks

NWRegisterNCPExtension() registers the NCP Extension with NNS by using the name of the NCP Extension associated with the application service. This returns a dynamic ID that is valid until the service providing the Handler is terminated.

After an NCP Extension has been registered, clients can access the NCP Extension. The Extension remains valid until the service-providing program deregisters the NCP Extension.

NCPExtensionName is the name that the NCP Extension uses as an identifier in the list of NCP extensions. NCP Extension names are case-sensitive and must be unique. They have a maximum length of 32 bytes plus a NULL terminator. For more information, see “NCP Extension Names” on page 296.

When you call **NWRegisterNCPExtension**() to register an NCP Extension, three of the parameters are functions that are called as part of the Handler service. These parameters are *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager*.

NCPExtensionHandler is a service routine (function) that is called when the client uses**NWSendNCPExtensionRequest**()to call the NCP Extension. In most cases, you want to provide an NCPX Handler. However, if your service can provide all information needed by updating the 32-byte *queryData* buffer, you do not need an NCPX Handler and can pass NULL for the parameter. The clients would then

obtain the information in the *queryData* buffer by calling **NWGetNCPExtensionInfo**() or **NWScanNCPExtensions**(). This is a passive method of passing information. The NCP Extension is not notified that an access has been made to its *queryData*.

The *NCPExtensionHandler* routine takes the following parameters:

client

(IN) A pointer to an NCPExtensionClient structure (shown below) containing the connection and task of the calling client. The client pointer can be used by the *ReplyBufferManager* to associate the request with the reply notification it receives.

```
typedef struct {
    LONG connection;
    LONG task;
} NCPExtensionClient;
```

requestData

(IN) A pointer to a buffer holding the request information.

requestDataLen

(IN) The size (in bytes) of the data in the request buffer.

replyData

(OUT) If the *ReplyBufferManager* parameter of **NWRegisterNCPExtension**() is set to NULL, this is a pointer to a buffer where the service routine can place its response data.

If a *ReplyBufferManager* was specified, this parameter points to the address of a pointer which the NCPX Handler sets to a valid buffer that it has created.

replyDataLen

(IN / OUT) On input, this is the maximum size (in bytes) of information that can be stored in the reply buffer.

On output, this is the actual number of bytes that NCPX Handler stored in the reply buffer.

You might want to have an NCPX Handler return other status information (such as failure reasons) to the client. If you do this, do not use any return values that have been defined for this call. The risk in returning values other than SUCCESSFUL is that future versions of

NNS might return values that you have defined, leaving you unsure of the return value's meanings.

A better way for your NCPX Handler to return status information is to have it always return SUCCESSFUL and then use a "status" field in the *replyData* buffer. This technique guarantees that the status information is always from the NCPX Handler.

The *ConnectionEventHandler* callback parameter keeps track of when connections are freed or logged out. If keeping track of connection status is not important to you, you can pass NULL for the *ConnectionEventHandler* when you register the NCP Extension. For a discussion on when to use connection event management, see "Connection Event Callback" on page 317.

The *ConnectionEventHandler* routine has the following parameters:

connection

(IN) The number of the connection that was logged out or cleared. (This notification is for all connections that are logged out or cleared. The connection information is *not* always about an NCPX client.)

eventType

(IN) This names the type of event that is being reported. This parameter returns the following values:

CONNECTION_BEING_FREED	Either the client has made a call to return its connection, or the server has cleared the connection (watchdog).
CONNECTION_BEING_LOGGED_OUT	The client has made a call to log out.

ConnectionEventHandler does not return a value.

ReplyBufferManager is a function that is used if the service-providing application wants to take care of reply buffer management for itself. This function takes the following parameters:

client

(IN) A pointer to an NCPExtensionClient structure (shown below) containing the connection and task of the calling client.

```
typedef struct {
    LONG connection;
    LONG task;
} NCPExtensionClient;
```

replyBuffer

(IN) A pointer to a buffer whose information has been returned to the client.

Most cases do not require a reply buffer manager. If you do not need one, pass NULL for this parameter. For a discussion of when to use a reply buffer manager, see “Reply Buffer Manager” on page 315.

The *majorVersion*, *minorVersion*, and *revision* parameters allow you to identify the version and revision of your service provider.

The *queryData* buffer is used by the service provider to return up to 32 bytes of information to the client.The pointer is also used by the registering NCPX application as the NCP Extension handle, when calling **NWDeRegisterNCPExtension**().

Returning the contents of the update buffer to the client also provides a one-way, passive information passing scheme. Your service provider can use the buffer to supply periodic update information to its clients. This information can then be retrieved with a call to **NWGetNCPExtensionInfo**() or **NWScanNCPExtensions**().

Example

```
int
HandlerMain( int argc, char *argv[] )
{
    int ccode;
    void *queryData;

    NCPX_EventLoopState el_state;
```

```
/*****************************************/
/* Register the extension. */

ccode = NWRegisterNCPExtension(“TEST EXTENSION”,
                           NCP_callback,
                           ConnectionEvent_callback,
                      NULL,    /* No reply-buffer-manager callback. */
                           1,   /* major version */
                           2,   /* minor version */
                           3,   /* revision */
                           &queryData);
if ( ccode != 0) {
    printf(“%s had failure (ccode %d) registering NCP Extension.\n”,
           ExecName, ccode);
    exit(1);
}
```

See Also

NWDeRegisterNCPExtension()
NWRegisterNCPExtensionByID()

NWRegisterNCPExtensionByID

Using a specific ID, register an NCP Extension with Novell Network Services 4.1 for AIX.

Syntax

```
#include “ncpx_app.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) Gives the well-known ID for the NCP Extension. This is a specific ID assigned by Novell’s Developer Support to be associated with your application service.

NCPExtensionName

(IN) Gives the name that identifies the NCP Extension.

NCPExtensionHandler

(IN) Specifies the function to be executed when the NCP Extension is called with **NWSendNCPExtensionRequest**().

ConnectionEventHandler

(IN) Specifies the function to be called when a connection is logged out or terminated on the server.

ReplyBufferManager

(IN) Specifies a reply buffer manager that can be used to manage buffers used to reply to NCP Extension requests.

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) Receives a pointer to a 32-byte area that the NCPX Handler library allocated. This buffer is used by the service provider to return up to 32 bytes of information to the client.

Return Values

0	(0x00)	SUCCESSFUL
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 request was made on a server version that does not support this function.
255	(0xFF)	ERR_BAD_PARAMETER The *NCPExtensionName* parameter was longer than the 32-byte limit.

Remarks

NWRegisterNCPExtensionByID() registers the NCP Extension with NetWare using a well-known ID that is assigned by Novell's Developer Support. This ID is always associated with the application's service.

After an NCP Extension has been registered, clients can access the NCP Extension. The Extension remains valid until the service-providing program deregisters the NCP Extension.

NCPExtensionID is the ID for the NCP Extension associated with your application service. To be assigned a well-known ID, you should contact Novell's Developer Support. Your ID must be unique and within the valid range. For more information, refer to "NCP Extension IDs" on page 296.

NCPExtensionName is the name that the NCP Extension uses as an identifier in the list of NCP Extensions. NCP Extension names are case-sensitive and must be unique. They have a maximum length of 32 bytes plus a NULL terminator. For more information about NCP extension names, refer to "NCP Extension Names" on page 296.

When you call **NWRegisterNCPExtensionByID**() to register an NCP Extension, three of the parameters are functions that are called as part

of the Handler service. These parameters are *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager*. For a more detailed description of the parameters for *NCPExtensionHandler*, *ConnectionEventHandler*, and *ReplyBufferManager*, see the documentation for **NWRegisterNCPExtension**() beginning on page 326.

NCPExtensionHandler is a service routine (function) that is called when the client calls the NCP Extension with **NWSendNCPExtensionRequest**() or **NWSendNCPExtensionFraggedRequest**().

ConnectionEventHandler keeps track of when connections are freed or logged out. If keeping track of connection status is not important to you, you can pass NULL for the *ConnectionEventHandler* when you register the NCP Extension. For a discussion on when to use connection event management, see “Connection Event Callback” on page 317.

ReplyBufferManager is a function that is used if the service-providing application wants to take care of reply buffer management for itself. Most cases do not require a reply buffer manager. If you do not need one, pass NULL for this parameter. For a discussion of when to use reply buffer management, see “Reply Buffer Manager” on page 315.

The *majorVersion*, *minorVersion*, and *revision* parameters allow you to identify the version and revision of your service provider.

The *queryData* buffer is used by the service provider to return up to 32 bytes of information to the client.The pointer is also used by the registering NCPX application as the NCP Extension handle, when calling **NWDeRegisterNCPExtension**().

Returning the contents of the update buffer to the client also provides a one-way, passive information passing scheme. Your service provider can use the buffer to supply periodic update information to its clients. This information can then be retrieved with a call to **NWGetNCPExtensionInfo**() or **NWScanNCPExtensions**().

See Also

NWRegisterNCPExtension()
NWDeRegisterNCPExtension()

NWDeRegisterNCPExtension

Remove an NCP Extension from NNS that was previously registered.

Syntax

```
#include “ncpx_app.h ”

int  NWDeRegisterNCPExtension(
   void *queryData);
```

Parameters

(IN) *queryData*

Specifies the Extension handle used to identify the NCP Extension.

Return Values

0	(0x00)	SUCCESSFUL The NCP Extension was deregistered.
255	(0xFF)	ERR_NO_ITEMS_FOUND The NCP Extension has already been deregistered.

Remarks

NWDeRegisterNCPExtension() removes an NCP Extension from the NWS server’s list of NCP Extensions. If a program has more than one NCP Extension registered, it must call **NWDeRegisterNCPExtension**() for each extension that it has registered.

Outstanding NCP Extension requests are not guaranteed to complete successfully after **NWDeRegisterNCPExtension**() is called.

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.

When an NCP Extension is registered with either **NWRegisterNCPExtension**() or **NWRegisterNCPExtensionByID**(), the address of the *queryData* pointer is passed as one of the parameters. This pointer is then initialized to point to a 32-byte area of memory in which the service provider can place data. This *queryData* pointer is used here as a handle for deregistering the NCP Extension.

See Also

NWRegisterNCPExtension()
NWRegisterNCPExtensionByID()

NCPX_GetObjectName

Returns the distinguished name of a logged-in object.

Syntax

```
#include “ncpx_app.h ”

int NCPX_GetObjectName(
    LONG connectionNumber,
    char *nameBuf,
    int bufLen);
```

Parameters

connectionNumber

(IN) The number of the connection slot where the object is logged-in.

nameBuf

(OUT) Passes a pointer to the buffer where the object’s name will be stored.

bufLen

(IN) Size (in bytes) of the *nameBuf*.

Return Values

0	SUCCESSFUL
0xFD	UNSUCCESSFUL Bad station number (connection).

Remarks

NCPX_GetObjectName() returns the name of the object logged-in at the given connection slot. It translates a connection number into the name of the object logged-in on the connection. This provides NCPX

Handlers with a means of identifying the client for which a request is being processed.

The *connectionNumber* parameter allows the Handler to “look up” the distinguished name of the object logged-in on the connection. If the given *connectionNumber* is invalid, return a 0xFD. Otherwise, fill in the *nameBuf* and return a zero.

If the connection isn’t logged in, the returned name is an empty string.

If there is an error building the name, the returned name is also an empty string.

Example

```
BYTE
NCPCallback(NCPExtensionClient *client,
void *requestData,
LONG requestDataLength,
void *replyData,
LONG *replyDataLength)
{
char buffer[ 256];

if ( NCPX_GetObjectName( client->connection, buffer, 256) == 0)
printf(“Object name is ‘%s’\n”, buffer);
else
printf(“Object not LOGGED IN.\n”);

return 0;
}
```

ConnectionIsLoggedIn

Determines whether a client is logged-in to the NNS server.

Syntax

```
#include “ncpx_app.h ”

int  ConnectionIsLoggedIn(
     LONG connectionNumber);
```

Parameters

connectionNumber

(IN) The number of the connection slot where the client is attached.

Return Values

FALSE	(0)	CONNECTION_NOT_LOGGED_IN
TRUE	(non-zero)	CONNECTION_LOGGED_IN

Remarks

ConnectionIsLoggedIn() determines whether the client originating a request is “logged-in”—that is both authenticated to the Directory tree and licensed to the NNS server. If **ConnectionIsLoggedIn**() returns FALSE, the client might be either “attached” (service connection only) or in a “temporary authenticated” state.

Clients are *not* required to log in to the NWS server before sending NCPX packets to the server. They can be merely authenticated to the Directory tree.

If you need to know whether the client is licensed to use the full resources of the server, use **ConnectionIsLoggedIn**() to ensure that clients are “logged in”.

Example

```
BYTE
NCPCallback(NCPExtensionClient *client,
void *requestData,
LONG requestDataLength,
void *replyData,
LONG *replyDataLength)
{
if ( ConnectionIsLoggedIn( client->connection))
printf("Connection is LOGGED IN\n");
else if ( ConnectionIsAuthenticatedTemporary( client->connection))
printf("Connection is AUTHENTICATED TEMPORARY\n");
else
printf("Connection is ATTACHED\n");

return 0;
}
```

See Also

ConnectionIsAuthenticatedTemporary()

ConnectionIsAuthenticatedTemporary

Determines whether a client's login state is temporary authentication.

Syntax

```
#include "ncpx_app.h"

int  ConnectionIsAuthenticatedTemporary(
     LONG connectionNumber);
```

Parameters

connectionNumber

(IN) The number of the connection slot where the client is attached.

Return Values

FALSE	(0)	Connection is *not* in the authenticated temporary state.
TRUE	(non-zero)	Connection is in authenticated temporary state.

Remarks

ConnectionIsAuthenticatedTemporary() determines whether a client originating a request has authenticated to the Directory tree. When a client "attaches" (establishes a service connection) to a NNS server, the client first authenticates to the Directory tree. This is a "temporary authenticated" state. When the client then logs in to the server, the client is both authenticated and licensed.

Clients are *not* required to log in to the NWS server before sending NCPX packets to the server. They can be merely authenticated to the Directory tree. Although **ConnectionIsAuthenticatedTemporary**() allows you to discriminate between clients in the "temporary

authenticated” state and those that are not (either “attached” or “logged in”), note that a “logged-in” client is also authenticated.

Example

```
BYTE
NCPCallback(NCPExtensionClient *client,
void *requestData,
LONG requestDataLength,
void *replyData,
LONG *replyDataLength)
{
if ( ConnectionIsLoggedIn( client->connection))
printf(“Connection is LOGGED IN\n”);
else if ( ConnectionIsAuthenticatedTemporary( client->connection))
printf(“Connection is AUTHENTICATED TEMPORARY\n”);
else
printf(“Connection is ATTACHED\n”);

return 0;
}
```

See Also

ConnectionIsLoggedIn()

NCPX_DetachForkedChildFromServer

Detaches a forked child process from NetWare resources.

Syntax

```
#include “ncpx_app.h”

int  NCPX_DetachForkedChildFromServer(
     void);
```

Parameters

None.

Return Values

0	Successful
-1	Unsuccessful

Remarks

NCPX_DetachForkedChildFromServer() detaches from NWS server resources and closes all file descriptors. It is called *only* by children processes which are forked off during NCP callback processing.

In the forked child, this call should be immediately followed by a call to **exec**().

NCPX Handlers are allowed to **fork**() and create independent child processes, but these children processes are *not* allowed to maintain any association with the server.

When a Handler **fork**()s, the child inherits all of the parent’s attachments and associations with the NWS server (shared memory, open files, signal disposition, etc.). If these associations are not removed

by detaching the child process from the server, the child prevents the server from going down.

To detach the forked child from the server, the forked child must call **NCPX_DetachForkedChildFromServer**()—This is not an optional step!

This library routine detaches the forked child process from the NWS server and allows the child process to continue running after the server is shut down. This functionality should be used whenever an NCPX Handler needs to fork off independent processes.

Example

```
BYTE
run_app(NCPExtensionClient *client,
void *requestData,
LONG requestDataLength,
void *replyData,
LONG *replyDataLength)
{
pid_tpid;
struct sigactionaction;
char *command;

/* Set signal action stuff so children don’t become zombies
 * AND we don’t get SIGCHLD when children stop and continue.
 */

/* First get old action information. */
if ( sigaction( SIGCHLD, NULL, &action)) {
printf(“run_app: sigaction() error (get), errno %d\n”, errno);
return 0xff;
}
action.sa_flags |= SA_NOCLDWAIT;/* Prevent children from becoming zombies.
*/
action.sa_flags |= SA_NOCLDSTOP;/* Don’t notify us when children stop/
continue. */
if ( sigaction( SIGCHLD, &action, NULL)) {
printf(“run_app: sigaction() error (set), errno %d\n”, errno);
return 0xff;
}

/* Since parent isn’t going to wait around for the child to finish,
 * we can’t depend on requestData being valid in the child (since the
 * parent will cause the buffer to be reused immediately).
```

```
 * So we have to make a copy of any data we're passing to the child
 * here, BEFORE the fork().
 */
command = strdup( requestData);
if ( command == NULL) {
return 0xff;/* failure to dup() the command line. */
}

pid = fork();
if ( pid == -1) {
free( command);
return 0xff;/* Failure to fork. */
}
if ( pid != 0) {
/* Parent. */
free( command);/* Get rid of unneeded-in-parent command-line copy. */
return 0;
}

/* Else we're the Child. */

/* Detach from the server environment.
 */
NCPX_DetachForkedChildFromServer();

/* Become session leader (detach from parent). */
if ( setsid() == -1) {
printf("setsid() failure, errno %d\n", errno);
}

execlp( command, command, NULL);/* Shouldn't return */

/* Exit with error code if execlp returns. */
exit( 1);

}
```

Index